

# Introducción a la PROGRAMACIÓN



Introducción a la

# PROGRAMACIÓN

Autor:  
Oscar Alejandro González Bustamante



## UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

### **Rector**

Juan Ramón de la Fuente

### **Secretario General**

Enrique del Val Blanco

### **Director General de Servicios de Cómputo Académico**

Alejandro Pisanty Baruch

### **Directora de Cómputo para la Docencia**

Carmen Bravo Chaveste

## **Guías y Textos de Cómputo. Introducción a la PROGRAMACIÓN**

### **Coordinación de la publicación**

Juana Figueroa Reséndiz

### **Autor**

Oscar Alejandro González Bustamante

### **Revisión técnica**

Juana Figueroa Reséndiz

Lidia Loreli Zamora Nunfio

### **Corrección de estilo**

Martha Patricia García Morales

Lucero Ledón Martínez

### **Diseño editorial y de portada**

Gabriela Lilí Morales Naranjo

### **Editor**

DR © Universidad Nacional Autónoma de México

Ciudad Universitaria, Coyoacán, CP. 04510, México, DF.



### **DIRECCIÓN GENERAL DE SERVICIOS DE CÓMPUTO ACADÉMICO**

Circuito exterior s/n, Ciudad Universitaria, Coyoacán, CP. 04510, México DF.

Prohibida la reproducción total o parcial por cualquier medio  
sin la autorización escrita del titular de los derechos patrimoniales

1ª. edición 2007

ISBN 978-970-32-3776-0

Impreso y hecho en México

## Contenido

---

|                                                              |    |
|--------------------------------------------------------------|----|
| Introducción .....                                           | 1  |
| 1. Lenguajes de programación .....                           | 3  |
| 1.1 Definiciones .....                                       | 4  |
| 1.2 Paradigmas de la programación .....                      | 4  |
| 1.2.1 Programación funcional y estructurada .....            | 4  |
| 1.2.2 Programación orientada a objetos .....                 | 5  |
| 1.2.3 Programación lógica .....                              | 8  |
| 1.2.4 Programación orientada a datos .....                   | 9  |
| 1.3 Traductores .....                                        | 9  |
| 1.3.1 Intérpretes .....                                      | 9  |
| 1.3.2 Compiladores .....                                     | 10 |
| 1.4 Código fuente, objeto y ejecutable .....                 | 11 |
| 1.4.1 Entornos de desarrollo integrado .....                 | 12 |
| 1.4.2 Depurador .....                                        | 12 |
| 2. Elementos básicos de la programación estructurada .....   | 14 |
| 2.1 Tipos de datos primitivos .....                          | 14 |
| 2.1.1 Numéricos (enteros y de punto flotante o reales) ..... | 15 |
| 2.1.2 Carácter y cadena .....                                | 15 |
| 2.1.3 Lógicos o Booleanos (falso, verdadero) .....           | 16 |
| 2.2 Palabras reservadas .....                                | 16 |
| 2.3 Variables .....                                          | 18 |
| 2.4 Constantes .....                                         | 18 |
| 2.5 Expresiones .....                                        | 19 |
| 2.6 Operadores y operandos .....                             | 19 |

|                                                                 |    |
|-----------------------------------------------------------------|----|
| 2.6.1 Operadores unarios.....                                   | 19 |
| 2.6.2 Operadores binarios.....                                  | 20 |
| 2.6.3 Operadores especiales .....                               | 20 |
| 2.7 Jerarquía de operadores.....                                | 20 |
| 2.7.1 Precedencia de operadores .....                           | 20 |
| 2.7.2 Reglas de evaluación de expresiones.....                  | 21 |
| 2.8 Operadores aritméticos .....                                | 21 |
| 2.8.1 Suma .....                                                | 21 |
| 2.8.2 Resta .....                                               | 22 |
| 2.8.3 Multiplicación .....                                      | 22 |
| 2.8.4 División .....                                            | 22 |
| 2.8.5 Módulo .....                                              | 22 |
| 2.8.6 Potencia .....                                            | 22 |
| 2.9 Operadores relacionales.....                                | 22 |
| 2.9.1 Mayor que .....                                           | 23 |
| 2.9.2 Menor que.....                                            | 23 |
| 2.9.3 Mayor o igual que .....                                   | 23 |
| 2.9.4 Menor o igual que.....                                    | 23 |
| 2.9.5 Igual a.....                                              | 23 |
| 2.9.6 Diferente de .....                                        | 24 |
| 2.10 Operadores lógicos o booleanos (and, or, not, xor ) .....  | 24 |
| 2.10.1 Tablas de verdad.....                                    | 24 |
| 2.10.2 Operadores de cadena o alfanuméricas.....                | 25 |
| 2.11 Precedencia de operadores y evaluación de expresiones..... | 25 |
| 2.11.1 Asignación .....                                         | 25 |
| 2.11.2 Funciones en las expresiones .....                       | 26 |
| 2.12 Uso de paréntesis.....                                     | 27 |
| 3. Control de flujo de sentencias.....                          | 29 |
| 3.1 Sentencias incondicionales.....                             | 32 |
| 3.1.1 Asignación.....                                           | 32 |
| 3.1.2 Lectura .....                                             | 33 |
| 3.1.3 Escritura .....                                           | 33 |
| 3.1.4 Transferencia incondicional de secuencia (Go to).....     | 34 |
| 3.1.5 Bloque de sentencias .....                                | 34 |
| 3.1.6 Identación.....                                           | 34 |
| 3.2 Sentencias condicionales .....                              | 35 |
| 3.2.1 Selección (if) .....                                      | 35 |
| 3.2.2 Ciclos (while, for).....                                  | 42 |
| 3.2.3 Selección múltiple (switch o select – case) .....         | 48 |

|                                                                   |     |
|-------------------------------------------------------------------|-----|
| 3.2.4 Ejemplos de estructuras de control .....                    | 52  |
| 4. Variables con subíndice o arreglos .....                       | 59  |
| 4.1 Definición .....                                              | 59  |
| 4.2 Arreglos unidimensionales.....                                | 59  |
| 4.3 Arreglos bidimensionales .....                                | 61  |
| 4.4 Arreglos multidimensionales (poliedros).....                  | 67  |
| 5. Funciones .....                                                | 69  |
| 5.1 Concepto de función en programación .....                     | 70  |
| 5.2 Llamada o invocación a una función .....                      | 70  |
| 5.3 Parámetros .....                                              | 70  |
| 5.3.1 Parámetros por valor .....                                  | 71  |
| 5.3.2 Parámetros por referencia .....                             | 71  |
| 5.4 Valor de retorno .....                                        | 71  |
| 6. Elementos básicos de la programación orientada a objetos ..... | 79  |
| 6.1 Concepto de objeto.....                                       | 79  |
| 6.2 Anatomía de un objeto .....                                   | 80  |
| 6.3 Beneficios de la programación orientada a objetos .....       | 81  |
| 7. Clases y objetos.....                                          | 82  |
| 7.1 Definición de una clase.....                                  | 82  |
| 7.2 Miembros de una clase.....                                    | 83  |
| 7.2.1 Propiedades .....                                           | 83  |
| 7.2.2 Métodos .....                                               | 87  |
| 7.2.3 Constructores y creación de objetos .....                   | 88  |
| 7.2.4 Acceso a propiedades y métodos .....                        | 89  |
| 7.2.5 Destructores .....                                          | 89  |
| 8. Encapsulamiento .....                                          | 94  |
| 8.1 Modularidad.....                                              | 94  |
| 8.2 Ocultamiento de la implementación .....                       | 95  |
| 8.3 Encapsulamiento eficaz.....                                   | 95  |
| 8.4 Protección de variables y métodos .....                       | 95  |
| 8.4.1 Miembros privados .....                                     | 96  |
| 8.4.2 Miembros públicos .....                                     | 96  |
| 8.4.3 Miembros protegidos .....                                   | 96  |
| 8.4.4 Implementando el encapsulamiento .....                      | 96  |
| 9. Herencia .....                                                 | 108 |
| 9.1 Jerarquía de clases.....                                      | 108 |

|                                                  |     |
|--------------------------------------------------|-----|
| 9.1.1 Generalización y especialización.....      | 108 |
| 9.2 Mecánica de la herencia.....                 | 110 |
| 9.2.1 El uso de super.....                       | 110 |
| 9.3 Clases abstractas .....                      | 110 |
| 9.4 Polimorfismo.....                            | 115 |
| 9.4.1 Enlazamiento temprano.....                 | 116 |
| 9.4.2 Enlazamiento tardío .....                  | 116 |
| 10. Interfaz gráfica .....                       | 121 |
| 10.1 Definición de GUI .....                     | 122 |
| 10.2 Formularios .....                           | 123 |
| 10.3 Contenedores .....                          | 124 |
| 10.4 Componentes .....                           | 124 |
| 10.4.1 Layouts o administradores de diseño ..... | 128 |
| 10.5 Eventos .....                               | 131 |
| Bibliografía .....                               | 136 |

## Introducción

---

Este manual de Introducción a la Programación tiene el objetivo de enseñarle al lector los principios básicos de la programación de equipos informáticos con lenguajes de programación; para ello, se discuten los dos principales paradigmas de la programación más utilizados en la actualidad, y que son: el procedural o funcional y el de la programación orientada a objetos.

En el primer capítulo se presentan los conceptos básicos de traductores, compiladores, intérpretes, ensamblador, lenguaje de programación, entorno de programación integrado, etc., los cuales son válidos para todos los lenguajes de programación.

Después viene la parte de la programación procedural o funcional. Desde el capítulo 2, se discuten conceptos tales como variables, constantes, operadores, jerarquía de operadores, tipos de datos. El capítulo 3 está dedicado a las estructuras de control de flujo, incondicionales, condicionales, ciclos, algoritmos y diagramas de flujo. El capítulo 4 trata los arreglos o variables con índice unidimensionales, bidimensionales y poliedros. Por su parte, el capítulo 5 trata el tema de funciones, módulos, procedimientos, parámetros por valor y por referencia, valor de retorno, etc.

En seguida de esto, viene la parte dedicada a la programación orientada a objetos. A partir del capítulo 6 se discuten conceptos tales como: objeto, atributos, comportamientos, identidad de los objetos, etc. En el capítulo 7, se ven los conceptos de clase, constructores, variables miembro, métodos, sobrecarga de métodos, instancias de una clase, etc. En el capítulo 8, se presenta el concepto de abstracción y encapsulamiento, ámbito de acceso, ocultamiento de la implementación, métodos accesorios. En tanto en el capítulo 9, se discute lo qué es la herencia, la sobreescritura de métodos, clases abstractas, y el polimorfismo con enlazamiento temprano y tardío.

El último capítulo trata sobre el concepto de interfaz gráfica de usuario, ya que a partir de la aparición de los nuevos sistemas operativos con ventanas, botones, menús, etc., muchísimas aplicaciones informáticas utilizan dicha interfaz, y los modernos lenguajes de programación permiten su diseño.

Todos estos conceptos se presentan con ejemplos en lenguaje de programación Java, lo cual no impide que el lector pruebe otro lenguaje de programación adecuado para ejemplificarlos.

Aunque este manual va dirigido a personas que quieran comprender los conceptos básicos de la programación para aprender un lenguaje de programación, esto no obstaculiza el que otras personas lo necesiten para un software a nivel avanzado donde se requiera programar, e igual en la reafirmación de sus conocimientos o como obra de consulta.

# CAPÍTULO

# 1

## Lenguajes de programación

---

Los primeros lenguajes de programación empezaron a crearse en la década de los 50, gracias al importante adelanto en el diseño de las computadoras, cuando el científico John Neumann tuvo la idea de que la computadora no debería ser “cableada” para ejecutar algo en particular, sino que podría lograrse con una serie de códigos almacenados como datos que determinaran las acciones ejecutadas por una unidad de procesamiento central.

Pronto los programadores se dieron cuenta de que sería de gran ayuda asignar símbolos a los códigos de instrucción, así como a las localidades de memoria, de esta manera nació el *lenguaje ensamblador*. Un lenguaje ensamblador es una forma simbólica del lenguaje de máquina de la computadora, caracterizado por el uso de nemónicos que representan operaciones de máquina y quizás direcciones simbólicas de la memoria de la computadora, por lo que las computadoras se programaban con instrucciones como las siguientes:

```
.model small
.stack
.data
Cadena1 DB 'Hola Mundo.$'
.code

programa:
    mov ax, @data
    mov ds, ax
    mov dx, offset Cadena1
    mov ah, 9
    int 21h
end programa
```

Pero el lenguaje ensamblador, depende directamente de la arquitectura de *hardware* de cada máquina en particular, tiene bajo nivel de abstracción y dificultad de escribirse y comprenderse, por tanto, no es lo que usualmente percibimos como un lenguaje de programación. Algunas veces al lenguaje ensamblador se le conoce como *lenguaje de bajo nivel*, a fin de distinguirlo de los *lenguajes de alto nivel*, que son propiamente los que utilizan los programadores hoy en día, para poder programar a las computadoras.

Ciertamente los programadores pronto se dieron cuenta de que un nivel más elevado de abstracción mejoraría su capacidad de escribir instrucciones concisas y comprensibles que podrían ser utilizadas con pocos cambios de una máquina a otra.

## **1.1 Definiciones**

Para los temas siguientes, requerimos entender lo qué es un lenguaje de programación cuya definición tenemos a continuación: un *lenguaje de programación* es un sistema notacional para describir computaciones en una forma legible tanto para el ser humano como para la máquina. Una computación es una operación o cálculo realizado por la computadora.

Un programa es un conjunto de sentencias o instrucciones escritas en algún lenguaje de programación, que le indican a la computadora lo que debe hacer<sup>1</sup>.

## **1.2 Paradigmas de la programación**

Por paradigma se entiende una de tantas formas en que se pueden hacer las cosas. Así, una computadora es posible programarla siguiendo algunos de los paradigmas siguientes.

### **1.2.1 Programación funcional y estructurada**

El paradigma funcional parte de la descripción de las computadoras al evaluar sus funciones o en la aplicación de dichas funciones con valores conocidos. Por esta razón, los lenguajes funcionales se conocen en ocasiones como lenguajes aplicativos.

Los algoritmos son una serie de pasos para obtener un resultado determinado y se implementan como procedimientos o funciones dentro de los programas.

Con la programación funcional y estructurada, los programas se organizan en un conjunto de funciones jerarquizadas como un árbol y, generalmente, al programa principal, se le conoce como nodo raíz (ver la figura 1.1). Estos programas dependen de las funciones o procedimientos que implementan los algoritmos en los que se basa el programa, frecuentemente, las partes más cercanas al nodo raíz contienen descripciones abstractas de los algoritmos empleados, y a medida que nos alejamos del nodo raíz, se van desglosando estas descripciones en otras cada vez menos abstractas, hasta llegar al nivel máximo de detalle en las capas inferiores.

La representación de datos está determinada por los algoritmos que se vayan a emplear y reflejan la estructura interna del algoritmo, más que la semántica del mismo. Además, los datos son entidades pasivas, meras zonas de memoria y aun cuando podamos definir

---

<sup>1</sup> Kenneth, C. Loudon, *Lenguaje de Programación– Principios y práctica*, Editorial Thomson, 2005, p. 3.

una cierta estructura jerárquica de los mismos, ésta puede, en muchos casos, reflejar más los detalles de implementación del algoritmo que el significado del tipo de datos así definido. Este modelo conceptual, reflejado en la siguiente figura, es el que motivó el nombre del libro clásico de Niklaus Wirth: “Algoritmos + Estructuras de datos = Programas”.

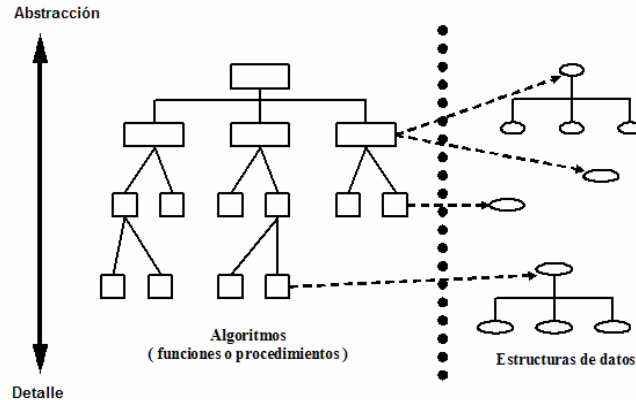


Figura 1.1. Árbol de funciones o procedimientos en la programación funcional y sus estructuras de datos.

Las metodologías destinadas a escribir programas basadas en el modelo conceptual anteriormente explicado reciben, en su conjunto, el nombre de programación estructurada.

La programación estructurada produjo en su momento una mejora considerable en la calidad de código con respecto a enfoques anteriores, pero no ha logrado resolver del todo problemas como la poca mantenibilidad del código y, sobre todo, la dificultad de construir componentes de *software* reutilizables de una aplicación a otra; es decir, módulos autocontenidos con entradas y salidas bien definidas y que sirvan como bloques prefabricados en la construcción de una amplia gama de sistemas con propósitos diversos.

La sintaxis de los lenguajes funcionales refleja fielmente las ideas de la programación estructurada. La unidad básica de un programa es el procedimiento o función, y los datos son bloques de memoria que esperan pasivamente a ser modificados por los algoritmos. No es posible, sólo con las reglas de ámbitos de acceso contempladas en estos lenguajes, conseguir mecanismos efectivos para ligar de una forma lógica y coherente los datos y el código necesario para manejarlos.

### 1.2.2 Programación orientada a objetos

Si nos ponemos a pensar en cuál es el modo natural como la mente organiza la información que nos llega a través de los sentidos, concluiremos fácilmente que no se apegan al modelo propuesto por el paradigma funcional y estructurado.

Las personas no ven el mundo dividido en dos grandes mitades, una compuesta por todos los datos del estado del mundo y otra por todas las posibles acciones que se ejercen sobre ellos. Al contrario, percibimos el mundo como dividido en cosas u objetos, cada uno de los cuales tiene asociado un estado y un conjunto de operaciones que producen cambios en éstos.

Así por ejemplo, cuando conducimos un coche, vemos como una unidad su estado y las acciones que son posibles realizar para modificar ese estado. El estado está compuesto por la velocidad, la aceleración, la velocidad de giro del motor (RPM) y otras medidas similares. Las acciones que se pueden realizar para modificar ese estado son pisar el acelerador, frenar, cambiar de marcha, etc. El único modo de cambiar el estado del coche es ejercer sobre él una o varias de las posibles acciones. Por otro lado, el estado actual del coche puede influir en el resultado de las acciones: obviamente, no se produce el mismo efecto al pisar el freno cuando la velocidad es baja que cuando ésta es alta, o cuando el coche circula en una recta o una curva. De este modo, estado y acciones están indisolublemente ligados en cualquier objeto de la vida real.

La programación orientada a objetos trata de reflejar estas características de los objetos reales, haciendo que un programa se divida en entidades que mantienen juntos los datos y el código que los manipula. Dichas entidades reciben, como sus contrapartidas del mundo real, el nombre de *objetos*. Un objeto consta de unas variables que contienen su estado actual, y de unos *métodos* o funciones ligadas al objeto para actuar sobre el estado. El único modo de acceder a las variables del objeto es a través de los métodos asociados a esas variables privadas del objeto. A este modo de disponer código y datos se le llama *encapsulación*. La encapsulación permite modificar la manera en que un objeto implementa sus métodos sin afectar a ningún otro objeto del sistema —modularidad—. Los servicios que proporciona el objeto pueden ser utilizados sin saber cuáles son los detalles de su implementación —ocultamiento de la información—.

La definición de los datos a través de las operaciones que es posible realizar con ellos, y no por su representación en memoria, recibe el nombre de *abstracción de datos*.

Los objetos se comunican entre sí enviando y recibiendo *mensajes*, que piden al objeto la realización de una acción (*ver la figura 1.2*). Un mensaje no es más que la invocación a un método del objeto con los parámetros contenidos en el mensaje.

En la siguiente figura se representa el modelo simplificado de la interacción entre coche y conductor que se emplearía para preparar un programa de simulación en un lenguaje orientado a objetos.

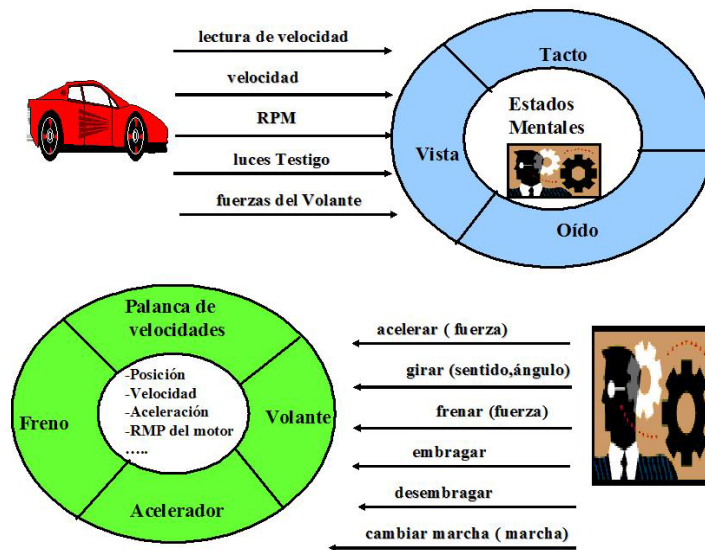


Figura 1.2. Envío de mensajes entre objetos.

Tanto los sentidos relevantes del conductor —vista, oído y tacto— como los mandos del coche —volante, acelerador, freno, etc.— se corresponden en el modelo con otros tantos métodos, que a su vez reciben como entrada los datos enviados por medio de mensajes desde el otro objeto. Los métodos deciden cuál va a ser el nuevo estado a partir de los datos de entrada y del estado actual.

Por ejemplo, si el objeto que representa al conductor decide que hay que aumentar la velocidad, envía un mensaje acelerar, el cual hace que se invoque el método acelerador ( ) con los parámetros proporcionados en el mensaje. Si el estado actual lo permite, es decir, el motor está en funcionamiento, la caja de cambios no está en punto muerto y se cumplen otras condiciones parecidas, aumentará la velocidad de giro del motor y, en consecuencia, la velocidad del coche.

Este cambio en la velocidad del coche provoca a su vez movimientos en las agujas del velocímetro, del tacómetro y del contador de revoluciones, las cuales son vistas por el conductor, y que en el modelo se representan mediante mensajes. Estos mensajes llegan al objeto conductor y producen llamadas al método que representa la vista. A través de él, cambian las variables internas del estado mental del conductor.

Siguiendo este ejemplo automovilístico, nosotros no nos conformamos con ver coches u objetos individuales. También observamos similitudes entre ellos, y podemos abstraer características comunes que nos sirven para clasificarlos. Así, existe un diseño llamado Tsuru GS1, del cual son casos particulares los coches de este modelo que vemos por las carreteras. A partir de ese diseño o plantilla se producen objetos individuales del mismo tipo, a esto se le llama *clase*, en programación orientada a objetos. La creación de un objeto a partir de una clase se le denomina *instanciación*, y es posible referirse al objeto como una *instancia* de esa clase (ver la figura 1.3):

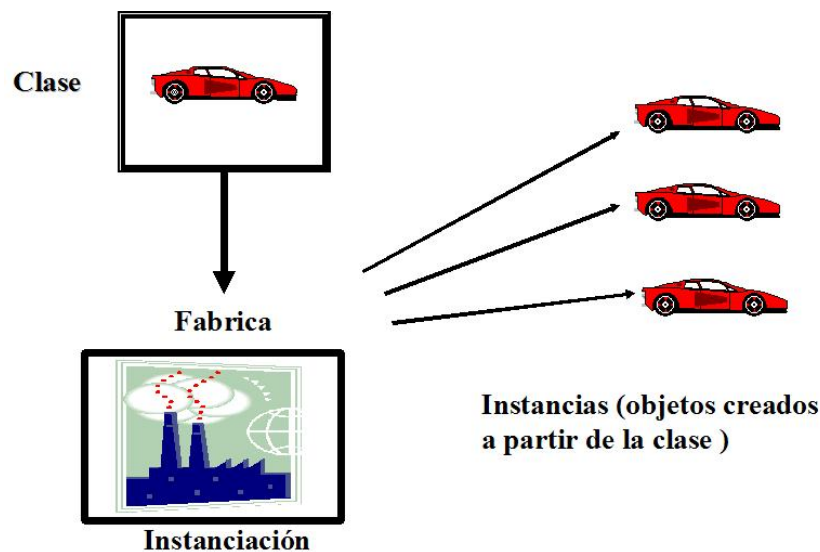


Figura 1.3. Instancias de objetos creados a partir de la clase.

Además de objetos y clases, la programación orientada a objetos posee mecanismos de abstracción y reutilización de código como la *herencia*. La herencia es una relación entre clases donde una clase nueva puede heredar los atributos y métodos de una clase existente. La clase existente se le conoce como superclase, y la clase nueva es una subclase. Gracias a la herencia, los lenguajes de programación orientada a objetos poseen una propiedad conocida como *polimorfismo*. El polimorfismo nos permite reutilizar el código de los métodos de las superclases para darles una nueva forma de comportamiento a las mismas, así el programador aprovecha lo que ya existe en la superclase y puede programar la nueva forma en que hará las cosas el nuevo método en la subclase. Más adelante en este manual, hablaremos ampliamente de este tema en el capítulo que trata sobre la herencia.

### **1.2.3 Programación lógica**

Este paradigma de lenguajes de programación se basa en la lógica simbólica. En un lenguaje lógico, un programa está formado por un conjunto de enunciados o proposiciones lógicas que describen lo que es verdad con respecto a un resultado deseado, en oposición a una secuencia particular de enunciados que deben ser ejecutados en un orden establecido para producir un resultado. Un lenguaje de programación lógico puro no tiene necesidad de sentencias de control de flujo como ciclos o selección. El control es suministrado por el sistema subyacente. Todo lo que se necesita en un programa lógico es el enunciado de propiedades del cómputo. Por esta razón, a la programación lógica algunas veces se le conoce como programación declarativa (recordar que también por razones similares a la programación funcional se le conoce como declarativa), dado que las propiedades se declaran, pero no se especifica una secuencia de ejecución.

Como ejemplos de lenguajes de programación lógica tenemos el Prolog y el Lisp. Uno de los más utilizados en el campo de la IA (Inteligencia Artificial) es el Prolog, lenguaje diseñado para manejar problemas lógicos, es decir, problemas en los que se necesita

tomar decisiones de una forma ordenada. Prolog<sup>2</sup> intenta hacer que la computadora “razone” la forma de encontrar la solución. Es particularmente adecuado para diferentes tipos de la IA, los dos más significativos de éstos son los *sistemas expertos* y el *procesamiento del lenguaje natural*.

### **1.2.4 Programación orientada a datos**

La programación con lenguajes orientados a datos se refiere al manejo de instrucciones o sentencias de consulta basados en el modelo entidad – relación o modelo de datos relacional. El modelo relacional<sup>3</sup> se basa en un conjunto de tablas. El usuario del sistema de bases de datos puede consultar estas tablas, insertar nuevas tuplas o registros, borrar, actualizar y modificar las tuplas. Hay varios lenguajes para expresar estas operaciones. El álgebra relacional define las operaciones fundamentales utilizadas en los lenguajes de consultas relacionales.

El álgebra relacional y los cálculos relacionales son lenguajes rígidos, formales, que no resultan adecuados para los usuarios ocasionales de los sistemas de bases de datos. Los sistemas comerciales de bases de datos, por lo tanto, utilizan lenguajes con menor rigidez. Los lenguajes de bases de datos comerciales incluyen el SQL (*Structured Query Lengue*, Lenguaje Estructurado de Consultas). Aunque el lenguaje SQL se considera un lenguaje de consultas, éste contiene muchas otras capacidades además de las consultas a la base de datos, por ejemplo, incluye características para definir la estructura de los datos, así como la modificación de los datos de la base de datos y la especificación de ligaduras de seguridad para la integridad referencial.

## **1.3 Traductores**

Para que un lenguaje de programación sea útil debe tener un *traductor*, esto es, un programa que acepte otros programas escritos en el lenguaje en cuestión, y que los ejecute directamente o los transforme en una forma adecuada para su ejecución.

Un traductor que ejecuta un programa directamente se conoce como *intérprete*. Un traductor que produce un programa equivalente en una forma adecuada para su ejecución se conoce como *compilador*.

### **1.3.1 Intérpretes**

Los intérpretes realizan el proceso que consta de un paso, en donde tanto el programa como la entrada le son dados al intérprete, y se obtiene una salida (ver la figura 1.4).

---

<sup>2</sup> Phillip, R. Robinson, *Aplique Turbo Prolog*, Editorial Osborne McGraw-Hill, 1991, p. xxii, xxiii.

<sup>3</sup> Silberschatz, Abraham, *Fundamentos de Bases de Datos*, Editorial McGraw Hill, p. 76.

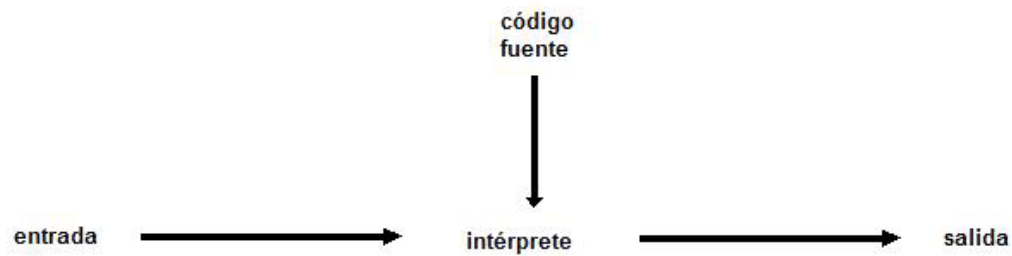


Figura 1.4. Proceso de traducción con un intérprete.

### 1.3.2 Compiladores

Un compilador es un programa que hace un proceso de traducción que consta de dos pasos:

1. Primero, el programa original (programa fuente) es la entrada al compilador, y la salida del compilador que es un nuevo programa (programa objetivo).
2. Segundo, dicho programa objetivo puede ser entonces ejecutado, siempre y cuando se encuentre en una forma adecuada para una ejecución directa, esto es, en lenguaje de máquina (ver la figura 1.5).

Usualmente el lenguaje objetivo es un lenguaje ensamblador, y el programa objetivo deberá ser traducido por un *ensamblador* en un programa objeto, posteriormente deberá ser *ligado* con otros programas objeto, y *cargado* en localidades de memoria apropiadas antes de que pueda ser ejecutado. A veces el lenguaje objetivo es incluso otro lenguaje de programación, y en tal caso, deberá utilizarse un compilador para que dicho lenguaje pueda obtener un programa objeto ejecutable.

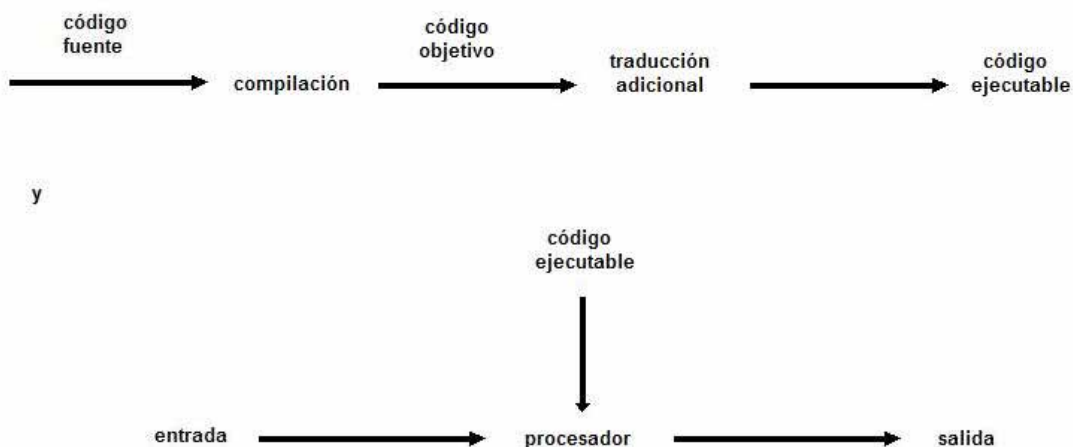


Figura 1.5. Proceso de traducción con un compilador.

- Ensambladores

Un *ensamblador* es un traductor para el *lenguaje ensamblador* de una computadora en particular. Como se mencionó, el lenguaje ensamblador es una forma simbólica del lenguaje de máquina de la computadora y está caracterizado por el uso de nemónicos que representan operaciones de máquina y quizás direcciones simbólicas de la memoria de la computadora. El lenguaje ensamblador es particularmente fácil de traducir debido a su bajo nivel de abstracción a código objeto de máquina de ceros y unos (código binario). En ocasiones el compilador generará lenguaje ensamblador como su lenguaje objetivo y dependerá entonces de un ensamblador para terminar la traducción a código objeto.

- **Ligadores**

Tanto los compiladores como los ensambladores a menudo dependen de un programa conocido como *ligador*, el cual recopila el código que se compila o ensambla por separado en diferentes archivos objeto, a un archivo que es directamente ejecutable. En este sentido, puede hacerse una distinción entre código objeto (código de máquina que todavía no se ha ligado) y el código de máquina ejecutable. Un ligador también conecta un programa objeto con el código de funciones de librerías estándar, así como con recursos suministrados por el sistema operativo de la computadora, por ejemplo asignadores de memoria y dispositivos de entrada y salida.

- **Cargadores**

Frecuentemente un compilador, ensamblador o ligador producirá un código que todavía no está completamente organizado y listo para ejecutarse, pero cuyas principales referencias de memoria se hacen relativas a una localidad de arranque indeterminada que puede estar en cualquier sitio de la memoria. Se dice que tal código es *relocalizable* y un cargador resolverá todas las direcciones relocalizables relativas a una dirección base, o de inicio, dada. El uso de un cargador hace más flexible el código ejecutable, pero el proceso de carga con frecuencia ocurre en segundo plano (como parte del entorno operacional) o conjuntamente con el ligado. Rara vez un cargador es en realidad un programa por separado.

## **1.4 Código fuente, objeto y ejecutable**

El *código fuente* es la entrada en el proceso de traducción (ya sea compilación o interpretación) y consiste de conjuntos de instrucciones o sentencias escritas en un lenguaje de programación. Estas sentencias se guardan en un archivo de texto ANSI o ASCII al cual se le denomina *archivo fuente*. Para crear los archivos fuentes y capturar estas sentencias, se utiliza generalmente un *editor* o programa que nos permite editar el código fuente desde el teclado, esto es, escribirlo, borrarlo, copiarlo, moverlo, insertarlo, guardarlo, etcétera.

- **Editores**

Los compiladores e intérpretes por lo regular aceptan archivos fuente escritos utilizando cualquier editor que pueda producir un archivo estándar, tal como un archivo ASCII. Desde hace ya varios años, los compiladores han sido integrados junto con editores y otros programas en un ambiente de desarrollo integrado o IDE. En un caso así, un editor, mientras produce archivos estándar, puede ser orientado hacia el formato o estructura del lenguaje de programación en cuestión. Tales editores se denominan *basados en estructura* e incluyen algunas de las operaciones del compilador, de manera que, por

ejemplo, pueda informarse al programador de los errores a medida que el programa se vaya escribiendo en lugar de hacerlo cuando está compilado. El compilador y sus programas acompañantes también pueden llamarse desde el editor, de modo que al programador le sea posible ejecutar el programa sin tener que abandonar el editor.

- **Código objeto**

El *código objeto* es la salida en el proceso de compilación, donde la entrada es el código fuente que es traducido por el compilador para convertirlo en código objeto, es decir, instrucciones en código de máquina. Un archivo que contiene código objeto se le denomina *archivo objeto*.

- **Código de máquina**

El código de máquina —código de máquina objetivo— es generalmente código en forma de *lenguaje ensamblador*, pero en su estado de menor abstracción es código binario de ceros y unos.

- **Lenguaje ensamblador**

Un *lenguaje ensamblador* es una forma más comprensible del código de máquina. Es el primer paso hacia una representación nemónica del programa.

### **1.4.1 Entornos de desarrollo integrado**

Un *entorno de desarrollo integrado* (IDE, *Integrated Development Environment*) es una aplicación de software que nos permite integrar en un solo ambiente mediante una interface gráfica de usuario (GUI, *Graphics User Interface*), un conjunto de herramientas de desarrollo de software, para que el programador o el equipo de programadores trabajen de manera interactiva y colaborativamente para así aumentar su productividad.

Las herramientas que suelen incluir los IDE generalmente son: un editor, un compilador, un intérprete, un depurador y un administrador de proyecto. Como ya habíamos visto, el editor es un programa que permite editar código fuente del programa en un lenguaje de programación. El compilador y el intérprete son traductores de un lenguaje de programación que permite trasladarlo a una forma adecuada para su ejecución en la máquina. El depurador detecta los errores de los programas y los corrige.

### **1.4.2 Depurador**

Un *depurador* es un programa que se utiliza para determinar los errores de ejecución de un programa compilado. A menudo está integrado con un compilador en un IDE. La ejecución de un programa con un depurador se diferencia de la ejecución directa en que el depurador se mantiene al tanto de la mayoría o la totalidad de la información sobre el código fuente, tal como los números de línea y los nombres de las variables y procedimientos. También en el proceso de depuración se pueden realizar *trazas* que es la interpretación de las sentencias una por una, como si estuviéramos revisando una cinta en cámara lenta. Asimismo, podemos colocar *puntos de ruptura* o puntos en el código fuente que marcamos, y en donde queremos que la ejecución del programa se pare, para así analizar el posible origen del error, además de proporcionar información de las funciones que se han invocado y sobre los valores actuales de las variables.

Tanto las trazas o *trace* como los puntos de ruptura o *break points*, se pueden utilizar en casi todos los compiladores modernos como son los de Delphi, Java, C/C++, Visual Basic, C# etc., debido a que los recientes IDE para estos lenguajes de programación vienen provistos de estas herramientas de depuración.

- **Administradores de proyecto**

Los modernos proyectos de *software* por lo general son tan grandes que tienen que ser emprendidos por grupos de programadores en lugar de un solo programador. En tales casos es importante que los archivos trabajados por distintas personas, se encuentren coordinados, y éste es el trabajo de un *programa de administración de proyectos*. Por ejemplo, un administrador de proyecto debería coordinar la mezcla de diferentes versiones del mismo archivo producido por programadores diferentes. También debería mantener una historia de las modificaciones para cada uno de los grupos de archivos, de modo que puedan mantenerse versiones coherentes de un programa en desarrollo (esto es algo que igual puede ser útil en un proyecto que lleva a cabo un solo programador). Un programa de administrador de proyecto es posible escribirlo de una forma independiente al lenguaje, pero cuando se integra junto con un compilador, puede mantener información acerca del compilador específico y las operaciones de ligado necesarias para construir un programa ejecutable completo.

- **Ayudas**

También en los IDE generalmente tenemos integrados programas asistentes y de ayuda para el programador, así como tutoriales. Estos programas dan información al usuario programador para familiarizarse con el ambiente y realizar tareas complejas. Actualmente, los modernos IDE tienen lo que se conoce como ayuda en línea, hipertexto, tecnología de punto, plantillas y demos.

- **Diseños**

Los IDE modernos integran herramientas de diseño de software como el lenguaje unificado de modelado (UML–*Unified Modeling Language*), y también para diseño de interfaces gráficas de usuario (GUI *Graphics User Interface*), entre otras más. Estas herramientas del IDE nos permiten desarrollar sistemas no sólo en la fase de programación o codificación de los programas, sino también desde la fase de análisis y diseño. Con las herramientas de diseño para GUI es posible desarrollar rápidamente prototipos de vistas con interfaces gráficas para la aprobación del cliente; además de facilitarnos, la creación de interfaces gráficas para nuestros programas, ya que su programación suele ser compleja y laboriosa.

- **Otros recursos**

Adicionalmente a lo anterior, los actuales IDE, además de permitirnos administrar proyectos a gran escala donde intervienen equipos de programadores, nos permiten integrar recursos tales como: orígenes de datos hacia bases de datos de diferentes proveedores, orígenes de datos a archivos planos, archivos de imágenes, de video, conexiones de red a servidores de aplicaciones, navegadores, servidores web, etc., ya que todos estos recursos podrían formar parte del desarrollo de la aplicación de software en un momento dado.

# CAPÍTULO 2

## Elementos básicos de la programación estructurada

En esta parte veremos algunos de los elementos básicos que todo lenguaje de programación tiene, los cuales son usados para describir los componentes que integran los programas, estos componentes son: comentarios, palabras reservadas, variables, tipos de datos, expresiones y operadores.

### 2.1 Tipos de datos primitivos

Antes de decir lo que son los datos primitivos, es necesario entender lo qué es un dato.

Dato: es un hecho que se representa en la computadora. Ejemplo: el color rojo es un hecho y dependiendo del contexto puede representar algo (ver la figura 2.1).

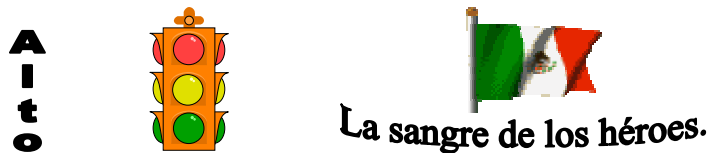


Figura 2.1. Concepto de dato.

