Introducción a la Programación Orientada a Objetos

Olinda de Barraza Msc.
Felícita de Krol Msc.
Ludia de Meléndez Msc.
Mitzi de Velásquez Msc.

PEARSON Prentice Hall ®

Introducción a la programación orientada a objetos

Introducción a la programación orientada a objetos

Msc. Olinda Velarde de Barraza
Msc. Mitzi Murillo de Velásquez
Msc. Ludia Gómez de Meléndez
Msc. Felícita Castillo de Krol
Universidad Tecnológica de Panamá

PEARSON Educación ®

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

VELARDE DE BARRAZA, OLINDA y cols.
INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

PEARSON EDUCACIÓN, México, 2006

ISBN: 970-26-0887-2
Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 168

Editor: Hugo Rivera Oliver
e-mail: hugo rivera@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: José D. Hernández Garduño

PRIMERA EDICIÓN, 2006

ISBN 10: 970-26-0887-2

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 - 09 08 07 06

CONTENIDO

Introducción

La programación Orientada a Objetos surge como el paradigma que permite manejar ampliamente las nuevas plataformas que garanticen desarrollar aplicaciones robustas, portables y reutilizables que puedan ofrecer una solución a largo plazo en un mundo donde los cambios se dan a cada momento.

Este libro se realiza con fines didácticos, por la necesidad de ofrecer un material de apoyo para aquellos estudiantes que se inician en el aprendizaje de la Programación Orientada a Objetos en un semestre regular y en él se plantean los conceptos básicos de esta metodología.

Es muy importante que los estudiantes sean capaces, no sólo de manejar los conceptos de orientación a objetos, sino también de aplicarlos de manera efectiva en el desarrollo de programas.

Este es un libro práctico, cada capítulo tiene un objetivo particular, ya que no sólo se explican los conceptos fundamentales sino que se presentan ejemplos, se proponen prácticas y se presentan ejemplos probados en los lenguajes C++ y Java, con los cuales se podrá tener un modelo que les permita probar los conceptos y los formatos de pseudo código aprendidos.

Organización de este Libro

Este libro cuenta con 6 capítulos que describimos a continuación:

- **Capítulo 1 "Introducción a la Programación"**, se revisan los conceptos fundamentales de la programación, antecedentes de la programación orientada a objetos, las etapas para la solución de problemas bajo un enfoque OO, donde se hace énfasis en las etapas de Análisis y Diseño OO, utilizando la técnica de modelado UML. Además se presenta una Metodología de Trabajo que permite al estudiante simplificar los conceptos y establecer los pasos a seguir para identificar las abstracciones.
- **Capítulo 2 "Elementos Fundamentales de la Programación"** se revisan los conceptos tales como: identificadores, variables, tipos de datos y operadores.
- **Capítulo 3 "Estructura de un Programa Orientado a Objetos",** abordaremos los conceptos teóricos y prácticos de los elementos que componen un Programa Orientado a Objetos como son: clases, métodos, objetos, mensajes y definiremos el formato de seudo código que utilizaremos para implementar los algoritmos.
- **Capítulo 4 "Estructuras de Control"**, se definen las estructuras fundamentales para el desarrollo lógico de algoritmos (secuencia, alternativas y de repetición) aplicándolas al desarrollo de algoritmos orientados a Objetos.
- **Capítulo 5 "Arreglos",** se definen los conceptos de arreglos unidimensionales y multidimensionales (haciendo énfasis en las matrices bidimensionales), cómo se manipulan los vectores y matrices, su lectura, impresión, operaciones aritméticas, se explica el paso por referencia, procedimientos de búsqueda en arreglos lineales y los métodos de clasificación.
- **Capítulo 6 "Herencia"** se revisan los conceptos básicos sobre la herencia y cómo se manejan los constructores en la herencia.

Éste es pues, un humilde aporte que será de gran utilidad para el estudiantado, ya que los prepara para las nuevas tendencias en programación que están ya en el mercado y a las cuales se deben enfrentar, de manera que no sientan que se le ha impuesto una nueva cultura que es ajena al conocimiento adquirido.

CAPÍTULO 1

Introducción a la programación

Objetivos:

- Definir los conceptos fundamentales involucrados en el proceso de resolución de un problema a través de la computadora.
- Definir lenguaje, programa, compilador e intérprete.
- Identificar los antecedentes de la programación orientada a objetos.
- Definir los conceptos involucrados en la programación orientada a objetos.
- Identificar las etapas para la resolución de un problema bajo un enfoque orientado a objetos.

Contenido

Encontrar solución a un problema y convertirlo en un programa de computación es una actividad compleja relacionada con una terminología precisa y particular. Por ello, antes de iniciar el aprendizaje de la programación orientada a objetos (POO), es necesario conocer los conceptos implicados en la resolución de un problema mediante una computadora.

1.1 Lenguajes de programación

1.1.1 Definición de lenguaje de programación

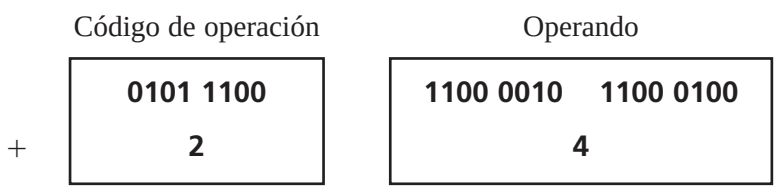
Un lenguaje de programación es una notación constituida por símbolos y reglas que permite escribir programas. Todo lenguaje de programación está compuesto por su sintaxis (reglas) y su semántica (significado de los símbolos y las palabras que utiliza). A través de los lenguajes de programación es posible establecer una comunicación sistematizada y precisa con una computadora.

1.1.2 Tipos de lenguaje

Lenguaje absoluto o de máquina

Es el lenguaje nativo de una unidad de procesamiento central (CPU o procesador). Está compuesto por instrucciones que la máquina entiende directamente y que se expresan en términos de la unidad de memoria más pequeña: el bit (código binario 1 o 0). En esencia, una instrucción es una secuencia de bits que especifican una operación y las celdas de memoria implicadas.

Por ejemplo:



Un lenguaje orientado a la máquina presenta las siguientes ventajas:

- No necesita traducción.
- Se aprovecha toda la capacidad de la computadora (uso de la memoria).
- El tiempo de acceso es menor.

Desventajas:

- Es difícil de escribir y entender.
- Su escritura toma mucho tiempo.
- Dado que es un lenguaje expresado en unos y ceros, se corre un riesgo grande de cometer errores.

Lenguaje simbólico

Es aquél en el cual las instrucciones o sentencias son escritas con palabras similares a las de los lenguajes humanos. Están compuestos por símbolos, letras y números.

Por ejemplo:

If (numero > 0) then print “El número es positivo”

Los lenguajes simbólicos están orientados al programador y presenta las siguientes ventajas:

- Se entienden y escriben fácilmente.
- Disminuye la probabilidad de cometer errores.
- Escribir en lenguajes simbólicos toma menos tiempo.
- Son independientes de la plataforma y, por tanto, pueden usarse en distintas plataformas.

Desventajas:

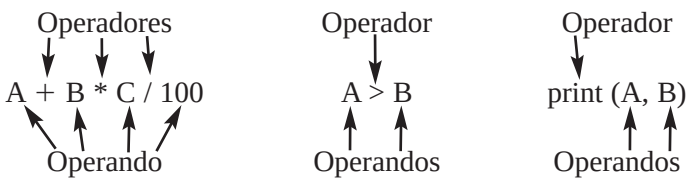
- Para que la máquina los entienda deben ser traducidos.
- Su tiempo de ejecución es mayor.

1.2 Programa

1.2.1 Definición de programa

Un programa es una secuencia lógica de instrucciones escritas en un determinado lenguaje de programación que dicta a la computadora las acciones que debe llevar a cabo. Una *instrucción* es una orden que se le da a la máquina para que ejecute una acción. Está compuesta por dos partes: operando y operador. El *operador* indica el tipo de operación a realizar sobre los datos; el *operando* es el conjunto de valores con los que el operador va a trabajar.

Éstos son algunos ejemplos:



El término *secuencia lógica* se refiere al orden de ejecución de cada instrucción del programa (figura 1.1).

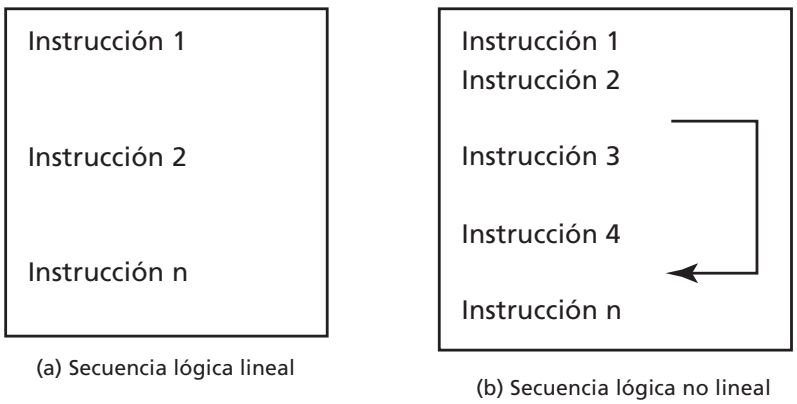


FIGURA 1.1
Secuencia lógica de un programa. [Luis Joyanes Aguilar (1998), Fundamentos de Programación.]

Las secuencias pueden ser de dos tipos:

- *Secuencia lógica lineal*. Las instrucciones se ejecutan en el orden en que aparecen, sin bifurcaciones, decisiones ni instrucciones repetitivas (figura 1.1a).
- *Secuencia lógica no lineal*. Contiene instrucciones de bifurcación que permiten dar saltos hacia delante o hacia atrás dentro del programa (figura 1.1b).

1.2.2 Tipos de Programa

De acuerdo con su función, pueden considerarse dos grandes tipos de programas:

- *Programa fuente*. Es un programa escrito en un lenguaje de programación generalmente simbólico que el programador desarrolla. Para que la

computadora pueda ejecutarlo, primero debe ser traducido (compilado). Algunos ejemplos de programas fuentes son los escritos en C, C++, JAVA, Pascal o Visual Basic.

- *Programa objeto*: Son los programas compuestos por unos y ceros, producto de la compilación de programas fuente. Es el programa que la máquina puede entender directamente.

1.3 Procesadores de lenguajes

Un procesador de lenguajes es el software que traduce los programas fuentes escritos en lenguajes de programación de alto nivel a código de máquina. Los procesadores de lenguaje pueden ser *compiladores* o *intérpretes*.

1.3.1 Compilador

Es un programa, suministrado por el fabricante del lenguaje, cuyo objetivo es convertir el programa fuente en un programa objeto. El compilador realiza las siguientes funciones:

- Traduce las instrucciones del programa fuente.
- Asigna áreas y direcciones de memoria.
- Suministra constantes y otros datos.
- Produce un diagnóstico de errores.
- Genera el programa objeto.

La compilación es, entonces, el proceso mediante el cual la computadora traduce las instrucciones escritas por el programador (programa fuente) a instrucciones escritas en el lenguaje propio de la máquina (programa objeto) (figura 1.2). Existen tantos compiladores como lenguajes hay; por ejemplo: compiladores para C++, C, Visual Basic, Pascal, etcétera.

FIGURA 1.2
Proceso de compilación.

Todo programa fuente debe ser traducido a programa objeto para que pueda ser ejecutado por la computadora.

1.3.2 Intérprete

Es un programa que va leyendo poco a poco el código que el programador escribe y va traduciéndolo y ejecutándolo según se traduce. En estos casos no hay una traducción completa, ya que no se genera un programa directamente ejecutable. Éste permite al programador ejecutar interactivamente las diferentes secciones del programa, en vez de esperar a que todo el programa sea traducido y probado, como lo haría con un lenguaje compilado. Los lenguajes interpretados son menos eficientes que los compilados.

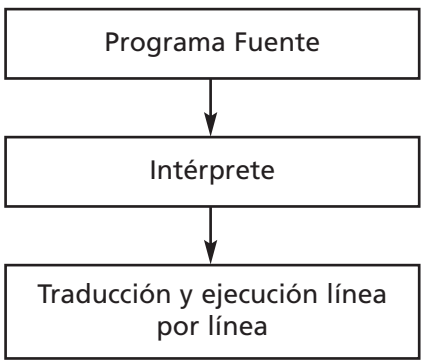


FIGURA 1.3
Intérprete (Luis Joyanes Aguilar, Fundamentos de Programación).

Los siguientes son ejemplos de intérpretes:

Lisp, Smalltalk y Prolog.

1.3.3 Lenguajes interpretados

Existen lenguajes que utilizan mecanismos tanto de compilación como de interpretación; por ejemplo, Visual Basic y JAVA. En el caso de JAVA, existe una "compilación" inicial donde el compilador traduce el código fuente a bytecode. El bytecode no es el programa objeto de ninguna CPU; más bien es un código

intermedio que será posteriormente interpretado (leído y ejecutado) por la máquina.

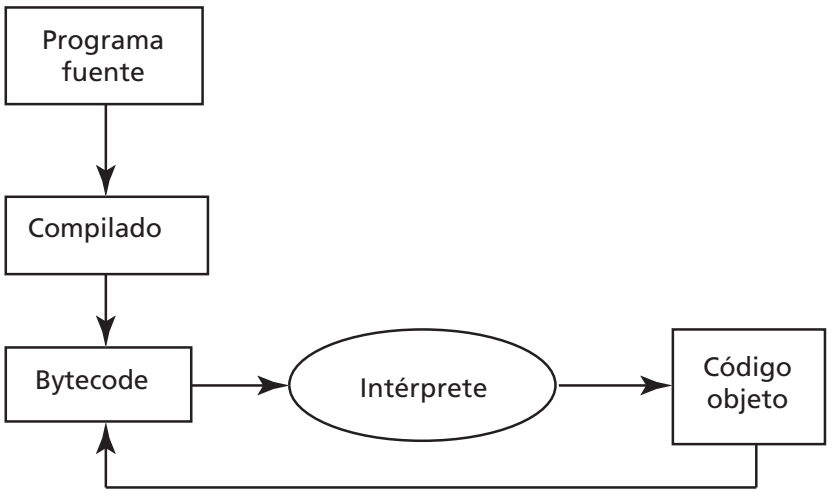


FIGURA 1.4
Proceso de compilación e interpretación en Java.

1.4 Antecedentes de la programación orientada a objetos (POO)

La POO es una metodología para el desarrollo lógico de programas que se remonta a la década de 1960. Sus conceptos fundamentales ya existían en lenguajes como Simula (desarrollado en 1967) y Smalltalk (que data de la década de 1970). A finales de la década de 1960 y principios de la de 1970 surgió una crisis de software derivada, por un lado, de la dificultad de cambiar y modificar los grandes y complejos programas existentes, y por otro, de la disminución de programadores dispuestos a dar mantenimiento a los programas. Esto trajo consigo la introducción de una nueva metodología de desarrollo denominada *programación estructurada*, mediante la cual se descompone en procedimientos individuales (funciones) a los programas. Un procedimiento realiza una tarea específica (esto es una abstracción funcional): este concepto constituye la base sobre el cual trabaja la programación estructurada. En un programa estructurado, cada procedimiento debe realizar una tarea específica; cómo se realiza la

tarea no es tan importante, pues mientras el procedimiento sea fiable, se puede utilizar.

En la programación estructurada es crítica la coordinación entre los desarrolladores implicados en el proyecto, pues dado que cada uno de ellos estará manipulando funciones separadas que pueden referirse a tipos de datos mutuamente compartidos, los cambios de un desarrollador deben reflejarse en el trabajo del resto del equipo.

Una debilidad de la programación estructurada es la separación conceptual de datos y código; esto se agrava a medida que el programa crece y se hace más complejo.

Actualmente, las sofisticadas interfaces de usuarios, los sistemas de ventanas, los programas multitareas, entre otros elementos, hacen más compleja la manipulación de las relaciones existentes entre los componentes de software. Por tanto, se requiere de una nueva técnica que mejore las limitaciones de los sistemas anteriores.