En el contexto de las reglas de tránsito el color rojo representa alto, y en el contexto de los símbolos patrios este color está asociado con la sangre de los héroes que nos dieron patria.

La computadora sólo puede representar números en su memoria dentro de sus circuitos, y para el dato color rojo puede asignarle un número entero por ejemplo el número 4 (ver la figura 2.2).

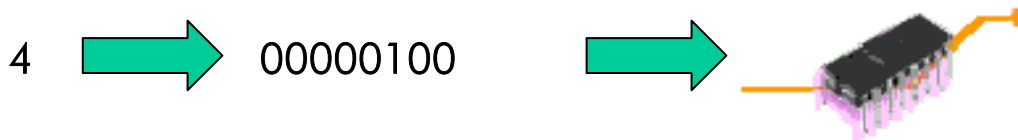


Figura 2.2. Representación de los datos como números en los circuitos de la computadora.

Existen datos simples llamados **primitivos**, y los tipos **compuestos** que se componen a su vez de tipos primitivos.

Un tipo primitivo es una abstracción del mundo real (como por ejemplo un número entero, carácter, un valor lógico, etc.), los cuales pueden ser representados internamente por la computadora.

### 2.1.1 Numéricos (enteros y de punto flotante o reales)

Tipo entero es un subconjunto finito de los números enteros, cuyo tamaño depende del lenguaje de programación y de la computadora utilizada, pueden ser negativos o positivos incluyendo el cero (ver la figura 2.3).

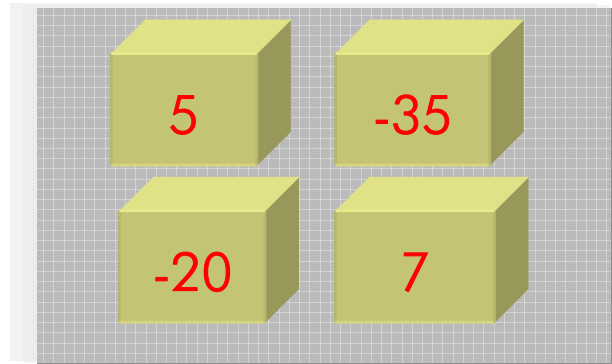


Figura 2.3. Representación de los números de tipo entero.

Tipo real es un subconjunto finito de los números reales, un número real consta de un entero y una parte decimal y pueden ser positivos o negativos incluyendo el cero (ver la figura 2.4).

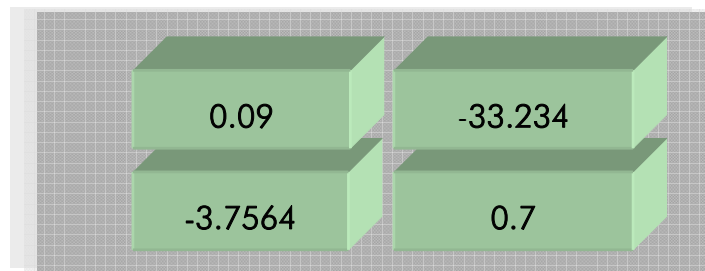


Figura 2.4. Representación de números de tipo real.

### 2.1.2 Carácter y cadena

Tipo carácter es la unidad básica e indivisible de una cadena, también suele ser llamado símbolo, como puede ser: a,@,#,1,etc. Cada carácter es representado por un conjunto de *bits*. Los caracteres suelen conformar conjuntos tales como el código ASCII o el UNICODE (ver la figura 2.5).

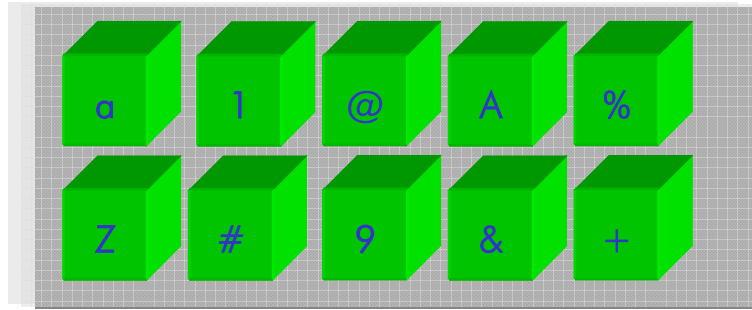


Figura 2.5. Representación de los caracteres.

Tipo cadena o *string*, es un tipo de dato compuesto debido a que consiste de una serie finita de caracteres que se encuentran delimitados por espacios en blanco y por una comilla (') o doble comillas (") dependiendo del lenguaje de programación (ver la figura 2.6).

Ejemplos:

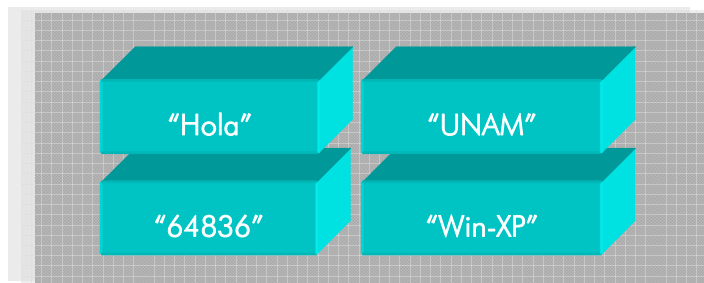


Figura 2.6. Representación de cadenas o *strings*.

### 2.1.3 Lógicos o Booleanos (*falso, verdadero*)

Tipo lógico o booleano, los booleanos o lógicos están formados por dos valores de verdad que son falso (*false*) y verdadero (*true*), (ver la figura 2.7).

Ejemplos:

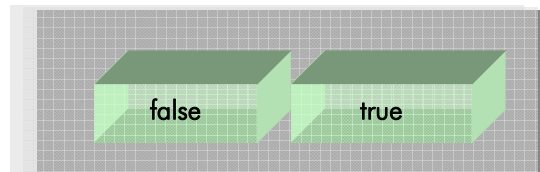


Figura 2.7. Representación de valores de tipo lógico o booleano.

## 2.2 Palabras reservadas

Todo lenguaje de programación tiene su conjunto de palabras reservadas. Las palabras reservadas no pueden ser empleadas como identificadores de funciones, clases, variables, constantes, etc., ya que como su nombre lo dice, son palabras reservadas para el lenguaje de programación y tienen un significado especial para dicho lenguaje.

Ejemplo: la tabla 2.1 es un resumen de las palabras reservadas del lenguaje de programación Java.

**Tabla 2.1. Conjunto de palabras reservadas del lenguaje Java.**

|          |            |           |              |
|----------|------------|-----------|--------------|
| abstract | double     | int       | static       |
| boolean  | else       | interface | super        |
| break    | extends    | long      | switch       |
| byte     | final      | native    | synchronized |
| case     | finally    | new       | this         |
| catch    | float      | null      | throw        |
| char     | for        | package   | throws       |
| class    | goto       | private   | transient    |
| const    | if         | protected | try          |
| continue | implements | public    | void         |
| default  | import     | return    | volatile     |
| do       | instanceof | short     | while        |

Ejemplo: las palabras reservadas de C, que tienen un significado para el compilador y no pueden usarse como identificadores, son las siguientes y deben escribirse como se muestran, es decir, con minúsculas:

- |            |            |            |
|------------|------------|------------|
| - auto     | - extern   | - signed   |
| - break    | - float    | - sizeof   |
| - case     | - for      | - static   |
| - char     | - goto     | - struct   |
| - const    | - if       | - switch   |
| - continue | - int      | - typedef  |
| - default  | - long     | - union    |
| - do       | - main     | - unsigned |
| - double   | - register | - void     |
| - else     | - return   | - volatile |
| - enum     | - short    | - while    |

Como se observa, las palabras reservadas de C y Java son muy parecidas debido a que Java es un lenguaje de programación orientado a objetos de aparición posterior y que aprovecha este aporte del lenguaje C incorporándolo a su sintaxis para facilitar su aprendizaje, ya que por razones históricas, la mayoría de los programadores que hay en el mundo conoce el lenguaje C.

## 2.3 Variables

Una variable es un lugar en la memoria de la computadora que guarda un valor, el cual cambia durante la ejecución del programa. Una variable posee un nombre y un lugar de almacenamiento o alojamiento en la memoria de la computadora y puede contener valores de cierto tipo de dato.

Ejemplo: en el lenguaje de programación Visual Basic para Aplicaciones (VBA), podemos definir una variable de tipo real así:

```
Dim Precio As Single
Y le podemos asignar un valor inicial así:
Precio = 16.50
```

## 2.4 Constantes

Una constante es un objeto cuyo valor no cambia durante la ejecución de un programa. Al igual que la variable, una constante tiene asignado un nombre y un lugar de alojamiento en la memoria de la computadora, además de tener un tipo de dato.

Ejemplo: en el lenguaje de programación Java podemos declarar una constante de tipo String así:

```
static final String UNAM = "Universidad Nacional Autónoma de México";
```

**Tipo de dato:** las variables y las constantes tienen o pertenecen a un tipo de dato, es decir, los valores de sus datos son simples o compuestos tales como: reales, enteros, carácter, booleanas, cadena, etcétera.

Ejemplo: en Visual Basic para aplicaciones tenemos estos tipos de datos:

Tabla 2.2. Tabla de los tipos de datos para Visual Basic.

Tipos de datos en Visual Basic para Excel

| Tipo de datos                               | Tamaño de almacenamiento | Intervalo                                                                                                                                           |
|---------------------------------------------|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Byte                                        | 1 byte                   | 0 a 255                                                                                                                                             |
| Boolean                                     | 2 bytes                  | True o False                                                                                                                                        |
| Integer                                     | 2 bytes                  | -32,768 a 32,767                                                                                                                                    |
| Long (entero largo)                         | 4 bytes                  | -2,147,483,648 a 2,147,483,647                                                                                                                      |
| Single (coma flotante/<br>precisión simple) | 4 bytes                  | -3,402823E38 a -1,401298E-45 para valores negativos;<br>1,401298E-45 a 3,402823E38 para valores positivos                                           |
| Double (coma flotante/<br>precisión doble)  | 8 bytes                  | -1.79769313486231E308 a<br>-4,94065645841247E-324 para valores negativos;<br>4,94065645841247E-324 a 1,79769313486232E308<br>para valores positivos |
| Currency (entero a escala)                  | 8 bytes                  | -922.337.203.685.477,5808 a<br>922.337.203.685.477,5807                                                                                             |

| Tipo de datos                             | Tamaño de almacenamiento           | Intervalo                                                                                                                                                                                                                              |
|-------------------------------------------|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Decimal                                   | 14 bytes                           | +/-79.228.162.514.264.337.593.543.950.335 sin punto decimal;<br>+/-7,9228162514264337593543950335 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es<br>+/-0,00000000000000000000000000000001 |
| Date                                      | 8 bytes                            | 1 de enero de 100 a 31 de diciembre de 9999                                                                                                                                                                                            |
| Object                                    | 4 bytes                            | Cualquier referencia a tipo Object                                                                                                                                                                                                     |
| String (longitud variable)                | 10 bytes + longitud de la cadena   | Desde 0 a 2.000 millones                                                                                                                                                                                                               |
| String (longitud fija)                    | Longitud de la cadena              | Desde 1 a 65.400 aproximadamente                                                                                                                                                                                                       |
| Variant (con números)                     | 16 bytes                           | Cualquier valor numérico hasta el intervalo de un tipo Double                                                                                                                                                                          |
| Variant (con caracteres)                  | 22 bytes + longitud de la cadena   | El mismo intervalo que para un tipo String de longitud variable                                                                                                                                                                        |
| Definido por el usuario (utilizando Type) | Número requerido por los elementos | El intervalo de cada elemento es el mismo que el intervalo de su tipo de datos.                                                                                                                                                        |

## 2.5 Expresiones

Las expresiones son fórmulas construidas con combinaciones de constantes, variables, operadores, operandos y nombres de funciones especiales.

Cada expresión al evaluarse, toma un valor que se determina tomando en cuenta los tipos de las variables y constantes implicadas, además de la ejecución de las operaciones indicadas por los operadores y los valores que devuelven las funciones.

## 2.6 Operadores y operandos

Los operadores son los símbolos matemáticos, lógicos, relacionales, etc, que efectúan una operación sobre los factores u operandos de una expresión, según el número de operandos que necesiten para realizar su cálculo, éstos los clasificamos en unarios, binarios y especiales.

### 2.6.1 Operadores unarios

Los operadores unarios operan sobre un solo factor o un solo operando.

operador (operando)

Ejemplo: si tenemos una variable entera **x** con valor asignado de **-10**

$$X = -10$$

y luego aplicamos el operador de inverso aditivo  $-x$ , entonces el resultado sería **10**, así:

$$X = 10$$

Porque la regla de los signos del álgebra nos dice que menos por menos da más.

### **2.6.2 Operadores binarios**

Como su nombre lo indica, los operadores binarios operan sobre dos factores o dos operandos.

(operando) operador (operando)

Ejemplo: tenemos una variable real **z** con valor asignado de **-8.5**.

$$Z = -8.5$$

Ahora restaremos el valor de **5**; **z - 5** así el valor de la variable **z**, que es **-8.5** menos el valor **5**, nos dará como resultado **-13.5** debido a que el operador *resta* es un operador binario, porque opera sobre el valor de la variable **z** (primer operando) y el valor de la constante **5** (segundo operando).

### **2.6.3 Operadores especiales**

Los operadores especiales son aquellos que operan sobre dos o más operadores u operandos, además, en algunos lenguajes de programación existen estos operadores especiales que efectúan un cálculo especial requerido.

Ejemplo: los lenguajes de programación C, C++, Java tienen un operador ternario **?:** que es una forma abreviada del **if - then**, el cual más adelante veremos en este manual en estructuras de control condicional; así entonces la expresión:

$$(x > z) ? x + 6 : 8.3;$$

Indica que, si es verdadero que **x** sea mayor que **z**, entonces el resultado de la expresión será la suma **x + 6** de lo contrario, si es falso, entonces el resultado será **8.3**.

## **2.7 Jerarquía de operadores**

El orden general de evaluación de los operadores de una expresión depende de su precedencia establecida en una jerarquía y de las reglas de la evaluación de expresiones.

### **2.7.1 Precedencia de operadores**

Cuando hay varias operaciones en una misma expresión, cada parte de la misma se evalúa y se resuelve en un orden predeterminado según la prioridad de los operadores. La siguiente tabla 2.3 muestra la prioridad o precedencia de operadores y está presente en la mayoría de los lenguajes de programación.

Tabla 2.3. Tabla de la prioridad o precedencia de los operadores.

| Prioridad o precedencia                                                                                 | Operador                        | Notas                                                                                       | Tipo de Operandos | Tipo de resultado | Ejemplo                                    |
|---------------------------------------------------------------------------------------------------------|---------------------------------|---------------------------------------------------------------------------------------------|-------------------|-------------------|--------------------------------------------|
| (Mayor)<br><br>(Menor) | ( )                             | Paréntesis                                                                                  | Todos los tipos   | Todos los tipos   | $-(A2+(A1/(5+3))^2)$                       |
|                                                                                                         | -                               | Negación (como en -1)                                                                       | Entero o Real     | Entero o Real     | -30                                        |
|                                                                                                         | <sup>^</sup> , **, <sup>↑</sup> | Exponente                                                                                   | Entero o Real     | Entero o Real     | 3 <sup>2</sup><br>3**2<br>3 <sup>↑</sup> 2 |
|                                                                                                         | *                               | Multiplicación<br>División<br>División entera<br>Módulo o residuo                           | Entero o Real     | Entero o Real     | 3*3                                        |
|                                                                                                         | /                               |                                                                                             | Entero o Real     | Entero o Real     | 3/3                                        |
|                                                                                                         | div                             |                                                                                             | Entero o Real     | Entero            | 3 div 3                                    |
|                                                                                                         | mod                             |                                                                                             | Entero o Real     | Entero            | 3 mod 3                                    |
|                                                                                                         | +, -                            | Suma y resta                                                                                | Entero o Real     | Entero o Real     | 5+4<br>5-4                                 |
|                                                                                                         | &, +                            | Conecta dos cadenas de texto (concatenación)                                                | Cadena            | Cadena            | "UNAM" & "-DGSCA"<br>"UNAM" + "-DGSCA"     |
|                                                                                                         | =<br><<br>><br><=<br>>=<br><>   | Igual a<br>Mayor que<br>Menor que<br>Menor o igual que<br>Mayor o igual que<br>Diferente de | Booleano          | Booleano          | A=B<br>A<B<br>A>B<br>A<=B<br>A>=B<br>A<>B  |
|                                                                                                         | not                             | Negación lógica                                                                             | Booleano          | Booleano          | not (P)                                    |
|                                                                                                         | and                             | Conjunción lógica                                                                           | Booleano          | Booleano          | P and Q                                    |
|                                                                                                         | or                              | Disyunción lógica                                                                           | Booleano          | Booleano          | P or Q                                     |

### 2.7.2 Reglas de evaluación de expresiones

Para evaluar las expresiones se deben seguir las siguientes tres reglas de evaluación:

- **Regla 1.** En todas las expresiones se evalúan primero las expresiones de los paréntesis más anidados (interiores unos a otros); y éstos modifican la prioridad. Es importante señalar que los paréntesis deben estar balanceados (el número de paréntesis que abren debe ser igual al número de paréntesis que cierran).
- **Regla 2.** Todas las expresiones se evalúan tomando en cuenta la prioridad o precedencia de los operadores.
- **Regla 3.** Todas las expresiones se evalúan de izquierda a derecha.

Las reglas están ordenadas de acuerdo a su importancia.

## 2.8 Operadores aritméticos

Dependiendo del resultado de la expresión al evaluarse, éstas se reducen usualmente a tres tipos: expresiones aritméticas, expresiones lógicas y expresiones de cadena.

En las expresiones numéricas el resultado es un valor de tipo real o entero y los operadores son generalmente aritméticos, éstos son: la suma, resta, multiplicación, división, módulo y potencia.

### 2.8.1 Suma

La suma ( + ) es un operador binario, y nos permite sumar dos operandos numéricos ya sean reales o enteros.

Ejemplo:  $5 + 10.3$   
da el resultado de  $15.3$ .

### **2.8.2 Resta**

La resta (  $-$  ) o sustracción es un operador binario que nos permite restar dos operandos numéricos, ya sean reales o enteros.

Ejemplo:  $7 - 9.8$   
el resultado es  $-2.8$ .

### **2.8.3 Multiplicación**

La multiplicación (  $*$  ) es un operador binario que efectúa una multiplicación entre dos operandos numéricos, ya sean reales o enteros.

Ejemplo:  $-3.5 * 4$   
el resultado es  $-14.0$ .

### **2.8.4 División**

La división (  $/$  ) es un operador binario que efectúa una división entre dos operandos numéricos, ya sean reales o enteros.

Ejemplo:  $-7.5 / 2$   
el resultado es  $-3.75$ .

### **2.8.5 Módulo**

El modulo (  $\%$  ) es un operador binario que efectúa una división entre dos operandos numéricos, ya sean reales o enteros dándonos como resultado el residuo.

Ejemplo:  $-7 \% 3$   
el resultado es  $1$ .

### **2.8.6 Potencia**

La potencia (  $^$  ) es un operador binario que efectúa una exponenciación entre dos operandos numéricos, uno es la base y otro el exponente, ya sean reales o enteros, que dan como resultado la potencia.

Ejemplo:  $25.0 ^ 3$   
el resultado es  $15625$ .

## **2.9 Operadores relacionales**

Las expresiones lógicas o booleanas son aquellas donde el resultado es un valor de tipo lógico, falso o verdadero, 0 o 1, y los operadores son relacionales y operadores lógicos. Los símbolos utilizados para denotar los operadores varían según el lenguaje de

programación por utilizar. Los operandos son generalmente numéricos, ya sean reales o enteros, aunque puede en ocasiones compararse caracteres.

### **2.9.1 Mayor que**

El operador mayor que (  $>$  ) es un operador binario que efectúa una comparación entre dos operandos. Si el valor de la izquierda es mayor que el de la derecha entonces da verdadero, de lo contrario da falso.

Ejemplo. La siguiente expresión:  $-7.5 > 2$

da el resultado de **falso** o 0

### **2.9.2 Menor que**

El operador menor que (  $<$  ) es un operador binario que efectúa una comparación entre dos operandos. Si el valor de la izquierda es menor que el de la derecha entonces da verdadero, de lo contrario da falso.

Ejemplo. La siguiente expresión:  $-7.5 < 2$

da el resultado de **verdadero** o 1

### **2.9.3 Mayor o igual que**

El operador mayor o igual que (  $>=$  ) es un operador binario que efectúa una comparación entre dos operandos. Si el valor de la izquierda es mayor o igual que el de la derecha entonces da verdadero, de lo contrario da falso.

Ejemplo. La siguiente expresión:  $-7.5 >= 2$

da el resultado de **falso** o 0

### **2.9.4 Menor o igual que**

El operador menor o igual que (  $<=$  ) es un operador binario que efectúa una comparación entre dos operandos. Si el valor de la izquierda es menor o igual que el de la derecha entonces da verdadero, de lo contrario da falso.

Ejemplo. La siguiente expresión:  $-7.5 <= -7.5$

da el resultado de **verdadero** o 1

### **2.9.5 Igual a**

El operador igual que (  $=$  o  $==$  ) es un operador binario que efectúa una comparación entre dos operandos. Si el valor de la izquierda es igual que el de la derecha entonces da verdadero, de lo contrario da falso.

Ejemplo. La siguiente expresión:  $-7.5 == -7.5$

da el resultado de **verdadero** o 1

### 2.9.6 Diferente de

El operador diferente de (  $<>$  ) es un operador binario que efectúa una comparación entre dos operandos. Si el valor de la izquierda es diferente que el de la derecha entonces da verdadero, de lo contrario da falso.

Ejemplo. La siguiente expresión:  $-7.5 <> -7.5$

nos da el resultado de **falso** o 0

## 2.10 Operadores lógicos o booleanos (and, or, not, xor )

En las expresiones lógicas o booleanas además de utilizar los operadores relacionales, también se utilizan los operadores lógicos. Los símbolos utilizados para denotar estos operadores varían según el lenguaje de programación por utilizar. Los operandos son generalmente lógicos o booleanos con sus dos posibles valores falso y verdadero, aunque en algunos lenguajes pueden ser números enteros, como sucede en el C / C++ donde el 0 representa falso y cualquier entero diferente de 0 representa verdadero.

### 2.10.1 Tablas de verdad

Las tablas de verdad indican el comportamiento de los operadores lógicos en la evaluación de expresiones condicionales. La tabla 2.4 muestra los símbolos utilizados para las operaciones lógicas, y la tabla 2.5 muestra los valores obtenidos cuando se evalúa una expresión lógica.

Tabla 2.4. Formas en que pueden representarse los operadores lógicos.

| Operación lógica | Símbolo matemático | Símbolo en inglés | Símbolo en español | Lenguaje Java | Visual Basic .Net |
|------------------|--------------------|-------------------|--------------------|---------------|-------------------|
| Negación         | $\neg$             | NOT               | NO                 | !             | Not               |
| Conjunción       | $\wedge$           | AND               | Y                  | &&            | And               |
| Disyunción       | $\vee$             | OR                | O                  |               | Or                |

Tabla 2.5. Tabla de la verdad o de las proposiciones lógicas y sus operadores.

| P         | Q         | No P      | P o Q     | P y Q     |
|-----------|-----------|-----------|-----------|-----------|
| Verdadero | Verdadero | Falso     | Verdadero | Verdadero |
| Verdadero | Falso     | Falso     | Verdadero | Falso     |
| Falso     | Verdadero | Verdadero | Verdadero | Falso     |
| Falso     | Falso     | Verdadero | Falso     | Falso     |

Los operadores lógicos actúan sobre proposiciones lógicas y son las siguientes:

La *negación* de  $P$ , o el operador  $\text{NO } ( P )$  ( $\text{NOT } ( P )$ ), niega el valor de una proposición. Así, si  $P$  es **verdadera**  $\text{NO } ( P )$  da como resultado **falso**.

La *conjunción* de  $P$  y  $Q$  ( $P \text{ and } Q$ ), sólo da **verdadero** cuando ambos  $P$ ,  $Q$  sean verdaderos, en caso de que una sola de las proposiciones  $P$ ,  $Q$  sean falsas, o ambas sean falsas la conjunción de  $P$  y  $Q$  será **falso**.

La *disyunción* de  $P$  o  $Q$  ( $P \text{ or } Q$ ), sólo da **verdadero** cuando alguna de las proposiciones  $P$ ,  $Q$  sean verdaderas o ambas sean verdaderas, la disyunción  $P$  o  $Q$  dará **falso** sólo cuando ambas proposiciones  $P$ ,  $Q$  sean falsas.

### 2.10.2 Operadores de cadena o alfanuméricas

El resultado es un valor de tipo carácter o una cadena de caracteres. No todos los lenguajes tienen operadores de cadena, pero el que más se utiliza es el de concatenación, que utiliza el  $( + )$  o el  $( \& )$  o el punto  $( . )$  para unir cadenas:

Ejemplo:

`"hola" + " " + "buenos días"`

`"hola" & " " & "buenos días"`

`"hola" . " " . "buenos días"`

Da como resultado **"hola buenos días"**.

## 2.11 Precedencia de operadores y evaluación de expresiones

Además de las reglas de precedencia o jerarquía de evaluación de expresiones, el resultado generalmente debe asignarse a una variable.

### 2.11.1 Asignación

Es una operación que tiene como efecto dar valor a una variable, resultado de la evaluación de una expresión.

Donde  $V$  denota una variable y  $E$  una expresión. Se lee:  $V$  se hace  $E$ , o hacer  $V$  igual a  $E$ , o a  $V$  se le asigna el resultado de  $E$ , (vea la figura 2.8).

$V \leftarrow E$

Figura 2.8. Representación de la operación de asignación.

El símbolo del operador de asignación, al igual que todos los otros operadores cambia de un lenguaje de programación a otro, pero generalmente se utilizan los siguientes:  $\leftarrow$ ,  $=$ ,  $:=$ . Una cosa que hay que tomar en cuenta es que el operador de igualdad relacional se confunde a veces con el de asignación, recuerde que una asignación no es una igualdad matemática.

Ejemplo:

| INSTRUCCIÓN                                       | MEMORIA            |
|---------------------------------------------------|--------------------|
| $I \leftarrow 0$                                  | <div>0</div> I     |
| $I \leftarrow I + 1$                              | <div>1</div> I     |
| $CAR \leftarrow 'a'$                              | <div>a</div> CAR   |
| $B \leftarrow (8 > 5) \text{ and } (15 < 2 ** 3)$ | <div>FALSO</div> B |
| $B \leftarrow B \text{ or } (I = 8)$              | <div>FALSO</div> B |

Figura 2.9. Evaluación de expresiones con asignación.

A reserva de ver más adelante en este capítulo otros ejemplos de evaluación de expresiones, vamos a dar algunas explicaciones para algunos de los casos mostrados en la figura 2.9. Para el caso de las expresiones:

$$I \leftarrow I + 1$$

Primero se efectúa la suma y después se realiza la asignación, entonces  $0 + 1$  nos da 1. Recuerde que anteriormente mencionamos que primero se evalúa la expresión a la derecha del operador de asignación y después el valor resultante se asigna a la localidad de memoria.

$$B \leftarrow (8 > 5) \text{ and } (15 < 2 ** 3)$$

Primeramente se evalúan las expresiones entre paréntesis porque esta es la regla de mayor prioridad, así  $(8 > 5)$  está a la derecha y por lo tanto se evalúa primero y nos da VERDADERO. Luego se evalúa la segunda expresión entre paréntesis  $(15 < 2 ** 3)$  y aquí se aplica la regla de prioridades de operadores y evaluamos 2 elevado a la 3 y nos da 8, entonces la expresión queda como  $(15 < 8)$ , lo cual da FALSO. Finalmente VERDADERO and FALSO nos da FALSO.

$$B \leftarrow B \text{ or } (I = 8)$$

En este caso, se evalúa primero la expresión entre paréntesis que es  $(I = 8)$ , lo cual nos da FALSO porque I vale uno y, uno no es igual a 8. Luego B nos había dado FALSO, entonces FALSO or FALSO nos da FALSO.

### 2.11.2 Funciones en las expresiones

En las expresiones pueden intervenir funciones que están predefinidas en los lenguajes de programación. Éstas aceptan argumentos y devuelven un valor de resultado.

Su formato general es:

**nombreFunción(arg1, arg2,...,argn)**

Algunos ejemplos de funciones que tienen la mayoría de los lenguajes de programación se muestran en la tabla 2.6.

Tabla 2.6. Algunas funciones matemáticas comunes.

| Nombre de la Función | Descripción              | Tipo de argumento | Resultado              |
|----------------------|--------------------------|-------------------|------------------------|
| Abs(x)               | Valor absoluto de x      | Entero o Real     | Igual que el argumento |
| Atan(x)              | Arco Tangente de x       | Entero o Real     | Real                   |
| Cos(x)               | Coseno de un ángulo x    | Entero o Real     | Real                   |
| Entero(x)            | Entero de x              | Real              | Entero                 |
| Exp(x)               | Logaritmo neperiano de x | Entero o Real     | Real                   |
| Log10(x)             | Logaritmo base 10 de x   | Entero o Real     | Real                   |
| Raiz(x)              | Raíz cuadrada de x       | Entero o real     | Real                   |
| Sen(x)               | Seno de un ángulo x      | Real o Entero     | Real                   |
| Truncar(x)           | Parte entera de x        | Real              | Entero                 |

Veamos algunos ejemplos de evaluación de expresiones:

Si  $A1 = 10$ ,  $A2 = -25$ , evaluar la siguiente expresión.

$$A3 = -A2 + \frac{A1}{5} + 3^2 + \sqrt{25}$$

Solución:

Si  $A1 = 10$  y  $A2 = -25$ . ¿Cómo se evalúa esta expresión?

$$\begin{aligned}
 A3 &= -A2 + A1/5 + 3^2 + \text{RAIZ}(25) \\
 A3 &= +25 + A1/5 + 3^2 + \text{RAIZ}(25) \\
 A3 &= +25 + A1/5 + 9 + \text{RAIZ}(25) \\
 A3 &= +25 + A1/5 + 9 + 5 \\
 A3 &= +25 + 2 + 9 + 5 \\
 A3 &= 41
 \end{aligned}$$

Figura 2.10. Evaluación de una expresión cuando no tiene paréntesis.

Aquí sólo aplicamos la regla 2 (prioridad de operadores) y la regla 3 (evaluación de izquierda a derecha). Note que el inverso aditivo del paso 1 tiene prioridad sobre la potencia en el paso 2, también recuerde que la raíz de un número es la potencia a la  $\frac{1}{2}$  y eso es lo que hace la función raíz (vea la figura 2.10).

## 2.12 Uso de paréntesis

Cuando en una expresión se agrupan los factores en paréntesis, se debe aplicar la regla de los paréntesis (regla 1, mencionada antes).

Ejemplo: Si  $A1 = 10$ ,  $A2 = -25$ , evaluar la siguiente expresión.

$$A4 = -\left( A2 + \left( \frac{A1}{5+3} \right)^2 + \sqrt{25} \right)$$

Solución:

$$\begin{aligned}
 A4 &= -(A2+(A1/(5+3))^2+RAIZ(25)) \\
 A4 &= -(A2+(A1/8)^2+RAIZ(25)) \\
 A4 &= -(A2+1.25^2+RAIZ(25)) \\
 A4 &= -(A2+1.5625+RAIZ(25)) \\
 A4 &= -(A2+1.5625+5) \\
 A4 &= -(-23.4375+5) \\
 A4 &= -(-18.4375) \\
 A4 &= +18.4375 \\
 \boxed{18.4375} & \text{ A4}
 \end{aligned}$$

Figura 2.11. Evaluación de una expresión con paréntesis de agrupación.

Aquí se aplica la regla 1 (la regla de los paréntesis), primero se evalúa la expresión con los paréntesis más anidados en el paso 1 y en el paso 2. En los pasos siguientes se aplica la regla 2 (prioridad o jerarquía de operadores), finalmente se asigna el resultado a la variable A4 (vea la figura 2.11).

Ejemplo: Si  $P = 2$ ,  $Q = 7$ , evaluar la siguiente expresión.

$$R = \neg(15 \geq Q^P) \vee (43 - 8P \text{ div } 4 \neq 3P \text{ div } P)$$

Solución:

$$\begin{aligned}
 R &= \text{NO}(15 \geq Q^P) \vee (43 - 8 * P \text{ div } 4 \neq 3 * P \text{ div } P) \\
 R &= \text{NO}(15 \geq 49) \vee (43 - 8 * P \text{ div } 4 \neq 3 * P \text{ div } P) \\
 R &= \text{NO(falso)} \vee (43 - 8 * P \text{ div } 4 \neq 3 * P \text{ div } P) \\
 R &= \text{NO(falso)} \vee (43 - 16 \text{ div } 4 \neq 3 * P \text{ div } P) \\
 R &= \text{NO(falso)} \vee (43 - 4 \neq 3 * P \text{ div } P) \\
 R &= \text{NO(falso)} \vee (43 - 4 \neq 6 \text{ div } P) \\
 R &= \text{NO(falso)} \vee (43 - 4 \neq 3) \\
 R &= \text{NO(falso)} \vee (39 \neq 3) \\
 R &= \text{NO(falso)} \vee \text{verdad} \\
 R &= \text{verdad} \vee \text{verdad} \\
 \boxed{\text{verdad}} & \text{ R}
 \end{aligned}$$

Figura 2.12. Evaluación de una expresión con valores lógicos.

En el ejemplo se aplica la regla 2 (prioridad de operadores) en el paso 1. Luego la regla 1 (la de los paréntesis) se aplica en el paso 2. Del paso 3 al paso 7 se aplica la regla 2 (prioridad de operadores), luego en el paso 8 se aplica la regla 1 (la de los paréntesis). En el paso 9 se aplica primero la negación porque ésta tiene prioridad sobre la operación lógica  $\vee$ . Finalmente se asigna el valor en el paso 11 (vea la figura 2.12).

# CAPÍTULO 3

## Control de flujo de sentencias

---

Antes de intentar elaborar algún programa, es importante conocer el orden en que se ejecutan las instrucciones o sentencias que se le dan, para lo cual tenemos que comprender los conceptos de algoritmo, diagrama de flujo, pseudocódigo, programa propio y el teorema de la programación estructurada.

### Algoritmo

- Un algoritmo es un conjunto de acciones que **determinan** la secuencia de los pasos por seguir para resolver un problema específico. Por ejemplo conjunto de acciones para preparar unos huevos fritos con tocino.
- Los pasos de un algoritmo deben estar definidos con **precisión**, no deben existir ambigüedades que den origen a elegir una decisión equivocada.
- Los algoritmos son **finitos**, esto es, sus pasos terminan en algún punto determinado.
- Un algoritmo es una forma de resolver un problema (ver la figura 3.1).

#### Etapas de la solución de un problema

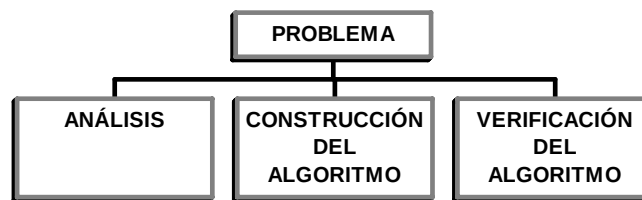


Figura 3.1. Etapas de la solución de un problema.

- Un algoritmo es un proceso con datos de entrada y de salida llamados resultados (ver la figura 3.2).

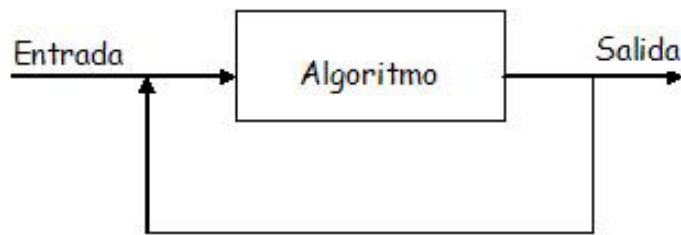


Figura 3.2 Diagrama del proceso de un algoritmo.

### Diagramas de flujo

- Es una representación gráfica de un algoritmo (ver la figura 3.3). Los símbolos más comúnmente utilizados en un diagrama de flujo son:

| Símbolo | Nombre                        | Función                                                                                                                                                                              |
|---------|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|         | Terminador                    | Se usa para marcar el <i>inicio</i> y <i>fin</i> del diagrama de flujo                                                                                                               |
|         | Datos                         | Se usa para introducir datos de entrada. Significa <i>Lectura</i>                                                                                                                    |
|         | Decisión                      | Se usa para representar una decisión. Este símbolo se emplea en la estructura selectiva <i>si entonces / sino</i> , y en las estructuras de ciclo <i>repetir</i> y <i>mientras</i> . |
|         | Selector múltiple             | Representa una decisión múltiple. Se utiliza para la estructura de decisión <i>si - múltiple</i> o <i>case</i> .                                                                     |
|         | Impresión                     | Representa la impresión, de un resultado. Expresa <i>Escritura</i> .                                                                                                                 |
|         | Conector                      | Representa una conexión dentro de la misma página.                                                                                                                                   |
|         | Conector fuera de ésta página | Representa una conexión entre páginas diferentes.                                                                                                                                    |
|         | Proceso predefinido           | Representa un módulo de un subproblema. Se utiliza cuando se definen funciones, subrutinas, procedimientos, definidas por el programador.                                            |
|         | Flechas de flujo              | Representa la dirección del flujo del diagrama.                                                                                                                                      |

Figura 3.3 Símbolos gráficos para diagramas de flujo.

Algunos consejos prácticos para utilizar diagramas de flujo son:

- Todos los diagramas de flujo van de arriba a abajo y de izquierda a derecha (ver la figura 3.4).



Figura 3.4. Los diagramas de flujo van de arriba a bajo y de izquierda a derecha.

- Sólo tienen un inicio y un final definido.

Algunas cosas que hay que evitar al hacer diagramas de flujo son:

- Las flechas de flujo no deben cruzarse (ver la figura 3-5).

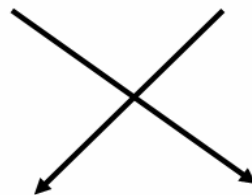


Figura 3.5. Las flechas de un diagrama de flujo no deben cruzarse.

- Cada elemento (a excepción de los de decisión) sólo tiene un flujo de entrada y uno de salida.

### Programación estructurada

- Conjunto de métodos y técnicas para diseñar y escribir programas utilizando el método científico y no solamente el método de ensayo y error.
- La programación es una actividad mental compleja que se divide en varias etapas y cuya finalidad es comprender con claridad el problema que va a resolverse o simularse en la computadora, así como entender detalladamente el procedimiento mediante el cual la computadora obtendrá la solución esperada.

### Teorema de la programación estructurada

Conocido también como el teorema de Böhm y Jacopini que dice:

- Un programa propio puede ser escrito utilizando únicamente tres tipos de estructuras de control de secuencia:  
Secuencial.  
Selectiva.  
Repetitiva.

### Pseudocódigo

- Es una mezcla de lenguaje de programación y español (o cualquier otro idioma).
- Se emplea en la *programación estructurada*, para realizar el diseño de un programa.
- Es un metalenguaje simbólico para la especificación de los pasos de un algoritmo.

### Programa propio

- Define un programa con propiedad o correcto si cumple lo siguiente:
  - Tiene un solo punto de entrada y uno de salida.
  - Toda sentencia (instrucción) del algoritmo es accesible, esto es, existe al menos un camino que va desde el inicio hasta el fin del algoritmo, se puede seguir y pasa a través de dicha sentencia.
  - No tiene ciclos o bucles infinitos.

## 3.1 Sentencias incondicionales

Se caracterizan porque una acción se ejecuta una tras otra o de manera secuencial.

### 3.1.1 Asignación

Como se había dicho anteriormente, la asignación es una operación que tiene como efecto dar valor a una variable. En los diagramas de flujo se representa con un rectángulo (ver la figura 3.6).

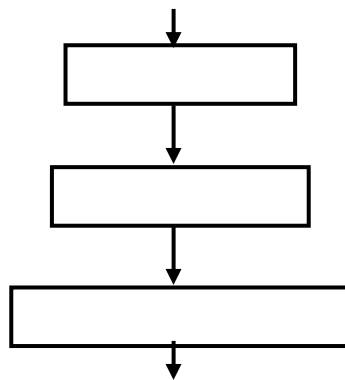


Figura 3.6. Sentencias de asignación.

Así por ejemplo, para una de las expresiones anteriores (ver la figura 3.7).

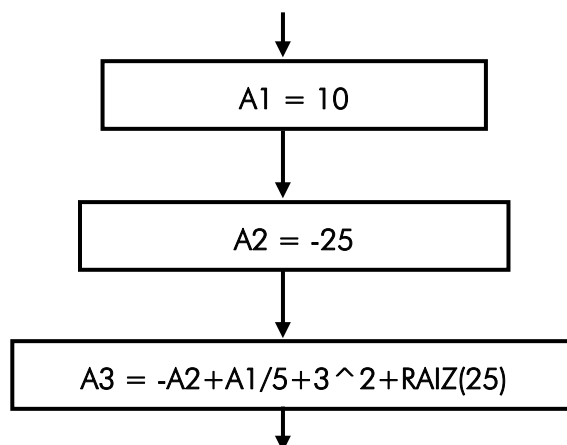


Figura 3.7. Sentencias de asignación donde se evalúan expresiones.