La programación orientada a objetos no es la solución definitiva, pero permite construir programas complejos a partir de entidades de software más simples llamadas *objetos*, que son instancias reales o muestras de clases, lo cual permite heredar datos y códigos de aplicaciones existentes. Esto mejora la productividad del programador y facilita la extensión y la reutilización de clases en otras aplicaciones con mínima modificación al código original.

Así como la programación estructurada o procedural se construye alrededor de funciones, la programación orientada a objetos se construye alrededor de clases. ¿Existe alguna relación entre ambas? Sí: las clases que hacemos son construidas usando elementos de la programación estructurada, llamadas *funciones* (lo que en POO llamamos *métodos*).

1.4.1 Definición

La programación orientada a objetos se puede definir como una técnica o estilo de programación que utiliza objetos como bloques esenciales de construcción. Los elementos básicos de la POO son: objetos, mensajes, métodos y clases.

La POO es una extensión natural de la actual tecnología de programación y representa un enfoque relativamente nuevo y distinto al tradicional. En la POO, los

objetos son los elementos principales de construcción, pero simplemente comprenderlos o usarlos dentro de un programa, no significa que se esté programando en un modo orientado a objetos. Los objetivos de esta modalidad de programación son el mejoramiento de la productividad del programador por medio del manejo de la complejidad del software a través del uso de clases y sus objetos asociados.

1.5 Características de la programación orientada a objetos

La potencia real de los objetos reside en sus propiedades: abstracción, herencia y polimorfismo.

1.5.1 Abstracción

Abstracción de datos. Es la capacidad de crear tipos de datos definidos por el usuario. Es una definición de tipo que incluye datos y funciones (métodos) en una sola estructura definida llamada *clase*. La abstracción de datos es un concepto fundamental en el diseño de un programa, pues permite definir el dominio y la estructura de los datos (atributos), junto con una colección o conjunto de operaciones que tienen acceso a los mismos (métodos). La abstracción permite no preocuparse de los detalles accesorios, y generalizar y centrarse en los aspectos que permiten tener una visión global del problema.

La abstracción y el encapsulamiento están íntimamente relacionados con la clase. El propósito de encapsular una definición de tipo de dato es imposibilitar el acceso a los componentes de los objetos. Equivale a empacar datos y/u operaciones dentro de una sola unidad de programación bien definida. El ocultamiento de información implica que los módulos se caractericen por especificaciones de diseño que los hacen independientes. Este ocultamiento se logra definiendo un objeto como parte privada y proporcionando mensajes para invocar el proceso adecuado; es decir, se dejan ocultos al resto de los elementos del programa los detalles de implementación del objeto. Mediante el ocultamiento se restringe el acceso a datos y métodos a partir de los objetos, dependiendo de los modificadores de acceso públicos, privados o protegidos especificados para la clase correspondiente y a los cuales tenga acceso el objeto.

El encapsulamiento y ocultamiento de información no son elementos separados: el segundo es consecuencia del primero, porque al tener los datos y procedimientos en un módulo, la información allí contenida queda empacada y oculta de cualquier manipulación inconveniente, lo cual permite tener datos seguros y confiables que podrán ser manejados sólo si el objeto lo permite.

En síntesis:

La abstracción es un nuevo tipo de dato definido por el programador; está compuesto por atributos (identificadores o datos) y métodos o acciones que se realizan sobre los atributos, definidos en una sola estructura llamada clase.

CLASE = atributos + métodos

El encapsulamiento permite concebir al objeto como una caja negra en la que se ha introducido toda la información relacionada con dicho objeto (tanto los datos que almacena el objeto como los métodos que permiten manipularlos); mediante el ocultamiento se restringe el acceso a los datos y a los métodos a partir de los objetos.

1.5.2 Herencia

Según Luis Joyanes (1998), la herencia "Es la capacidad para crear nuevas clases de objetos que se construyen basados en clases existentes". La herencia es una propiedad que permite a un objeto poseer propiedades de otras clases. Además, a estos nuevos objetos creados es posible asignarles nuevos atributos y métodos.

La clase que puede ser heredada se denomina *clase base* (*superclase*) y la clase que hereda se denomina *clase derivada* (*subclase*).

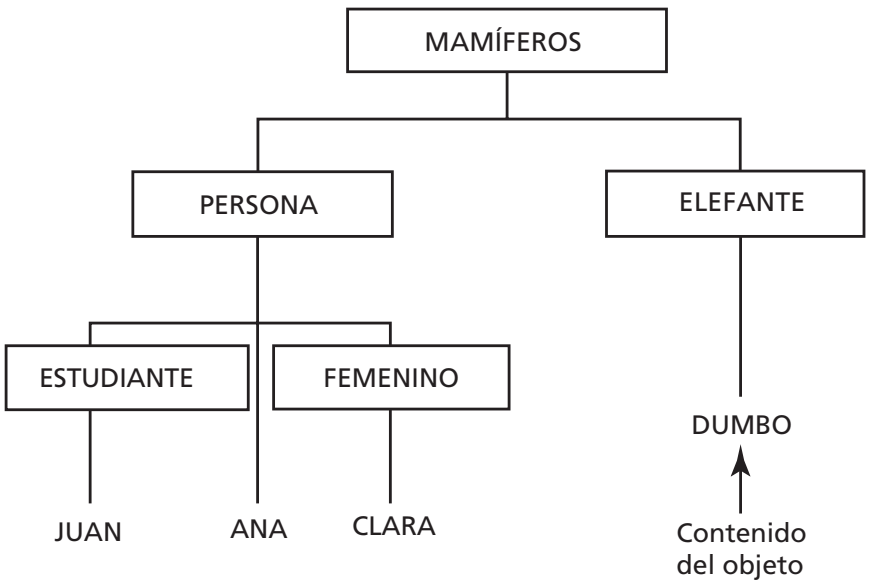


FIGURA 1.5
Herencia.

1.5.3 Polimorfismo

Polimorfismo proviene de dos raíces griegas: *poli*, múltiples y *morfismo*, formas. Esta característica es la capacidad que objetos similares tienen para responder de diferentes formas al mismo mensaje, y permite al programador implementar múltiples formas de un mismo método, dependiendo cada una de ellas de la clase sobre la que se realice la implementación. Esto permite acceder a varios métodos distintos utilizando el mismo medio de acceso (el mismo nombre). El polimorfismo está muy relacionado con la herencia.

1.6 Ventajas de la programación orientada a objetos

Las principales ventajas de la programación orientada a objetos son:

- Las herramientas POO nos ayudan a mejorar la complejidad.
- Mejora la productividad debido a la reutilización de código (herencia).

- Mejora la seguridad y calidad.
- Permite generar programas modulares mejor estructurados.

1.7 Etapas para la solución de problemas por computadora de acuerdo con un enfoque orientado a objetos

Una vez que se plantea una solución específica, es preciso escribirla en un lenguaje de programación —una notación intermedia entre el lenguaje natural y el de la máquina— para desarrollar un programa que después podrá ejecutarse en la computadora. Aunque el proceso de diseñar programas es esencialmente creativo, se pueden señalar una serie de etapas o pasos comunes que generalmente deben seguir todos los programadores. Las etapas para la solución de un problema por computadora de acuerdo con un enfoque orientado a objetos son:

- Definición del problema o dominio del problema.
- Análisis orientado a objetos y diseño orientado a objetos.
- Programación.
- Documentación.

1.7.1 Definición del problema o dominio del problema

Permite describir en forma narrativa o esquemática, de modo claro y concreto, y en un lenguaje corriente el problema que ha de resolverse. Presenta la procedencia y el aspecto de los datos a procesar, y describe los resultados y la manera de presentarlos.

EJEMPLO 1.1

Definición o dominio del problema: Leer dos lados de un triángulo, calcular la hipotenusa $H = \sqrt{a2 + b2}$ y presentar el resultado.

1.7.2 Análisis orientado a objetos y diseño orientado a objetos

El análisis orientado a objetos (AOO) se centra en la investigación del problema, buscando identificar y describir los objetos en el dominio del mismo. A partir de los objetos, que surgen a consecuencia de los requisitos del problema, se definen sus atributos (datos), relaciones y procedimientos (métodos).

En el diseño orientado a objetos (DOO) se procura definir los objetos lógicos (clases) que finalmente serán implementados en un lenguaje de programación. Las clases no son consecuencia de los requisitos del problema, sino de la solución propuesta (dominio de la solución).

Es difícil determinar dónde acaba el AOO y dónde comienza el DOO; lo que algunos autores incluyen en el AOO, otros lo consideran dentro del DOO. El objetivo del AOO es modelar la semántica del problema en términos de objetos distintos pero relacionados. Por su parte, el DOO implica reexaminar los objetos del dominio del problema para refinarlos, extenderlos y reorganizarlos en clases para mejorar su reutilización y aprovechar la herencia. El AOO se concentra en los objetos del dominio del problema; el DOO, en las clases del dominio de la solución.

Para representar las etapas de análisis y diseño orientados a objetos se creó la notación UML (*Unified Modeling Language*, Lenguaje unificado para la construcción de modelos). El UML se define como un "*lenguaje que permite especificar, visualizar y construir los artefactos de los sistemas de Software...*" (Largman, 99) y ahora es un estándar emergente en la industria para modelado orientado a objetos. Nació en 1994, gracias al esfuerzo combinado de Grady Booch y James Rumbaugh para combinar sus famosos métodos: el de Booch y el OMT (*Object Modeling Technique*, Técnica de modelado de objetos), y cuya primera versión se denominó *Unified Metod v. 0.9*.

En 1995, se unió al esfuerzo Ivar Jacobson, creador del método OOSE (*Object Oriented Software Engineering*, Ingeniería de software orientada a objetos), y en 1997 se presentó una notación unificada llamada UML como estándar de modelado.

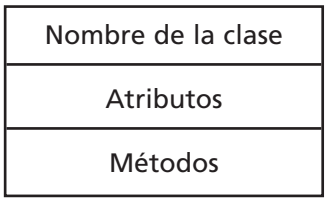
UML permite presentar los diagramas de clase del dominio del problema; no constituye una guía para efectuar el análisis y diseño orientados a objetos, o una lista de procesos de desarrollo a seguir.

Los siguientes son algunos de los elementos más importantes a utilizar en el desarrollo de diagramas de clases con UML.

Diagramas de clases

Permiten describir la composición de las clases que van a estar definidas por atributos, métodos y una especificación de acceso.

Formato:



Atributos. Describen las características de los objetos: tipo de acceso (privado, protegido, publico) y tipo de dato (entero, real, booleano, etcétera).

Métodos. Describen lo que puede hacer la clase; es decir, el método define las instrucciones necesarias para realizar un proceso o tarea específicos. La definición del método se compone de tipo de acceso, tipo de retorno, nombre del método, parámetros, si los requiere, y el cuerpo del método.

Los tipos de acceso se emplean para controlar la visibilidad de los miembros de una clase y pueden ser los siguientes:

Públicos. Son accesibles desde cualquier parte del programa. Se representan con el signo más (+).

Privados. Son accesibles sólo por los métodos que pertenecen a la clase. Se representan con el signo menos (−).

Protegidos. Se utilizan sólo con la herencia; son accesibles por los métodos de la clase base y también por los métodos de las clases derivadas. Se representan por el signo de número (#).

EJEMPLO 1.2

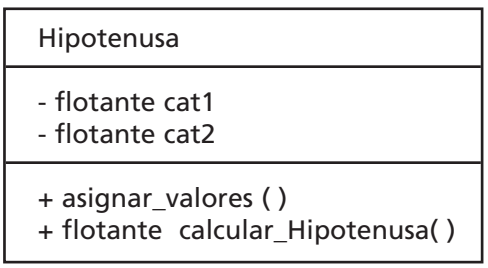
De acuerdo con el enunciado del ejemplo 1.1, se presenta la siguiente metodología para resolver el problema.

Análisis orientado a objetos (AOO)

1. Identificar la abstracción (clase)
   Nombre de la clase: **Hipotenusa**
2. Identificar los atributos de la clase
   Datos: privado flotante cat1
   privado flotante cat2
3. Identificar los métodos de la clase
   Métodos: público asignar_valores( ),
   público calcular_Hipotenusa ( )
4. Aplicar la herencia donde sea necesario.
   En este ejemplo no se aplica.

Diseño Orientado a Objetos

5. Representación del diseño de clases utilizando un diagrama UML:



1.7.3 Programación

Aunque esta etapa consiste en escribir un programa para resolver el problema, hacerlo correctamente no es sólo redactar código en un lenguaje de programación específico. El programa resultante debe ser, además, claro, eficiente y fácil de modificar y usar. Esto implica una disciplina y una metodología de programación mediante las cuales pueda establecerse un estilo de desarrollo que garantice la calidad del producto.

La correcta consumación de esta etapa implica seleccionar una técnica que permita tener una visión clara y detallada de los pasos lógicos a seguir: construir algoritmos, elaborar diagramas de flujo y acción, programar con un estilo funcional,

etcétera. Posteriormente se realiza una prueba de escritorio, a la que siguen la codificación, compilación, ejecución y prueba del programa.

1.7.3.1 Algoritmo

Desarrollar software sin un buen algoritmo es como construir una casa sin los planos necesarios: los resultados pueden ser catastróficos.

Un algoritmo puede definirse como una técnica de solución de problemas que consiste en una serie de instrucciones paso por paso y que produce resultados específicos para un problema determinado. Como *planos* previos a la *construcción* de programas de computación, los algoritmos permiten esquematizar los pasos a seguir usando un lenguaje de especificaciones llamado *seudocódigo*, que requiere menos precisión que un lenguaje de programación formal. De hecho, la computadora no puede ejecutar el seudocódigo de un algoritmo: sólo ayuda al programador a determinar cómo escribir la solución planteada en un lenguaje de programación.

Un buen algoritmo debe ser independiente pero fácilmente traducible a cualquier lenguaje formal de programación. Por tanto, todo algoritmo debe poseer las siguientes características:

a. Debe estar compuesto por acciones bien definidas (especificidad).
b. Debe constar de una secuencia lógica de operaciones.
c. Debe ser finito.

Los algoritmos son técnicas que empleamos en la vida diaria de forma inconsciente y automática para llevar a cabo acciones tan cotidianas como preparar una gelatina, lavarse el cabello, cambiar una llanta, salir del salón de clase, quitarse las medias, ir al cine. Por ejemplo:

Problema: ¿Qué hacer para ver la película *Inteligencia Artificial*?

Algoritmo:

1. Averiguar por medio de una cartelera (en Web o en un periódico) dónde y en qué horario se exhibe esa película concreta.
2. Estar en la taquilla del cine donde se exhibe un poco antes del horario especificado.
3. Comprar una entrada y entrar a la sala correspondiente.

4. Sentarse en la butaca.
5. Ver la película.
6. Salir del cine.

Este algoritmo consta de 6 pasos genéricos, que pueden ser más detallados y específicos para lograr ir al cine a ver esa película.

Un algoritmo orientado a objetos consta de dos partes:

- La definición de la clase, que está compuesta de sus atributos y sus métodos.
- El cuerpo principal, en donde se hace la declaración del objeto, la lectura de datos, la llamada a los métodos de la clase y la impresión de salida, entre otros.