### 3.1.2 Lectura

La lectura es otra sentencia incondicional que permite obtener datos del teclado, de un archivo u otro dispositivo de entrada. En los diagramas de flujo se representa con el símbolo conocido con el nombre de datos (ver la figura 3.8), y que es un romboide, así:

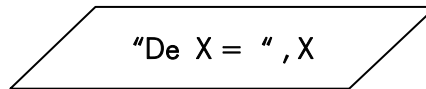


Figura 3.8. Símbolo gráfico de lectura de datos.

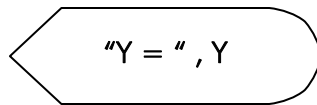
En pseudocódigo, este símbolo se traduce a las palabras Lee, o Leer, así para el caso anterior:

Leer "De X = ", X

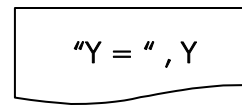
Tiene el efecto de mandar por pantalla de la computadora la cadena<sup>4</sup> "De X = " e inmediatamente después el usuario teclaea el valor de la variable **X**, al darle *Enter* o *Intro*, dicho valor se asigna a dicha variable. La palabra **Leer** o **Lee** puede ir o no subrayada, en este manual se subraya para denotar que es una sentencia incondicional, es decir, que se ejecuta invariablemente y para darle mayor claridad al pseudocódigo.

### 3.1.3 Escritura

La escritura es también una sentencia incondicional para escribir los datos en algún dispositivo de salida, ya sea la pantalla, hoja impresa, archivo en disco, etc. (ver la figura 3–9). En los diagramas de flujo la escritura se representa con los símbolos conocidos como nombre de pantalla y documento.



Símbolo pantalla



Símbolo documento

Figura 3.9. Símbolos gráficos de escritura.

En pseudocódigo, este símbolo se traduce a las palabras Escribe, o Escribir, así, para el caso anterior, para ambos casos:

Escribir "Y = ", Y

Esto tiene el efecto de mandar por pantalla, impresora de la computadora la cadena "Y = " e inmediatamente después el valor almacenado de la variable **Y**. La palabra **Escribe** o **Escribir** puede ir o no subrayada, en este manual se subraya para denotar que es una sentencia incondicional y para darle mayor claridad al pseudocódigo.

<sup>4</sup> Como recordará el lector, en el capítulo 2 vimos que una cadena es un conjunto de caracteres. A las cadenas también se les llaman o se les conocen como *strings*.

### 3.1.4 Transferencia incondicional de secuencia (Go to)

La instrucción **goto** (ir a) fue muy utilizada por los primeros programadores de la vieja guardia, ahí por los 50, 60 y 70 del siglo pasado. Fue a partir de los 70 cuando empezó a abandonarse la práctica de utilizar esta instrucción, debido a que el abuso al utilizarla en los programas, produce códigos complejos e ininteligibles (código espagueti).

En este manual no se utiliza en ningún caso esta sentencia siguiendo el consejo del desaparecido científico computacional Edsger Dijkstra, quien en un artículo publicado en 1968 titulado *Go To Statement Considered Harmful*, nos aconseja evitar el uso de esta sentencia y utilizar en su lugar las sentencias de control de flujo, que más adelante veremos en este manual (ver la figura 3.10).



Figura 3.10. Página web en *wikipedia.org* donde se menciona el artículo “Go To Statement Considered Harmful”.

A pesar de esta advertencia, muchos programadores aún la siguen utilizando, y los modernos lenguajes de programación orientados a objetos como el lenguaje Java, hacen uso del famoso goto, pero tan sólo para casos extremos y no para abusar de él.

### 3.1.5 Bloque de sentencias

Se denomina bloque de sentencias a un conjunto de instrucciones delimitadas, ya sea por llaves que abren y cierran { } como sucede en Lenguaje C, C++, Java, etcétera.

En lenguajes como Pascal, Modula, Delphi, Ada, las llaves se substituyen con dos palabras reservadas: **Begin** para la llave que abre y **End** para la llave que cierra.

Estas instrucciones o bloque se toman como una sola sentencia en las estructuras de control.

### 3.1.6 Identación

La indentación es la posibilidad de escribir los programas con sangrías y espacios desde el margen izquierdo para que el código se vea claro o fácil de entender. Los actuales

lenguajes de programación nos permiten la libre indentación, esto es, nosotros podemos poner código utilizando espacios y sangrías necesarios para que nuestro código sea legible, e incluso algunos editores ponen automáticamente la indentación de las sentencias.

Es sumamente recomendable, por cuestiones de mantenimiento del código, hacer uso de esta práctica. Como se ha mencionado, los proyectos grandes requieren de un equipo de programadores y es importante que todos puedan leer fácilmente el código. Aun en proyectos pequeños, resulta muy cómodo y legible.

## 3.2 Sentencias condicionales

Las sentencias condicionales son aquellas que dependen de que una condición en el flujo del programa se cumpla para ejecutar ciertas instrucciones (recuerde que una condición es una expresión lógica). Dentro de las sentencias condicionales tenemos las estructuras selectivas (**if**) y las repetitivas (**while**, **for**).

### 3.2.1 Selección (if)

De las sentencias condicionales selectivas **if** (**si**) tenemos tres casos: la selectiva simple, la doble y la múltiple.

- **Selectiva simple**

Se evalúa la condición, si es verdadera se ejecutan las sentencias del bloque que viene a continuación; en caso contrario se salta dicho bloque de sentencias.

```
Si (condición) entonces
    Sentencia (s);5
```

Ejemplo:

```
a = 8;

Si ( a <> 8 ) entonces
    Escribe "a es diferente de ocho";
```

En este ejemplo como **a** tiene el valor de 8 por asignación, entonces la condición (**a <> 8**) es falsa, por tanto, se brincaré la instrucción **Escribe "a es diferente de ocho"**, esto es, no la ejecutará.

Ejemplo:

```
a = 8;    b = 16;

Si ( a < b ) entonces
    Escribe "a es menor que b";
    c = a + b;
    Escribe "a más b es igual a ", c ;

FinSi
```

<sup>5</sup> En la mayoría de los ejemplos se utilizará el punto y coma como indicador del final de la instrucción, ya que es muy común en muchos lenguajes de programación.

En el ejemplo anterior, **a** tiene el valor de 8 y **b** el de 16, luego la condición (  $a < b$  ) es verdadera , entonces el programa ejecutará las instrucciones dentro del bloque Si – FinSi y escribirá en la pantalla:

```
a es menor que b
a más b es igual a 24
```

En la figura 3.11 vemos el diagrama de flujo y su equivalente en pseudocódigo de la estructura selectiva simple. Recuerde que la decisión (condición o expresión lógica) se representa con un rombo.

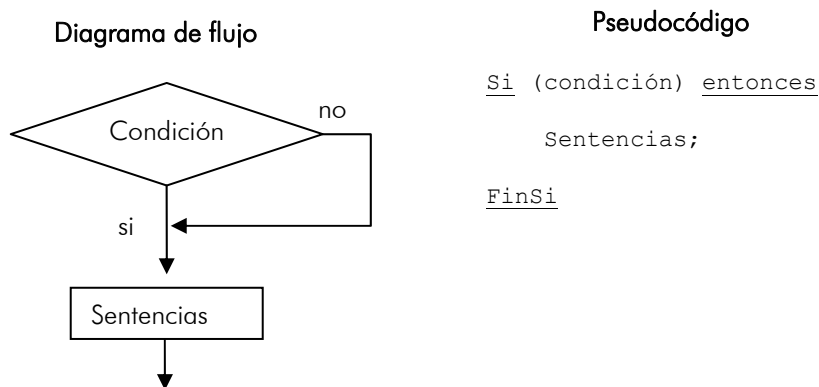


Figura 3.11. Diagrama de flujo y pseudocódigo de la estructura selectiva simple.

- **Selectiva doble**

Es similar a la sentencia selectiva simple, if (si), pero incluye dos sentencias en lugar de una. Permite la ejecución de una sentencia o bloque de sentencias si se cumple la condición lógica, al igual que la sentencia si, pero en el caso de que no se cumpla esa condición, permite la ejecución de otra sentencia u otro bloque de sentencias diferente. La sintaxis de la sentencia si–sino es:

```

Si (condición) entonces
    sentencia 1;
sino
    sentencia 2;

```

Ejemplo:

```

día = 7;
Si ( día <> 7 ) entonces
    Escribe "Hoy no es domingo";
    Escribe "Hoy tienes que trabajar";
sino
    Escribe "Hoy hay que descansar porque es domingo";
    Escribe "Y como es domingo hay que ver el football";
Finsi

```

En el ejemplo anterior, como día es una variable numérica con un valor asignado de 7, entonces la condición (día <> 7) será falso, entonces se escribirá en la pantalla:

Hoy hay que descansar porque es domingo

Y como es domingo hay que ver el football

Formalizando:

Si la condición lógica es verdadera, se ejecutará la sentencia 1. Después, la ejecución se reanuda tras la sentencia si – sino, esto es, tras la sentencia 2 (que no se ejecutará).

Si la condición lógica es falsa, se ejecutará la sentencia 2, y no se ejecutará la sentencia 1. Después proseguirá la ejecución del programa (ver la figura 3.12).

Tanto la sentencia 1 como la sentencia 2, pueden ser bloques de sentencias delimitadas por llaves de bloque.

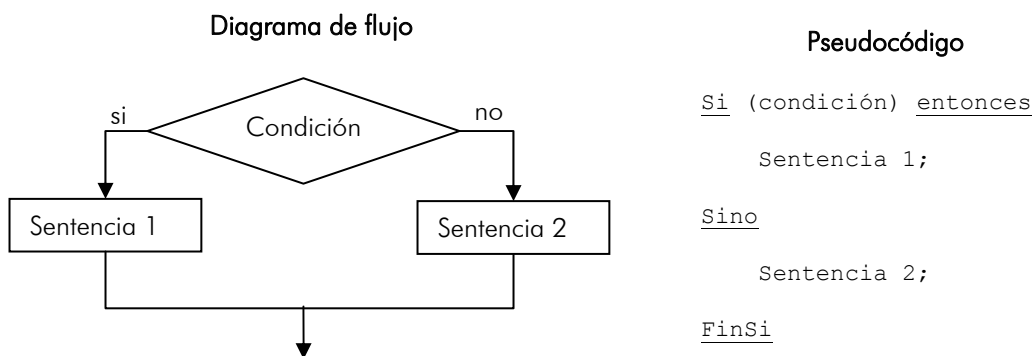


Figura 3.12. Diagrama de flujo y pseudocódigo de la estructura selectiva doble.

- **Selectiva múltiple**

Esta estructura se aplica cuando tenemos condiciones múltiples, es decir, muchas condiciones que verificar, veamos un ejemplo:

El siguiente fragmento de un algoritmo, pregunta el nombre y la profesión de una persona y da una respuesta por pantalla:

```

Inicio
Cadena  resp;
Cadena  profesion;
Lee  "¿Hola cómo te llamas?", resp;
Lee  "¿Cuál es tu profesión? ", profesion;
Si  (profesion = "Físico") Entonces
    Escribe "Tu profesión es del área de Físico-Matemáticas ", resp;
    Sino Si (profesion = "Matemático") Entonces
        Escribe "Tu profesión es del área de Físico-Matemáticas ", resp;
        Sino Si (profesion = "Informática") Entonces
            Escribe "Tu profesión es del área de Físico-Matemáticas ", resp;
          
```

```

Sino Si (profesion = "Médico") Entonces
    Escribe "Tu profesión es del área de Ciencias
    Biológicas", resp;

    Sino Si (profesion = "Biólogo") Entonces
        Escribe "Tu profesión es del área de
        Ciencias Biológicas ", resp;

        Sino Si (profesion = "Dentista") Entonces
            Escribe "Tu profesión es del área de
            Ciencias Biológicas ", resp;

            Sino
                Si (profesion = "Contador") Entonces
                    Escribe "Tu profesión es del
                    área de Ciencias Sociales ",
                    resp;

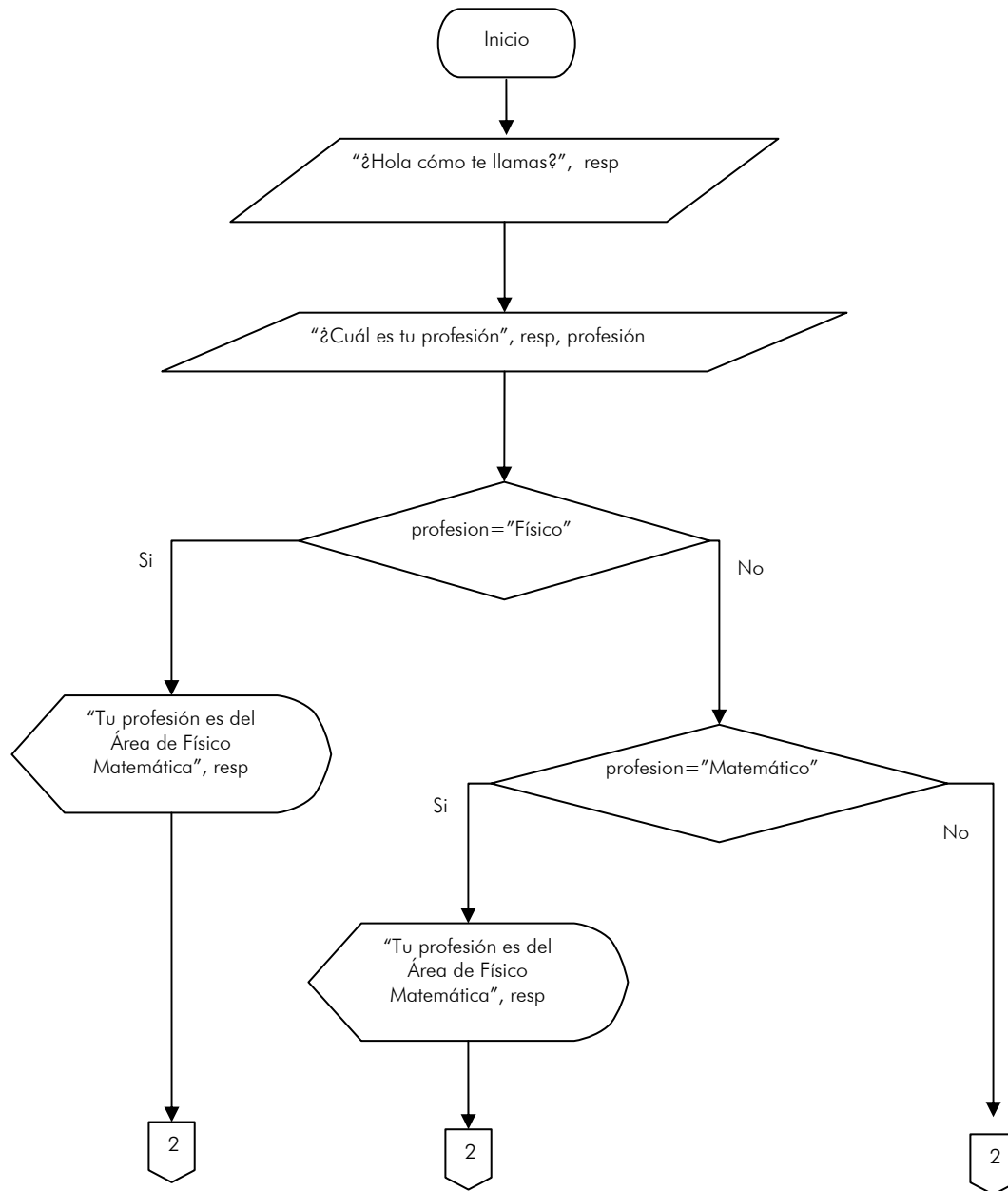
                    Sino
                        Si (profesion = "Administrador")
                        Entonces
                            Escribe "Tu profesión es del
                            área de Ciencias Sociales ",
                            resp;

                            Sino
                                Escribe "Tu profesión no la
                                conozco ", resp;

                                FinSi
                            FinSi
                        FinSi
                    FinSi
                FinSi
            FinSi
        FinSi
    FinSi
Fin

```

Observe el lector que el número de FinSi corresponde al número de Si o condiciones que tenga el algoritmo; además de la indentación en forma de "escalerita" nos facilita visualizar esta estructura de control. Ahora revise el diagrama de flujo del anterior algoritmo (ver las figuras 3.13 y 3.14).

Figura 3.13. Ejemplo de la estructura selectiva múltiple con *if* anidados.

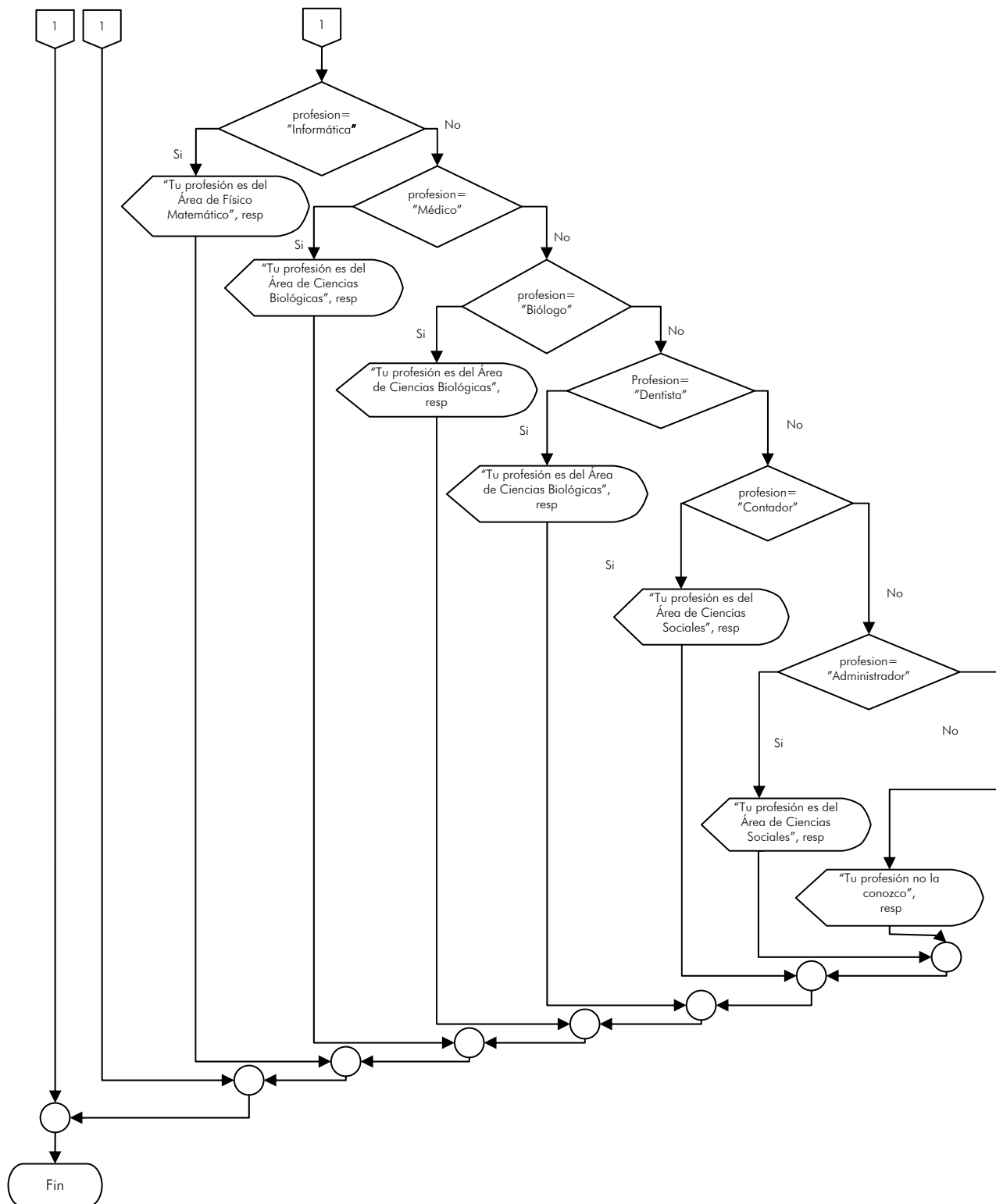


Figura 3.14. Ejemplo de la estructura selectiva múltiple con *if* anidados.

En este ejemplo, se evalúa la condición 1, si es verdad ( si ), se ejecutan el bloque de sentencias 1, sino, si es falsa la condición 1 ( no ), entonces se evalúa la condición 2.

Si la condición 2 es verdad (si) se ejecutan el bloque de sentencias 2, sino, si es falsa la condición 2 (no), entonces se evalúa la condición 3, y así sucesivamente hasta llegar a evaluar la condición N, si es verdad (si) entonces se ejecuta el bloque de sentencias N.

Finalmente si da falso en todas las condiciones desde la 1 hasta la N, entonces se ejecuta el bloque de sentencias

$N + 1$  (ver las figuras 3.15 y 3.16).

### Selectiva múltiple a if anidados

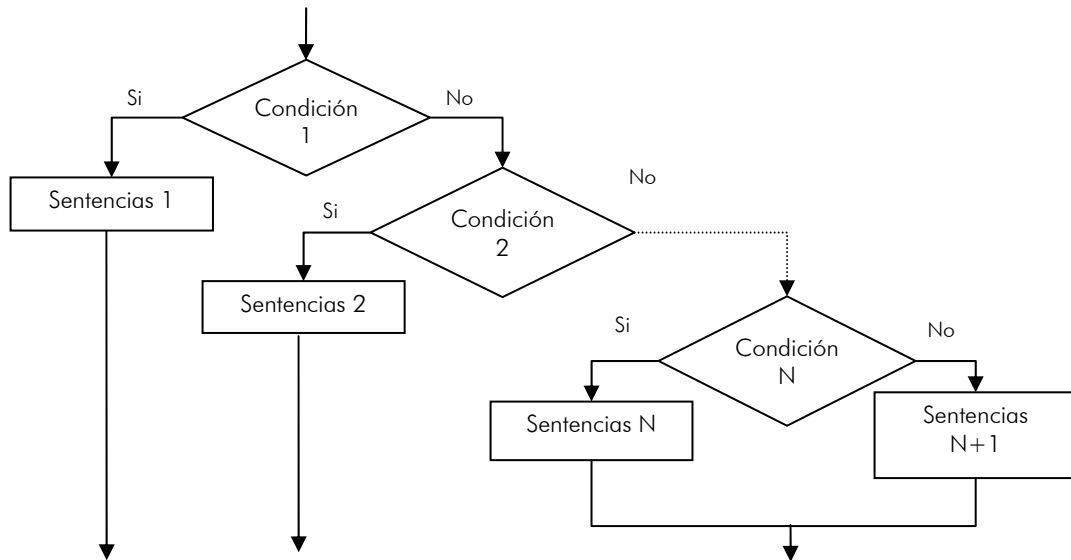


Figura 3.15. Diagrama de flujo de la estructura de selección múltiple o if anidados.

### Pseudocódigo selectiva múltiple o if anidados

```

Si ( condición 1 ) entonces
    Sentencias 1;
Sino
    Si ( condición 2 ) entonces
        Sentencias 2;
    Sino
        .....
        Si ( condición N ) entonces
            Sentencias N;
        Sino
            Sentencias N+1;
        FinSi
    FinSi
FinSi
  
```

Figura 3.16. Pseudocódigo de la estructura de selección múltiple o if anidados.

### 3.2.2 Ciclos (*while*, *for*)

Las instrucciones repetitivas o iterativas más comunes son los ciclos *while* (mientras) y *for* (para), que nos ayudan a repetir un determinado número de veces una o varias sentencias. El número de veces que se ejecutan dependen de la condición que se establezca. A este tipo de sentencias también se les conocen como sentencias de bucle (loop).

- **Ciclo *while* (mientras)**

El ciclo *while* tiene la característica de ser un ciclo que se repite mientras la condición que se evalúa sea verdadera.

Supongamos que deseamos realizar un programa para ejecutar una tabla de conversión de grados Fahrenheit a grados Celsius, empezando de un límite superior a un límite inferior. Usaremos el ciclo *while* para este propósito:

```
Inicio
Entero fahr, limite, decre;
Escribe "Convierte a grados centígrados partiendo de Fahrenheit";
Lee "Dame el límite de grados a calcular ", limite;
Lee " Dame el decremento ", decre;
fahr = limite;
mientras ( fahr > 0 ) hacer
    Escribe fahr, ( 5.0 / 9.0 ) * ( fahr - 32 ) ;
    fahr = fahr - decre ;
FinMientras
Escribe "Pulse cualquier tecla para continuar..." ;
llama obtencar( ) ;
Fin
```

Observe que la condición lógica ( $fahr > 0$ ) controla la repetición de las sentencias que se encuentran entre el *mientras* y el *FinMientras*. Cuando sea verdadera se repetirán las sentencias, pero cuando sea falsa se terminará el ciclo repetitivo para ejecutar las sentencias que estén inmediatamente después, entre las cuales esta la invocación o llamado a una función `obtencar( )` que tiene el efecto de pedir un carácter del teclado al usuario para terminar el programa (en el capítulo de Funciones veremos a detalle como crear y utilizar una función).

Ahora revise el diagrama de flujo del anterior algoritmo (ver la figura 3.17).

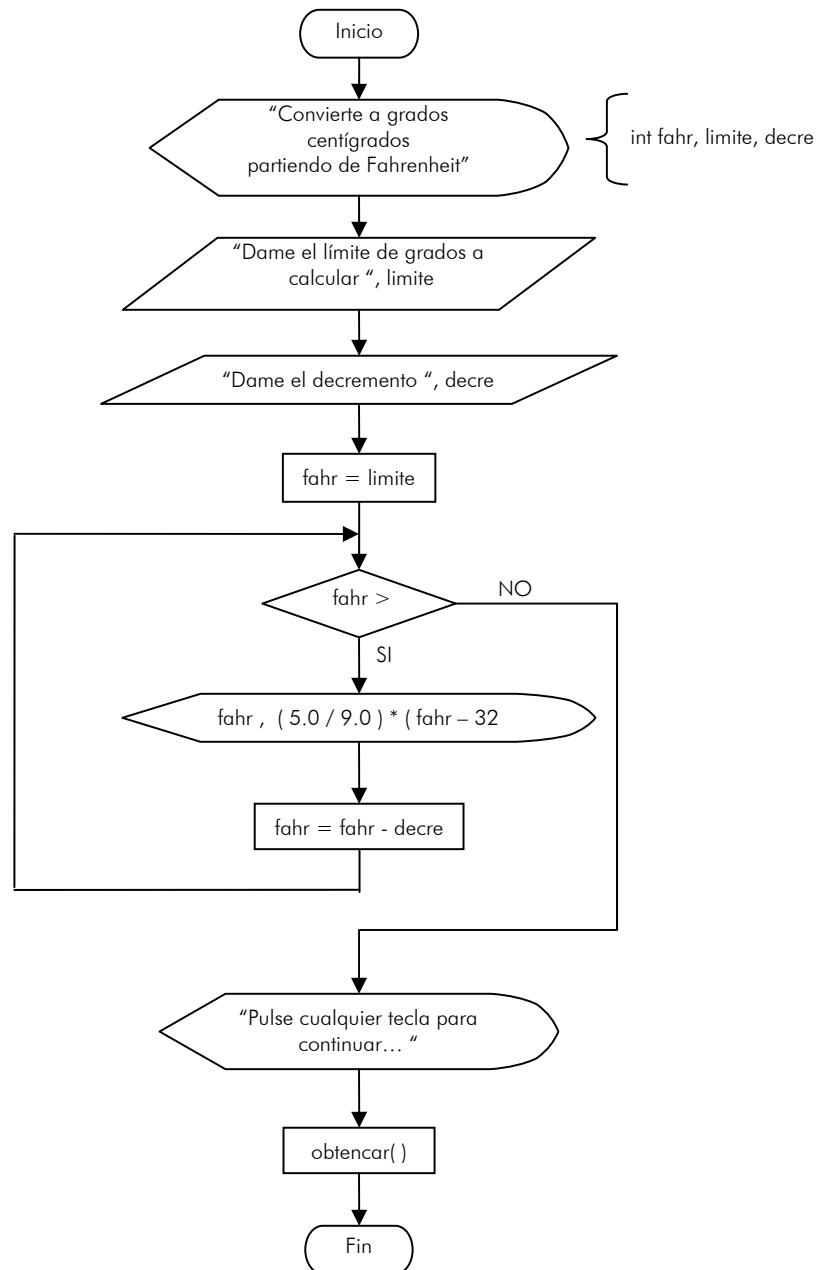


Figura 3.17. Diagrama de flujo del ejemplo del ciclo *while* (mientras).

En esta estructura de ciclo o bucle *while* se evalúa la condición o expresión lógica y según sea el resultado, si es verdad se ejecutan las sentencias N veces, si es falsa no se ejecutan las sentencias. Debido a la condición o expresión lógica que se evalúa al principio, este ciclo se ejecuta de 0 a N veces (ver la figura 3.18).

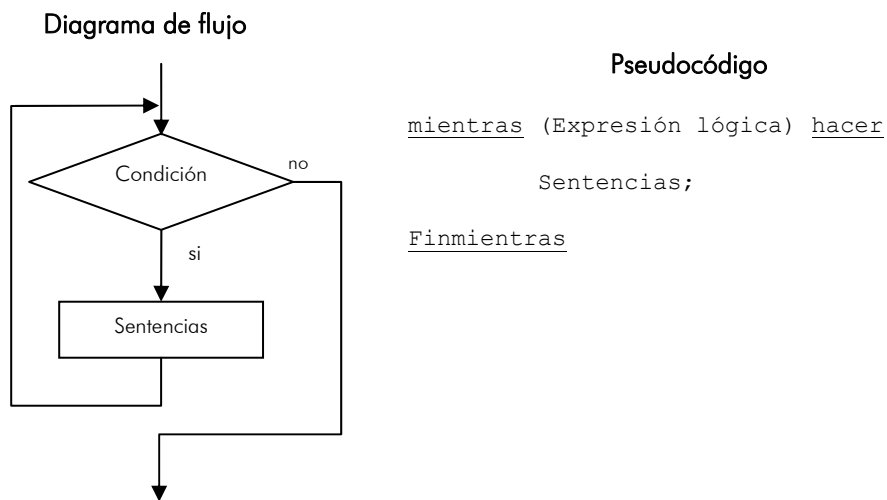


Figura 3.18. Diagrama de flujo y pseudocódigo de la estructura repetitiva *while* (mientras).

- **Estructura repetitiva do – while (repetir–hasta)**

Este ciclo o bucle nos permite por lo menos ejecutar una repetición del bloque de sentencias. Veamos el ejemplo del programa que hace una tabla de conversión de grados Fahrenheit a grados Celsius, empezando de un límite superior a un límite inferior. Usaremos el ciclo do – *while* (repetir – hasta) para este propósito:

```

Inicio
Entero fahr, limite, decre;
Escribe "Convierte a grados centígrados partiendo de Fahrenheit";
Lee "Dame el límite de grados a calcular ", limite;
Lee "Dame el decremento ", decre;
fahr = limite;

Repetir
    Escribe fahr, ( 5.0 / 9.0 ) * ( fahr - 32 );
    fahr = fahr - decre ;
Hasta ( fahr <= 0 );

Escribe "Pulse cualquier tecla para continuar...";
llama obtencar( ) ;
Fin
  
```

Vea en el ejemplo que las sentencias dentro del bloque Repetir – Hasta se ejecutan por lo menos una vez y que la condición lógica ( $\text{fahr} \leq 0$ ) controla la repetición de las sentencias. Cuando la condición o expresión lógica sea falsa se repetirán las sentencias, pero cuando

sea verdadera se terminará el ciclo repetitivo para ejecutar las sentencias que estén inmediatamente después.

Ahora revise el diagrama de flujo del anterior algoritmo (ver la figura 3.19).

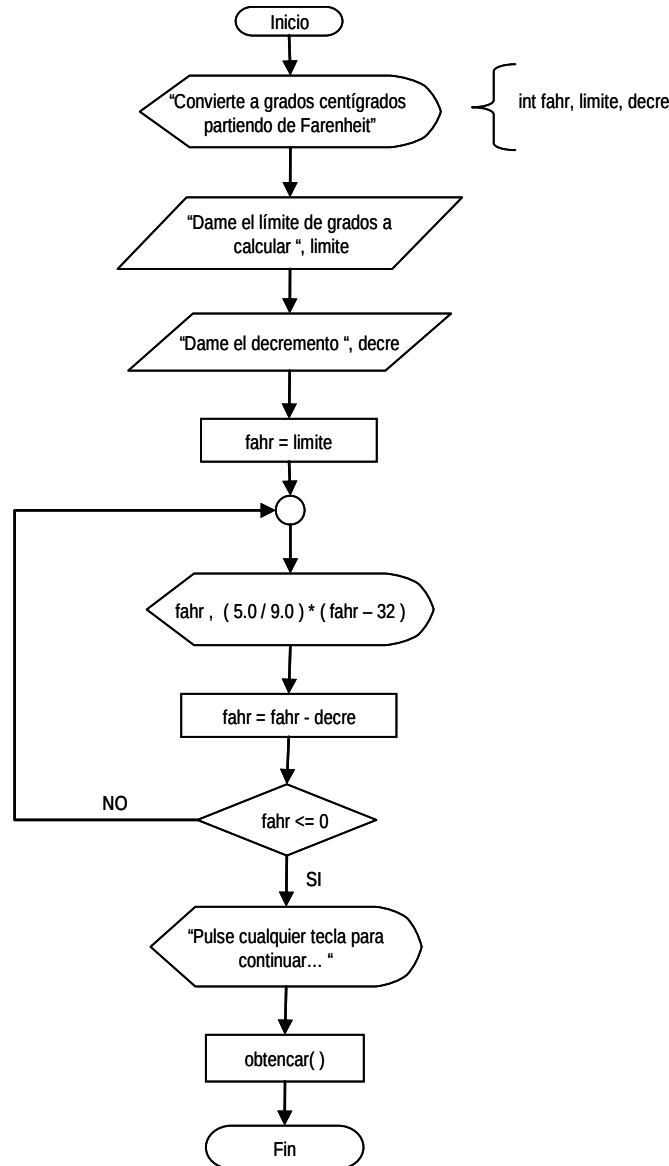


Figura 3.19. Diagrama de flujo del ejemplo de grados Fahrenheit con el ciclo *do-while* (repetir-hasta).

Ahora para formalizar, el ciclo *do-while* (repetir-hasta) es una variación del ciclo *while* y en esta estructura se ejecutan las sentencias y luego se evalúa la condición o expresión lógica, si es falsa se ejecutan las sentencias N veces, si es verdadera se sale del ciclo. Debido a que la condición o expresión lógica que se evalúa al final, este ciclo se ejecuta de 1 a N veces (ver la figura 3.20).

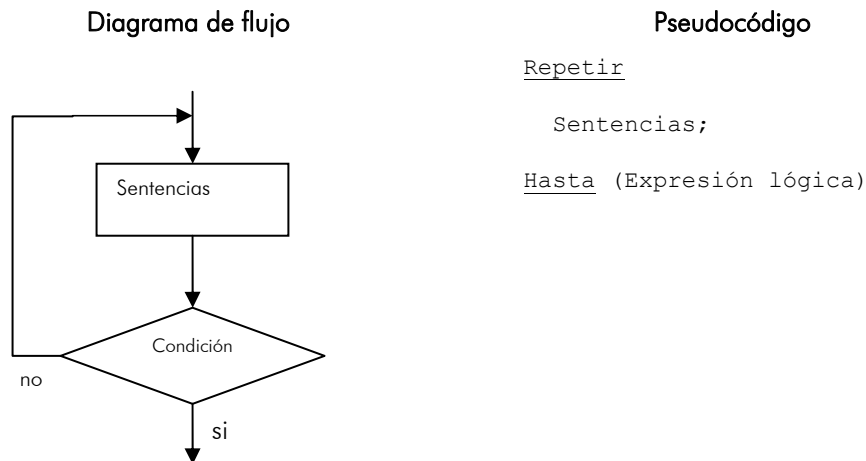


Figura 3.20. Diagrama de flujo y pseudocódigo del ciclo do-while (repetir-hasta).

### La estructura repetitiva for (para)

Este es uno de los ciclos o bucles más utilizados en los lenguajes de programación. Veamos el mismo ejemplo del programa que hace una tabla de conversión de grados Fahrenheit a grados Celsius, empezando de un límite superior a un límite inferior. Usaremos el ciclo *for* (para) para este propósito:

```

Inicio
Entero  fahr, limite, decre;
Escribe "Convierte a grados centígrados partiendo de Fahrenheit";
Lee "Dame el límite de grados a calcular ", limite;
Lee " Dame el decremento ", decre;
fahr = limite;

Para fahr = limite hasta 0 decremento decre hacer

    Escribe fahr, ( 5.0 / 9.0 ) * ( fahr - 32 ) ;

FinPara

Escribe "Pulse cualquier tecla para continuar...";
llama  obtencar( ) ;
Fin
  
```

Vea el lector que la variable que controla las repeticiones es *fahr* la cual es inicializada con el valor *limite* y hasta cuando llegue al valor final 0 se repetirá el bloque de sentencias entre el **Para – FinPara**. También observe que la variable *decre* es un decremento y equivale a hacer  $fahr = fahr - decre$ .

Ahora cheque el diagrama de flujo del anterior algoritmo (ver la figura 3.21).

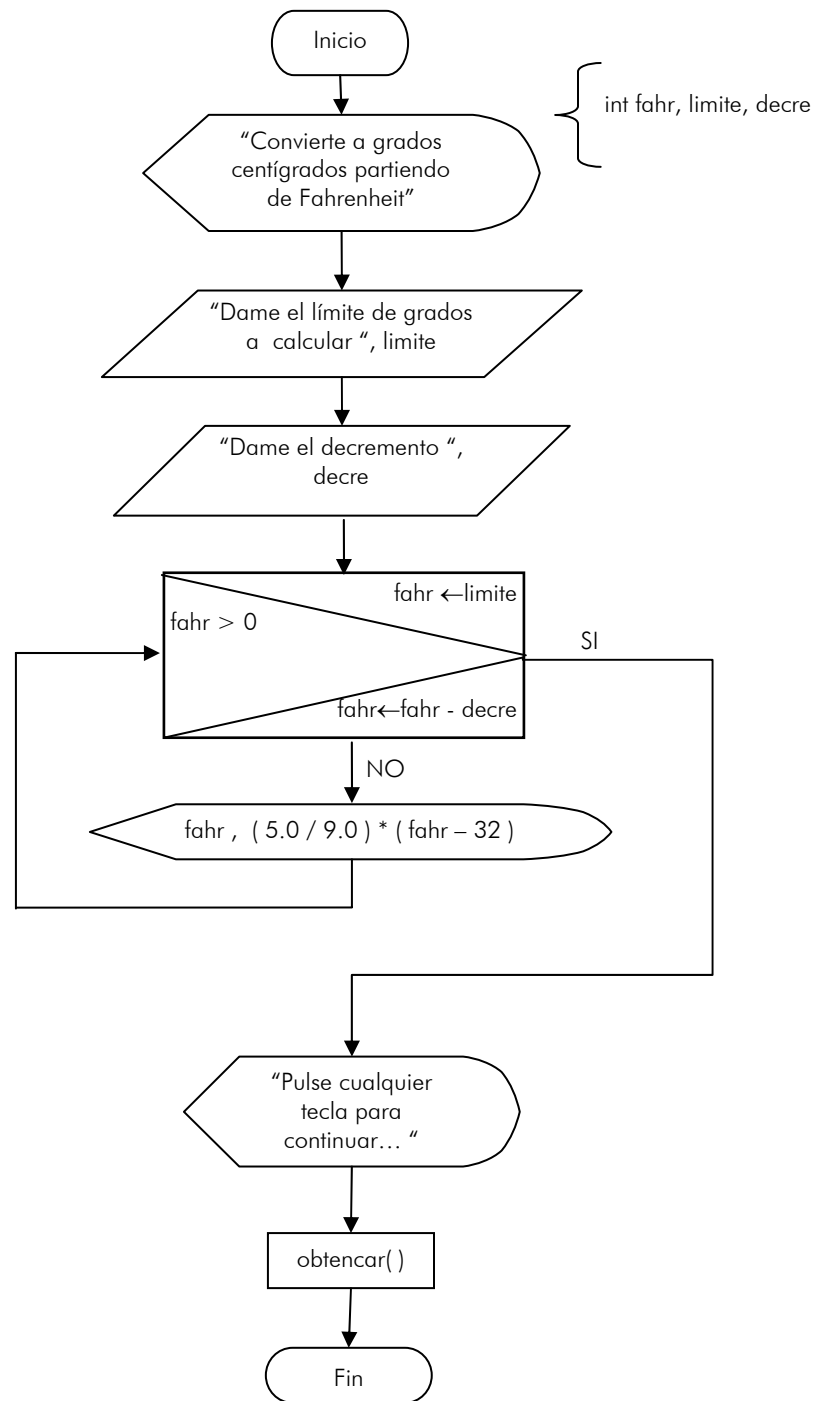


Figura 3.21. Diagrama de flujo del ejemplo de grados Fahrenheit con ciclo *for* (para).