EJEMPLO 1.3

El algoritmo correspondiente al problema planteado en el ejemplo 1.1 tendría los siguientes pasos:

Definir la clase **Hipotenusa**

Declarar los atributos de la clase

Definir el método **asignar_val:** que da valores a los atributos de la clase

Definir el método **calcular_hip:** que calcula la hipotenusa

Crear el cuerpo principal del algoritmo

Declarar el objeto y las variables a utilizar

Leer los datos de entrada

Llamar al método **asignar_val**

Llamar al método **calcular_hip**

Imprimir la hipotenusa calculada

Su seudocódigo sería el siguiente:

```
Algoritmo
/* Definición de la clase  */
   clase Hipotenusa {
    /* Declaración de datos de la clase */
    privado  flotante cat1, cat2
    /* Implementación  de métodos */
      publico  asignar_val ( flotante a, flotante b) {
          cat1 = a
          cat2 = b }
```

```
    publico flotante calcular_hip ( )

     {  flotante h
     √──────────
              h =    cat1 ** 2 + cat2 **2

       retornar h
     }
/* definición del cuerpo principal del algoritmo  */
  INICIO.
    /* Declaración de variables */
    flotante a, b , res

/* se crea el objeto de la clase */
    Hipotenusa obj_hip

/*Crear el objeto con valores leídos */
    IMPRIMIR ( "ENTRAR EL VALOR DE A :")
    LEER (a)
    IMPRIMIR ( "ENTRAR EL VALOR DE B :")
    LEER (b)

   /* Activación de mensajes (llamada a los métodos de la clase)      */
    obj_hip.asignar_val( a, b)
     res = obj_hip.calcular_hip()

  /* Impresión de  salida            * /
    IMPRIMIR ("HIPOTENUSA ES :", res)

  FIN.
```

1.7.3.2 Prueba de escritorio

Consiste en examinar la solución exhaustivamente con el fin de que produzca los resultados deseados; al mismo tiempo, la prueba detecta, localiza y elimina errores. Debe considerar varias posibilidades de valores de entrada para garantizar que éstos produzcan siempre un resultado correcto. Trate de incluir valores no muy comunes, para analizar el comportamiento del algoritmo bajo condiciones extremas.

Por ejemplo, suponga que un algoritmo requiere que el usuario ingrese un valor para buscar su raíz cuadrada. El usuario no debería introducir un valor negativo porque la raíz cuadrada de un número negativo es imaginaria. Sin embargo, usted debe probar qué hará el algoritmo si el usuario lo hace. Otra posibilidad de ingreso que siempre debe tomarse en cuenta para una división es un cero como divisor.

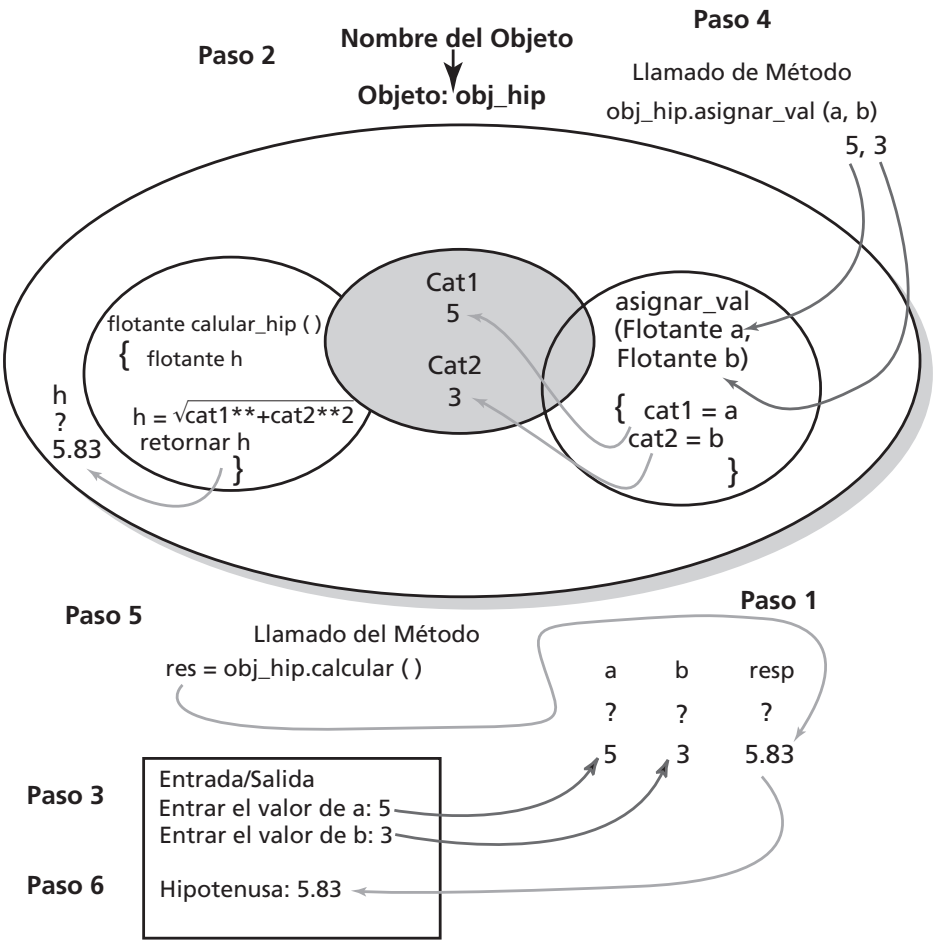


FIGURA 1.6
Desarrollo de una prueba de escritorio para cálculo de la hipotenusa.

1.7.3.3 Codificación

La codificación implica convertir el algoritmo resultante en un código escrito en lenguaje de programación. Al resultado se le denomina programa fuente.

EJEMPLO 1.4

Partiendo del ejemplo 1.3, el algoritmo se traduce a dos lenguajes de programación: C++ y Java.

Programa en C++

```
#include <math.h>
 #include  <iostream.h>
 #include  <conio.h>
 class Hipotenusa {
     float cat1, cat2;
  public:
    void asignar_val(float
    ➥a , float b)
     {  cat1 = a ;
         cat2 = b;
     }
    float calcular_hip( )
     {  float h ;
         h = sqrt (pow-
         ➥(cat1,2) + pow
         ➥(cat2, 2));
         return(h); }
    };

// Programa Principal
      void main( )       {
         float a, b;
         float res;
         clrscr( );
       /* se crea el objeto */
         Hipotenusa obj_hip;
         cout << "Entrar el
         ➥valor de a : ";
         cin>> a;
         cout << "Entrar el
         ➥valor de b : ";
         cin>> b;

        // Activación de
           ➥mensajes
         obj_hip.asignar_
         ➥val(a, b);
          res = obj_hip.calcular
          ➥hip();

          cout<<"Hipotenusa es
          ➥= "<< res;
          getch();
        }
```

Programa en Java

```
 import java.io.*;
class Hipotenusa
{ private float cat1,cat2;
    public void asignar_
    ➥val(float a, float b)
  { cat1=a;
    cat2=b; }

  public double calcular_hip()
  {double h;
   h=Math.sqrt(Math.pow(cat1,2)+
Math.pow(cat2,2));
   return (h);
   }
  }
  //Programa Principal

class Hipotenusap
{ public static void main
➥(String [] arg) throws
➥IOException {

  BufferedReader br = new
  ➥BufferedReader (new Input-
  ➥StreamReader (System.in));
  float a,b;
  double res;
    System.out.println("Entrar
    ➥el valor de a:");
  a= Float.parseFloat(br.read-
  ➥Line());
  System.out.println("Entrar
  ➥el valor de b:");
  b= Float.parseFloat(br.read-
  ➥Line());

  /* se crea el objeto   */
  Hipotenusa obj_hip = new
  ➥Hipotenusa();

  //Activación de mensajes
  obj_hip.asignar_val(a,b);
  res=obj_hip.calcular_hip();
  System.out.println("Hipotenusa
  ➥es = "+res);
   }
 }
```

1.7.3.4 Compilación y Ejecución

Compilación

Una vez que el algoritmo ha sido convertido en un programa fuente, es preciso compilarlo —traducirlo— en un programa objeto. Si tras la compilación se presentan errores (de sintaxis) en el programa fuente, éstos se deben corregir para poder volver a compilarlo.

Ejecución

La ejecución de un programa consiste en que la computadora procese cada una de las instrucciones que el primero contenga. Debe emplearse una amplia variedad de datos de entrada, conocidos en esta fase como *datos de prueba*, para determinar si el programa tiene errores. Esta gama de datos de prueba debe incluir:

- Valores normales de entrada.
- Valores extremos de entrada que comprueben los límites del programa.
- Valores que comprueben aspectos especiales del programa.

1.7.3.4.1 Tipos de errores

Durante el desarrollo de un programa es necesario poner especial cuidado en evitar que el producto obtenido presente errores que lo hagan inservible. Los errores se clasifican de la siguiente manera:

- *Errores de compilación*. Los errores en tiempo de compilación, o errores sintácticos, se derivan del incumplimiento de las reglas sintácticas del lenguaje como, por ejemplo, una palabra reservada mal escrita o una instrucción incompleta. Si existe un error de sintaxis, la computadora no puede comprender la instrucción y no puede generarse el programa objeto. El compilador despliega una lista de todos los errores encontrados.
- *Errores de ejecución*. Estos errores se producen por instrucciones que la computadora puede comprender pero no ejecutar; por ejemplo, la división entre cero o el cálculo de raíces cuadradas de números negativos. En estos casos se detiene la ejecución del programa y se imprime un mensaje de error. Son más difíciles de detectar y corregir que los errores

sintácticos, ya que ocurren o no dependiendo de los datos de entrada que se utilicen.

- *Errores de lógica*. Consisten en resultados incorrectos obtenidos por el programa. Son los más difíciles de detectar, ya que el programa no puede producir errores de compilación ni de ejecución, y sólo puede advertirse el error comparando los resultados del programa con los obtenidos por un método distinto que se considere confiable (por ejemplo, operaciones matemáticas hechas *a mano*). En este caso se debe volver a la etapa de diseño, modificar el algoritmo, cambiar el programa fuente y compilar y ejecutar una vez más.

1.7.4 Documentación

La documentación de un programa consta de las descripciones de los pasos a seguir en el proceso de resolución de un problema. La importancia de la documentación debe ser destacada por su decisiva influencia en el producto final. Los programas deficientemente documentados son difíciles de leer, más difíciles de depurar y casi imposibles de mantener y modificar.

La documentación de un programa se hace a través de las líneas de comentarios, y se incluyen tantas como sean necesarias para aclarar o explicar el significado de las líneas de código que no son obvias, especialmente en lo que respecta a:

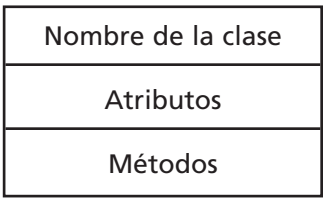
- Identificadores y estructuras de datos declaradas.
- Estructuras de control.
- Métodos y sus parámetros.

En resumen:

Una solución orientada a objetos implica los siguientes pasos necesarios:

Análisis y diseño orientado a objetos

- Identificar las abstracciones (clases).
- Identificar los atributos (datos) de cada abstracción.
- Identificar las operaciones (métodos) de cada abstracción.
- Aplicar la herencia donde sea necesario.
- Construir el diagrama de clases.



Programación

- Desarrollo del algoritmo.
- Prueba del algoritmo (prueba de escritorio).
- Codificación del algoritmo en un lenguaje de programación.

CAPÍTULO 2

Elementos fundamentales de la programación

Objetivos:

- Identificar los tipos de datos que se emplean en el desarrollo de un programa.
- Definir los conceptos de variables, expresiones y constantes.
- Describir el procedimiento para declarar los formatos de las sentencias de asignación y de entrada y salida.
- Resolver problemas relacionados con tipos de datos y expresiones aritméticas.

Contenido:

La programación orientada a objetos permite construir una aplicación utilizando como herramienta a las clases, que, a su vez, permiten generar objetos. El resultado de esto es un programa, una estructura lógica integrada por un conjunto de elementos de construcción (variables, expresiones, instrucciones, etcétera) que permite dar solución a un problema. Al interior de los programas, esos elementos se combinan de acuerdo con ciertas reglas con el fin de que funcionen y puedan ser entendidos por la computadora.

En este capítulo se describen los elementos básicos de programación comunes a casi todos los lenguajes: tipos de datos, variables, constantes, expresiones, sentencias de control, además de los elementos característicos de la programación orientada a objetos: clases, métodos, objetos y mensajes.

2.1 Tipos de datos

En términos generales, un dato es la representación simbólica de un atributo de una entidad; en programación, los datos expresan características de las entidades sobre las que opera un algoritmo. Los datos representan hechos, observaciones, cantidades o sucesos y pueden tomar la forma de números, letras o caracteres especiales. Por ejemplo, la edad y el domicilio (atributos) de una persona (entidad). Los datos pueden ser procesados por la computadora para producir nueva información.

Un tipo de dato define el conjunto de valores que es posible crear y manipular, y las operaciones que con ellos pueden llevarse a cabo. Cada variable, constante o expresión tiene asociado un tipo de dato que determina el conjunto de valores que puede tomar. La figura 2.1 muestra claramente los tipos de datos principales clasificados en dos grandes grupos: *simples* o *primitivos* y *compuestos* o *estructurados*.

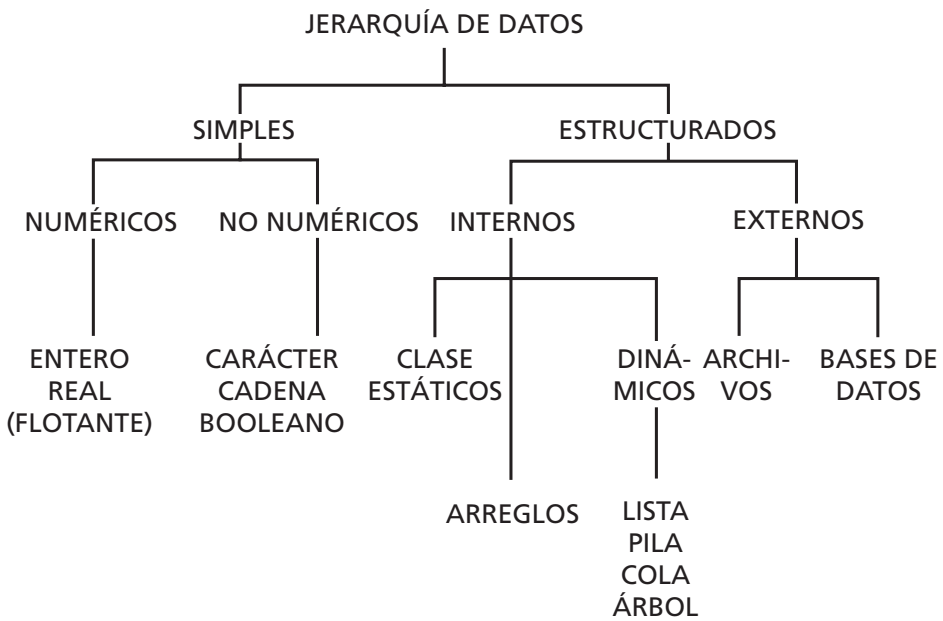


FIGURA 2.1
Clasificación de los tipos de datos principales.

1 *Entero*. Es un valor numérico sin punto decimal. El valor más grande que puede asumir está definido por el espacio que al dato se le haya especificado usar. Puede ir precedido por + o –, lo cual le permite representar tanto valores negativos como positivos. Los siguientes son ejemplos de datos de tipo entero: 2003, –54, +3.

2. *Real*. Es un valor numérico con punto decimal. El valor más grande que puede asumir está definido por el espacio que al dato se le haya especificado usar. Puede ir precedido por + o –, lo cual le permite representar tanto valores negativos como positivos. Se expresa de dos maneras distintas denominadas: notación de punto fijo y notación exponencial.

   a) Notación de Punto fijo (flotante): es aquella que sólo maneja números del 0 al 9 y el punto decimal. Maneja un máximo de 10 dígitos.

   b) Punto Flotante (flotante): un valor en punto flotante tiene la forma "mantisa exponente E", donde la mantisa es un número real y exponente un número entero el cual representa la cantidad mantisa multiplicada por 10 elevado al exponente. Su representación máxima es de 14 dígitos

   La diferencia entre punto fijo y punto flotantes es la cantidad de valores que se puede almacenar y que dependerá del microprocesador que se utilice. Ejemplos de datos de tipo carácter son 129.22, 0.005, 3.6×10^{-4951}.