Formalizando, este ciclo *for* (para), comienza con un valor inicial **vi** que se asigna a la variable índice **v**, ésta se incrementa o decrementa en 1 o en un valor que se especifique, y si el nuevo valor no excede al valor final **vf** se continua repitiendo las sentencias, en caso contrario se sale del ciclo (ver la figura 3.22).

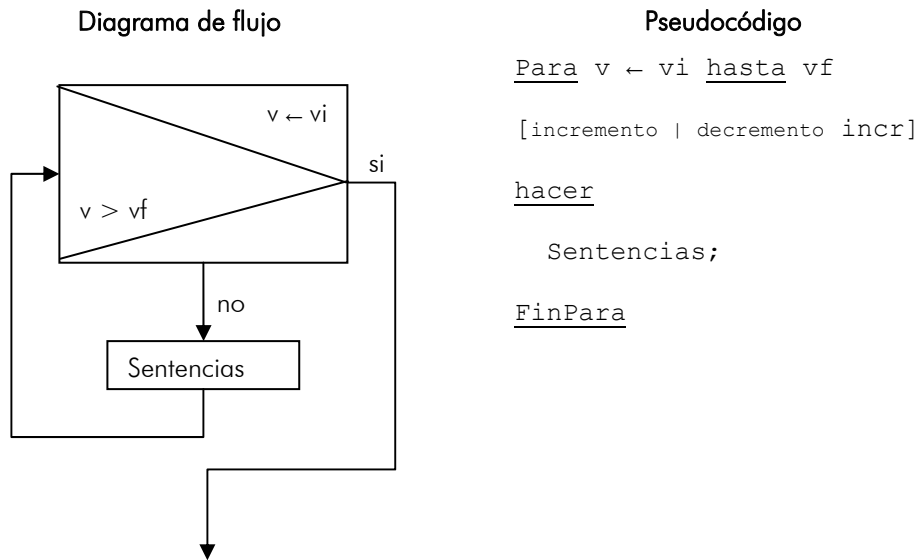


Figura 3.22. Diagrama de flujo y pseudocódigo del ciclo *for* (para).

### 3.2.3 Selección múltiple (*switch* o *select – case*)

Algunos lenguajes de programación tienen una estructura de control selectiva múltiple conocida como *select – case* o *switch – case*. Es utilizada cuando hay que seleccionar entre más de dos alternativas como falso o verdadero, sino más bien de un rango de posibles valores resultantes de la evaluación de una expresión. Estos valores generalmente son discretos, como caracteres, números enteros, etc. (Aunque hay lenguajes como el Visual Basic y el VBA que permiten rangos de valores continuos como valores reales o de punto flotante).

Ejemplo:

Supongamos que desea hacer un algoritmo que le pida dos números reales por teclado y luego un carácter (+, -, \*, /) para efectuar dicha operación entre esos dos números e imprimir el resultado por pantalla. El programa se repetirá hasta que el usuario teclee una N o un NO y terminará el programa, pero en caso contrario volverá a iniciar el proceso de preguntarnos dos nuevos números reales y una opción para calcular.

```

Inicio
Real  num1, num2;
Carácter  operador;
Cadena  respuesta;
Escribe  "El programa realiza una operación ( + , - , * , / ) con
           dos números reales leídos por teclado
           y muestra el resultado en la pantalla" ;
Repetir
    Lee  "Dame el primer número ", num1 ;
    Lee  "Dame el segundo numero ", num2 ;
    Escribe  "¿Qué operación desea hacer?
  
```

```

        Introduzca el símbolo de la
        Operación a calcular:
        '+' para Sumar
        '-' para Restar
        '*' para Multiplicar
        '/' para Dividir";
Lee "Indique el símbolo " , operador;
Caso ( operador ) hacer
    '+' : Escribe "La suma de ", num1, " + " , num2 , " es
          igual a " , (num1 + num2) ;
          break;
    '-' : Escribe "La resta de ", num1, " - " , num2 , " es
          igual a " , (num1 - num2) ;
          break;
    '*' : Escribe "La multiplicación de ", num1, " * " , num2 , "
          es igual a " , (num1 * num2) ;
          break;
    '/' : Escribe "La división de ", num1, " / " , num2 , " es
          igual a " , (num1 / num2) ;
          break;
    Sino : Escribe "¡Símbolo incorrecto! Solo son validos los
          símbolos ( +,- , * , / ) "
          break;

FinCaso

Lee "¿Desea hacer otro cálculo ( S / N ) ? " , respuesta;

Hasta (respuesta = "S" or respuesta = "SI" or respuesta = "s" or
respuesta = "si") ;

Escribe "Pulse cualquier tecla para continuar...";
llama obtencar( ) ;
Fin

```

Observe que hay que utilizar la palabra *break* para poder terminar cada uno de los casos o de lo contrario, se seguirá ejecutando el siguiente caso. Esta palabra reservada *break* es de uso frecuente en lenguajes como C / C++, Java, PHP, etcétera.

Ahora revise el diagrama de flujo del anterior algoritmo (ver las figuras 3.23 y 3.24).

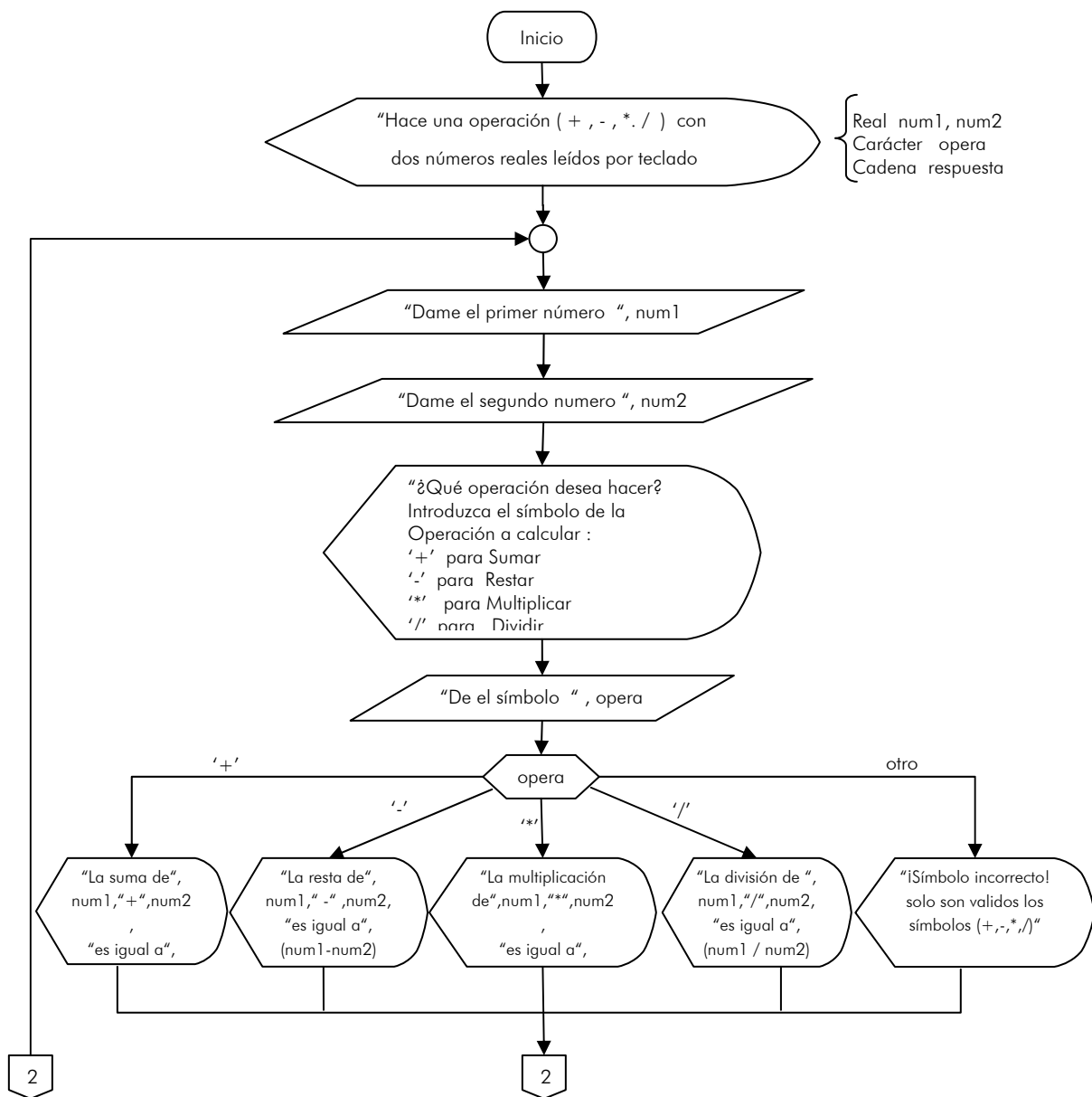
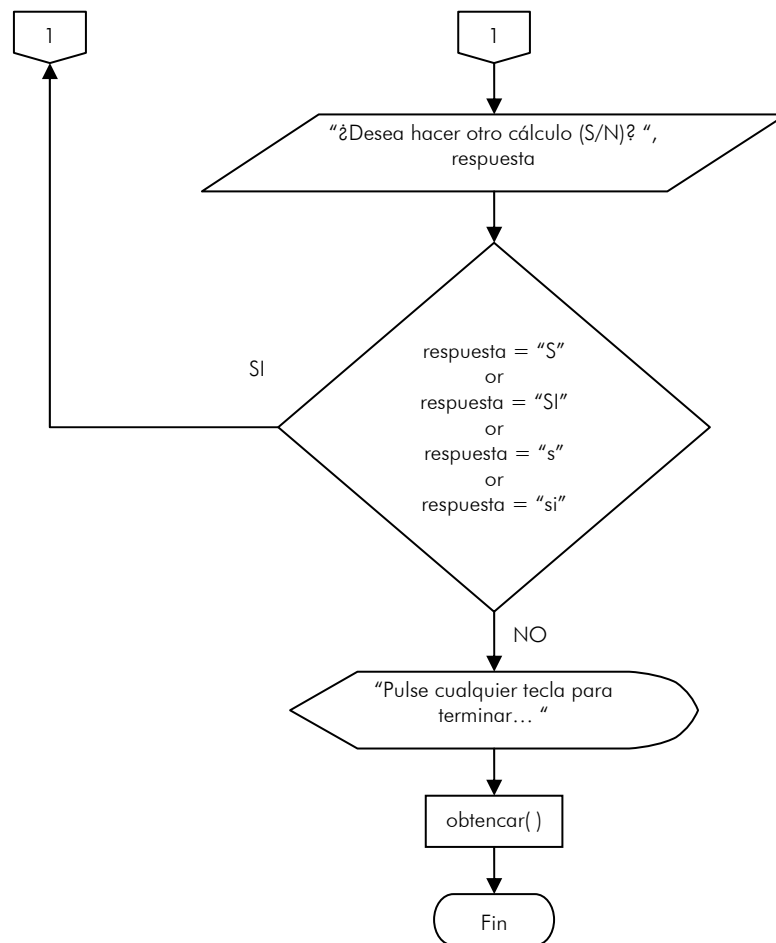


Figura 3.23. Diagrama de flujo del ejemplo de la calculadora con selección múltiple (switch o select – case).



3.24. Diagrama de flujo del ejemplo de la calculadora con selección múltiple (switch o select – case).

Formalizando, esta estructura de control selectiva múltiple conocida como *select – case* o *switch – case* es utilizada cuando se evalúa la expresión (una expresión numérica o una expresión que nos de un valor discreto como un carácter por ejemplo), según sea el resultado, si es el valor v1 se ejecutan las sentencias 1, si es el valor v2 se ejecutan las sentencias 2, etc. Si no es ningún caso se ejecutan las sentencias N (ver la figura 3.25).

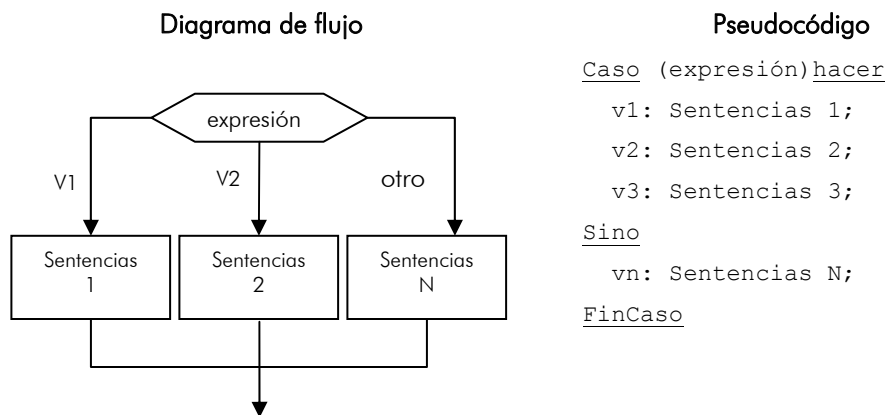


Figura 3.25. Diagrama de flujo y pseudocódigo de la estructura de selección múltiple (switch o select – case).

### 3.2.4 Ejemplos de estructuras de control

A continuación veremos algunos ejemplos sencillos de las estructuras de control expuestas.

**Ejemplo 1.** Realice el algoritmo con un diagrama de flujo y pseudocódigo tal que, dados los valores de  $a = 5.6$ ,  $b = 7.0$ , y  $c = 3.0$  (lados de un triángulo), calcule  $p$  y su área  $S$  (ver figura 3.26).

Donde:

$$p = (a + b + c)/2 \text{ y}$$

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

Solución:

Ejemplo 1

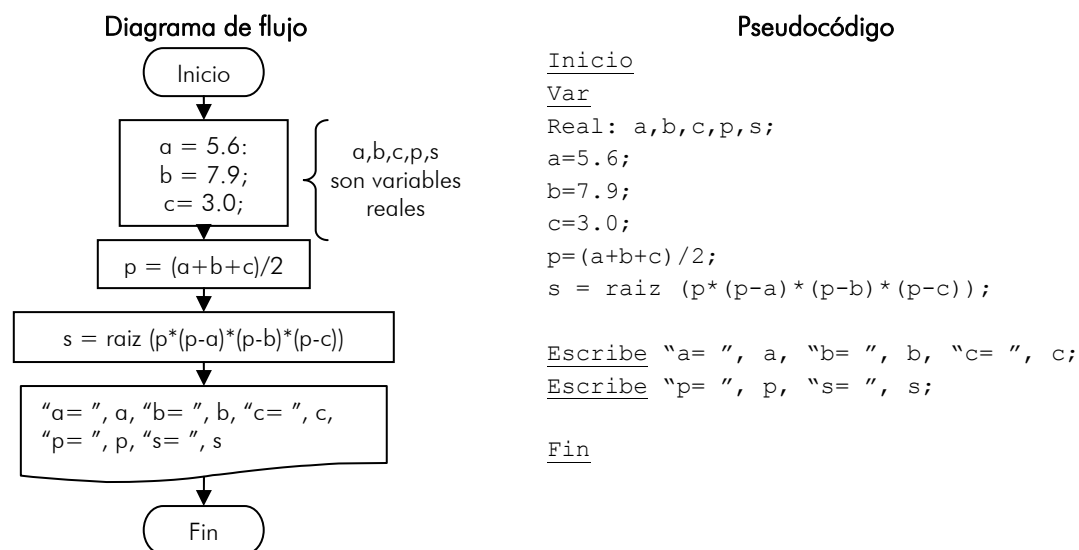


Figura 3.26. Diagrama de flujo y pseudocódigo del ejemplo 1.

**Ejemplo 2.** Construya un algoritmo con diagrama de flujo y en pseudocódigo tal que, dados los valores enteros P y Q, determine si los mismos satisfacen la siguiente expresión (ver figura 3.27).

$$P^3 + Q^4 - 2P^2 < 680$$

En caso afirmativo debe imprimir los valores P y Q.

Solución:

### Ejemplos Control de flujo. Selección simple

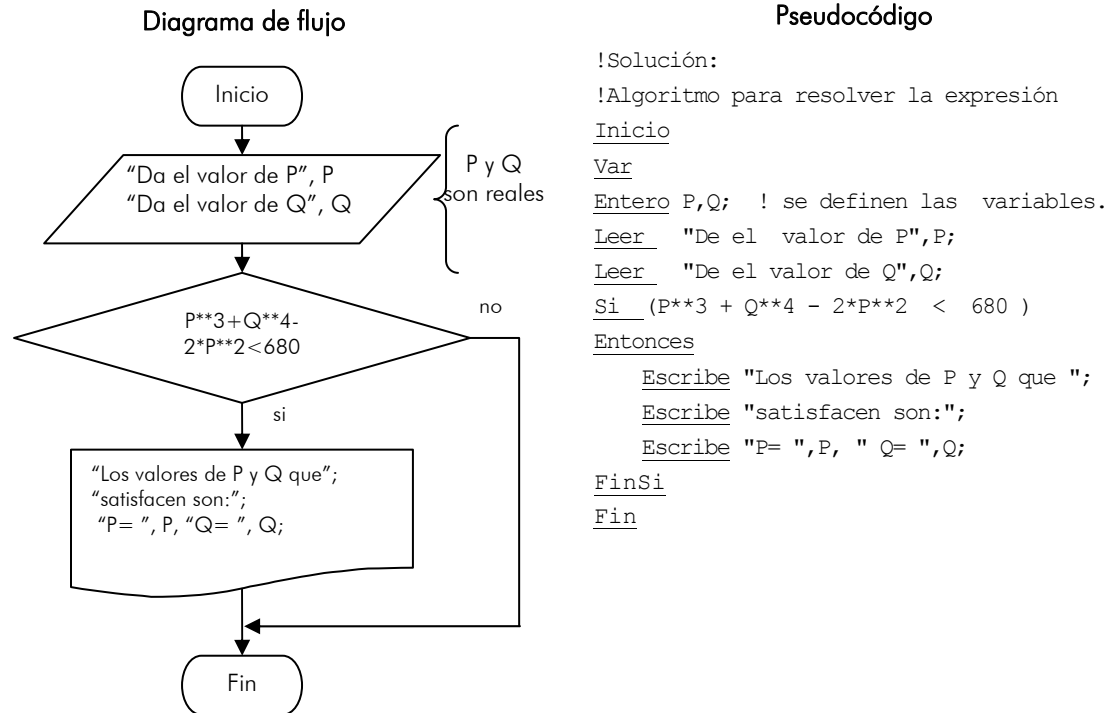


Figura 3.27. Diagrama de flujo y pseudocódigo del ejemplo 2.

**Ejemplo 3.** Construya un algoritmo con diagrama de flujo y en pseudocódigo tal que, dada la calificación de un alumno en un curso de computación que consiste en el promedio de tres exámenes de la siguiente forma:  $CAL = (EX1 + EX2 + EX3) / 3.0$ ; escriba "aprobado" si su calificación es mayor que 7.5 y "reprobado" en caso contrario. Las calificaciones deben ser leídas para calcular la variable CAL (ver figuras 3.28, 3.29 y 3.30).

Solución:

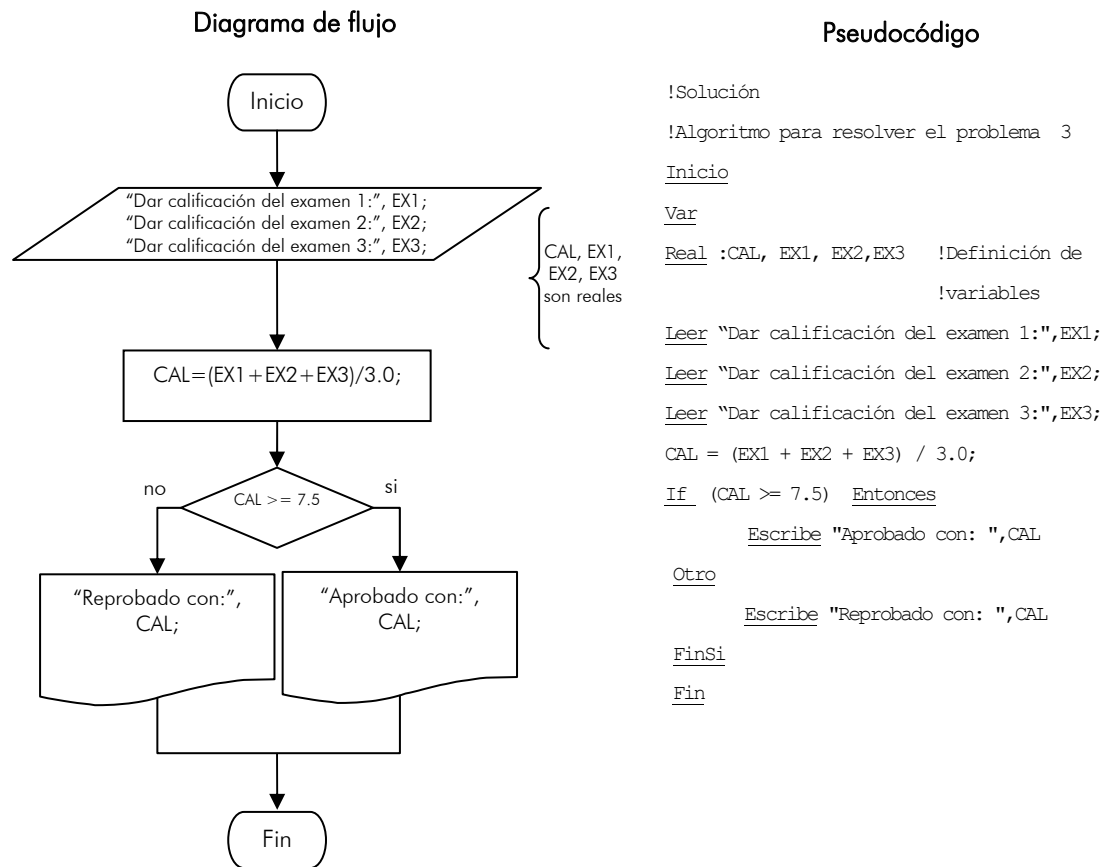


Figura 3.28. Diagrama de flujo y pseudocódigo del ejemplo 3.

**Ejemplo 4.** Elabore un algoritmo con diagrama de flujo y pseudocódigo tal que, dado como dato una temperatura en grados Centígrados, calcule los grados Fahrenheit y determine el deporte que es apropiado practicar a esa temperatura, teniendo en cuenta la siguiente tabla, vea figura:

| DEPORTE  | TEMPERATURA en grados Fahrenheit |
|----------|----------------------------------|
| Natación | > 85                             |
| Tenis    | $70 < TEMP \leq 85$              |
| Golf     | $32 < TEMP \leq 70$              |
| Esquí    | $10 < TEMP \leq 32$              |
| Marcha   | $\leq 10$                        |

Solución (pseudocódigo *if* anidado):

```

!Solución:
!Algoritmo para resolver el problema 4 (1ra. forma)
Inicio
Var Entero : TEMPF ! Temperatura en grados Fahrenheit
Leer "Dar el valor de TEMPF:",TEMPF
Si (TEMPF>85) Entonces
    Escribe "Natación";
otro
    Si (TEMPF>70) Entonces
        Escribe "Tenis";
    otro
        Si (TEMPF>32) Entonces
            Escribe "Golf";
        otro
            Si (TEMPF>10) Entonces
                Escribe "Esquí";
            otro
                Escribe "Marcha";
            finsi
        finsi
    finsi
finsi
fin
  
```

Figura 3.29. Pseudocódigo del ejemplo 4.

Solución (diagrama de flujo *if* anidados):

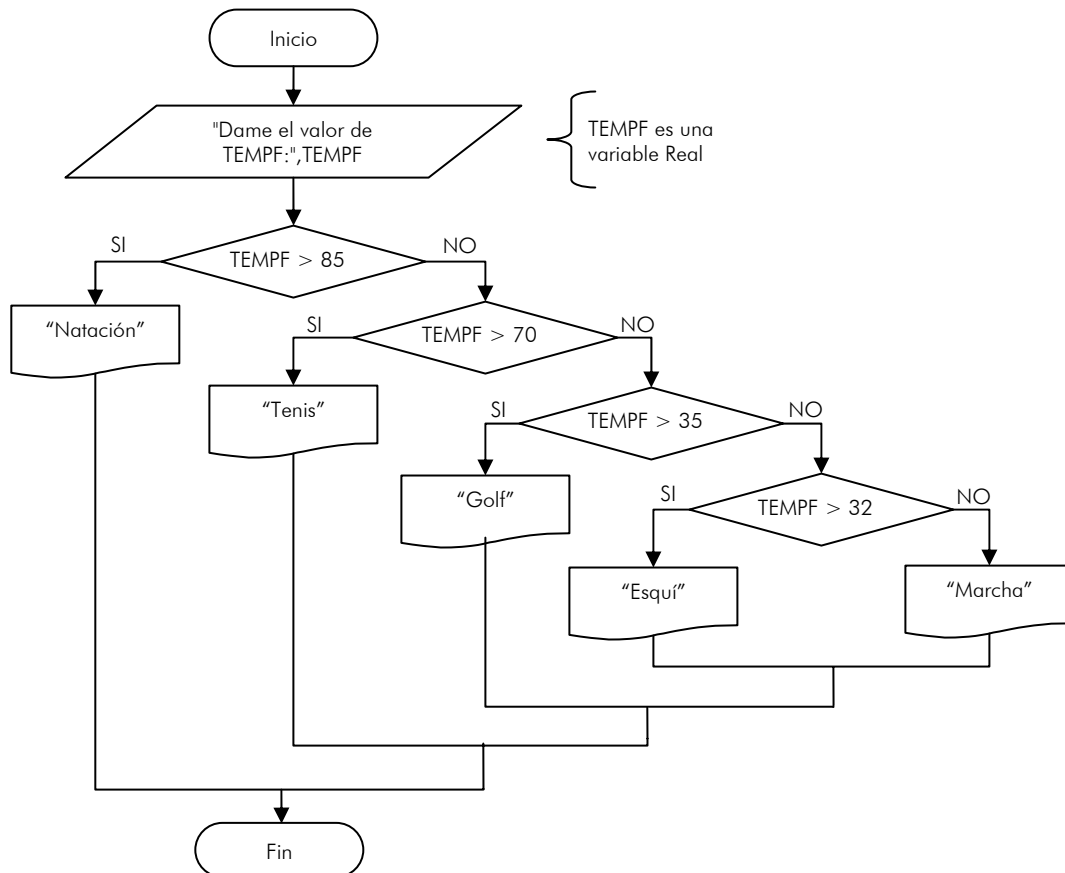


Figura 3.30. Diagrama de flujo del ejemplo 4.

Solución (diagrama de flujo con switch – case)

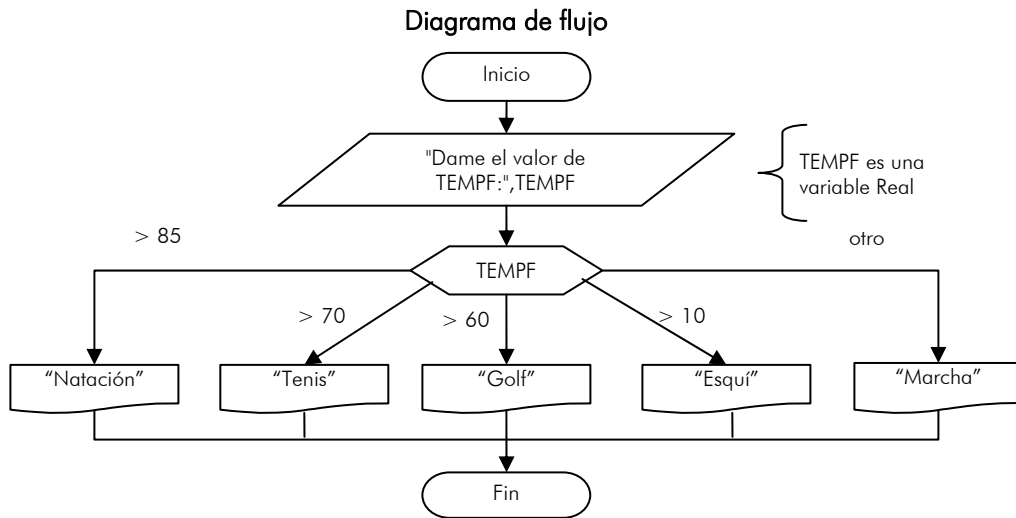


Figura 3.31. Diagrama de flujo del ejemplo 4.

**Ejemplo 5.** Realice el algoritmo en diagrama de flujo y en pseudocódigo que calcule e imprima la suma de los primeros N números enteros positivos (ver figuras 3.31, 3.32 y 3.33).

Solución (ciclo while):

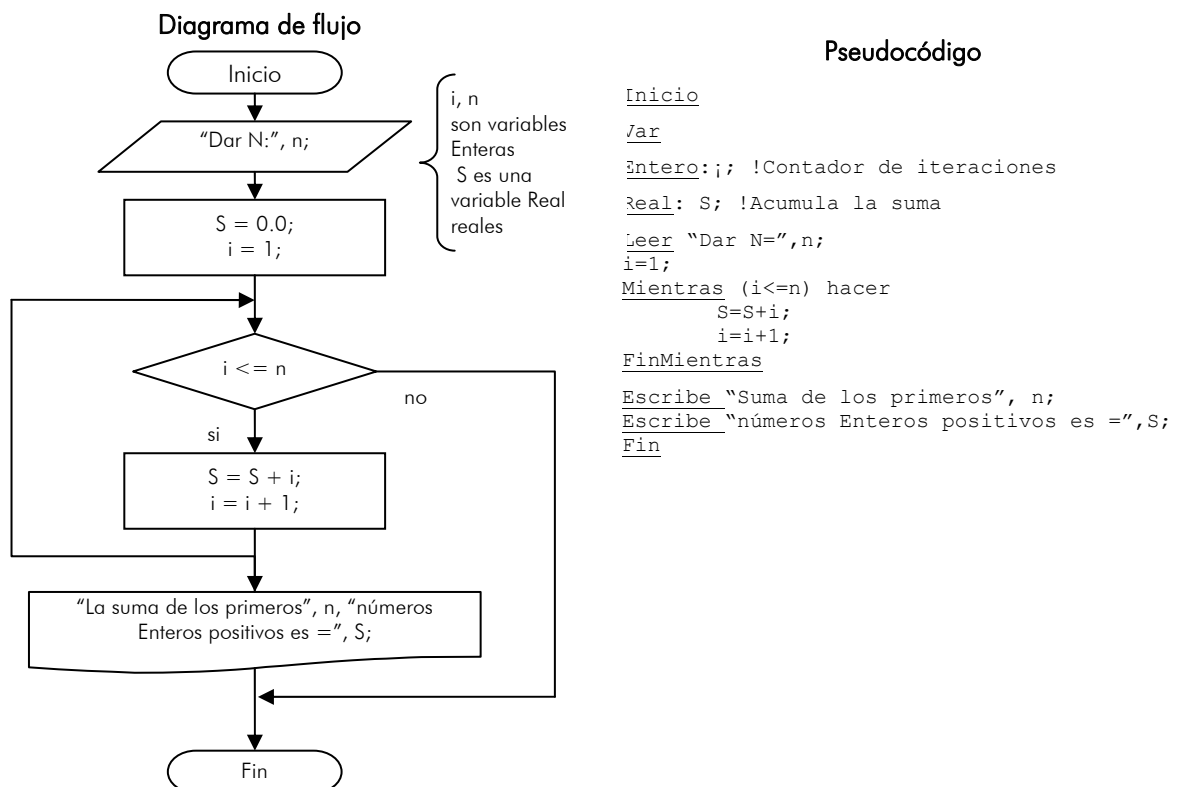
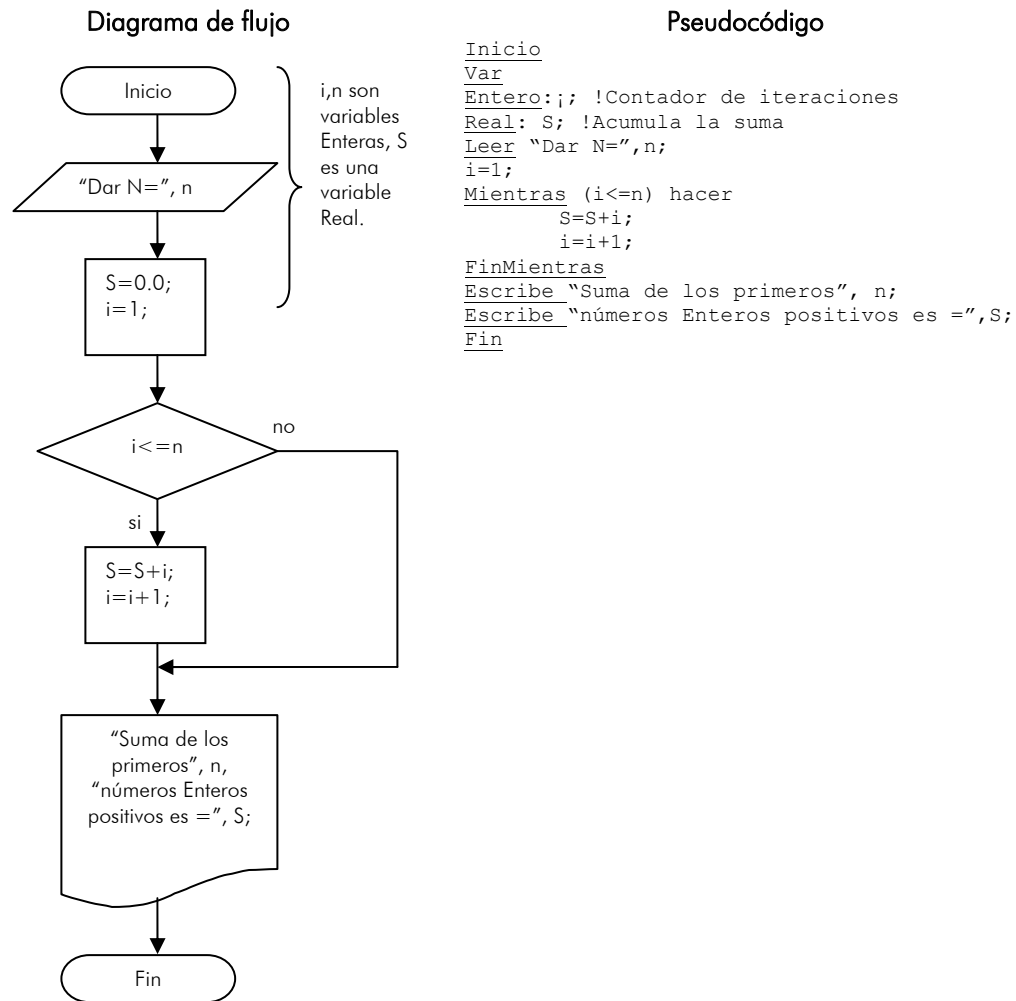
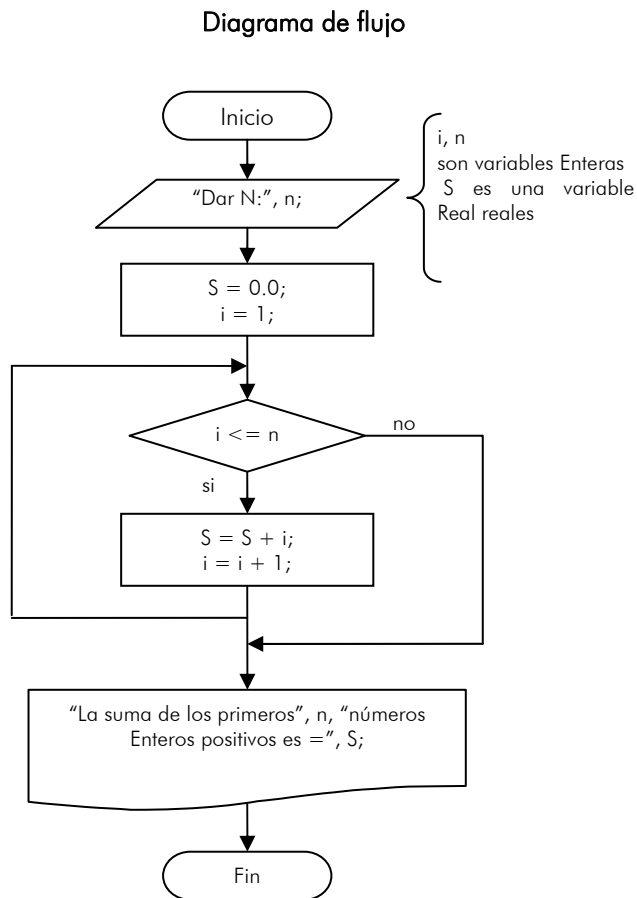


Figura 3.32. Diagrama de flujo del ejemplo 5, con ciclo *while* (mientras).Solución (ciclo *do – while*):Figura 3.33. Diagrama de flujo del ejemplo 5, con ciclo *do–while* (repetir–hasta).

Solución (ciclo for):



### Pseudocódigo

!Solución:

!Algoritmo para resolver el problema 5

!con un ciclo Mientras

Inicio

Var

Entero : si !Contador de iteraciones

Real : si ! Acumula la suma

Leer : "Dar N=", n;

Para i=1 hasta n hacer

s =s +i;

FinPara

Escribe "La suma de los primeros "n;

Escribe "números Enteros positivos es =", s;

Fin

Figura 3.34. Diagrama de flujo del ejemplo 5, con for (para).

**Ejemplo 6.** Realice el algoritmo en diagrama de flujo y pseudocódigo que calcule el aumento de sueldo para un grupo de empleados de una empresa, teniendo en cuenta el siguiente criterio:

Si el sueldo es inferior a \$1,000: Aumento 15%

Si el sueldo es mayor o igual a \$1,000: Aumento 12%

Imprima el nombre del trabajador, su sueldo sin aumento, su sueldo nuevo con aumento y el total de la nómina de la empresa, considerando este nuevo aumento.

(Este ejemplo hazlo de tarea).

# CAPÍTULO 4

## Variables con subíndice o arreglos

---

Frecuentemente se requiere trabajar con colecciones de datos del mismo tipo y guardarlas en una variable con nombre o identificador para poder manipularlas; para estos casos, los lenguajes de programación tienen lo que se conoce como estructuras de datos llamadas arreglos.

### 4.1 Definición

Los *Arreglos* (*Array* en inglés) son variables indexadas o estructuras de datos homogéneas en el sentido de que todos los elementos que la componen son del mismo tipo de dato; y se almacenan en posiciones consecutivas en la memoria principal de la computadora.

Todos estos elementos tienen un mismo nombre o identificador y lo que los diferencia el uno del otro es el subíndice o posición que ocupan en el arreglo.

- **Propiedades básicas de un arreglo**

Los datos individuales y homogéneos del arreglo se llaman elementos.

Todos los elementos son del mismo tipo de dato.

Los arreglos según el número de índices que tienen se clasifican en:

- Arreglos unidimensionales (vectores), con un solo índice.
- Arreglos bidimensionales (tablas o matrices), con dos índices.
- Arreglos multidimensionales, con tres o más dimensiones (poliedros), con más de dos índices.

### 4.2 Arreglos unidimensionales

Los arreglos unidimensionales o de un solo índice tienen que declararse dando su nombre o identificador, su tipo y sus límites inferior y superior. Normalmente en la mayoría de los lenguajes (como el C/C++, Java, PHP, JavaScript, etcétera). El límite inferior siempre es 0 y  $N - 1$  el límite superior.

Por ejemplo, si tenemos un arreglo unidimensional de elementos de tipo carácter como el siguiente:

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| 0   | 1   | 2   | 3   | 4   |
| 'a' | 'b' | 'c' | 'd' | 'e' |

Podemos ver claramente que el número de elementos es igual a 5, que el índice sólo puede tomar los siguientes valores: 0 1 2 3 4, además que el elemento 'e' es el elemento 5 y que su índice es igual a  $5 - 1 = 4$ , esto es, el límite superior del arreglo; a su vez, el primer elemento 'a' ocupa la localidad señalada por el índice 0 y es el límite inferior del arreglo.

En algunos otros lenguajes (como el Pascal, Modula, Delphi), se pueden crear tipos de *array* y luego utilizar ese tipo para definir las variables.

#### Tipo

**array**[<liminf>...<limsup>] **de** <tipo\_base> : <nombre\_del\_tipo>

#### Var

<nombre\_del\_tipo>: <nombre\_del\_vector>

Para el ejemplo:

#### Tipo

**array**[0..3] **de** **Real** : salarios;

#### Var

salarios : sueldo;

○ también:

#### Var

**Real** : sueldo[3];

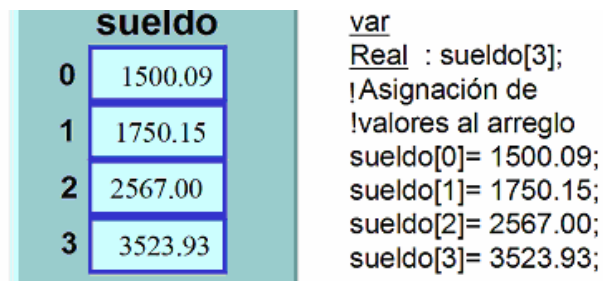


Figura 4.1. Ejemplo de arreglos unidimensionales para calcular el promedio de sueldos.

Ejemplo: algoritmo que suma el valor de los sueldos y obtiene su promedio, (ver figura 4.1).