3. *Carácter*. Este tipo de dato acepta un solo carácter, alfanumérico (un dígito o una letra mayúscula o minúscula), símbolo o carácter especial, que se encuentre en el conjunto ASCII ampliado. El valor debe ir entre apóstrofos. Ejemplos de datos de tipo carácter son: 'A', 'b', '9' y '/'.

4. *Cadena*. Este tipo de dato aloja un valor formado por una sucesión de caracteres (letras, dígitos y caracteres especiales). El valor debe ir entre comillas dobles. Ejemplos de datos de tipo cadena son: "Universidad Tecnológica", "$123.67" o "17/03/2003".

5. *Booleano o lógico*: Este tipo de dato puede alojar uno de dos valores que representan falso o verdadero.

Problemas propuestos:

Valores	*Tipo de datos*	*Correcto S/N*	*Si es incorrecto, ¿por qué?*
5280	Entero	S	
2.0e-8			
1000,0			
'A'			
'verde'			
"8:15 p.m."			
"R"			
$77.7			
+12.34			
'+'			
"–127.5"			
2.0e+10.2			

2.2 Identificadores

Los identificadores son palabras creadas por el programador para denominar los elementos que necesita declarar en un programa, tales como variables, clases, métodos, objetos y estructuras de datos.

Los identificadores deben crearse de acuerdo con las siguientes reglas:

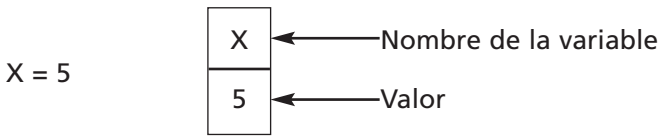
1. Deben comenzar con una letra.
2. Pueden combinar letras, números y el carácter de subrayado (_).
3. Los identificadores de clases comienzan con mayúscula; para cualquier otro elemento deben iniciar con minúscula.

Algunos ejemplos de identificadores son `x1`, `Estado`, `edad`, `nombre`, `num_seguro`, `calcular_Suma`, `Cpersona`.

2.3 Variables y constantes

2.3.1 Variables

Una variable es un área de almacenamiento temporal a la que se ha asignado un nombre simbólico y cuyo valor puede ser modificado a lo largo de la ejecución de un programa. Por ejemplo:



2.3.2 Constante

Una constante es un valor definido que no cambia durante la ejecución de un programa. Por ejemplo: 3.1416, 3, “Universidad”, ‘a’ o –256.

2.4 Declaración de variables

Todos las variables deben ser declaradas antes de ser utilizadas. Esto significa que el compilador debe conocer, *por adelantado*, la variable y el tipo de dato que se estará utilizando. En términos generales, la sintaxis que se emplea para declarar una variable es

```
Tipo_dato  iden1, iden2,...,iden n
```

Una variable queda determinada por:

- *Nombre*. Permite referirse a la misma; el nombre es su identificador.
- *Tipo de dato*. Permite conocer qué valores se pueden almacenar en ella.
- *Rango*. Determina el conjunto de valores que puede admitir.

En resumen:

- Existen cinco tipos básicos de datos: *entero* (para valores numéricos sin punto decimal), *real* (para valores numéricos con punto decimal), *carácter* (para un solo símbolo o carácter), *cadena* (para una sucesión de símbolos o caracteres) y *booleano* (para uno de dos valores: falso o verdadero).

2.5 Expresiones y operadores aritméticos

Una *expresión aritmética* es un conjunto de valores, constantes, variables y/o funciones combinadas con operadores aritméticos. Estas expresiones producen resultados de tipo numérico y se construyen mediante operadores aritméticos y funciones matemáticas intrínsecas. Un *operador* es un símbolo que le indica al compilador desarrollar una operación matemática o lógica específica.

La siguiente tabla lista los operadores aritméticos:

Operadores	*Descripción*	*Ejemplo*
+	Suma	A + B
–	Resta	A – B
*	Multiplicación	A * B
/	División	A / B
**	Exponenciación	A ** B
( )	Se utiliza para alterar el orden jerárquico de evaluación de los operadores.	(A+B)/2

Los operadores aritméticos deben aplicarse siguiendo cierta jerarquía:

1) ( ) paréntesis
2) ** exponenciación
3) * / multiplicación , división
4) + – suma, resta

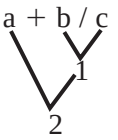
Las operaciones con igual jerarquía, se resuelven de izquierda a derecha. Por ejemplo, la expresión matemática

$$a + \frac{b}{c}$$

en la computadora se convierte en:

$$a + b / c$$

El orden de evaluación de la siguiente expresión aritmética es:



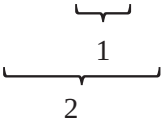
Problemas de operadores aritméticos

Convierta a expresión de computadora y grafique el orden de evaluación:

Por ejemplo:

$$\frac{B}{C+D}$$

se expresa como B/ (C+D)C



Expresión matemática	
a) $(A+B)\frac{C}{D}$	g) $\left(\frac{a}{b}\right)^{x-1}$
b) $A+B.C$	h) $x + \frac{y}{2} + z$
i) $Y^3 + 4$	i) $6 + \frac{a}{b} * \frac{c}{d}$
e) $A^{x+3} - F$	j) $a - \frac{b}{x + y}$
f) $\frac{a - b}{c - d}$	k) $\frac{1}{2}\left(x - \frac{a}{x}\right)$
g) $(x^2 - y^2)^3$	

2.6 Asignación

Es la operación de asignar o cambiar un valor a una variable. Se lleva a cabo de acuerdo con la siguiente sintaxis:

nombre_de_variable = expresión

La asignación se lleva a cabo evaluando la expresión y asignando el resultado obtenido a la variable, definida a la izquierda del símbolo de asignación. Por ejemplo:

y = m + 3.25 – 11.0 / b

x = 5.7

2.7 Entrada/salida

Para que la computadora pueda realizar cálculos requiere de los datos necesarios para ejecutar las operaciones; es decir, necesita una *entrada* que posteriormente se convertirá en resultados o *salida*. En términos muy genéricos, la inserción de datos de entrada a la computadora se conoce como *lectura* y la transferencia de datos hacia un dispositivo de salida de una computadora se denomina *escritura*.

2.7.1. Entrada

Las operaciones de entrada permiten leer valores y asignarlos a determinadas variables. Los datos se introducen a la computadora mediante dispositivos de entrada (teclado, unidades de disco, escáneres, etcétera) y son almacenados en las direcciones de memoria asignadas a las variables.

En términos generales, el proceso se lleva a cabo dentro de la siguiente estructura:

`Leer (Var1, Var2, Var3,...,VarN)` donde Var# es el nombre de una variable

La figura 2.2 ilustra la función de un segmento de código de entrada como el siguiente:

```
/* Declaración de variables  */
          cadena nombre
          flotante  salario
          /* Proceso de lectura de datos  */
          Leer (nombre, salario)
```

Datos de entrada:

nombre	salario
Pedro	550.50

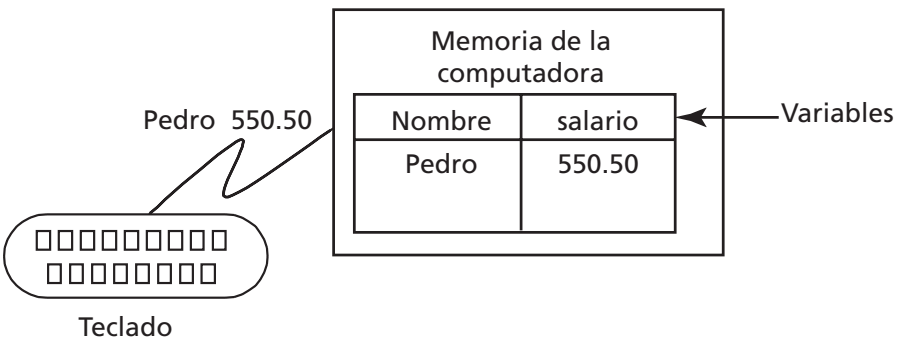


FIGURA 2.2
Diagrama de lectura.