Solución:

```

Inicio
! Programa que suma los sueldos y su promedio.
var
  Real : sueldo[3];
  Real : suma, promedio;

```

```

Entero : cont;
!Asignación de valores al arreglo
sueldo[0]= 1500.09;
sueldo[1]= 1750.15;
sueldo[2]= 2567.00;
sueldo[3]= 3523.93;
cont = 0;
suma = 0;
promedio = 0;
Mientras (cont <= 3) hacer
    suma = suma + sueldo[cont];
    cont = cont + 1;
FinMientras;
promedio = suma / cont;
Escribe "Suma de sueldos = ", suma;
Escribe "Promedio de sueldos = ", promedio;
Fin.

```

### 4.3 Arreglos bidimensionales

Los arreglos bidimensionales o de dos índices tienen que declararse dando su nombre o identificador, su tipo y sus límites inferior y superior, para cada dimensión. También generalmente, su límite inferior empieza desde 0 y el límite superior en  $N - 1$ .

#### Tipo

**array**[<linf>..de <tipo\_base>: <nombre\_del\_tipo>

#### Var

<nombre\_del\_tipo> : <nombre\_de\_la\_matriz>

Para el ejemplo:

#### Tipo

**array** [0..3,0..3] **de** **Entero** : demos;

#### Var

**demos** : **votos**;

O también:

#### Var

**Entero** **votos**[3,3];

Arreglos bidimensionales (tablas o matrices)

| votos |     |     |     |     |
|-------|-----|-----|-----|-----|
|       | 0   | 1   | 2   | 3   |
| 0     | 789 | 425 | 567 | 355 |
| 1     | 734 | 765 | 733 | 543 |
| 2     | 567 | 354 | 234 | 435 |
| 3     | 454 | 546 | 345 | 523 |

#### Var

```

Entero : votos[3,3];
votos[0,0]=789;
votos[0,1]=425;
votos[0,2]=567;
votos[0,3]=355;
votos[1,0]=734;
votos[1,1]=765;
.....

```

Figura 4.2. Ejemplo de arreglo bidimensional para calcular la suma de votos electorales.

Ejemplo: algoritmo que suma el total de los votos electorales, vea la figura 4.2, para cada uno de los de cuatro partidos diferentes en cuatro estados diferentes, con base en la siguiente tabla, (ver *tabla 4.1*).

**Tabla 4.1. Votos electorales por partido y estado, para el ejemplo del arreglo bidimensional**

| VOTOS    |          |        |                  |                  |
|----------|----------|--------|------------------|------------------|
| PARTIDO  | VERACRUZ | PUEBLA | DISTRITO FEDERAL | ESTADO DE MÉXICO |
| RIP      | 789      | 425    | 576              | 355              |
| BOLILLO  | 734      | 765    | 733              | 543              |
| SOLECITO | 567      | 354    | 234              | 435              |
| PAJARITO | 454      | 546    | 345              | 523              |

Solución:

La solución está en pseudocódigo, en diagrama de flujo, (vea *las figuras 4.3 y 4.4*) y finalmente, también se tiene la solución escrita en código fuente con el lenguaje de programación Java.

```

! Pseudocódigo del algoritmo de votos electorales
Inicio
! Programa que suma el total de los votos electorales para cada
! uno de cuatro partidos diferentes en cuatro estados diferentes.

! Declaración y asignación de valores al array bidimensional
Var
Entero: votos[3,3] = {
    {789, 425, 576, 355},
    {734, 765, 733, 543},
    {567, 354, 234, 435},
    {454, 546, 345, 523}
} ;

! Declaración y asignación de valores a un arreglo unidimensional
! de cadenas
Cadena nombrePartido[3] = {"RIP" , "BOLILLO" , "SOLECITO" , "PAJARITO"};

Entero sumaPartido[3];          ! variable para acumular la suma de
                                ! votos por cada partido
Entero i, j;    ! variables índices, i para filas-partidos,
                ! j para columnas - estados

i = 0;  ! inicializamos contador de filas - partidos
j = 0;  ! inicializamos contador de columnas - estados

Mientras (i <= 3) hacer
```

```

sumaPartido[i] = 0; ! inicializamos con 0 las sumas de
                    ! votos de cada partido

Mientras (j <= 3) hacer
    sumaPartido[i] = sumaPartido[ i ] + voto[i,j];
    j = j + 1;      ! incrementamos contador de
                    ! columnas -estados
FinMientras; ! fin del ciclo interno con j como contador

i = i + 1; ! incrementamos contador de
           ! filas - partidos
j = 0;      ! reinicializamos a cero la variable
           ! contador j del ciclo interno
FinMientras; ! fin del ciclo externo con i como contador

! a continuación escribimos los resultados de la suma de votos
! electorales por partido

i = 0; ! volvemos a inicializar el contador para filas

Mientras (i <= 3) hacer
    Escribe "La suma de votos electorales del partido " +
            nombrePartido[i] + " es igual a + sumaPartido[i];
    i = i + 1; ! incrementamos contador de filas - partidos
FinMientras; ! fin del ciclo i como contador

Fin

```

Se puede ver que declaramos en el pseudocódigo anterior un arreglo bidimensional (de dos índices) llamado votos[3,3]. Entre los corchetes indicamos el límite superior tanto de filas como de columnas, así el arreglo tendrá de 0 a 3 filas (o sea 4 filas) y de 0 a 3 columnas (esto es 4 columnas). Luego asignamos los valores enteros al arreglo utilizando llaves para delimitar las filas, y comas para separar los elementos de cada columna del arreglo, así por ejemplo, los valores asignados a las variables son los siguientes: votos[1,2] corresponderá al valor 733, votos[3,3] será 523 y votos[0,0] será 789.

```

Entero: votos[3,3] = {
                        {789, 425, 576, 355},
                        {734, 765, 733, 543},
                        {567, 354, 234, 435},
                        {454, 546, 345, 523}
                    };

```

Se puede observar que tenemos 4 filas delimitadas con { }, y cada elemento separado de otro elemento con una coma. Finalmente ponemos punto y coma (;) para terminar la sentencia de declaración y asignación de valores al arreglo bidimensional.

También tenemos una declaración de un arreglo de elementos de tipo cadena o string:

```

Cadena nombrePartido[ 3 ] = { "RIP", "BOLILLO", "SOLECITO", "PAJARITO" };

```

Aquí como el arreglo es unidimensional, sólo tendrá un índice que puede tomar valores del 0 al 3, así por ejemplo, el valor "SOLECITO" corresponde al índice 2 y para hacer referencia a él usamos ese valor del índice, así: nombrePartido[2].

También declaramos una variable *array* unidimensional de tipo entero llamada sumaPartido para acumular la suma de cada una de las filas, desde la fila 0 a la fila 4.

**Entero** sumaPartido[ 3 ]; !variable para acumular la suma de votos por partido.

Luego utilizamos dos ciclos *while* (o mientras), para manejar cada uno de los índices, donde el ciclo externo maneja la variable *i* para el índice de las filas y el ciclo interno la variable *j* para el índice de las columnas, así:

```
Mientras (i <= 3) hacer
    sumaPartido[i] = 0; ! inicializamos con 0 las sumas de
                        ! votos de cada partido

    Mientras (j <= 3) hacer
        sumaPartido[i] = sumaPartido[ i ] + voto[i,j];
        j = j + 1;    ! incrementamos contador de
                    ! columnas -estados
    FinMientras; ! fin del ciclo interno con j como contador

    i = i + 1; ! incrementamos contador de
                ! filas - partidos
    j = 0;      ! reinicializamos a cero la variable
                ! contador j del ciclo interno
FinMientras;      ! fin del ciclo externo con i como contador
```

Utilizamos la fórmula:

```
sumaPartido[ i ] = sumaPartido[ i ] + voto[ i , j ];
```

Para ir acumulando la suma de cada una de las filas de votos de cada partido.

Finalmente en un solo ciclo *while* manejamos la variable índice *i* para recorrer los índices de los arreglos nombrePartido[ i ] y sumaPartido[ i ] el nombre del partido y su suma de votos electorales, así:

```
Mientras (j <= 3) hacer
    sumaPartido[i] = sumaPartido[ i ] + voto[i,j];
    j = j + 1;    ! incrementamos contador de
                ! columnas -estados
FinMientras; ! fin del ciclo interno con j como contador
```

En todos los casos, las variables índices van del 0 al 3 y antes de salir de cada ciclo se incrementan en uno.

Ahora veamos el diagrama de flujo.

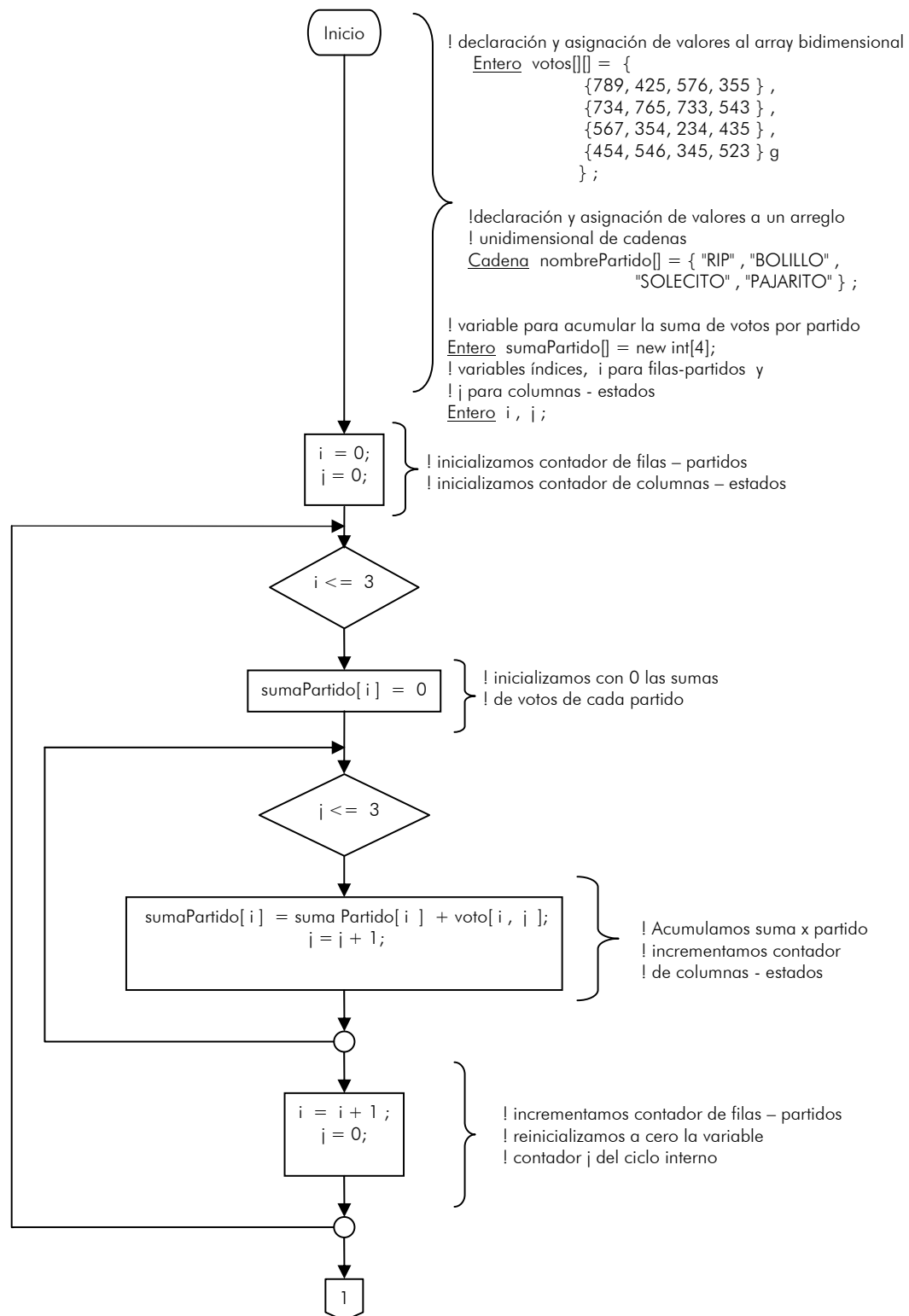


Figura 4.3. Hoja 1 del diagrama de flujo del ejemplo de votos electorales.

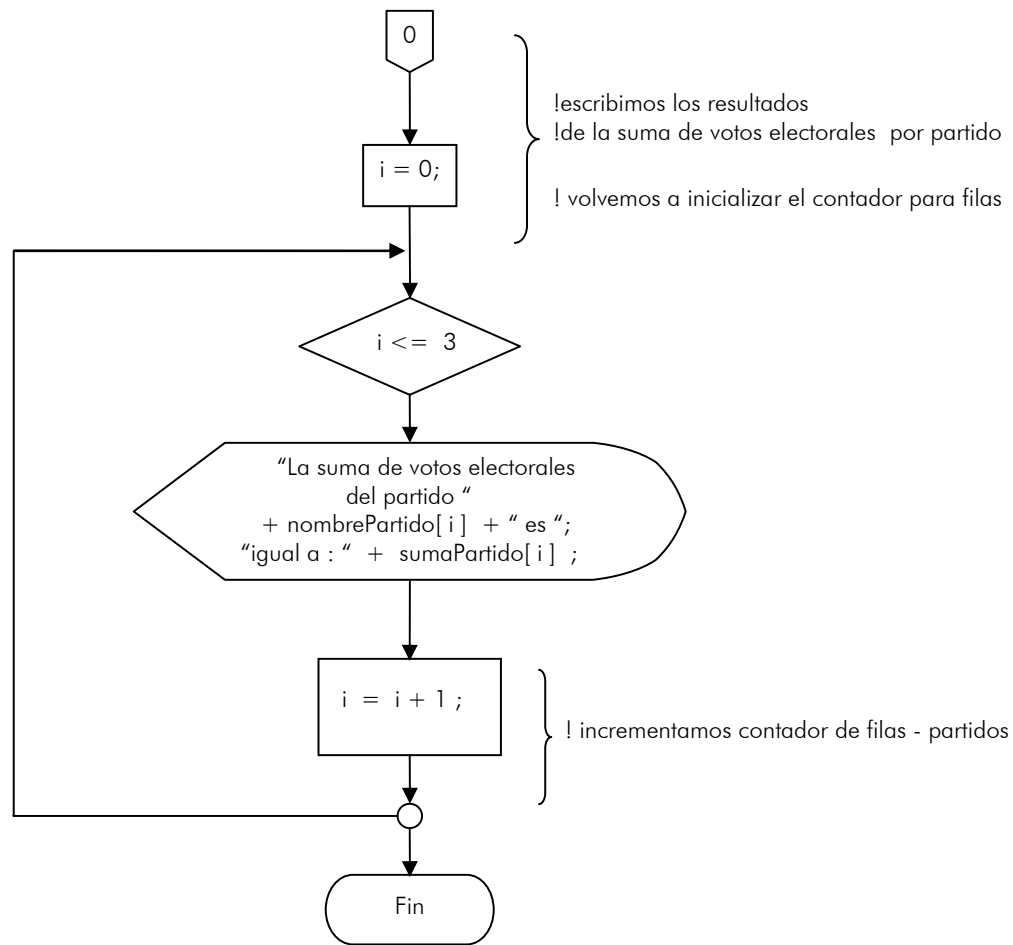


Figura 4.4. Hoja 2 del diagrama de flujo del ejemplo de votos electorales.

Ahora veamos el código fuente en lenguaje de programación Java.

```

package ejemplos;
// Programa que suma el total de los votos electorales para cada uno de
// cuatro partidos diferentes en cuatro estados diferentes.

// Declaración y asignación de valores al array bidimensional
public class Votos {
    int votos[][] = {
        {789, 425, 576, 355 },
        {734, 765, 733, 543 },
        {567, 354, 234, 435 },
        {454, 546, 345, 523 }
    };

    // Declaración y asignación de valores a un arreglo unidimensional
    // de cadenas
    String nombrePartido[] = {"RIP", "BOLILLO", "SOLECITO", "PAJARITO"};

    int sumaPartido[] = new int[4]; // variable para acumular la suma
                                    // de votos por partido
    int i, j; //variables índices, i para filas-partidos,
              // j para columnas - estados
  
```

```

public Votos() {
    i = 0;          // inicializamos contador de filas - partidos
    j = 0;          // inicializamos contador de columnas - estados

    while(i <= 3) {
        sumaPartido[i] = 0;    // inicializamos con 0 las sumas de votos
                                // de cada partido

        while (j <= 3) {
            // acumula la suma de votos por partido
            sumaPartido[i] = sumaPartido[ i ] + votos[i][j];
            // incrementamos contador de columnas - estados
            j = j + 1;
        } // fin del while o del ciclo interno con j como contador

        i = i + 1 ;    // incrementamos contador de filas - partidos
        j = 0;         // reiniciamos a cero la variable contador
                        // del ciclo interno
    } //fin del while o del ciclo externo con i como contador

    // escribimos los resultados de la suma de votos electorales
    // por partido
    i = 0; // volvemos a inicializar el contador para filas

    while (i <= 3) {
        System.out.print("La suma de votos electorales del partido "
            + nombrePartido[i] + " es igual a : " + sumaPartido[i]);
        i = i + 1;    // incrementamos contador de filas - partidos
    } // fin del while del ciclo i como contador

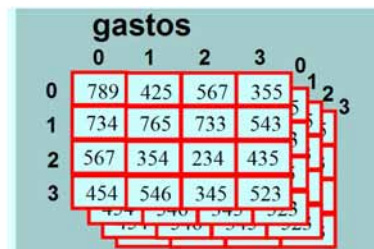
} // fin del constructor

public static void main(String[] args) {
    new Votos(); // arranca el programa creando un objeto e
                // invocando al constructor
} // fin del método main
} // fin de la clase Votos

```

#### 4.4 Arreglos multidimensionales (poliedros)

Los arreglos de multidimensionales o de más de dos índices tienen que declararse dando su nombre o identificador, su tipo de dato por almacenar, y sus límites inferior y superior, para cada dimensión. También generalmente su límite inferior empieza desde 0 y el límite superior en  $N - 1$ , (vea la figura 4.5).



El diagrama muestra un arreglo de 4x4 con el título "gastos". Las filas están indexadas de 0 a 3 y las columnas de 0 a 3. Los valores dentro de las celdas son:

|   | 0   | 1   | 2   | 3   |
|---|-----|-----|-----|-----|
| 0 | 789 | 425 | 567 | 355 |
| 1 | 734 | 765 | 733 | 543 |
| 2 | 567 | 354 | 234 | 435 |
| 3 | 454 | 546 | 345 | 523 |

Figura 4.5. Arreglos de más de dos dimensiones o poliedros.

Los arreglos multidimensionales rara vez son utilizados, pero son muy útiles en el procesamiento de datos numéricos.

# CAPÍTULO 5

## Funciones

Una función es un bloque de código llamado subprograma que resuelve una parte del problema. Es un proceso con cierto grado de independencia del módulo o función principal (programa principal).

En los lenguajes de programación de computadoras, las funciones son sinónimo de:

- subrutinas (Basic, FORTRAN).
- procedimientos, funciones (Pascal, Delphi, Ada, etcétera).
- funciones, métodos (Lenguaje C, C++, Java, C#, etcétera).

Las funciones nacen ahí donde se requiera reducir la complejidad del problema “divide y vencerás”, (esta frase se le atribuye al emperador romano Julio César cuando venció a los Celtas dividiendo sus tribus para que pelearan entre si, y ya debilitadas vencerlas con sus legiones, esto fue hace casi 2000 años); en subproblemas más sencillos cuya solución total del problema sea el resultado de la suma de soluciones parciales de cada función, (vea la figura 5.1).

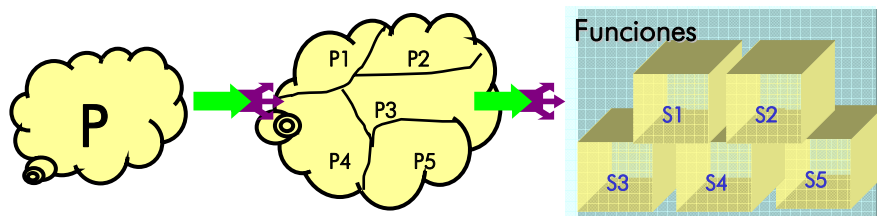


Figura 5.1. Reducir la complejidad del problema “divide y vencerás”.

### Características deseables en una función

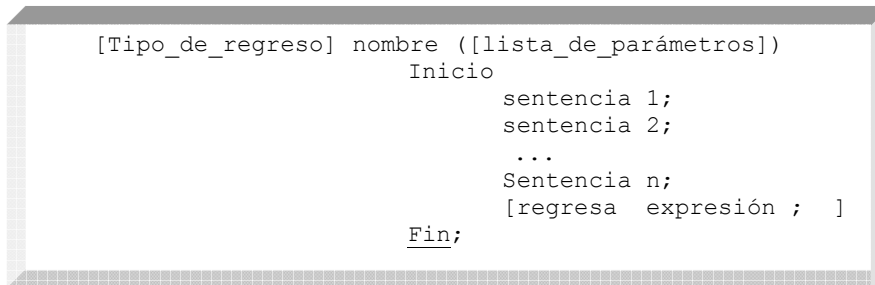
- a) Deberán estar jerarquizados.
- b) Deberán ser pequeños y sencillos.
- c) Deberán “esconder” los detalles poco importantes a módulos superiores en jerarquía.
- d) Deberán, a su vez, usar tantos módulos de más baja jerarquía como sea necesario para cumplir con el punto B.

- e) Deberán usar estructuras de datos y control adecuadas para cumplir con el punto B.
- f) Deberán ser legibles; esto es que no sólo su autor sea capaz de entenderlos, sino cualquiera que tenga acceso a ellos y a un conocimiento elemental de programación.

## 5.1 Concepto de función en programación

Una función es un conjunto de sentencias encargadas de resolver una tarea o rutina que se repite frecuentemente en la solución de un problema, y todas las funciones deben tener las siguientes características, (vea figura 5.2).

- Las funciones tienen un *nombre* (identificador).
- Pueden tener o no una *lista de parámetros*.
- Tienen un *cuerpo* (conjunto de sentencias).
- Pueden o no *regresar* un valor de algún tipo de dato primitivo o definido por el usuario.

El diagrama muestra la estructura general de una función en programación. Está representada como un bloque rectangular con un efecto de sombra 3D. El contenido del bloque es el siguiente código: 

```
[Tipo_de_regreso] nombre ([lista_de_parámetros])  
    Inicio  
        sentencia 1;  
        sentencia 2;  
        ...  
        Sentencia n;  
        [regresa expresión ; ]  
    Fin;
```

Figura 5.2. Características de una función.

## 5.2 Llamada o invocación a una función

Una función es invocada o llamada mediante su nombre y su lista de argumentos (si los tiene), en cualquier punto del programa; y en una sentencia puede haber una invocación a una o más funciones.

$$\text{valor} = \text{nombre} ( [ \text{lista\_de\_argumentos} ] )$$

No todas las funciones regresan un valor, esto depende de las necesidades del programa y si tienen definido en el cuerpo de la función una sentencia de retorno o regreso (return en inglés) de algún tipo de valor.

La lista de argumentos debe coincidir en número y tipo de dato asociado, con los parámetros declarados previamente en la cabecera de la función que es invocada.

## 5.3 Parámetros

Los parámetros son un mecanismo de intercambio de información entre las funciones de un programa. Se relacionan de acuerdo con su posición y el tipo de dato asociado. Existen dos tipos de parámetros, por valor y por referencia.

### 5.3.1 Parámetros por valor

Estos parámetros son constantes. Son copias de la variable que se le da a la función, estos valores no los puede modificar la función; aunque la copia puede cambiar de valor dentro de la función.

### 5.3.2 Parámetros por referencia

Parámetros que son variables. Son la referencia o dirección de memoria de las variables que se le dan a la función; con esto se modifican directamente los valores como parte de la referencia, es decir, la función podrá modificar los valores que se le dieron como argumentos en la invocación.

## 5.4 Valor de retorno

Un valor de retorno es el resultado de la evaluación de una expresión realizado dentro del bloque de sentencias de la función y que es devuelto al punto del programa de donde la función fue invocada o llamada.

Este es también un mecanismo adicional al de los parámetros de intercambio de información entre las funciones de un programa.

Ejemplo:

Definición de una función en pseudocódigo que intercambia los valores de las variables X, Y que tiene como parámetros. Por referencia y que regresa su suma o el resultado de la expresión  $X + Y$ , (vea figura 5.3).

```
Real  intercambia (Var Entero X, Y)
      Var Entero  Temporal;

      Temporal = X;
      X = Y;
      Y = Temporal;

      regresa X + Y;

Fin; ! intercambia
```

Figura 5.3. Pseudocódigo de la función que intercambia valores enteros y devuelve su suma.

A continuación tenemos ahora el programa o función principal en pseudocódigo, que invoca o llama a la función **intercambia** y escribe los resultados por pantalla, (vea figura 5.4).

## Tomando en cuenta la definición anterior de intercambia

```

Inicio      ! Ejemplo de uso de un módulo
  Var      Entero    A, B, C;

  A = 11;
  B  = 16;
  Escribe  "a= ", A , "b= ", B; !<== ( A= 11  B=16 )
  C = intercambia(A, B); ! Se invoca con su identificador.
                           ! A continuación se especifica la
                           ! lista de argumentos (si los hay)
  Escribe  "a = ", A , " b = ", B , " suma a + b = ", C ;
  Fin      ! Programa principal

```

Figura 5.4. Pseudocódigo donde se invoca a la función intercambia(A,B).

Ahora este mismo ejemplo con diagrama de flujo, observe que los *parámetros por referencia* X, Y tienen flechas rojas de entrada y salida para denotar que son variables, y hay una sentencia que *regresa* dentro del bloque de la función, la cual devuelve el resultado de sumar  $X + Y$ , por eso, también hay otra flecha roja que denota el tipo de valor de retorno (vea figura 5.5).

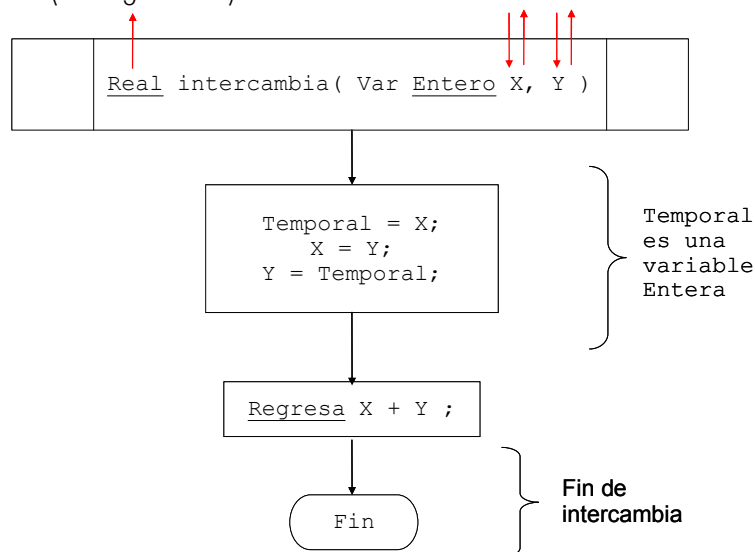


Figura 5.5. Diagrama de flujo de la función que intercambia valores enteros y devuelve su suma.

A continuación la función principal o programa principal que *invoca* a la función “intercambia” en diagrama de flujo. Vea cómo se invoca la función “intercambia”, la cual envía dos *argumentos* A, B y cómo recibe el *valor de retorno* en la variable C (vea figura 5.6).

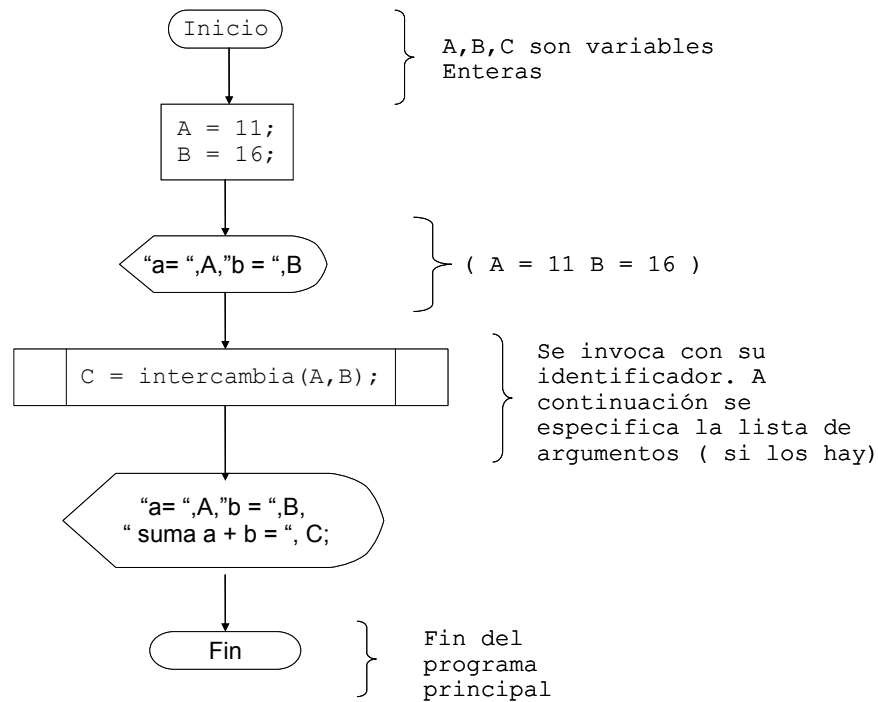


Figura 5.6. Diagrama de flujo donde se invoca a la función intercambia(A,B).

### Ejemplo de refinación sucesiva con funciones

Haga un programa que calcule e imprima la suma y el promedio de una lista de números reales.

#### Requerimientos

##### Datos de entrada

NumElem : Entero; ! Número de elementos a sumar

Elemento : Real; ! Cada elemento o número real

##### Datos de salida

Suma : Real; ! Sumatoria de los elementos

Promedio : Real; ! Promedio o media aritmética

#### Fórmulas importantes

Promedio = suma de elementos / número de elementos

#### Diseño

##### ! Primer refinamiento

##### Inicio

! Algoritmo que calcula la suma y promedio de un conjunto de elementos.

! 1. Leer el número de elementos.

! 2. Calcula la suma de los elementos.

! 3. Calcula el promedio de los elementos.

! 4. Imprime la suma y el promedio.

**Fin**

Aquí ya subdividimos el problema en subproblemas y podemos hacer un diagrama de bloques para visualizarlo gráficamente (vea figura 5.7).

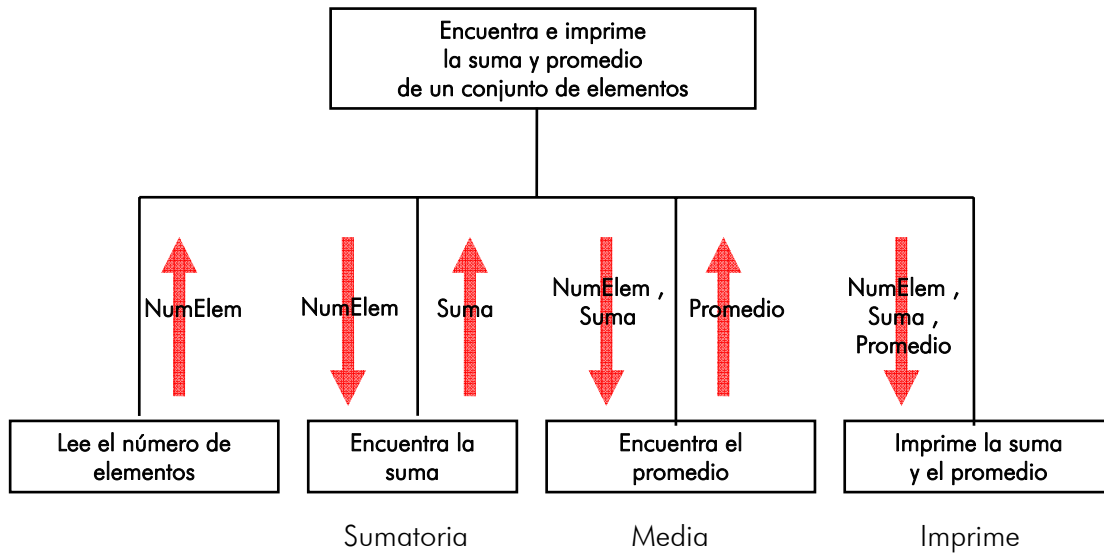


Figura 5.7. Subdividimos el problema en subproblemas.

### Implementación de la función principal

#### **Inicio! Codificación módulo principal**

**! Algoritmo que calcula la suma y promedio de un conjunto de elementos**

```

Var   Entero  : NumElem,   ! entrada - No de elementos
      Real    : Suma,      ! salida - acumulador de la suma
      Promedio;           ! salida - promedio de los elementos
Leer  "Dar número de elementos = ", NumElem;
! Calcula la suma de los elementos.
Sumatoria( NumElem, Suma); ! invoca función sumatoria
! Calcula el promedio de los elementos.
Promedio = Media( NumElem, Suma); ! Invoca función de Media
! Imprime la suma y el promedio.
Imprime( NumElem, Suma, Promedio);

```

**Fin**

### Función Sumatoria

#### **Requerimientos**

#### **Entradas del módulo**

NumElem : Entero ; ! Número de elementos

#### **Salidas del módulo**

Suma : Real; ! Sumatoria de los elementos

#### **Variables Locales**

Item : Real; ! Cada elemento de dato

Cont : Entero; ! Contador de los elementos sumados

### Algoritmo para función Sumatoria

**Inicio**

```

! 1. Inicializa la Suma = 0;
! 2. Para cada valor de cont de 1 hasta NumElem hacer
    ! 2.1 Lee el siguiente elemento.
    ! 2.2 Acumúlalo a la suma.

```

**Fin**

!Implementación de la función Sumatoria

```
Sumatoria( Entero NumElem, Var Real Suma )
```

```
Var      ! Variables locales
```

```
    Entero : cont; ! Contador de elementos
```

```
    Real   : item; ! Siguiendo elemento a ser sumado
```

**Inicio**

```
Suma = 0;
```

```
Para cont = 1 hasta NumElem hacer
```

```
    Lee "Siguiendo elemento a ser sumado=", item;
```

```
    Suma = Suma + item;
```

**FinPara;**

```
Fin ! De Sumatoria
```

**Requerimientos para la función Imprime*****Entradas del módulo***

```
NumElem : Entero ; ! Número de elementos
```

```
Suma      : Real;           ! Sumatoria de todos los elementos
```

```
Promedio : Real;           ! Promedio de todos los elementos
```

**Algoritmo para la función Imprime****Inicio**

```
3. Si el NumElem es positivo Entonces
```

```
    1.1 Despliega la Suma y el Promedio de los elementos
```

**Fin****Implementación de la función Imprime**

```
Imprime ( Entero NumElem, Real Suma, Promedio )
```

**Inicio**

```
Si ( NumElem > 0 ) Entonces
```

```
    Escribe "La suma es = ", Suma;
```

```
    Escribe "El promedio es = ", Promedio;
```

**Sino**

```
    Escribe "La suma y el promedio no están definidos ";
```

**FinSi;**

```
Fin ! De Imprime
```

**Requerimientos para la función Media*****Entradas del módulo***

```
NumElem : Entero ; ! Número de elementos
```

```
Suma      : Real;           ! Sumatoria de todos los elementos
```

***Salidas del módulo***

```
Promedio : Real;           ! Promedio de todos los elementos
```

**Algoritmo para la función Media**

**Inicio**

4. Si el NumElem es positivo Entonces

1.1 Calcula Promedio dividiendo Suma entre NumElem

Sino

1.2 Asigna a Promedio el valor de 0.

**Fin****Implementación de la función Media**

**Real** Media( **Entero** NumElem , **Real** Suma )

**Inicio**

**Si** ( NumElem > 0 ) **Entonces**

Regresa ( Suma / NumElem );

**Sino**

Regresa 0.0;

**FinSi**;

**Fin**; **! De Media**

Ahora en diagrama de flujo de la función o módulo principal, (ver figura 5.8).

**Del algoritmo del módulo principal del programa que calcula la suma y promedio de un conjunto de elementos (1ª. parte)**

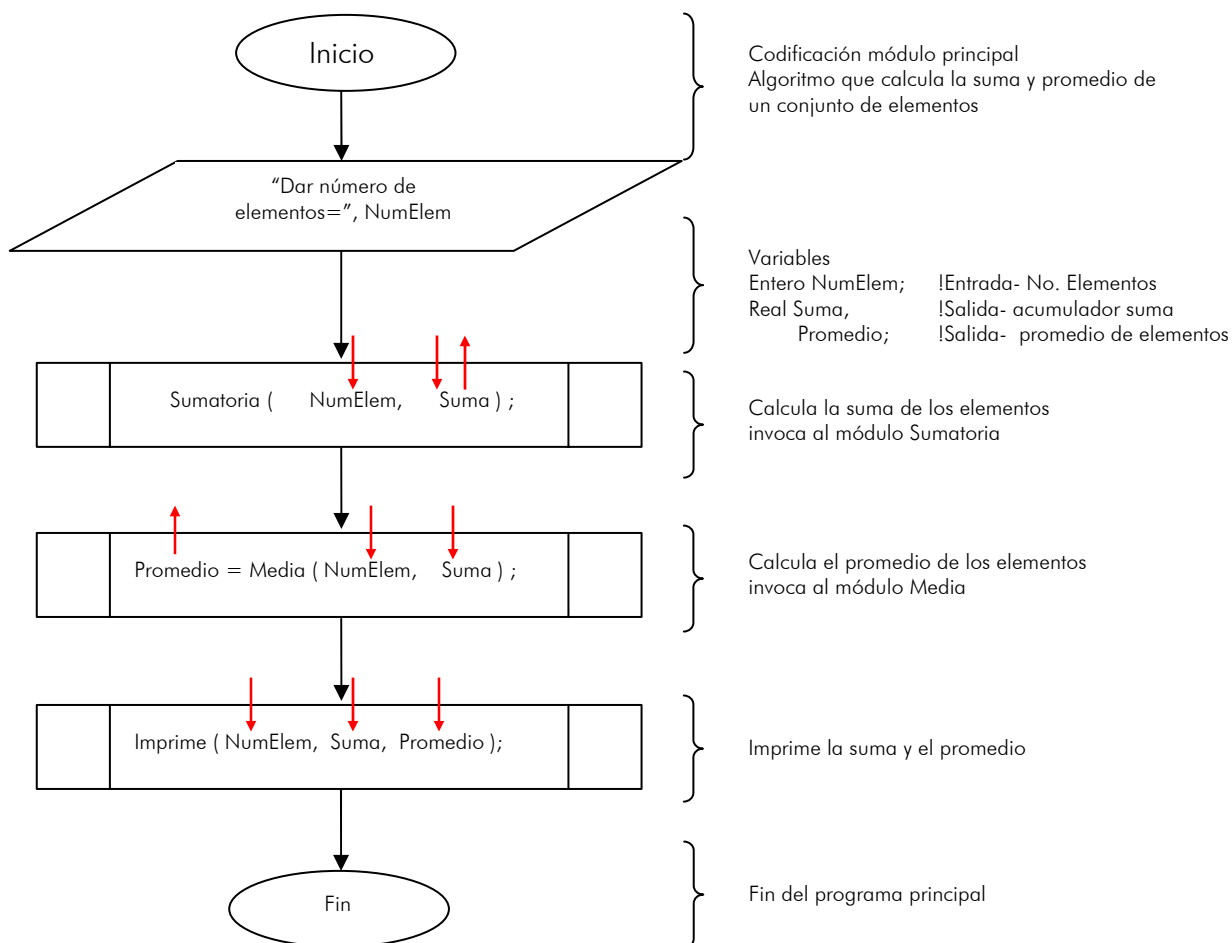


Figura 5.8. Diagrama de flujo del modulo principal o función principal.

Diagrama de flujo de la función Sumatoria, (ver figura 5.9).

**Del algoritmo del módulo principal del programa que calcula la suma y promedio de un conjunto de elementos (2ª. parte)**

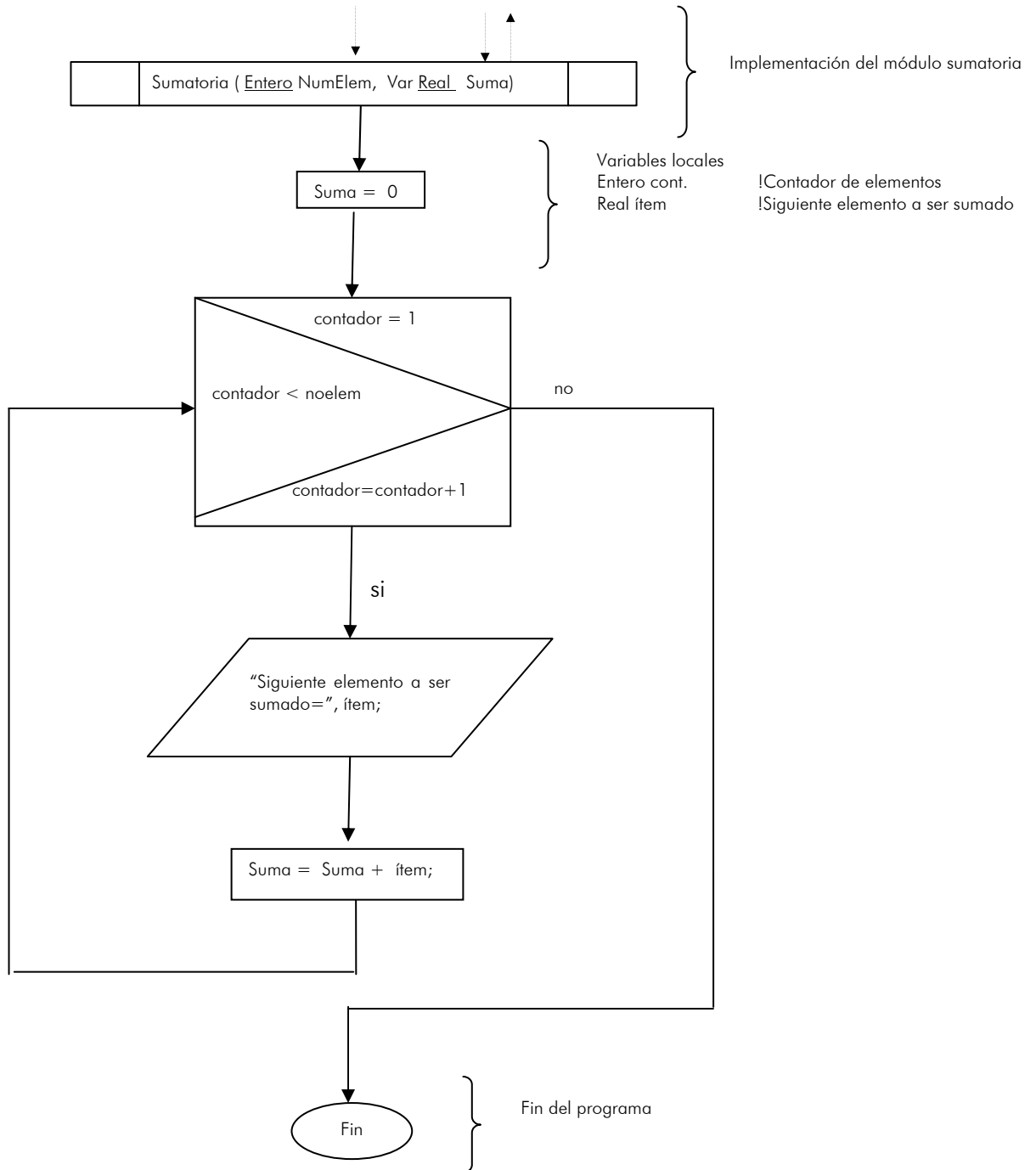


Figura 5.9. Diagrama de flujo de la función Sumatoria.

Es recomendable que se haga como un ejercicio el diagrama de flujo de las funciones Media e Imprime.

# CAPÍTULO 6

## Elementos básicos de la programación orientada a objetos

La programación orientada a objetos es una técnica para el modelado de sistemas, esto es, para el análisis, diseño e implementación de sistemas y programas de cómputo.

Se modela el sistema a través de un número de objetos relacionados que interactúan entre sí, y es similar a la manera en que la gente ve el mundo en que vive y su medio ambiente. Es una forma de pensar y concebir el mundo. Para entender esta manera de hacer programas, debemos entender primero qué es un objeto.

### 6.1 Concepto de objeto

Los objetos pueden ser una oración, una cuenta en un banco, un número, un auto, una persona, etc., (vea figura 6.1).

Los objetos son:

- Cosas.
- Reales o imaginarias.
- Simples o complejas.

#### Identificando a los objetos



Figura 6.1. Ejemplos de objetos.

## 6.2 Anatomía de un objeto

Todos los objetos están constituidos o tienen *atributos* y *comportamientos*, (vea figura 6.2).

- Los *atributos* son las características de los objetos.
- Los *comportamientos* son lo que un objeto puede hacer.

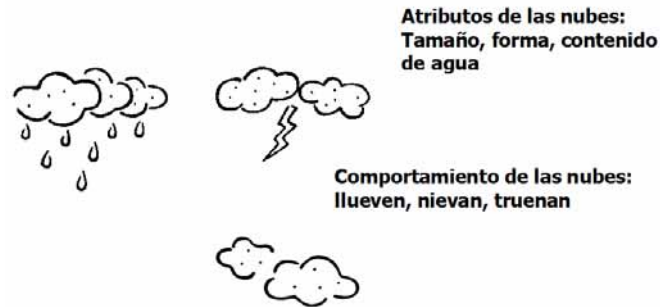


Figura 6.2. Identificando los atributos y comportamientos de los objetos.

Además de los *atributos* y *comportamientos*, un objeto tiene una *identidad*.

- La *identidad* de un objeto es lo que lo diferencia y distingue de otros objetos similares.

Luego entonces, podemos resumir que todo objeto está compuesto de tres elementos:

$$\text{Objeto} = \text{atributos} + \text{comportamientos} + \text{identidad}$$

Los criterios que debemos seguir para identificar a los objetos son:

- Su relevancia para el dominio del problema.
  - ¿Existe dentro de las fronteras del enunciado del problema?
  - ¿Es requerido para alcanzar completamente las responsabilidades del sistema?
  - ¿Los objetos son características de otros objetos?
- Necesidad de su existencia independiente.
  - Para que un objeto no sea una característica de otro objeto, necesariamente debe existir con independencia.
  - Por ejemplo: considere una computadora que tiene Pentium III de procesador. En una perspectiva (para un vendedor), el procesador es una característica o atributo de la computadora. Pero si usted fuera diseñador de componentes de computadoras o PC en una fábrica, el procesador por sí mismo es un objeto independiente.
- Que tenga atributos y comportamientos.
  - Un objeto debe tener atributos y operaciones.
  - Si no los tiene, es probablemente un atributo u operación de otro objeto.

### **6.3 Beneficios de la programación orientada a objetos**

La programación orientada a objetos (POO) es una manera más natural de pensar, concebir y programar sistemas, debido a que en la vida cotidiana, las personas están relacionándose con objetos, así por ejemplo, un objeto Persona Cliente de un Banco, está relacionada con un objeto Cuenta Bancaria. Ambos objetos tienen sus atributos y comportamientos, así entonces una Persona Cliente de un Banco, tiene los atributos; nombre, domicilio, fecha de nacimiento y los comportamientos; esperar, trabajar, dormir, etc., y el objeto Cuenta Bancaria tiene los atributos; tipo de cuenta (crédito, débito, cheques), número de cuenta, etc., y los comportamientos; transferir fondos, abrir cuenta, verificar el saldo, etcétera.

De esta manera, el equipo de desarrollo de sistemas ya no tiene que pensar en términos de la arquitectura de la computadora para desarrollar sistemas (direcciones de memoria hexadecimales, celdas de memoria RAM, sectores de disco, estructuras de datos y algoritmos, etcétera), sino en objetos y clases de objetos, sus relaciones e interacciones para abordar la complejidad.

A propósito de la complejidad, los sistemas cada vez son más grandes y complejos, debido a las crecientes necesidades de la industria y el comercio. Los usuarios y los clientes demandan soluciones cada vez más eficientes, fiables y complejas. Aquí la POO entra en escena, debido a que es posible abordar con muchísima más facilidad la complejidad de la realidad a la que tienen que enfrentarse los clientes y usuarios de la información.

La POO aborda la complejidad mediante la abstracción, el encapsulamiento, la herencia, el polimorfismo, etc., que son propiedades principales de este tipo de programación, y que vamos a ver en capítulos posteriores de este manual. Estas propiedades de la POO garantizan la producción de componentes de software reutilizables e intercambiables (abstracción y encapsulamiento), y la estrategia de reutilización del código (herencia y polimorfismo), así el programador utiliza lo que ya hicieron otros programadores antes que él y le saca provecho en nuevas implementaciones. Si hay algo que cambiar para adaptarse a las nuevas necesidades del sistema, lo puede modificar, gracias al polimorfismo.

Cuando son aplicaciones sencillas, y no muy grandes, se recomienda la programación funcional o procedural. Ahora cuando los sistemas crecen y se hacen más complejos, se recomienda la POO debido a que es más fácil actualizarlos, mantenerlos y ampliarlos ya que este tipo de programación nos permite adaptarnos a la complejidad de las necesidades de los usuarios.

# CAPÍTULO 7

## Clases y objetos

---

Los objetos usualmente no nacen solos, sino que existen dentro de una colección, conjunto o comunidad de objetos similares. En esta sección veremos cómo es que los objetos forman parte de una clase.

### 7.1 Definición de una clase

*Clase*: es una descripción de atributos y comportamientos de un conjunto de objetos similares. Los objetos de una clase o tipo determinado comparten los mismos comportamientos y atributos. Las clases actúan en forma muy parecida a una plantilla o molde para galletas en el sentido de que una clase se utiliza para crear o *instanciar* objetos, (ver figura 7.1).

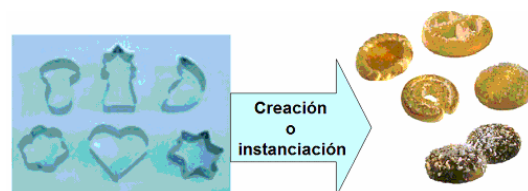


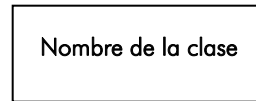
Figura 7.1. Creación o instanciación de objetos.

Así tenemos entonces que un objeto es una instancia: una específica galleta, hoja, balón, carro o moneda, (ver figura 7.2).



Figura 7.2. Clases de objetos.

En UML (Unified Modeling Language) que es un lenguaje gráfico, unificado de varios otros lenguajes gráficos, para el modelado de aplicaciones orientadas a objetos, el símbolo gráfico para representar una clase es un rectángulo, con el nombre de la clase, (ver figura 7.3).



Por ejemplo:

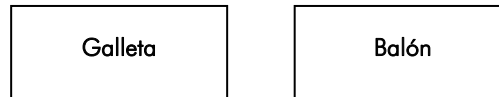
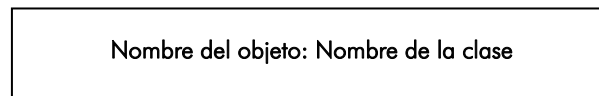


Figura 7.3. El símbolo gráfico para representar una clase es un rectángulo en UML.

Los objetos se representan en UML con un rectángulo con nombre del objeto y dos puntos para separarlo del nombre de la clase, (ver figura 7.4).



Por ejemplo:

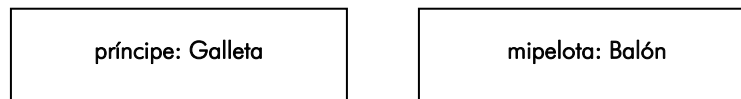


Figura 7.4. Símbolo gráfico para representar objetos en UML.

## 7.2 Miembros de una clase

En una clase, los atributos y comportamientos de un objeto son sus *miembros*. Estos miembros del objeto son similares a los de otros objetos de la misma clase a la que pertenezcan y se declaran en la definición de la clase.

### 7.2.1 Propiedades

Los atributos se convierten en *variables miembro* de la clase (también se les conoce en algunos ambientes como *campos*). Los atributos se representan en UML, dentro del rectángulo de la clase abajo del nombre de la clase, (ver figura 7.5).

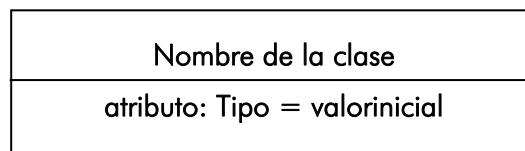


Figura 7.5. Las propiedades o atributos van abajo del nombre de la clase.

La variable miembro de una clase es muy similar al concepto clásico de una variable de programación. Una variable miembro tiene la siguiente sintaxis.

```
[ ámbito ]   Tipo  nombreVariable = [valorinicial]
```

donde :

**ámbito** pueden ser cualquiera de los siguientes:

- public
- private
- protected
- default

**Tipo** puede ser cualquiera de los siguientes tipos primitivos de datos:

- int
- short
- byte
- long
- float
- double
- boolean
- char

ejemplos:

```
public   int  x = 20;
private  short sh = 100;
protected byte bt = 15;
long lo = 1000000;
float y = 10.3;
private  double d = 3.141659;
public   boolean b = x > y ;
char    car = '@';
```

También un tipo puede ser de una clase específica de objetos, aquí la clase se comporta como si fuera un tipo de dato abstracto, por ejemplo:

- Date
- Persona
- String
- Balon
- Galleta
- etc.

ejemplos:

```
private Date fechanac = new Date ( 01, 13, 1961 - 1900 ) ;

String nombre = new String( "Oscar Alejandro González
Bustamante" );

public Persona oscar = new Persona( nombre , fechanac );

protected Balon pelota = new Balon( " Football Socker" );

protected Galleta marinela = new Galleta();
```

Veamos ahora un ejemplo de una clase en Java llamada VariablesMiembro con las declaraciones de variables miembro anteriores. Primero su diagrama de UML (las clases

Date y String no se ven en el diagrama porque son clases estándar o API y forman parte de la biblioteca de clases de Java). La clase VariablesMiembro tiene flechas que la conectan a la clase Galleta, Balon y Persona. Esto significa que la clase VariablesMiembro tiene atributos o variables de esas clases; marinela es de la clase Galleta, pelota es de la clase Balón y oscar es de la clase Persona, (ver figura 7.6). Esta relación entre clases es conocida como *composición*.

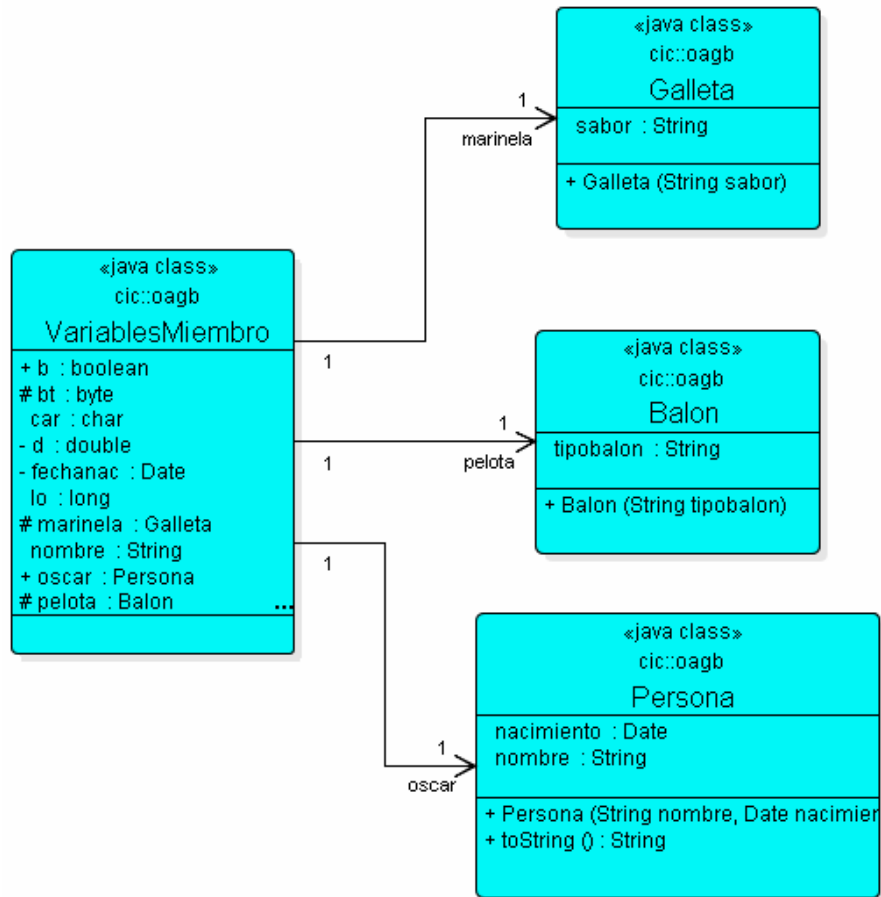


Figura 7.6. Diagrama de clases de UML.

Ahora el código en Java.

```
// Archivo : VariablesMiembro.java
package cic.oagb;
import java.util.Date;
import cic.oagb.Persona;
// los import es algo análogo a un include de Lenguaje C
// y son las referencias de donde se sacan las
// utilidades del lenguaje

import cic.oagb.Balon;
import cic.oagb.Galleta;

class VariablesMiembro
{
```

```
// Los atributos se convierten en variables miembro de
// la clase
// (También se les conoce en algunos ambientes como
// campos ).
public      int x = 20;
private    short sh = 100;
protected byte bt = 15;
long lo = 1000000;
float y = 10.3f;
private    double d = 3.141659;
public     boolean b = x > y ;
char car   = '@' ;

private    Date fechanac = new Date(1961-1900, 1, 13);
String     nombre      = new String("Oscar Alejandro González
Bustamante");
public     Persona oscar = new Persona(nombre, fechanac);
protected Balon pelota = new Balon("Football Socker");
protected Galleta marinela = new Galleta( "Cholate" );

} // fin de la clase VariablesMiembro
```

---

```
// Archivo Persona.java
package cic.oagb;
import java.util.Date;

public class Persona
{
    String nombre;    // atributo o variable miembro
    Date nacimiento; // atributo o variable miembro

    // constructor
    // Los constructores son métodos miembros de una
    // clase que tiene el mismo nombre o
    // identificador igual que el nombre de la clase
    // y tienen la tarea de construir un objeto,
    // esto es, inicializan las variables
    // miembro del objeto ( variables de instancia ),
    // en el proceso de la construcción del objeto
    //(instanciación).
    // Esto se explica mas detalladamente en la siguiente
    // sección.

    public Persona(String nombre , Date nacimiento)
    {
        this.nombre = nombre;
        this.nacimiento = nacimiento;
    } // fin del constructor

    public String toString()
    {
        return " Nombre: " + nombre + "\n Fecha de Nacimiento: " +
nacimiento ;
    } // fin del método sobrecargado toString

} // fin de la clase Persona
```

---

```
// Archivo: Galleta.java
```

```

package cic.oagb;

public class Galleta
{
    String sabor;
    public Galleta( String  sabor )
    {
        this.sabor = sabor;
    } // fin del constructor
} // fin de la clase Galleta

```

```

// Archivo: Balon.java
package cic.oagb;

public class Balon
{
    String tipobalon ; // atributo

    public Balon(String  tipobalon )
    {
        this.tipobalon = tipobalon;
    } // fin del constructor

} // fin de la clase Balon

```

### 7.2.2 Métodos

Los comportamientos se convierten en *métodos miembro* de la clase (también se les conoce en algunos ambientes como *operaciones*). Su representación en UML, es dentro del rectángulo de la clase, debajo los atributos de la clase, (ver figura 7.7).

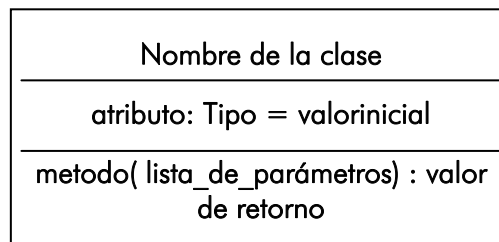


Figura 7.7. Los métodos van debajo de los atributos.

El concepto de método *miembro* de una clase es muy similar al concepto clásico de una función de programación. Un método es un conjunto de sentencias encargadas de implementar el comportamiento del objeto y todos los métodos deben tener las siguientes características:

- Los métodos tienen un **nombre** (identificador).
- Pueden tener o no una **lista\_de\_parámetros** (que son un conjunto de parejas *Tipo nombre* separados por coma).
- Tienen un **cuerpo** (conjunto de sentencias).
- Pueden o no **regresar** un valor de algún tipo de dato primitivo o definido por el usuario, por lo tanto tienen un **Tipo\_de\_regreso**.

- Tienen un **ámbito** de aplicación y pueden ser: private, public, protected, default.

```
[ ambito ] [ Tipo_de_regreso ] nombre([ lista_de_parámetros ])
{
    sentencia 1;
    sentencia 2;
    ...
    Sentencia n;
    [regresa expresión ;]
}
```

Ejemplo de un método que devuelve o regresa los valores de las variables **miembro** de la clase Persona convertidos a una cadena (String) .

```
public String toString()
{
    StringBuffer cad = new StringBuffer("Nombre: " + nombre + "\n" );
    // append es un método de concatenación o
    //unión de cadenas.
    cad.append( "Dirección: " + direccion + "\n" );
    cad.append( "Estatura: " + estatura + "\n" );
    cad.append( "Fecha de Nac.: " + fnac );

    return cad.toString() ;
} // fin de método toString()
```

### 7.2.3 Constructores y creación de objetos

Los **constructores** son métodos miembros de una clase que tiene el mismo nombre o identificador, igual que el nombre de la clase. Los métodos constructores tienen la tarea de construir un objeto, esto es, inicializan las variables miembro del objeto (variables de instancia), en el proceso de la construcción del objeto (instanciación).

Para poder instanciar un objeto utilizamos el operador **new**. Este operador se aplica cuando se invoca o llama al método constructor del objeto. Por ejemplo, si tenemos una clase llamada Persona, tendremos por lo menos un método constructor llamado Persona para construir los objetos de esta clase:

```
// constructor por omisión o default
public Persona()
{
    this.nombre = null;
    this.direccion = null;
    this.estatura = -1.0;
    this.fnac = null;
} // fin de constructor
```

Y para invocar a este método en la construcción de un objeto, utilizamos lo siguiente:

```
Persona p1;           // declaro una referencia de la
                       // clase Persona

new Persona ( ); // construyo el objeto
```

Pero a veces es necesario poder construir objetos de varias formas, p. ejemplo:

```
Persona p1 = new Persona( "Oscar González");
○ así:
Persona p1 = new Persona( "Oscar González", dp1 );
```

Donde: dp1 es la dirección de la persona p1, esto es, un objeto de la clase Dirección

```
Persona.Direccion dpl = new Persona.Direccion("Farallones No. 5",
"Acueducto de Gpe." , "Distrito Federal", " México", "07270" );
```

Esto se conoce en los lenguajes de POO como *sobrecarga de métodos*. La sobrecarga de métodos constructores me permite construir un objeto de una clase invocando a su constructor con diferente número y tipo de parámetros. Dentro de los constructores se utiliza la palabra reservada `this([ Lista_de_argumentos ])` para invocar al constructor de ese número de parámetros y armar una cadena de invocaciones.

#### 7.2.4 Acceso a propiedades y métodos

La manera de acceder a los miembros de una clase, esto es, a los valores almacenados en las variables *miembro* (propiedades) y a sus métodos *miembro* (comportamientos), es mediante el operador punto ( `.` ), conocido como el operador de acceso. Primero se pone el nombre o identificador del objeto, después el punto y luego el nombre de la variable o método.

```
objeto.variable

objeto.método([ Lista_de_argumentos ])
```

#### 7.2.5 Destructores

Los destructores son métodos que liberan los recursos ocupados por un objeto y que fueron utilizados todo el tiempo de vida del mismo. Estos recursos son por ejemplo, la cantidad de memoria principal necesaria para alojarlo, etc. En lenguajes como C++, es obligado destruir los objetos cuando dejan de ser útiles en un programa. Por otro lado, en el lenguaje Java, no es necesario destruir los objetos, debido a que el lenguaje tiene un mecanismo automático para la destrucción de los objetos, denominado Garbage Collectger (Colector de Basura), lo cual minimiza la aparición de errores y hace más fiable y robusto este lenguaje.

**Ejemplo:** ahora veamos un ejemplo que resuma todo lo anterior.

Tenemos la clase *Persona* con varios constructores. Esta clase tiene las variables privadas dirección de la clase *Direccion*, estatura de tipo *double*, *fnac* de la clase estándar de Java *Date*, y nombre de la clase *String*.

Los constructores de la clase *Persona* inicializan estas variables *miembro* privadas. El método `toString()` dentro de *Persona*, arma una cadena de los valores de las variables *miembro* y la regresa.

La clase *Persona* tiene dentro una clase interna estática llamada *Dirección*, y nos sirve para crear un objeto que maneje la dirección.

Para usar esta clase *Persona*, hacemos un programa de Java. Este programa se llama *UsaLaPersona* y es una clase que contiene método `main()`. Este método `main` es donde empiezan a ejecutarse las aplicaciones de Java cuando se corren con el intérprete.

Este programa *UsaLaPersona* construye un objeto de la clase *Persona* llamado *p1*, (ver figura 7.8), y luego presenta los datos de las variables miembro de *p1* en un *ScrollPane* (panel de desplazamiento) dentro de un cuadro de diálogo para visualizar esos datos, (ver figura 7.9).

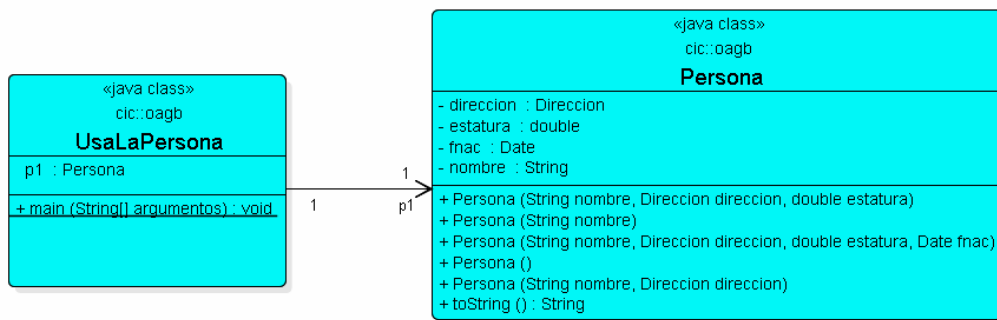


Figura 7.8. Diagrama de UML de las clases Persona y UsaLaPersona.

Se aprecia la relación de asociación entre las clases, representada con una flecha que las conecta.

A continuación se listan los códigos de las clases Persona y UsaLaPersona.

```

package cic.oagb;
import java.util.Date;
// Archivo : Persona.java

public class Persona
{
    private String nombre;
    private Direccion direccion;
    private double estatura;
    private Date fnac;

    // constructor por default
    public Persona()
    {
        this.nombre = null;
        this.direccion = null;
        this.estatura = -1.0;
        this.fnac = null;
    } // fin de constructor

    public Persona(String nombre)
    {
        this.nombre = nombre;
    } // fin de constructor

    public Persona(String nombre, Direccion direccion)
    {
        this( nombre );
        this.direccion = direccion;
    } // fin de constructor

    public Persona(String nombre, Direccion direccion, double estatura)
  
```

```

    {
        this(nombre, direccion);
        this.estatura = estatura;
    } // fin de constructor

    public Persona(String nombre, Direccion direccion, double estatura,
Date fnac)
    {
        this(nombre, direccion, estatura);
        this.fnac = fnac;
    } // fin de constructor

    // método que devuelve los valores de las variables
    // miembro de la clase convertidos a una cadena ( String )
    public String toString()
    {
        StringBuffer cad = new StringBuffer("Nombre: " + nombre + "\n"
);

        // append es un método de concatenación o unión de cadenas.
        cad.append( "Dirección: " + direccion + "\n" );
        cad.append( "Estatura: " + estatura + "\n" );
        cad.append( "Fecha de Nac.: " + fnac );

        return cad.toString(); // el objeto StringBuffer se
                                //convierte a su vez en cadena
    } // fin de método toString

    // definición de una clase interna estática.
    // Usada para manejar la dirección.
    public static class Direccion
    {
        String calle;
        String colonia;
        String provincia;
        String país;
        String cp;

        // constructor por default de la clase
        // interna estática
        public Direccion( )
        {
            this.calle = null;
            this.colonia = null;
            this.provincia = null;
            this.país = null;
            this.cp = null;
        } // fin de constructor

        public Direccion(String calle, String colonia, String
provincia, String país, String cp)

```

```
        {
            this.calle = calle;
            this.colonia = colonia;
            this.provincia = provincia;
            this.país = país;
            this.cp = cp;
        } // fin de constructor

        public String toString()
        {
            StringBuffer cad = new StringBuffer("\n  Calle: " + this.calle
+ "\n");
            cad.append(" Colonia : " + this.colonia + "\n" );
            cad.append(" Provincia : " + this.provincia + "\n" );
            cad.append(" País : " + this.país + "\n" );
            cad.append("Codigo Postal : " + this.cp );

            return cad.toString();
        } // fin del método toString() de la clase
        // interna Direccion
    } // fin de la clase interna Direccion

} // fin de la clase Persona
```

---

```
package cic.oagb;
import java.awt.BorderLayout;
import java.util.Date;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class UsaLaPersona
{
    Persona p1; // variable de instancia p1 de
               // la clase Persona.

    public static void main(String [] argumentos)
    {
        Persona.Direccion dp1 = new Persona.Direccion("Farallones No. 5",
"Acueducto de Gpe." , "Distrito Federal", " México", "07270" );

        UsaLaPersona up = new UsaLaPersona(); // construyo un
                                                // objeto de esta
                                                // clase

        // construyo el objeto p1 de la clase Persona y que es
        // miembro de la clase UsaLaPersona.
```

```

        up.p1 = new    Persona ( "Oscar A. Gonzalez B.", dp1, 1.74d, new
Date(1961-1900, 1,13));

        JTextArea tap1 = new    JTextArea( up.p1.toString(), 4, 20);
        JScrollPane sptap1      =      new    JScrollPane(      tap1,
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        JPanel pan = new    JPanel();
        pan.setLayout( new BorderLayout() );
        pan.add( new    JLabel( "Datos de la Persona p1 " ) ,
BorderLayout.NORTH );
        pan.add( sptap1, BorderLayout.CENTER );
        pan.add( new    JLabel("Dele clic al boton de Aceptar para
terminar") , BorderLayout.SOUTH );
        JOptionPane.showMessageDialog( null, pan , "Datos de p1",
JOptionPane.INFORMATION_MESSAGE );
        System.exit( 0 ); // fin del proceso sin error.

    } // fin del método main()

} // fin de la clase UsaLaPersona

```

La salida del programa cuando se ejecuta es la siguiente:

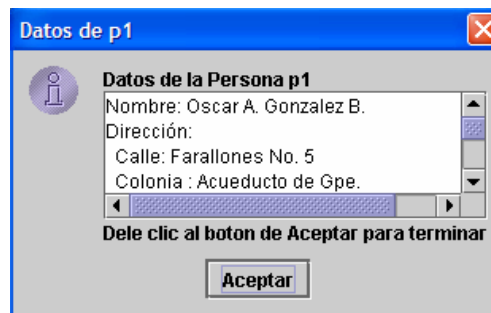


Figura 7.9. Pantalla de salida del programa UsaLaPersona.java.

# CAPÍTULO 8

## Encapsulamiento

---

El *encapsulamiento* es una metodología para ocultar ciertos elementos de la implementación (sentencias) de una clase, pero proveyendo una interface pública para el cliente o usuario del software, o dicho de otro modo: “el encapsulamiento separa los aspectos externos de un objeto de sus detalles internos de implementación”.

Así los cambios internos necesarios no afectan a la interface externa.

- Por ejemplo: usted no necesita saber cómo un teléfono conecta a las dos partes para poder utilizarlo. Sólo necesita entender la interface, cuál es la composición de botones, el auricular (para el oído) y la bocina (para la boca). Usted puede ignorar los procesos internos de cómo es hecha la conexión, los switches que cruzan los países, etcétera, (vea figura 8.1).



Figura 8.1. Teléfono compuesto de interface y procesos internos.

### 8.1 Modularidad

También el *encapsulamiento* es una propiedad de los programas orientados a objetos que nos permite crear componentes reutilizables (módulos) para uno o varios dominios del problema; pero primero es necesario entender algunos conceptos, antes de plantear lo qué es el encapsulamiento:

- *Código moderadamente ligado*. Es aquel que es independiente de la implementación de otros componentes.
- *Código estrechamente ligado*. Es aquel que depende intensamente de la implementación de otros componentes.

La modularidad significa que cada objeto debe realizar una función —su responsabilidad— y hacerla bien (*División de la responsabilidad*). Una división apropiada de la responsabilidad también significa que el objeto es consistente, esto es, no tiene sentido encapsular al azar una enorme cantidad de métodos y variables, ya que éstas deben contar con un estrecho vínculo conceptual entre sí. Todas las funciones tienen que enfocarse a una responsabilidad común.

## 8.2 Ocultamiento de la implementación

Un objeto oculta lo que hace a otros objetos y al mundo exterior, por lo cual al *encapsulamiento* también se le conoce como *ocultamiento de la información*. Pero un objeto tiene que presentar un “rostro” al mundo exterior para poder iniciar sus operaciones.

Por ejemplo, una lavadora tiene diversas perillas que le permiten establecer los niveles de temperatura y agua; un elevador tiene también botones para subir, bajar, abrir las puertas, cerrarlas, etc. Las perillas y botones de la lavadora y del elevador se conocen como *interfaces*.

El ocultamiento de la información nos permite escribir *código independiente y moderadamente ligado* con otros componentes. El *código moderadamente ligado* es menos frágil y más flexible al cambio. El código flexible facilita la reutilización y las mejoras, porque los cambios a una parte del sistema no afectan a las partes que no tienen relación con ésta.

## 8.3 Encapsulamiento eficaz

Para que dicho encapsulamiento sea eficaz, debe componerse de:

- *Abstracción*: es el proceso de ignorar los detalles para concentrarse en las características esenciales. Si se ignora la abstracción obtendremos código no reutilizable.
- *Ocultamiento de la implementación*: si eliminamos el ocultamiento de la implementación, el resultado será código frágil y estrechamente ligado.
- *División de la responsabilidad*: si eliminamos la responsabilidad, obtendremos código orientado a datos, procedural, estrechamente ligado y descentralizado.

## 8.4 Protección de variables y métodos

En los lenguajes de POO hay modificadores de acceso que se utilizan para controlar la accesibilidad de las clases y la de sus miembros a otras clases y objetos, éstos son: *private*, *protected* y *public*.

El delimitador por omisión es el conocido como acceso de paquete o acceso *por default*. Cuando no se especifica ningún modificador, se dice que se tiene un acceso

por *default*, esto es, un miembro con acceso por defecto proporciona a otras clases dentro del mismo *paquete*, acceso a miembros de clase.

- *Paquete*: un *paquete* es un conjunto de clases relacionadas en una unidad. Se agrupan clases en paquetes debido a que pueden estar relacionadas de acuerdo con su funcionalidad; así por ejemplo, en el lenguaje de programación Java, tenemos el paquete `java.sql`, el cual está dedicado al acceso e interacción de las aplicaciones de Java con las bases de datos mediante el puente JDBC.

#### **8.4.1 Miembros privados**

El delimitador de acceso ***private*** restringe el acceso a los miembros de su clase, de tal forma, que ninguna otra pueda llamar a sus métodos o acceder directamente a sus variables. Este modificador protege a nivel de clase, pero no a nivel de instancia. Todas las instancias de una clase pueden acceder a todas las variables privadas en todas las instancias, no sólo en su propia instancia.

#### **8.4.2 Miembros públicos**

El modificador de acceso ***public*** no restringe el acceso. Cualquier parte del código de cualquier objeto, puede acceder a clases, métodos y variables de miembros públicos.

#### **8.4.3 Miembros protegidos**

Un miembro con el modificador de acceso ***protected*** es algo parecido al acceso por omisión o acceso por *default*, pero otorga cierto acceso privilegiado a las subclases de otros paquetes. Frecuentemente los métodos se crean para utilizarse dentro de la propia clase (y no para su utilización general). Si se declaran como *protected*, las podremos utilizar dentro de la clase y todas sus subclases.

#### **8.4.4 Implementando el encapsulamiento**

Cuando implementamos el encapsulamiento hay que considerar que los miembros de un objeto encapsulado pueden tener acceso ***public*** o ***private***.

En sistemas orientados a objetos puros, todos los atributos son ***private***, y pueden ser cambiados o accedidos sólo a través de métodos ***public***.

Estos métodos son conocidos como *métodos accesorios* y se dividen en *getters* y *setters*. En los IDE de programación de lenguajes orientados a objetos, hay comandos para producirlos automáticamente, (ver la figura 8.2):

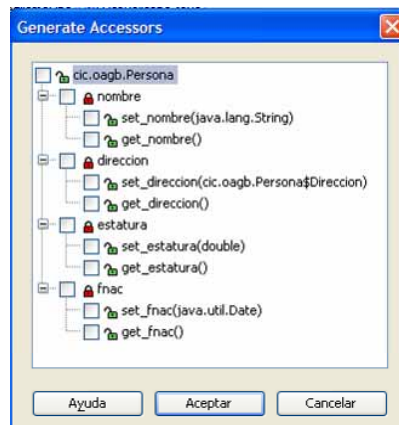


Figura 8.2. Cuadro de diálogo para generar los métodos accesores –getters y setters– de una clase llamada Persona en lenguaje Java.

Los métodos accesores *getters* se usan para leer el estado de un objeto, esto es, acceden al valor de las variables privadas de la clase y nos devuelven su estado. Tienen la siguiente forma general:

```
public      tipo      getXxxx( )
{
    return      valorxxxx;
}
```

Donde **Xxxx** es el nombre de la variable miembro de la clase que se desea obtener su valor.

Los métodos *setters* nos permiten escribir el estado de un objeto, esto es, acceden al valor de las variables privadas de la clase y lo cambian o mutan. Tienen la siguiente forma general:

```
public void  setXxxx( tipo  valorXxxx )
{
    this.xxxx =  valorXxxx;
}
```

Donde **Xxxx** es el nombre de la variable miembro de la clase a la cual se desea asignar o establecer un valor.

- **Mensajes**

Un solo objeto aislado generalmente no es muy útil, en lugar de uno solo, usualmente aparece como componente de una gran aplicación que contiene muchos otros objetos. A través de la interacción de estos objetos, los programadores alcanzan una funcionalidad superior y un comportamiento más complejo en sus sistemas. Por ejemplo, un objeto Automóvil, únicamente es un conjunto de piezas mecánicas de metal, llantas de goma, etc., que por sí solo no es capaz de alguna actividad. El automóvil es útil para cuando otro objeto (usted el conductor) interacciona con éste (con la palanca de cambios o embrague), (vea figura 8.3).

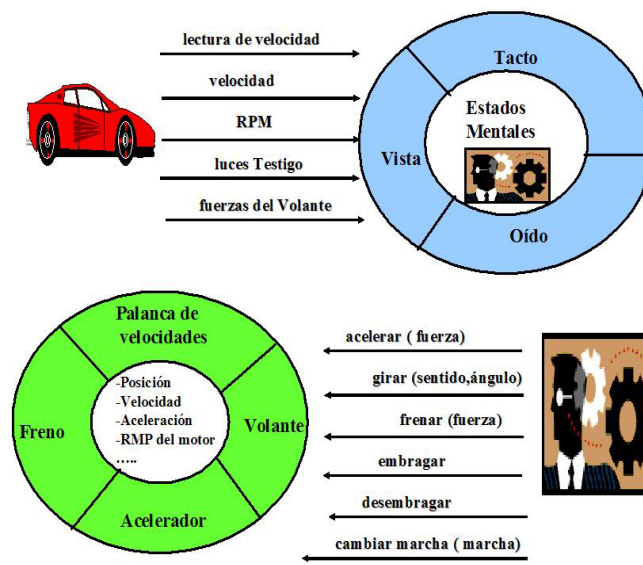


Figura 8.3. Mensajes entre el objeto Automóvil y el objeto Conductor.

Los objetos interactúan y se comunican unos con otros mediante el envío de *mensajes* de uno a otro. Cuando un objeto A quiere que el objeto B ejecute uno de sus métodos, el objeto A envía un mensaje al objeto B, (ver figura 8.4).

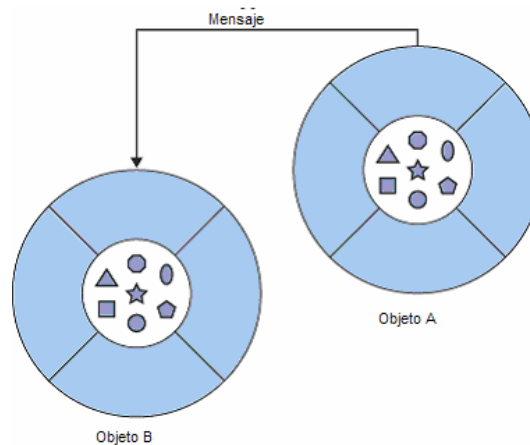


Figura 8.4. Envío de mensajes entre objetos.

A veces, el objeto receptor necesita más información sobre qué exactamente debe hacer: por ejemplo, cuando se requiere cambiar las velocidades del automóvil, se tiene que indicar la velocidad requerida. Esta información se transfiere a través de los *parámetros* del mensaje.

La figura 8.5 muestra las tres partes de un mensaje:

- El objeto al cual es enviado el mensaje (automóvil).
- El nombre del método para ejecutarlo (cambia velocidad).
- Cualquier número de parámetros necesarios para el método (velocidad).

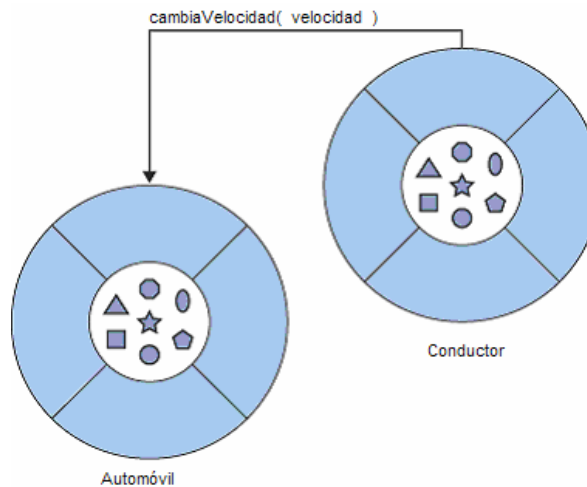


Figura 8.5. Las tres partes de un mensaje.

Estos tres componentes son suficiente información para el objeto receptor, el cual ejecuta el método deseado. No hay otra cosa que se requiera.

Los mensajes proveen dos importantes ventajas:

- El comportamiento de un objeto es expresado a través de sus métodos, así el paso de mensajes soportan todas las posibles interacciones entre los objetos.
- Los objetos no necesitan estar en el mismo proceso o incluso en la misma computadora para enviar y recibir mensajes de uno a otro.

Así en el encapsulamiento:

- Un *mensaje* es la invocación de una operación o método de la interfaz pública de un objeto encapsulado.
- En los objetos encapsulados, los *mensajes* son entre los métodos públicos de los objetos.
- El acceso al estado de un objeto se logra mediante estos mensajes a métodos accesoros y mutadores (los getters y setters).
- El *estado de un objeto* son los valores actuales de los atributos o variables de un objeto en un instante dado.
- La instancia de una clase (es decir un objeto) cambia o muta sus valores en el transcurso del tiempo de vida del objeto, esto es, desde su construcción hasta su destrucción.
- Este *estado del objeto* sólo es posible accederlo o modificarlo mediante el envío de *mensajes* o *invocaciones* de un objeto a otro a los métodos de la interface pública de los objetos —accesores (getters y setters)—, para no perder la propiedad del encapsulamiento.

Veamos ahora un ejemplo de encapsulamiento: haremos una clase AccesoBanco que utilice dos objetos de las clases CuentaDatos y ClienteDatos, todas ellas pertenecen al paquete dgsc.aogb (vea la figura 8.6).

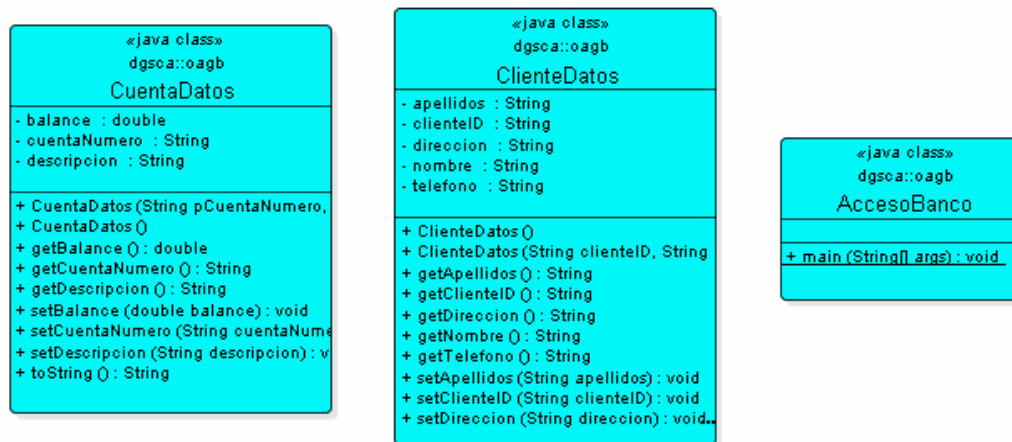


Figura 8.6. Diagrama de clases.

La clase **CuentaDatos** con las variables *miembro* privadas:

balance de tipo double

cuentaNumero de tipo String

descripción: de tipo String

Esta clase CuentaDatos tiene dos constructores públicos:

public CuentaDatos() que es el constructor por default —porque no tiene parámetros—.

public CuentaDatos( String pCuentaNumero, double pBalance, String pDescripcion ) que es el constructor con tres parámetros para inicializar las variables las variables privadas de la clase.

Como CuentaDatos tiene variables privadas, sólo pueden ser accedidas o modificadas con los métodos de interface pública llamados accesoros —getters y setters—.

También tenemos un método `toString()` que regresa y arma una cadena de los valores actuales de las variables privadas – estado del objeto – de tipo CuentaDatos —vea el siguiente código—.

```

package dgsca.oagb;

public class CuentaDatos
{
    private String    cuentaNumero;
    private double    balance;
    private String    descripción;

    // constructor por default
    public CuentaDatos()
    {
    } // fin del constructor

    public CuentaDatos(String pCuentaNumero, double pBalance, String
pDescripcion )
    {
        System.out.println("Creando una nueva CuentaDatos");
    }
}
  
```

```

        this.cuentaNumero = pCuentaNumero;
        this.balance = pBalance;
        this.descripcion = pDescripcion;
    } // fin del constructor

//Métodos de la interface publica. Accesores - getters y setters.
    public void setCuentaNumero(String cuentaNumero)
    {
        this.cuentaNumero = cuentaNumero;
    }

    public String getCuentaNumero()
    {
        return cuentaNumero;
    }

    public void setBalance(double balance)
    {
        this.balance = balance;
    }

    public double getBalance()
    {
        return balance;
    }

    public void setDescripcion(String descripcion)
    {
        this.descripcion = descripcion;
    }

    public String getDescripcion()
    {
        return descripcion;
    }

    // método toString que devuelve una cadena con los valores del
    estado - valores de las variables miembro-
    public String toString()
    {
        // creamos una cadena modificable
        StringBuffer sb = new StringBuffer("\nNumero de cuenta: ");
        //unimos o concatenamos las cadenas
        sb.append( this.cuentaNumero );
        sb.append( "\nDescripción: ");
        sb.append( this.descripcion );
        sb.append( "\nBalance: ");
        sb.append( this.balance );

        return sb.toString(); // convertimos la cadena StringBuffer
                               // a String y la retornamos.
    } // fin del método toString()

} // fin de la clase CuentaDatos

```

De manera similar, también tenemos una clase llamada `ClienteDatos` con las variables, miembro privadas:

- apellidos de tipo `String`
- clienteID de tipo `String`
- direccion de tipo `String`
- nombre de tipo `String`
- telefono de tipo `String`

La clase `Cliente` tiene dos constructores públicos

- `public ClienteDatos()` que es el constructor por default y el constructor
- `public ClienteDatos(String clienteID, String nombre, String apellidos, String direccion, String telefono)`

que es el constructor con cinco parámetros para inicializar las variables las variables privadas de la clase:

Como `ClienteDatos` tiene variables privadas, sólo pueden ser accedidas o modificadas con los métodos de interface pública llamados accesores —getters y setters—.

También tenemos un método `toString()` que regresa y arma una cadena de los valores actuales de las variables privadas – estado del objeto – de tipo `ClienteDatos` —vea el siguiente código—.

```
package dgsca.oagb;
import java.util.Date;

public class ClienteDatos
{
    // Atributos privados del objeto - Encapsulados -
    private String clienteID;
    private String nombre;
    private String apellidos;
    private String direccion;
    private String telefono;

    // Constructores: El constructor por default no tiene parámetros
    public ClienteDatos()
    {
        this.clienteID = null;
        this.nombre = null;
        this.apellidos = null;
        this.direccion = null;
        this.telefono = null;
    } // fin del constructor por default

    public ClienteDatos(String clienteID, String nombre, String
    apellidos, String direccion, String telefono)
    {
        this.clienteID = clienteID;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

```
        this.direccion = direccion;
        this.telefono = telefono;
    } // fin del constructor con cinco argumentos

    // Métodos de interface publica
    // Accesos: getters y setters.
    // Utilizados para modificar o accesar el estado del objeto en sus
    // variables privadas

    public void setClienteID(String clienteID)
    {
        this.clienteID = clienteID;
    }

    public String getClienteID()
    {
        return clienteID;
    }

    public void setNombre(String nombre)
    {
        this.nombre = nombre;
    }

    public String getNombre()
    {
        return nombre;
    }

    public void setApellidos(String apellidos)
    {
        this.apellidos = apellidos;
    }

    public String getApellidos()
    {
        return apellidos;
    }

    public void setDireccion(String direccion)
    {
        this.direccion = direccion;
    }

    public String getDireccion()
    {
        return direccion;
    }

    public void setTelefono(String telefono)
    {
        this.telefono = telefono;
    }
}
```

```
public String getTelefono()
{
    return telefono;
}

// método toString() regresa una cadena del estado del
// objeto - valores de sus variables privadas -
public String toString()
{
    StringBuffer cad = new StringBuffer("Cliente ID: " + getClienteID() +
"\n" );
    cad.append( "Nombre: " + getNombre() + "\n" );
    cad.append( "Apellidos: " + getApellidos() + "\n" );
    cad.append( "Dirección: " + getDireccion() + "\n" );
    cad.append( "Teléfono: " + getTelefono() + "\n" );
    return cad.toString();
} // fin del método toString()

} // fin de la clase ClienteDatos
```

Ahora la clase AccesoBanco, hace uso de estas clases creando dos objetos:

Cliente y cuenta dentro de su método *main()* :

```
ClienteDatos cliente = new ClienteDatos( "8310606-6" , "Oscar
Alejandro" , "González Bustamante", "Farallones No. 5, Residencial
Acueducto de Guadalupe", "5598-92-93");

CuentaDatos cuenta = new CuentaDatos("3637-2637-1737-2647" ,
17505.30d , "Tarjeta de Credito - Clasica -" );
```

Una vez creados o instanciados estos objetos, podemos invocar a sus métodos *toString()* de ambas clases para recuperar los datos encapsulados en las variables *miembro* privadas, y concatenarlos a una nueva variable *cadenaClienteCuenta* :

```
String cadenaClienteCuenta = cliente.toString() + cuenta.toString() ;
```

Finalmente, imprimimos en la consola de salida, el valor de la variable *cadenaClienteCuenta* con el método *println()* :

```
System.out.println( cadenaClienteCuenta );
```

Y también creamos una pequeña interface gráfica de usuario GUI, mediante la utilización de las clases estándar o API de Java del paquete *javax.swing*, para dar la salida en un cuadro de diálogo de mensaje. Primero creamos el objeto *tap1* de la clase *JTextArea*, de ocho filas por 25 columnas, así:

```
JTextArea tap1 = new JTextArea(cadenaClienteCuenta, 8, 25);
```

Luego creamos un objeto *sptap1* de la clase *JScrollPane* con el objeto *tap1* de la clase *JTextArea* para desplazar la información horizontalmente y verticalmente, así:

```
JScrollPane sptap1 = new JScrollPane( tap1,
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );
```

Luego creamos un objeto *pan* de la clase *JPanel* con distribución *BorderLayout()*– para agregar los componentes a cinco áreas de distribución posibles, Norte, Sur, Este, Oeste o Centro, así:

```
JPanel pan = new JPanel();
pan.setLayout(new BorderLayout());
```

Agregamos una etiqueta al objeto *pan* de la clase *JPanel*, construyendo un objeto de la clase *JLabel* al área Norte, así:

```
pan.add(new JLabel("Datos del cliente " + cliente.getNombre() + " " +
cliente.getApellidos()), BorderLayout.NORTH);
```

Cheque el uso de los métodos accesorios; *getNombre()* y *getApellidos()* para acceder al valor de las variables privadas *nombre* y *apellidos* del objeto *cliente* de la clase *ClienteDatos*.

Luego agregamos al objeto *pan* de la clase *JPanel*, el objeto *sptap1* de la clase *JScrollPane* al área Centro, así:

```
pan.add( sptap1, BorderLayout.CENTER );
```

creamos un etiqueta construyendo un objeto de la clase *JLabel* y lo colocamos en el área Sur, así:

```
pan.add( new JLabel("Dele clic al boton de Aceptar para terminar") ,
BorderLayout.SOUTH );
```

y finalmente, agregamos el objeto *pan* de la clase *JPanel* a un cuadro de diálogo de mensaje, mediante el método *showMessageDialog()* de la clase *JOptionPane* así:

```
JOptionPane.showMessageDialog( null, pan , "Datos de la cuenta: " +
cuenta.getCuentaNumero() , JOptionPane.INFORMATION_MESSAGE );
```

Cheque el uso del método accesor *getCuentaNumero()* para acceder al valor del número de cuenta del objeto *cuenta* de la clase *CuentaDatos*.

El código de la clase *AccesoBanco* se lista a continuación:

```
package dgsca.oagb;
import java.awt.BorderLayout;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class AccesoBanco
{
    public static void main(String[] args)
    {
```

```

        ClienteDatos cliente = new ClienteDatos( "8310606-6" , "Oscar
Alejandro" , "González Bustamante", "Farallones No. 5, Residencial
Acueducto de Guadalupe", "5598-92-93");

        CuentaDatos cuenta = new CuentaDatos("3637-2637-1737-2647" ,
17505.30d , "Tarjeta de Credito - Clasica -" );

        // recuperamos valores de las variables privadas con el
        // metodo toString()
        String cadenaClienteCuenta = cliente.toString() +
cuenta.toString();

        // imprime en consola de salida
        System.out.println( cadenaClienteCuenta );

        // crea un TextArea con la cadenaClienteCuenta
        JTextArea tap1 = new JTextArea(cadenaClienteCuenta, 8, 25);
        JScrollPane sptap1 = new JScrollPane( tap1,
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );
        JPanel pan = new JPanel();
        pan.setLayout(new BorderLayout());
        pan.add(new JLabel("Datos del cliente " +
cliente.getNombre() + " " + cliente.getApellidos() ) ,
BorderLayout.NORTH );
        pan.add(sptap1, BorderLayout.CENTER);
        pan.add(new JLabel("Dé clic al botón de Aceptar para terminar") ,
BorderLayout.SOUTH );
        JOptionPane.showMessageDialog(null, pan, "Datos de la cuenta: " +
cuenta.getCuentaNumero() , JOptionPane.INFORMATION_MESSAGE) ;
    } // fin del método main()
} // fin de la clase AccesoBanco

```

Al ejecutar este programa, vea figura 8–7, nos da el siguiente resultado:

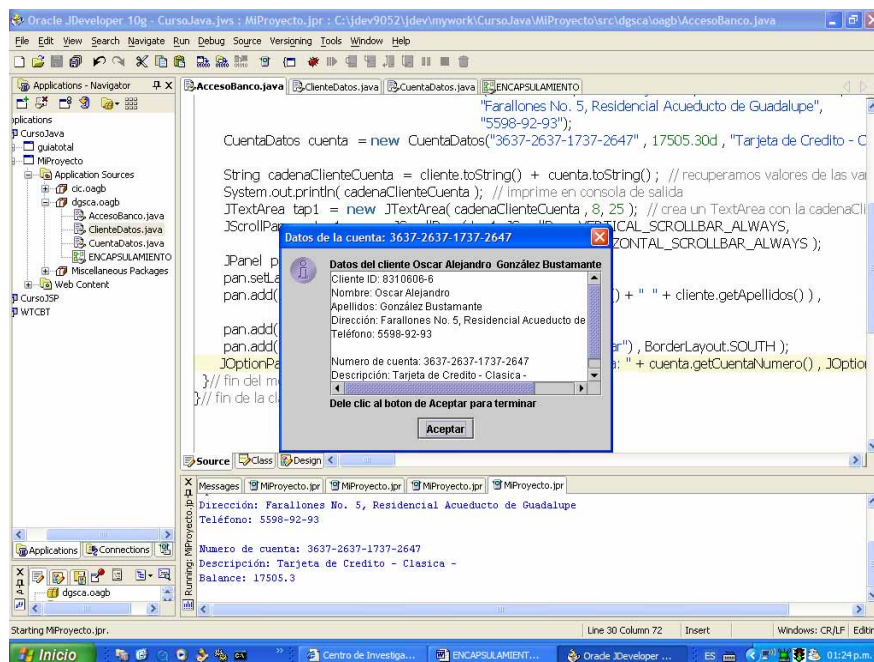


Figura 8.7. Pantalla de ejecución del programa.

En este caso se editó el código y los diagramas de UML de Java, e igual se compiló y ejecutó con el JDeveloper, que es un IDE de distribución libre de la empresa Oracle; aunque cabe señalar que puede usarse cualquier IDE de Java como el Eclipse, el NetBeans, JBuilder, etc., y en todos corre igual.

# CAPÍTULO 9

## Herencia

---

En esta sección, veremos cómo la relación entre clases llamada *herencia* o relación *is-a* (*es-un*), es una estrategia de la POO para la reutilización de código y para la especialización y extensión de código existente en algo nuevo y mejor.

### 9.1 Jerarquía de clases

En los lenguajes de POO, una jerarquía de clases está conformada por un conjunto de clases que forman un árbol de nodos, donde los nodos son clases, y este árbol tiene en su parte superior una clase denominada *clase raíz*, árbol que termina en una(s) clase(s) *hoja(s)*.

#### 9.1.1 Generalización y especialización

En la figura 9.1, se muestra la diferencia entre *generalización* y *especialización*. En la medida en que se recorra hacia abajo una jerarquía, se estará *especializando*. Conforme lo haga hacia arriba estará *generalizando*. Entre más generales, más clases caerán dentro de esta categoría, y entre más se especialicen, menos podrán conocer todos los criterios para categorizarse en ese nivel.

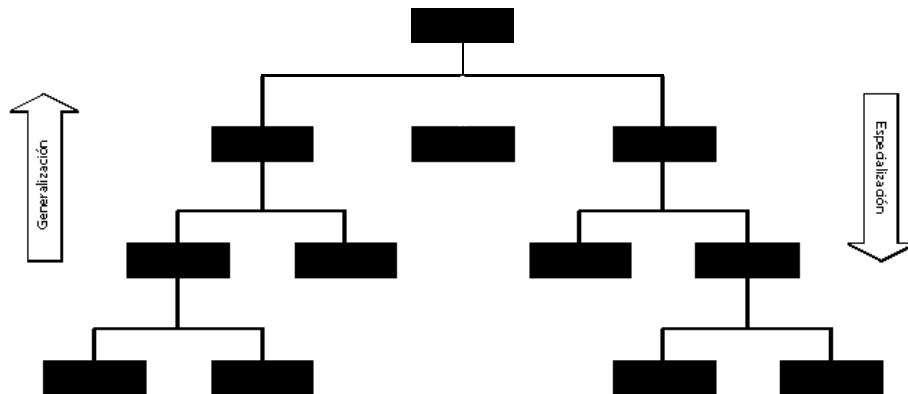


Figura 9.1. Generalización y especialización en la jerarquía de clases.

Como veremos más adelante, entre más arriba o general sea una clase, éstas tienden a ser clases abstractas (abstract); entre más abajo en la jerarquía, la clase será más específica o más especializada, tendremos clases concretas hasta llegar a las hojas del árbol de jerarquía.

- **Clase raíz**

La *clase raíz* es la clase base de todas las clases en una jerarquía de clases. Así por ejemplo, en el lenguaje de programación Java todas las clases heredan por omisión de la clase Object (conocida como la clase raíz), todos sus métodos y atributos de manera implícita, pero el programador puede crear sus clases e implementar la herencia de manera explícita con la palabra reservada *extends*, y así, construir un sub-árbol dentro de la jerarquía de clases que tiene Java.

- **Clase Hoja**

Una clase Hoja es aquella que ya no tiene descendencia y en la jerarquía de clases está en el nivel más especializado o concreto posibles. En el lenguaje Java, a estas clases se les conoce como clases finales.

- **Superclase y subclase**

La herencia define la nueva clase, la *hija*, en función de una clase anterior ya existente denominada, la *madre*. Esta relación *madre-hija* es la más sencilla de la herencia. De hecho, todas las jerarquías de herencia empiezan con una madre y una hija.

La *clase hija* es la que recibe la herencia, también se conoce como *subclase*.

La *clase madre* es de la cual hereda directamente la hija, también se conoce como *superclase*, (ver la figura 9.2).

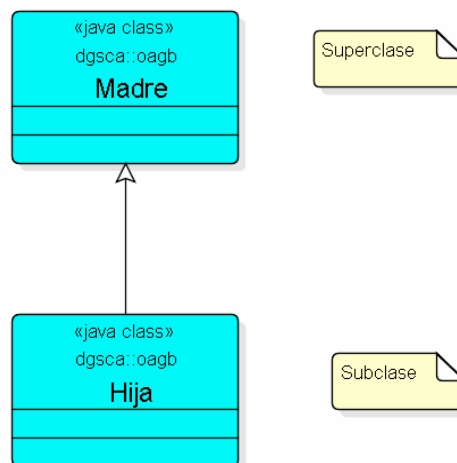


Figura 9.2. Diagrama de una superclase y una subclase.

La relación entre la superclase y la subclase en diagramas de UML<sup>6</sup> es representada mediante una flecha que va de la subclase a la superclase y con la punta del lado de la superclase.

## 9.2 Mecánica de la herencia

Cuando una clase hereda de otra su implementación, comportamientos y atributos, esto significa que todos los métodos y atributos disponibles en la interfaz pública de la madre, aparecerán en la interfaz de la hija. Una clase elaborada a través de la herencia, puede tener tres tipos importantes de métodos y atributos.

- **Redefinidos:** la nueva clase hereda el método o el atributo de la madre, pero proporciona una nueva definición. (El concepto *redefinidos* también se le conoce como *sobreescritos* o *sobrepuestos*, veremos esto con más detalle en la siguiente sección).
- **Nuevos:** la nueva clase agrega un método o un atributo completamente nuevo.
- **Rekursivos:** la nueva clase simplemente hereda de la madre un método o un atributo.

La mayoría de los lenguajes orientados a objetos no permiten redefinir un atributo o variable a excepción de lenguajes de POO modernos como en Java, que sí lo permite.

### 9.2.1 El uso de *super*

La palabra reservada *super* es utilizada en la herencia cuando queremos invocar a un método constructor de la superclase, a un método cualquiera de la superclase o a una variable de la superclase.

- `super([lista_de_argumentos])`
- `super.metodo([lista_de_argumentos])`
- `super.variable`

## 9.3 Clases abstractas

Son aquellas clases que son definidas como *abstract*, en tanto sólo se espera que tenga rasgos comunes a subclases que se derivarán de ella. Una clase abstracta tiene las siguientes reglas:

- No permite definir instancias de ella (no permite crear objetos).
- El que una clase abstracta no pueda crear objetos, no le imposibilita tener constructores. Estos constructores serán invocados por los constructores de las subclases con *super*, para crear un objeto de esa subclase.

---

<sup>6</sup> UML. (Unified Modeling Language ) Es un lenguaje gráfico, que unifica varios otros lenguajes gráficos, y es utilizado para el modelado de aplicaciones Orientadas a Objetos, en particular para las fases de análisis y diseño en el desarrollo de un sistema.

- Normalmente las clases que están por encima en la jerarquía son más abstractas que las inferiores que son más concretas.
- También podemos considerar una clase como abstracta cuando una clase tiene por lo menos un método declarado pero sin implementación, a este tipo de métodos se les denomina *métodos abstractos*.
- Al declarar métodos abstractos, se obliga a sus subclases a especializarse sobre la clase base abstracta proporcionando una implementación para los métodos abstractos (métodos redefinidos o sobrescritos).
- El que una clase sea abstracta no quiere decir que sólo tienen métodos abstractos, por lo tanto, pueden tener métodos comunes y corrientes, esto es métodos concretos.
- A través de hacer abstracta una clase base y de crear métodos abstractos se planea de antemano lo que las subclases deben redefinir.

Por ejemplo: supongamos que queremos hacer un software que maneje figuras. Nuestro software tendrá como característica principal, dibujar figuras geométricas, borrarlas, calcular sus áreas correspondientes, etc. Podemos partir de una definición abstracta de figura. Cuando mis alumnos no entienden qué es clase abstracta, les pido que pasen al pizarrón a dibujar una figura, unos dibujan círculos, otros triángulos, otros rectas, polígonos, etc. "*todos dibujan figuras concretas diferentes pero nunca una figura abstracta en sí*", esto es, el concepto figura es un concepto abstracto, entonces partimos de la definición de una clase abstracta como la que se ve en la figura 9.3.

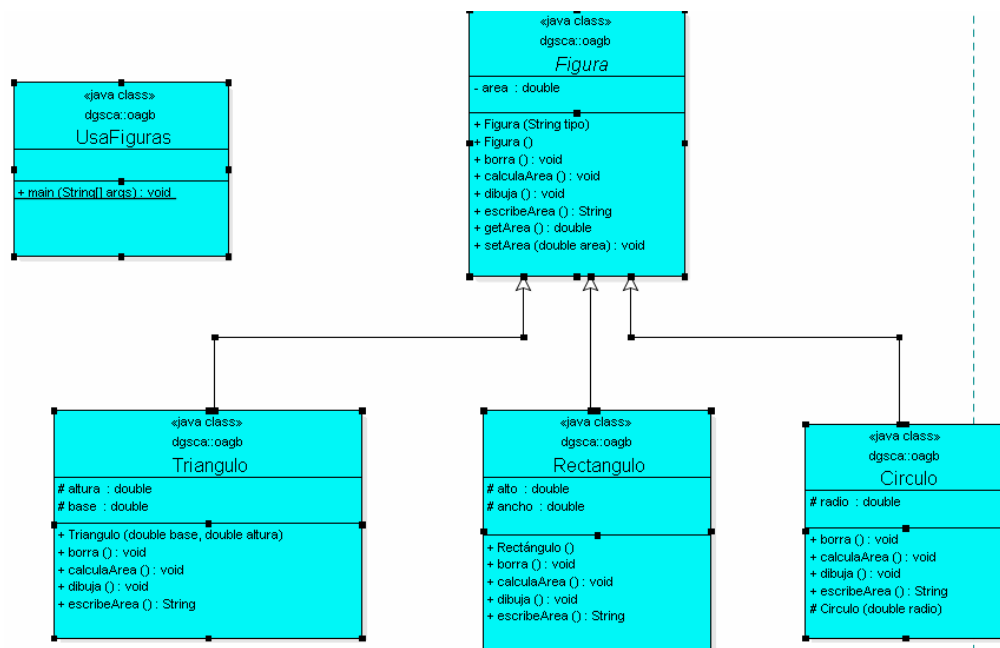


Figura 9.3. Jerarquía de clases donde *Figura* es una clase abstracta.

Nótese que el nombre de la clase abstracta *Figura* está escrita en *italicas*, por convención del UML. En ella, hay métodos abstractos como *dibuja()*, *borra()*, *calculaArea()* y *escribeArea()*, y éstos deberán ser sobre-escritos o redefinidos en las subclases *Triangulo*,

**Rectángulo y Círculo.** También tenemos un constructor en la clase abstracta **Figura**, el cual sólo puede ser usado por las subclases mediante su invocación con la palabra *super*. No nos es permitido crear un objeto de la clase **Figura** directamente, esto es, hacer un:

```
Figura f new Figura("Figura");    // oops!! No se puede instanciar
                                   //una clase abstracta.
```

Si lo intentamos hacer, el compilador de Java nos enviará un mensaje de error parecido al comentario. También podemos tener métodos concretos, en el ejemplo son los métodos accesores *getArea()* y *setArea double area*, y que nos permitirán tener acceso a la variable privada *area* debido a que no es posible heredar miembros privados a las subclases (en la herencia sólo se heredan los miembros *public*, *protected* o *default* ). El código de la clase abstracta **Figura** es el siguiente:

```
package dgsca.oagb;
// esta es la clase abstracta
abstract class Figura
{
    private double area; // variable miembro privada

    // constructores
    public Figura() // constructor por default
    {
        System.out.println("soy el constructor de la clase abstracta
Figura() ");
    } // fin del constructor

    public Figura(String tipo) // constructor con un argumento String
    {
        System.out.println("soy el constructor de la clase abstracta
Figura() y voy a crear un " + tipo);
    } // fin del constructor

    // métodos abstractos
    public abstract void dibuja();
    public abstract void borra();
    public abstract void calculaArea();
    public abstract String escribeArea();
    // métodos accesores
    // observe que una clase abstracta puede tener métodos concretos
    public double getArea()
    {
        return this.area;
    }

    public void setArea( double area )
    {
        this.area = area;
    }
} // Fin de la clase abstracta Figura
```

En la clase **Círculo**, en el constructor invocamos al de la superclase **Figura** con *super* ("**Círculo**"). Los métodos abstractos que son heredados de **Figura** tienen que ser redefinidos, así por ejemplo, el método redefinido *calculaArea()* establece el valor del área del círculo con el método accesor *this.setArea( Math.PI \* radio \* radio )*. El método

`getArea()` es utilizado para devolvernos el valor del área del círculo para formar la cadena que nos devuelve el método redefinido `escribeArea()`. Todos estos métodos accesorios y la variable privada `area`, pasan como miembros recursivos, con la herencia de la clase **Figura** a la clase **Circulo**. El código de la clase **Circulo** es el siguiente:

```
package dgsca.oagb;

public class Circulo extends Figura
{
    protected double radio;
    protected Circulo(double radio)
    {
        super( "Circulo" ) ; // invoco al constructor de la superclase
    }
    abstracta // Figura con un argumento String
        this.radio = radio;
    } // fin del constructor

    // métodos redefinidos o sobreescritos.
    public void borra()
    {
        System.out.println("borro circulo");
    }

    public void dibuja()
    {
        System.out.println("dibujo circulo");
    }
    public void calculaArea()
    {
        this.setArea( Math.PI * radio * radio ) ;
    }

    public String escribeArea()
    {
        this.calculaArea();
        return "\nEl área de un circulo de radio: " + this.radio + " es
        igual a : " + getArea() ;
    }
} // fin de la clase Circulo
```

El código de la clase **Rectangulo** es el siguiente:

```
package dgsca.oagb;

public class Rectangulo extends Figura
{
    protected double alto , ancho;

    public Rectangulo( double alto , double ancho )
    {
        super("Rectángulo");
        this.alto = alto;
        this.ancho = ancho;
    } // fin del constructor

    // métodos redefinidos o sobreescritos.
    public void borra()
    {
```

```
        System.out.println("borro rectángulo" );
    }

    public void dibuja()
    {
        System.out.println("dibujo rectángulo");
    }
    public void calculaArea()
    {
        this.setArea(this.ancho * this.alto) ;
    }

    public String escribeArea()
    {
        this.calculaArea();
        return "\nEl area de un rectángulo de ancho: " + this.ancho + "
y altura : " + this.alto + " es igual a : " + getArea();
    }
} // fin de la clase Rectangulo
```

El código de la clase **Triangulo** es el siguiente:

```
package dgsca.oagb;

public class Triangulo extends Figura
{
    protected double base, altura;

    //Constructores
    public Triangulo( double base, double altura )
    {
        super ( "Triángulo");
        this.base = base;
        this.altura = altura;
    } // fin del constructor

    // métodos redefinidos o sobreescritos.
    public void borra()
    {
        System.out.println("borro triángulo" );
    }

    public void dibuja()
    {
        System.out.println("dibujo triángulo");
    }
    public void calculaArea()
    {
        this.setArea( this.base * this.altura / 2.0d ) ;
    }

    public String escribeArea()
    {
        this.calculaArea();
        return "\nEl area de un triángulo de base: " + this.base + " y
altura : " + this.altura + " es igual a : " + getArea() ;
    }
} // fin de la clase Triangulo
```

La clase **UsaFiguras** utiliza referencias *f1*, *f2* y *f3* de la clase abstracta **Figura** y los instancia o construye con las subclases concretas **Circulo**, **Triangulo** y **Rectangulo** respectivamente. Para ello utiliza el *Upcasting* para asignarle a las referencias de una superclase, un objeto de la subclase. Después invoca a los métodos polimórficos —mas adelante veremos que es el polimorfismo— con enlazamiento tardío, para dibujar, borrar e imprimir el área de cada uno de estos objetos.

```
package dgsca.oagb;

public class UsaFiguras
{
    public static void main( String[] args )
    {
        Figura f1 = new Circulo( 30.0d ); // upcasting
        Figura f2 = new Triangulo( 30.0d , 40.0d ); // upcasting
        Figura f3 = new Rectangulo( 40.0d , 30.0d ); // upcasting
        f1.dibuja();
        f1.borra();
        f2.dibuja();
        f2.borra();
        f3.dibuja();
        f3.borra();
        System.out.println( f1.escribeArea() + f2.escribeArea() +
f3.escribeArea());
        System.out.println("Fin del programa");
        System.exit( 0 ); // fin del proceso
    } // fin del método main()
} // fin de UsaFiguras
```

La salida al ejecutar el programa *UsaFiguras.java* es:

```
soy el constructor de la clase abstracta Figura() y voy a crear un Circulo
soy el constructor de la clase abstracta Figura() y voy a crear un Triángulo
soy el constructor de la clase abstracta Figura() y voy a crear un Rectángulo
dibujo circulo
borro circulo
dibujo triángulo
borro triángulo
dibujo rectángulo
borro rectángulo

El área de un circulo de radio: 30.0 es igual a : 2827.4333882308138
El área de un triángulo de base: 30.0 y altura : 40.0 es igual a : 600.0
El área de un rectángulo de ancho: 30.0 y altura : 40.0 es igual a : 1200.0
Fin del programa

Process exited with exit code 0
```

## 9.4 Polimorfismo

El polimorfismo (poli = muchas, morfismo = formas) es una de las propiedades importantes de la Programación Orientada a Objetos (POO). Para entender el polimorfismo, debemos entender primeramente la herencia, ya que dicho concepto está muy relacionado con el polimorfismo.

El polimorfismo es una de las técnicas más importantes que permiten al programador "*Separar las cosas que cambian de las cosas que deben permanecer igual*", así se puede reutilizar código existente para las cosas que quedan sin cambios y crear código nuevo para las cosas que cambian.

#### **9.4.1 Enlazamiento temprano**

En el polimorfismo, cuando conectamos una llamada (invocación) de un método con su cuerpo de sentencias, realizamos un proceso denominado *ligado*. Cuando el ligado es ejecutado antes de que el programa sea ejecutado, a esto se le conoce como *enlazamiento temprano* (*early binding*).

Con el polimorfismo es posible que objetos similares respondan al mismo mensaje de diferentes maneras. Si ese mensaje o ligado se hace en tiempo de compilación, entonces estamos hablando de enlazamiento temprano.

Por ejemplo, se puede tener un sistema con muchas figuras. Sin embargo, un círculo, un cuadrado, y un triángulo se dibujan de diferente forma y tienen diferentes áreas. Usando el polimorfismo se puede enviar a cada una de las figuras el mismo mensaje (por ejemplo, *dibuja*), y cada forma es responsable de dibujarse así misma, (vea figura 9.4).

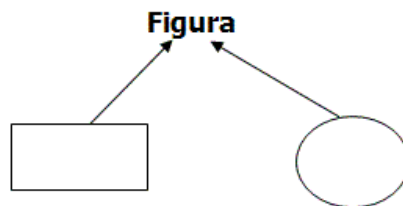


Figura 9.4. Figuras que se dibujan de distinta forma.

#### **9.4.2 Enlazamiento tardío**

En la sección anterior ya habíamos visto que en el polimorfismo, cuando conectamos una llamada (invocación) de un método con su cuerpo de sentencias, esto es denominado *ligado*, y que cuando el ligado es ejecutado antes de que el programa sea ejecutado, se le conoce como *enlazamiento temprano* (*early binding*). ¿Qué ocurre cuando el ligado es en tiempo de ejecución?

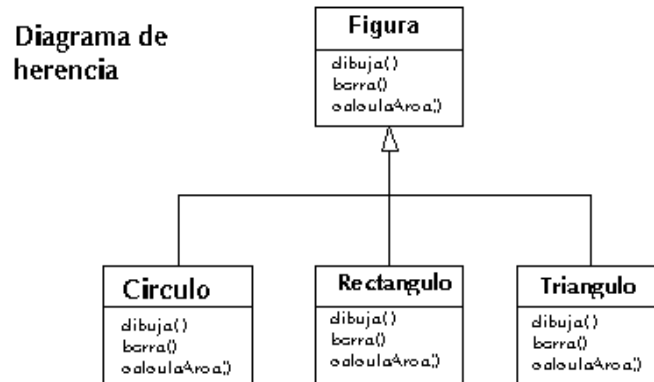
Cuando el ligado ocurre en tiempo de ejecución sobre el tipo del objeto, a esto se le conoce como *enlazamiento tardío*. El enlazamiento tardío también es denominado *enlazamiento dinámico* o *enlazamiento en tiempo de ejecución*.

Cuando un lenguaje de programación implementa el enlazamiento tardío, debe haber algún mecanismo que determine el tipo del objeto en tiempo de ejecución para llamar al método apropiado. Esto significa que el compilador sigue sin saber el tipo del objeto, pero dicho mecanismo de llamada del método lo resuelve y llama al cuerpo del método correcto.

Una vez que usted está consciente de que todo ligado o invocación de métodos ocurre polimórficamente vía *enlazamiento tardío*, se puede escribir su código para llamar a la clase base y saber que todas las clases derivadas trabajarán correctamente usando el

mismo código. O poniéndolo de otra forma: se envía un mensaje a un objeto y deja al objeto que decida la acción correcta a hacer).

El clásico ejemplo de figuras geométricas es muy ilustrativo para ejemplificar el polimorfismo con ligado dinámico. El ejemplo de figuras tiene una superclase abstracta base llamada **Figura** y varios tipos de clases derivadas: **Circulo**, **Rectangulo**, **Triangulo**, etc. Se utiliza dicho ejemplo porque se entiende al decir que un círculo es un tipo de figura. Ver el diagrama de clases:



Con la herencia relacionando las clases, podemos hacer upcasting o conversión de la clase (como vimos anteriormente) de un objeto a la de su clase base, y se realiza con:

```
Figura f = new Circulo();
```

Aquí, un objeto **Circulo** es creado, y la referencia resultante es inmediatamente asignada a **Figura**, lo cual podría ser un error a primera vista, (asignación de un tipo por otro); no obstante esto es correcto, porque un *Circulo* es una *Figura* por herencia. Así que el compilador está de acuerdo con esto y la sentencia no arroja ninguna clase de error.

Suponga ahora que se invoca a un método de la clase base (que ha sido sobre escritos en las clases derivadas).

```
f.dibuja();
```

Y de nueva cuenta, usted podría esperar que el método *dibuja()* de **Figura** es ejecutado, porque después de todo **f** es una referencia a **Figura**, y se preguntaría cómo podría hacer el compilador lo correcto, y a pesar de esto, el método correcto *Circulo.dibuja()* es invocado debido al *enlazamiento tardío* (polimorfismo).

El siguiente programa, al cual se le ha agregado una clase **Triangulo** para el caso del triángulo, ejemplifica estos conceptos. La clase **UsaFigurasDos** hace uso de las clases **Figura**, **Rectangulo**, **Circulo** y **Triangulo** para invocar o llamar al método *escribeArea()* de manera polimórfica mediante el enlazamiento tardío, al invocarlo en el ciclo for en:

```
// llena el arreglo con figuras
for (int i=0; i < f.length; i++)
    f[i] = figuraalAzar();
```

aquí el método *figuraalAzar()* devuelve un objeto de las clases **Rectangulo**, **Circulo** y **Triangulo** pero lo hace al azar y en tiempo de ejecución. Luego, al invocar al método *escribeArea()* en el ciclo **for** así:

```
// Hace llamadas a los metodos polimórficos con
// enlazamiento dinámico o early binding
for (int i=0; i < f.length; i++)
    System.out.println(f[i].escribeArea());
```

se consigue escribir el área de la figura adecuada gracias al enlazamiento que se hace de forma dinámica o en tiempo de ejecución y que determina la clase de objeto a que se está haciendo referencia. A continuación se presenta todo el código del programa:

```
/**
 * Programa de java que ejemplifica el uso
 * de las clases que heredan métodos
 * de una clase base para así poder
 * implementar polimorfismo y el enlazamiento
 * tardío o enlazamiento dinámico
 * que ocurre en tiempo de ejecución.
 * @autor Oscar A. González Bustamante
 * @version 1.0
 * Archivo: UsaFigurasDos.java
 */
package dgsca.oagb;

public class UsaFigurasDos {

    // este metodo regresa figuras al azar
    public static Figura figuraalAzar() {

        switch ( (int) (Math.random() * 3) ) {
            case 0 : return new Circulo(30);
            case 1 : return new Rectangulo(40,30);
            case 2 : return new Triangulo(30, 40 );

            default : return null;
        }
    }
}
```

```

public static void main(String argumentos[]) {

    Figura f[]= new Figura[9]; // crea un arreglo de 10 referencias a Figura

    // llena el arreglo con figuras
    for ( int i=0; i < f.length; i++ )
        f[i] = figuraalAzar();

    // Hace llamadas a los metodos polimórficos con
    // enlazamiento dinámico o early binding
    for ( int i=0; i < f.length; i++ )
        System.out.println(f[i].escribeArea());

    System.out.println("Fin del programa");
}

} // fin de UsaFigurasDos

```

Para correr el programa, hay que ejecutar el siguiente comando **java UsaFigurasDos** y la salida es la siguiente:

```

soy el constructor de la clase abstracta Figura() y voy a crear un Circulo
soy el constructor de la clase abstracta Figura() y voy a crear un Circulo
soy el constructor de la clase abstracta Figura() y voy a crear un Circulo
soy el constructor de la clase abstracta Figura() y voy a crear un Triángulo
soy el constructor de la clase abstracta Figura() y voy a crear un Rectángulo
soy el constructor de la clase abstracta Figura() y voy a crear un Circulo
soy el constructor de la clase abstracta Figura() y voy a crear un Rectángulo
soy el constructor de la clase abstracta Figura() y voy a crear un Circulo
soy el constructor de la clase abstracta Figura() y voy a crear un Triángulo

El área de un circulo de radio: 30.0 es igual a : 2827.4333882308138

El área de un circulo de radio: 30.0 es igual a : 2827.4333882308138

El área de un circulo de radio: 30.0 es igual a : 2827.4333882308138

El área de un triángulo de base: 30.0 y altura : 40.0 es igual a : 600.0

El área de un rectángulo de ancho: 30.0 y altura : 40.0 es igual a :
1200.0

El área de un circulo de radio: 30.0 es igual a : 2827.4333882308138

```

```
El área de un rectángulo de ancho: 30.0 y altura : 40.0 es igual a :  
1200.0
```

```
El área de un círculo de radio: 30.0 es igual a : 2827.4333882308138
```

```
El área de un triángulo de base: 30.0 y altura : 40.0 es igual a : 600.0  
Fin del programa  
Process exited with exit code 0.
```

# CAPÍTULO 10

## Interfaz gráfica

---

Durante mucho tiempo, los programadores desarrollaron aplicaciones donde la entrada de datos era principalmente por el teclado, y la salida con los resultados era principalmente, mediante una pantalla negra llamada consola de una sesión de terminal de UNIX o MS-DOS. Estas entradas y salidas estaban basadas en comandos de cadenas de texto, así el usuario que explotaba estos sistemas tenía que aprenderse multitud de líneas de texto de comandos para decirle a la aplicación lo que quería ejecutara.

Uno de los principales problemas de uso de estas aplicaciones con interfaz de texto, que no eran “amigables” para el usuario, era tener que aprender cientos de comandos en línea de texto con sus respectivas variantes.

Otra desventaja era la salida, ya que los usuarios querían ver los resultados en un formato adecuado sobre papel, tal y como los habían visto en la pantalla, pero a la hora de imprimirlos salían de diferente forma, y no se diga si querían por ejemplo mandar a imprimir una gráfica o una imagen de los resultados, pues parecía en ocasiones una tarea muy difícil de implementar.

Al evolucionar los dispositivos de entrada y salida, éstos ofrecieron al usuario mayores resoluciones de píxeles y puntos por pulgada. La capacidad de cómputo necesaria para procesar imágenes y gráficos aumentó al paso de los años y en la actualidad pueden procesarse imágenes con cientos de miles de colores en tiempos razonables para el usuario.

Por la década de 1980, los adelantos en dispositivos de entrada y salida y de cómputo, dieron origen a una nueva manera de comunicación del usuario con las aplicaciones mediante una interfaz gráfica y no sólo mediante cadenas de texto; es cuando aparecen las primeras aplicaciones con esta tecnología en el mercado de software, así por ejemplo, aparecen los primeros sistemas operativos con interfaz gráfica como las computadoras de la compañía Apple, y en esta misma época es cuando Microsoft comienza a desarrollar sus primeros sistemas operativos con estos elementos gráficos y que en la actualidad se conocen como Windows.

## 10.1 Definición de GUI

Las interfaces de usuario son aquellos mecanismos que permiten al usuario interactuar con los objetos de su aplicación de manera directa y libre, dando la impresión de utilizar los mismos objetos correspondientes a su interés. En esta definición se usa el concepto de objeto y se reconoce también la división entre aplicación y la interfaz de un sistema.

Los objetos de la aplicación son los objetos semánticos y los de la interfaz, denominados también objetos de la presentación, son los objetos sintácticos.

La Interfaz Gráfica de Usuario son programas que ahora están proliferando, ya que el usuario con sólo teclear, oprimir un botón de ratón, mover íconos y ventanas, puede completar una tarea sin importar el dominio de la aplicación.

Los objetos de la presentación permanecen ocultos e independientes de los objetos de la aplicación, y viceversa.

Los intercambios entre los objetos de la presentación y de la interfaz son a través del paso de mensajes. Por ejemplo, si se tiene un menú de opciones donde el botón*i* selecciona la opción de editor de íconos, cuando sea presionado el objeto botón*i* del menú se pasará un mensaje al objeto editor de iconos, ver la figura 10-1, para la creación y mapeo de una ventana del editor de iconos.

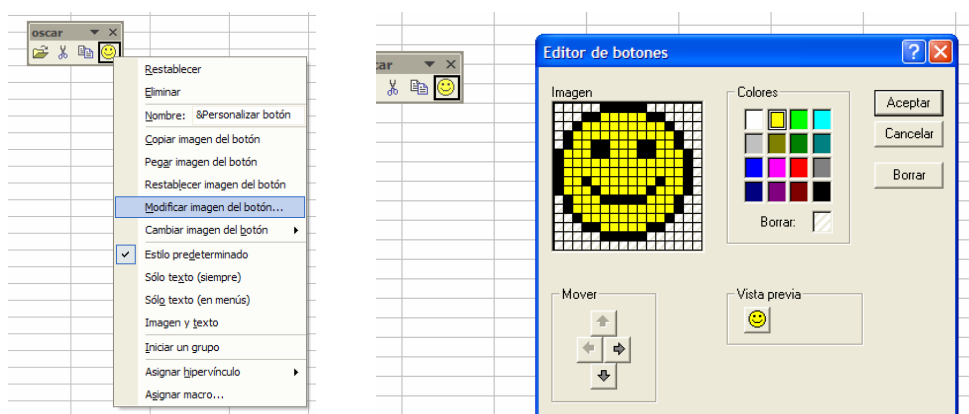


Figura 10.1. Un editor de iconos sencillo para personalizar botones de barras de herramientas de Excel.

Las clases de la presentación en interfaces de usuario son generalmente: etiquetas, botones, (opciones a presionar), menús, menús *pull-down*, despliegues gráficos, separadores, cajas de diálogo, ventanas de texto, campos de texto, etc. Los objetos de presentación instanciarán a ese conjunto de clases.

La jerarquía de clases y la relación de herencia con la orientación a objetos se aplican en estas clases.

Una clase *menú* puede especializarse en un menú *pull-down* o en un menú contextual. Una clase *ventana* con datos encapsulados como: encabezado, tipo de encabezado, límite inferior, límite superior, tipo borde, recorte, etc., y métodos como: apertura, cambio de tamaño, movimientos en todas direcciones, rotación, almacenamiento, cierre,

podrá heredar sus atributos y métodos a una subclase especializada, ventana de texto con datos encapsulados como: tipo de letra, longitud de la cadena, tipo de edición y métodos especializados como: edición y borrado de texto.

La mayoría de las aplicaciones que utilizan interfaz gráfica de usuario proporcionan la extensibilidad a los objetos definidos, para que el usuario pueda construir objetos muy particulares a sus necesidades.

## 10.2 Formularios

Los formularios son generalmente plantillas de acopio de datos para la entrada de una aplicación. Utilizan elementos de una Interface Gráfica de Usuario como botones, cuadros de texto, etc., y pueden ser para aplicaciones *stand-alone* o para el web. En la figura 10.2 tenemos un ejemplo de un formulario.

Centro de Investigación en Computación - Guestbook - Microsoft Internet Explorer

Archivo Edición Ver Favoritos Herramientas Ayuda

Dirección <http://www.ute.icp.mx:8080/oscaruno/curso/librovisitasagb/guestbook.jsp>

**Libro de Visitas**

Nombre: Juana Figueroa  
 e-mail: juana@servidor.unam.mx  
 Página Web (URL): <http://www.dgsca.unam.mx>  
 Asunto: saludos a oscar  
 País / Región: México  
 Provincia: Distrito Federal

**Comentarios y/o ¿Preguntas?**  
 Hola Oscar. Por favor ya apurate a terminar el manual de Técnicas de Programación. Saludos Juana

**¿Opiniones?**  
☒ Yo encontré que el sitio es muy útil.  
☒ El sitio es fácil de navegar.  
☒ Me gusta el diseño del sitio web.

Submit Reset

Figura 10.2. Formulario de acopio de datos de un libro de visitas.

Generalmente los datos seleccionados van a ser almacenados en tablas de una base de datos, o en archivos de texto o binarios, pero también pueden ser procesados mediante la lógica de la aplicación para darle una respuesta al usuario, (ver figura 10.3).

Documento sin título - Microsoft Internet Explorer

Archivo Edición Ver Favoritos Herramientas Ayuda

Dirección [http://www.ute.icp.mx:8080/oscaruno/curso/librovisitasagb/guestbook\\_act.jsp](http://www.ute.icp.mx:8080/oscaruno/curso/librovisitasagb/guestbook_act.jsp)

**Libro de Visitas**

nombre: Juana Figueroa  
 email: juana@servidor.unam.mx  
 url: <http://www.dgsca.unam.mx>  
 asunto: saludos a oscar  
 preguntas: null  
 comentario: Hola Oscar. Por favor ya apurate a terminar el manual de Técnicas de Programación. Saludos Juana

es útil: yes  
 es facil de navegar: yes  
 buen diseño: yes

Gracias, por firmar nuestro Guestbook

[Regresar al libro de visitas](#)  
[Regresar a sitio principal](#)

Figura 10.3. Datos del formulario ya procesados por la aplicación.

Los IDE actuales tienen herramientas para el diseño de estos formularios, ver figura 10.4, como plantillas de componentes donde se pueden seleccionar y ser arrastrados con el puntero de ratón a un contenedor o formulario (Frame) en blanco (sin elementos) para irlos diseñando.

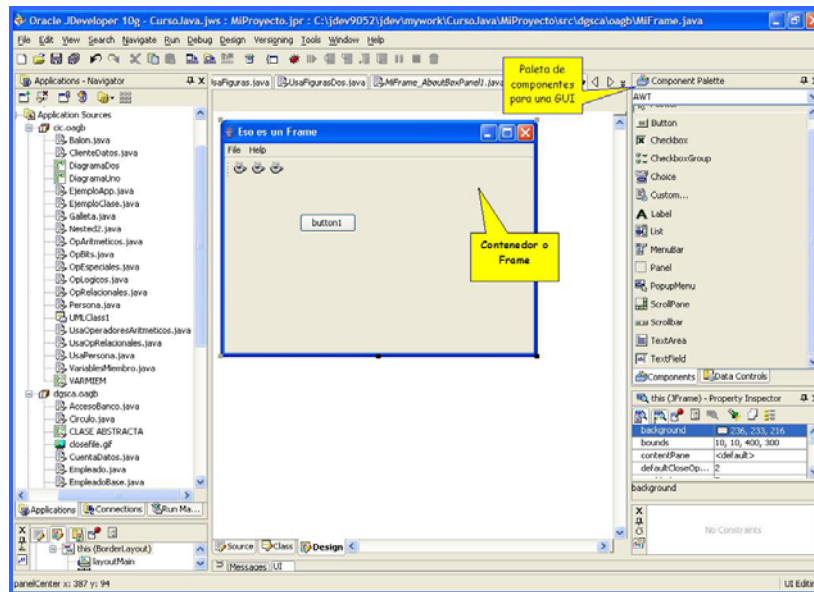


Figura 10.4. Un IDE con una paleta de componentes para diseñar un GUI.

### 10.3 Contenedores

Son clases de objetos que permiten crear objetos gráficos para contener a los componentes, tales como: paneles, cuadros de diálogo, marcos, ventanas, etcétera.

### 10.4 Componentes

Son clases de objetos para crear los objetos gráficos que componen una GUI tales como; botones, listas desplegables, cuadros de texto, casillas de verificación, botones de opción, campos de texto, etiquetas, menús, etc. En los lenguajes de programación orientada a objetos como Java tenemos dos; un conjunto de clases agrupadas en un paquete llamado AWT (Abstract Windows Toolkit) y en el paquete SWING, algunos de estos componentes del paquete AWT están resumidos en la siguiente tabla:

**Las AWT del lenguaje de programación Java y sus componentes:**

| Tipo de componente | Descripción                                                                      |
|--------------------|----------------------------------------------------------------------------------|
| Button             | Es un botón usado para recibir el clic del ratón.                                |
| Canvas             | Un lienzo o panel usado para dibujar.                                            |
| Checkbox           | Cuadro de verificación. Es un componente que le permite seleccionar un elemento. |

| Tipo de componente | Descripción                                                                                             |
|--------------------|---------------------------------------------------------------------------------------------------------|
| CheckboxMenuItem   | Es un cuadro de verificación dentro de un menú.                                                         |
| Choice             | Es una lista desplegable de elementos estáticos.                                                        |
| Component          | Es el padre de todos los componentes AWT, excepto de los componentes de tipo menú.                      |
| Container          | Es el padre de todos los contenedores.                                                                  |
| Dialog             | Es un cuadro de diálogo o una ventana de alto nivel con título y bordes.                                |
| Frame              | Es un marco o ventana y es la clase base de todas las ventanas GUI con controles para ventana.          |
| Label              | Etiqueta. Es una cadena de texto como componente.                                                       |
| List               | Un componente que contiene un conjunto dinámico de elementos.                                           |
| Menu               | Es un elemento dentro de la barra de menú, el cual contiene un conjunto de elementos de tipo menú.      |
| MenuItem           | Un elemento dentro de un menú.                                                                          |
| Panel              | Una clase contenedora básica, usado frecuentemente para crear diseños (layouts) complejos.              |
| Scrollbar          | Un componente que permite al usuario hacer una selección dentro de un rango de valores.                 |
| ScrollPane         | Una clase contenedora que implementa un deslizador horizontal y vertical para un único componente hijo. |
| TextArea           | Un componente que permite al usuario introducir texto en un bloque o rectángulo.                        |

A continuación se presenta un ejemplo de algunas de estas clases:

- **MenuBar**

Un menú es un componente diferente a otros componentes porque no se puede agregar un Menú a los contenedores comunes. Sólo es posible agregar menús a un "menú container". Se puede comenzar un árbol de menú poniendo un **MenuBar** "una barra de menú" en un **Frame**, usando el método **setMenuBar()**. Desde este punto, se pueden agregar menús a la barra de menú y menús o elementos de menú dentro del menú, (ver la figura 10.5):



Figura 10.5. Ejemplo de menú.

Los menús *Pop-up* son una excepción porque éstos aparecen como ventanas flotantes y no requieren un administrador de diseño, pero es importante agregar el menú *Pop-up* al contenedor padre, de lo contrario no funciona.

Los *Help menu* (menús de ayuda) pueden ser implementados en un **MenuBar** mediante el uso del método **setHelpMenu(Menu)**. Se debe agregar el menú que será tratado como un Help menu a la barra de menú; entonces será tratado en la misma forma que un Help menu para la plataforma en que se esté trabajando. Para *X/Motif-type systems*, éste será colocado como una entrada de menú al final derecho de la barra de menú.

El siguiente programa maneja estos diversos tipos de menús y al ejecutarlo, despliega un **JFrame** con un N, y un Help menu. Al darle clic al botón, desplegará un menú **popupMenu**.

```

1. package mypackage1;
2. import javax.swing.JFrame;
3. import java.awt.Dimension;
4. import java.awt.Button;
5. import java.awt.Rectangle;
6. import java.awt.event.ActionListener;
7. import java.awt.event.ActionEvent;
8. import java.awt.PopupMenu;
9. import java.awt.MenuItem;
10. import java.awt.Font;
11. import java.awt.Color;
12. import java.awt.MenuBar;
13. import java.awt.Menu;
14. /**
15.      * Programa de Java que te enseña a utilizar componentes
16.      * del paquete java.awt. Este demuestra el uso de los
17.      * objetos de la clase PopupMenu.
18.      * @autor Oscar A. González Bustamante
19.      * @version 1.0
20.      * Archivo: PopUpMenu.java

```

```

21. */
22.
23. public class PopUpMenu extends JFrame {
24. String [] elementos = {"Nuevo", "Abrir", "Re Abrir", "Eliminar",
    "Guardar", "Cargar", "Salir"};
25. private Button b = new Button();
26. private PopupMenu popupMenu1 = new PopupMenu(); // se instancia un
    PopupMenu
27. private MenuItem menuItem1 = new MenuItem(); // se instancian los
    elementos
28. private MenuItem menuItem2 = new MenuItem(); // del PopupMenu
29. private MenuItem menuItem3 = new MenuItem();
30. private MenuItem menuItem4 = new MenuItem();
31. private MenuItem menuItem5 = new MenuItem();
32. private MenuItem menuItem6 = new MenuItem();
33. private MenuBar menuBar1 = new MenuBar();
34. private Menu a = new Menu ( "Archivo");
35. private Menu e = new Menu ( "Editar");
36. private Menu h = new Menu ( "Ayuda" );
37.
38. public PopUpMenu()
39. {
40.     try
41.     {
42.         jbInit();
43.     }
44.     catch(Exception e)
45.     {
46.         e.printStackTrace();
47.     }
48. }
49. }
50.
51. private void jbInit() throws Exception
52. {
53.     this.getContentPane().setLayout(null);
54.     this.setSize(new Dimension(400, 282));
55.     this.setTitle("Un Pop up Menú");
56.     this.setBackground(Color.blue);
57.     b.setLabel(";dame clic y verás!");
58.     b.setBounds(new Rectangle(80, 60, 195, 65));
59.     b.setActionCommand(";dame clic y verás! ");
60.     b.setFont(new Font("Tahoma", 1, 16));
61.     b.addActionListener(new ActionListener() // oyente al botón
62.     {
63.         public void actionPerformed(ActionEvent e)
64.         {
65.             b_actionPerformed(e);
66.         }
67.     });
68.
69.     popupMenu1.setLabel("Un popup"); // se establecen las etiquetas
70.     menuItem1.setLabel( elementos[0] );
71.     menuItem2.setLabel( elementos[1] );
72.     menuItem3.setLabel( elementos[2] );
73.     menuItem4.setLabel( elementos[3] );
74.     menuItem5.setLabel( elementos[4] );
75.     menuItem6.setLabel( elementos[5] );
76.     popupMenu1.addActionListener( new ActionListener() // oyente al
        botón para el popupMenu
77.     {

```

```

78.         public void actionPerformed(ActionEvent e)
79.         {
80.             popupMenu1_actionPerformed(e);
81.         }
82.     });
83.     this.getContentPane().add(b, null); // se agrega el botón al
        contenedor
84.     popupMenu1.add(menuItem1);
85.     popupMenu1.add(menuItem2);
86.     popupMenu1.add(menuItem3);
87.     popupMenu1.add(menuItem4);
88.     popupMenu1.add(menuItem5);
89.     popupMenu1.add(menuItem6);
90.
91.     menuBar1.add( a ); // se agrega el menú Archivo al MenuBar
92.     menuBar1.add( e ); // sea agrega el menú Editar al MenuBar
93.     menuBar1.setHelpMenu( h ); // agrega un menú de ayuda al MenuBar
94. // agregar el PopupMenu al Contendor padre JFrame
95. // si no funciona
96. // al agregarlo también se agregan todos sus items.
97.     this.getContentPane().add(popupMenu1);
98.
99. }
100.
101.     public static void main( String args[] )
102.     {
103.         PopUpMenu pum = new PopUpMenu();
104.         pum.setVisible( true ); // hace visible al JFrame
105.         pum.setMenuBar( pum.menuBar1 );
106.
107.         System.out.println("Fin del programa");
108.     } // fin del main()
109.
110.     private void b_actionPerformed(ActionEvent e)
111.     {
112.         popupMenu1.show( b , 70, 70); //
            muestra PopupMenu
113.     }
114.
115.     private void popupMenu1_actionPerformed(ActionEvent e)
116.     {
117.         String item = null;
118.         int i;
119.         // maneja el evento de ver cual MenuItem fue
            seleccionado.
120.         for ( i=0; i < elementos.length; ++i )
121.             if ( e.getActionCommand().equals( elementos[i] ) )
122.                 item = new String ( elementos[i] );
123.
124.         System.out.println("comando: " + item );
125.
126.     }
127. } // fin de la clase PopUpMenu

```

#### 10.4.1 Layouts o administradores de diseño

Layouts son clases de objetos que permiten crear objetos que administren el diseño, la distribución y colocación de los objetos componentes dentro de los objetos contenedores. Así por ejemplo, en el lenguaje de programación Java, el objeto **FlowLayout** distribuye los

componentes de izquierda a derecha y de arriba a abajo, y el **BorderLayout** los distribuye en cinco áreas geográficas; norte, sur, este, oeste y centro, etc. Veamos un ejemplo de esto:

- **El BorderLayout**

Es el administrador de diseño por default de la clase **Frame**. Los componentes son agregados a cinco regiones específicas dentro de la ventana o frame:

- NORTH ocupa la parte de arriba
- EAST ocupa el lado derecho
- SOUTH ocupa la parte inferior
- WEST ocupa la parte izquierda
- CENTER ocupa la parte central

Cuando ajustamos verticalmente el tamaño de la ventana o *frame*, las regiones EAST, WEST y CENTER son ajustadas.

Cuando ajustamos horizontalmente el tamaño de la ventana o *frame*, las regiones NORTH, SOUTH y CENTER son ajustadas.

Cuando añadimos botones a las posiciones relativas los botones no cambian si se modifica el tamaño de la ventana; pero los tamaños de los botones si cambian.

El siguiente constructor crea un administrador de diseño de tipo BorderLayout sin espacios entre sus componentes:

```
setLayout(new BorderLayout());
```

Usando el siguiente constructor podemos indicarle los espacios entre los componentes especificados por hgap y vgap:

```
BorderLayout(int hgap, int vgap);
```

Se deben agregar los componentes en las regiones específicas respetando mayúsculas y minúsculas, ya que no es lo mismo **add(boton, BorderLayout.CENTER)** que **add(boton, BorderLayout.center)** en el administrador de diseño, o de otra forma no serán visibles. Si se quiere evitar esto se puede usar **add(boton, "center")**.

Si se deja una región sin utilizar, ésta se comportará como si se hubiera preferido un tamaño de 0 x 0. La región **CENTER** seguirá apareciendo como fondo, incluso cuando no tiene componentes.

Se puede agregar sólo un componente por cada una de las cinco regiones. Si se trata de agregar más de uno, sólo el último agregado será visible. Cuando deseamos agregar más componentes por región, se utiliza la clase Panel.

```
1. package oscar230604;
2.
3. import java.awt.*;
4. import java.awt.event.*;
5.
6. /**
```

```
7. * <p>Titulo: EjemploBorderLayout.java </p>
8. * <p>Descripción: Te enseña a usar el FlowLayout</p>
9. * <p>Copyright: Totalmente libre</p>
10. * <p>Empresa: El patito Feo Inc.</p>
11. * @author Oscar Alejandro González Bustamante
12. * @version 1.0
13. */
14.
15. public class EjemploBorderLayout extends Frame {
16.     Button button1 = new Button();
17.     Button button2 = new Button();
18.     Button button3 = new Button();
19.     BorderLayout borderLayout1 = new BorderLayout();
20.     Button button4 = new Button();
21.     TextArea textArea1 = new TextArea();
22.
23.     public EjemploBorderLayout() {
24.         try {
25.             jbInit();
26.         }
27.         catch(Exception ex) {
28.             ex.printStackTrace();
29.         }
30.     }
31.
32.     void jbInit() throws Exception {
33.         button1.setLabel("Aceptar");
34.         button1.setLocale(java.util.Locale.getDefault());
35.         button1.addActionListener(new
EjemploBorderLayout_button1_actionAdapter(this));
36.         button2.setLabel("Abrir");
37.         button3.setLabel("Cerrar");
38.         this.setSize(400, 300);
39.         this.setBackground(Color.cyan);
40.         this.setResizable(true);
41.         this.setTitle("Ejemplo de BorderLayout - Curso de Java Básico -");
42.         this.setLayout(borderLayout1);
43.         button4.setBackground(Color.red);
44.         button4.setLabel("Aplicar");
45.         borderLayout1.setHgap(20);
46.         borderLayout1.setVgap(40);
47.         textArea1.setColumns(20);
48.         textArea1.setEditable(false);
49.         textArea1.setEnabled(true);
50.         textArea1.setFont(new java.awt.Font("Arial", 3, 12));
51.         textArea1.setLocale(java.util.Locale.getDefault());
52.         textArea1.setSelectionEnd(20);
53.         textArea1.setSelectionStart(20);
54.         textArea1.setText("\t Hola esto es un TextArea  \n " +
55.             "\t dentro de un Frame con un \n " +
56.             "\t BorderLayout en la región \n " +
57.             "\t CENTER");
58.         textArea1.setVisible(true);
59.         this.add(button1, BorderLayout.NORTH);
60.         this.add(button2, BorderLayout.WEST);
61.         this.add(button3, BorderLayout.SOUTH);
62.         this.add(button4, BorderLayout.EAST);
63.         this.add(textArea1, BorderLayout.CENTER);
64.     }
65.
66.     public static void main(String[] args) {
```

```

67.     EjemploBorderLayout ejemploBorderLayout = new EjemploBorderLayout();
68.     ejemploBorderLayout.setVisible( true );
69. }
70.
71. void boton1_actionPerformed(ActionEvent e) {
72.
73. }
74. }
75.
76. class EjemploBorderLayout_boton1_actionAdapter implements
    java.awt.event.ActionListener {
77.     EjemploBorderLayout adaptee;
78.
79.     EjemploBorderLayout_boton1_actionAdapter(EjemploBorderLayout
        adaptee) {
80.         this.adaptee = adaptee;
81.     }
82.     public void actionPerformed(ActionEvent e) {
83.         adaptee.boton1_actionPerformed(e);
84.     }
85. } // fin de EjemploBorderLayout
86.

```

En la figura 10.6, podemos ver que al ejecutar el programa con el comando **java EjemploBorderLayout** presenta una ventana con cuatro botones y un área de texto en la región **CENTER**.

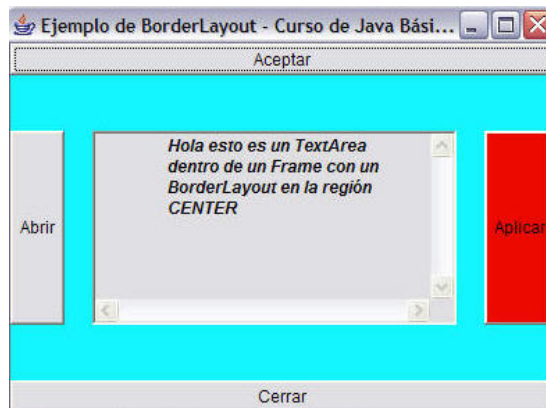


Figura 10.6. Diseño BorderLayout.

## 10.5 Eventos

Son las clases o interfaces que permiten crear objetos para capturar y manejar los eventos. Un evento es una acción sobre algún componente, por ejemplo, clic a un botón, pulsar la tecla Enter en un botón, mover un elemento con las teclas de navegación, eventos especiales como los programados por tiempo, etc. Sin los eventos, las GUI serían interfaces gráficas sin vida, y por lo tanto no serían muy útiles que digamos.

A continuación, examinaremos un ejemplo en el lenguaje de programación Java sobre esto. Los eventos son objetos que describen lo que ha sucedido. Hay diferentes clases de eventos para describir diferentes categorías de acciones por parte del usuario.

- Fuentes de eventos

Una fuente de un evento es el generador de un evento, así por ejemplo, el clic del ratón sobre un componente botón genera un **ActionEvent** con el botón como origen o fuente del evento, ver figura 10–7. La instancia de un **ActionEvent** es un objeto que contiene información acerca de los eventos que acaban de darse. Éste contiene:

- `getActionCommand()` – Devuelve el nombre del comando asociado con la acción.
- `getModifiers()` – Regresa cualquier modificador que se haya dado durante la acción.

- **Manejadores de eventos**

Un *manejador de evento* es un método que recibe un objeto de tipo evento y decide y procesa la interacción con el usuario.

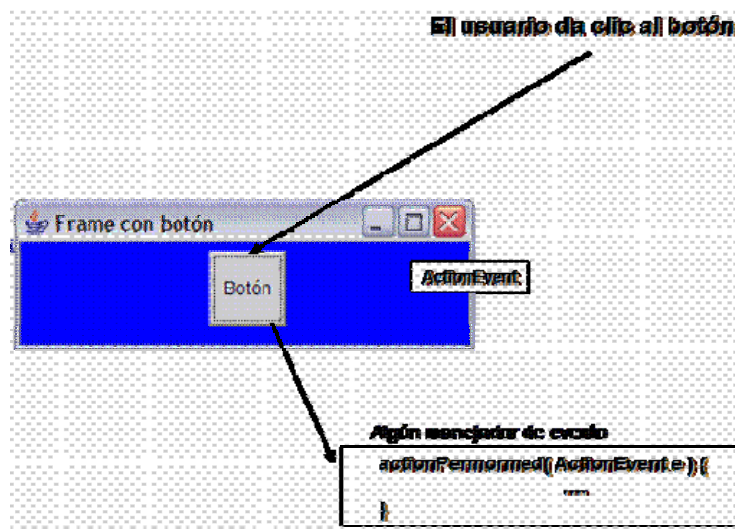


Figura 10.7. Fuentes y manejadores de eventos.

- **El modelo de delegación de eventos**

Con este modelo, los eventos son enviados al componente desde donde el evento fue originado, pero cada componente propaga el evento a una o más clases llamadas oyentes (*listeners*).

Los oyentes contienen *manejadores de eventos* que reciben y procesan el evento, (ver figura 10.8). De esta forma, el manejador del evento puede ser un objeto separado del componente. Los oyentes son clases que implementan la interfaz **EventListener**.

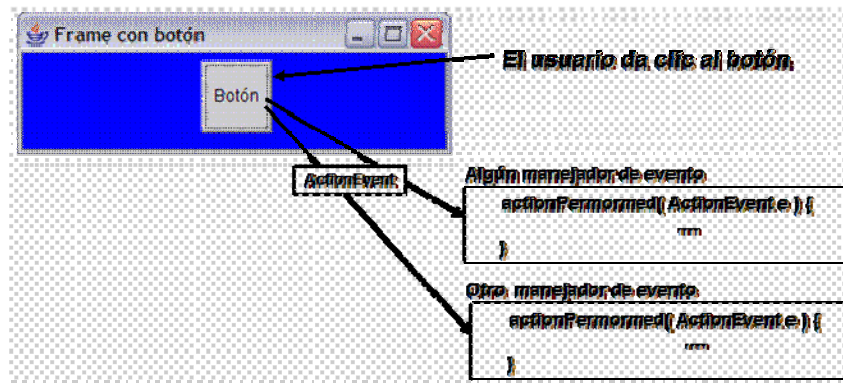


Figura 10.8. Delegación de eventos.

Los eventos son objetos que reportan solamente a los oyentes registrados. Cada evento tiene una interface oyente correspondiente, que le indica cuáles son los métodos adecuados por definirse dentro de la clase para recibir tales tipos de eventos. La clase que implementa la interface define todos esos métodos, que a su vez pueden ser registrados como un oyente.

Los componentes que no tienen oyentes registrados no son propagados.

Por ejemplo, veamos el siguiente código de un *frame* con un simple botón:

```

1. import java.awt.*;
2. import java.awt.event.*;
3.
4. /**
5.  * <p>Título: PruebaBoton.java </p>
6.  * <p>Descripción: Te enseña a usar delegación de eventos</p>
7.  * <p>Copyright: Totalmente libre</p>
8.  * <p>Empresa: El patito Feo Inc.</p>
9.  * @author Oscar Alejandro González Bustamante
10. * @version 1.0
11. */
12.
13. public class PruebaBoton
14. extends Frame {
15. Button button1 = new Button();
16. FlowLayout flowLayout1 = new FlowLayout();
17.
18. public PruebaBoton() {
19.     try {
20.         jbInit();
21.     }
22.     catch (Exception ex) {
23.         ex.printStackTrace();
24.     }
25. }

```

```
26.
27. void jbInit() throws Exception {
28.     button1.setLabel("Botón");
29.     button1.setActionCommand("¡Dame clic y verás que bonito!");
30.     // registrar un oyente al botón
31.     button1.addActionListener(new
        PruebaBoton_button1_actionAdapter(this));
32.
33.     this.setBackground(Color.blue);
34.     this.setTitle("Frame con botón");
35.     this.setLayout(flowLayout1);
36.     this.add(button1, null);
37. }
38.
39. public static void main(String[] args) {
40.     PruebaBoton pruebaBoton = new PruebaBoton();
41.     pruebaBoton.setSize(300, 100);
42.     pruebaBoton.setLocation(300, 200);
43.     pruebaBoton.setVisible(true);
44.     pruebaBoton.button1.setSize(50, 50);
45. }
46.
47. void button1_actionPerformed(ActionEvent e) {
48.     System.out.println("" + e.getActionCommand());
49. }
50.} // fin de la clase PruebaBoton
51.
52.// La clase PruebaBoton_button1_actionAdapter es la
53.// clase manejadora en la cual el evento es delegado.
54.class PruebaBoton_button1_actionAdapter
55.implements java.awt.event.ActionListener {
56.PruebaBoton adaptee;
57.
58. PruebaBoton_button1_actionAdapter(PruebaBoton adaptee) {
59.     this.adaptee = adaptee;
60. }
61.
62. public void actionPerformed(ActionEvent e) {
63.     adaptee.button1_actionPerformed(e);
64. }
65.} // fin de la clase PruebaBoton_button1_actionAdapter
```

La figura 10.9 muestra la ejecución del programa anterior en JBuilder X (versión de prueba). Cuando el usuario le da clic al botón, se dispara el evento y escribe en la ventana de mensajes "¡Dame clic y verás!".

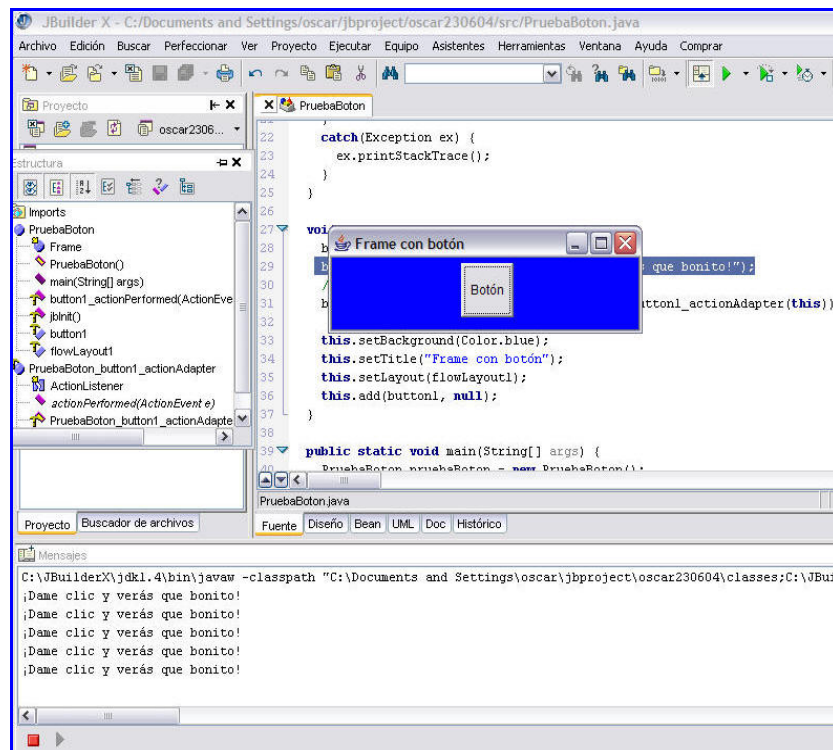


Figura 10.9. Salida del programa PruebaBoton.java.

Este ejemplo tiene las siguientes características:

La clase **Button** tiene un método **addActionListener (ActionListener)**

La interface **ActionListener** define un método **actionPerformed**, el cual recibe un **ActionEvent**.

Una vez creado, un objeto **Button** puede tener un objeto registrado como un oyente (listener) para el **ActionEvents** a través del método **addActionListener()**. El oyente registrado es instanciado desde una clase que implementa la interface **ActionListener**.

Cuando al objeto **Button** se le da clic con el ratón, un **ActionEvent** es enviado. El **ActionEvent** es recibido a través del método **actionPerformed()** para cualquier **ActionListener** que sea registrada sobre el objeto **Button** a través de su método **addActionListener()**.

El método **getActionCommand()** de la clase **ActionEvent** regresa el nombre del comando asociado con esta acción. En la línea 29, la acción del comando para este botón es establecida con "¡Dame clic y verás!".

## Bibliografía

---

- Fundamentos de Programación. Algoritmos,  
Joyanes, L., L. Rodríguez, M. Fernández Estructuras de datos y Objetos. Libro de  
problemas, 2ª Ed., McGraw-Hill, 2003.
- Lenguajes de Programación  
Kenneth C. Louden.  
Internacional Thomson Editores, S.A. de C.V.  
Edición México, 2004.
- Metodología de la Programación.  
Alcalde Eduardo / García Miguel.  
Mc Graw-Hill, 2da. Edición, México 1993.
- Desarrollo de Algoritmos y sus Aplicaciones en Basic, Pascal, Cobol y C.  
Correa Uribe Guillermo.  
Mc Graw-Hill, 3ra. Edición, Colombia 1992
- Metodología de la Programación  
Algoritmos, Diagramas de flujo y Programas  
Cairo Battistutti Osvaldo.  
Computec-Alfaomega Grupo Editor, S.A. de C.V., 1ra. Edición, México 1995
- Fundamentos de Programación – Libro de Problemas  
Joyanes Aguilar Luis, Rodríguez Baena Luis, Fernández Azuela Matilde  
Mc Graw-Hill / Interamericana de España, S.A. 1ra. Edición, España 1996
- Introducción a la Computación y a la Programación Estructurada.  
Levine Guillermo  
Mc Graw-Hill, 2da. Edición, México 1991
- Structured Programing Techniques  
Lectures on Burroughs Implementation of  
Levin Jacques.  
Estados Unidos 1976

- Ingeniería del Software  
Un Enfoque Práctico  
Pressman Roger S.  
McGraw-Hill 1ra. Edición, México 1988
- Understanding Computers - Computer Languages  
Time-Life Books Inc.  
Autores Varios. Estados Unidos 1986
- Computadoras y Programación de Algoritmos  
Vera Badillo Fernando  
Universidad la Salle, México 1994
- Introducción a la Programación Sistemática.  
Writh Niklaus.  
Ed. "El Ateneo", 2da. Edición, Argentina 1986
- Java 2 J2SE 1.4  
Anaya multimedia  
Zukowski John, Madrid España, 2003  
ISBN: 84-415-1559-x
- Aprendiendo Programación Orientada A Objetos en 21 Lecciones Avanzadas  
Sintes, Anthony,  
Pearson Educación, México, 2002,  
ISBN: 970-26-0366-8
- Como Programar en Java Quinta Edición  
Deitel, H.M.,  
Pearson Educación, México 2004,  
ISBN 979-26-0518-0
- Migrating to OO Programming With Java Technology  
SSL-210-V013 - Student Guide  
Sun Microsystems, EUA, 2000
- Aprendiendo UML en 24 horas  
Joseph Schmuller  
Pearson Educación, México, 2000  
ISBN: 968-444-463-X
- Cómo programar en Java  
Enrique Velasco y Javier del Pino  
Prensa Técnica, Madrid España, 1999  
ISBN: 84-89245-69-X
- Perspectivas en la Automatización de las Metodologías de Ingeniería de Software  
Sergio V. Chapa Vergara y Pedro Alday Echavarría.  
Publicación del Departamento de Ingeniería Eléctrica CINVESTAV-IPN, México  
D.F., 1993
- Developing J2EE Compliant Applications  
SFJ-310-V033 – Student Guide  
Sun Microsystems, EUA, 2003

# UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

**DR. JUAN RAMÓN DE LA FUENTE**  
Rector

**LIC. ENRIQUE DEL VAL BLANCO**  
Secretario General

**DR. DANIEL BARRERA PÉREZ**  
Secretario Administrativo

**DRA. ROSAURA RUIZ GUTIÉRREZ**  
Secretaria de Desarrollo Institucional

**MTRO. JOSÉ ANTONIO VELA CAPDEVILA**  
Secretario de Servicios a la Comunidad

**MTRO. JORGE ISLAS LÓPEZ**  
Abogado General

## DIRECCIÓN GENERAL DE SERVICIOS DE CÓMPUTO ACADÉMICO

**DR. ALEJANDRO PISANTY BARUCH**  
Director General

**MAT. CARMEN BRAVO CHAVESTE**  
Directora de Cómputo para la Docencia

**MTRO. JESÚS DÍAZ BARRIGA ARCEO**  
Subdirector de Planeación Académica

**ACT. FABIÁN ROMO ZAMUDIO**  
Subdirector de Tecnología para la Educación

**ING. SERGIO ALVA ARGUINZONIZ**  
Coordinador del Centro Mascarones

**MTRA. ALEJANDRINA SAN JUAN REYES**  
Coordinadora del Centro de Extensión en  
Cómputo y Telecomunicaciones Nuevo León

**LIC. JOSÉ LUIS MEDINA FLORES**  
Coordinador del Centro Coapa de Extensión  
en Cómputo y Telecomunicaciones

**ING. PABLO DE LA O CRUZ**  
Coordinador del Centro San Agustín

**MTRA. ALMA IBARRA OBANDO**  
Responsable de los cursos de Cómputo  
del Palacio de la Autonomía, Fundación UNAM

**LIC. PATRICIA ROMERO ELÍAS**  
Coordinadora del Centro Educativo  
Multidisciplinario Polanco

**LIC. JOSÉ ANTONIO SÁNCHEZ YLLANEZ**  
Subdirector de Comunicación

**DG TOÑA ZIMMERMAN SONABEND**  
Jefa del Departamento de Diseño

*Guías y Textos de Cómputo  
Introducción a la programación  
Mayo de 2007*