2.7.2. Salida

Las operaciones de salida permiten desplegar o imprimir los resultados o valores guardados en memoria en un dispositivo de salida (pantalla, impresora, trazador, etcétera). Se llevan a cabo bajo el siguiente esquema genérico:

```
Imprimir (lista de variables y/o constantes)

Imprimir ("cadena Constante")
Imprimir (var1,..., varn)
Imprimir( )
Imprimir ("cadena constante", var)
```

Donde las variables y constantes van separadas por comas.

La figura 2.3 ilustra la función de un segmento de código de salida (impresión) como el siguiente:

Imprimir ("Nombre del Empleado = ", nombre, "Sueldo =", salario)

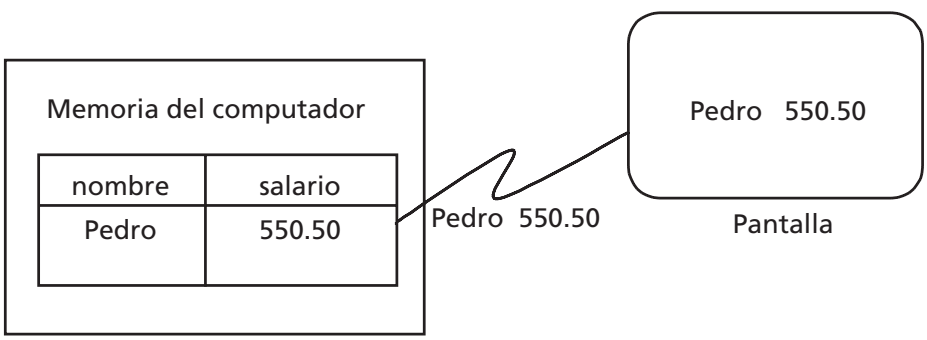
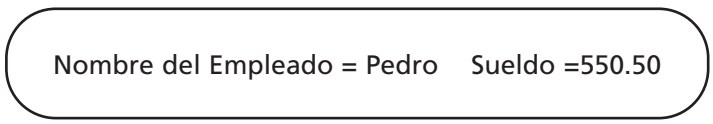


FIGURA 2.3
Diagrama de impresión.

El resultado en pantalla sería:



Problemas propuestos

Escriba una instrucción que lea el nombre, cédula y edad del estudiante.

Escriba una instrucción que imprima el nombre del estudiante.

Escriba una instrucción que imprima el mensaje: **UNIVERSIDAD TECNOLÓGICA**.

Escriba las instrucciones necesarias para lo siguiente: leer el nombre y dos notas del estudiante, calcular la nota promedio y después imprimir el nombre y la nota promedio.

Escriba las instrucciones necesarias para obtener la siguiente salida:

```
Compañía Buenas Ventas
Nombre
Departamento
Salario
```

Capítulo 3

Estructura fundamental de un programa orientado a objetos

Objetivos:

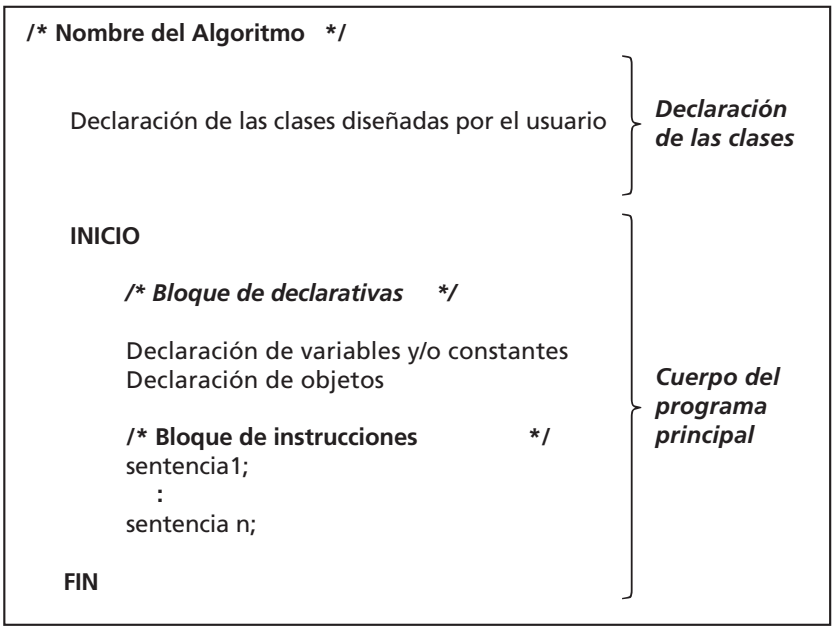
- Identificar los elementos que componen el modelo de programación orientada a objetos (POO).
- Definir los conceptos de clases, métodos, objetos y mensaje.
- Declarar los formatos de clase, métodos, objetos y mensaje.
- Desarrollar ejemplos de los conceptos de clases, métodos, objetos y mensajes.
- Resolver problemas en donde se manejen los conceptos de clase, métodos y objetos a través de seudocódigos.

Contenido

Para desarrollar un programa específico de manera lógica y completa es necesario conocer los elementos que lo conforman. Como metodología de programación, la POO establece una manera particular de agrupar los componentes y organizar su estructura para posteriormente desarrollar aplicaciones. La POO opera con base en un conjunto de objetos que interactúan entre sí y que deben estar organizados en clases. La POO emplea un formato que permite definir la solución de un problema: el seudocódigo.

3.1 Formato de escritura de un algoritmo a través de un seudocódigo

El siguiente esquema ilustra el formato general en seudocódigo para plantear la solución de un problema bajo un enfoque orientado a objetos:



Este formato está compuesto por dos partes fundamentales:

- *Área de declaración de clases definidas por el usuario*. Se definen las clases creadas por el usuario; debe haber por lo menos una.
- *Área del programa principal o cuerpo del programa.* Es el bloque de instrucciones que permitirán la utilización de las clases, además de otras tareas para la solución del problema.

3.1.1 Reglas para la escritura del seudocódigo

1. Todas las clases deben ser definidas antes del cuerpo del programa principal.

2. El cuerpo del programa principal debe empezar con la palabra INICIO y finalizar con la palabra FIN.
3. El seudocódigo debe ser escrito en letra de imprenta.
4. Los comentarios se harán en una de dos formas:
   a) /* ….. comentario ……. */
   b) // ……comentario…..
5. Apegarse a los formatos establecidos para cada instrucción.

3.2 Clase

Todo programa orientado a objetos basa su construcción en un elemento fundamental que es la clase, la cual es manipulada por los objetos a través de los métodos. La clase es un tipo de dato definido por el usuario que especifica las estructura de los datos y los métodos correspondientes a ese tipo de dato particular. Existen un par de definiciones de clases que pueden ilustrar mejor su naturaleza:

> "Son los moldes con los que se producen los objetos." (Deitel/Deitel.)
>
> "Es un término técnico que se aplica a los lenguajes orientados a objetos para describir grupos de estructuras de datos caracterizados por propiedades comunes." (Meyer.)

Una clase se compone de atributos y métodos, y es posible definir muchos objetos de la misma clase. Todos los miembros de una clase son globales dentro de la clase, independientemente del tipo de acceso que tengan; es decir, todos los métodos pueden utilizar los atributos de la clase.

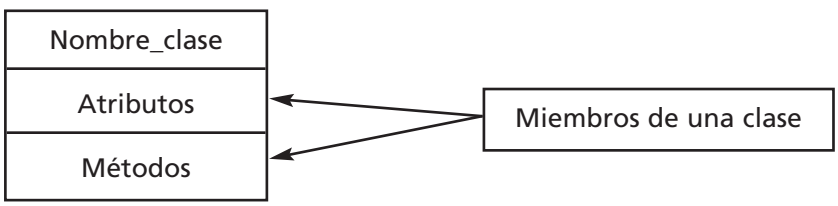


FIGURA 3.1
Estructura de una clase.

En seudocódigo, la creación de clases sigue la sintaxis que se ilustra a continuación:

```
clase  Nombre_clase {
    Acceso Tipo de dato  nombre_Identificador_1
                    :
    Acceso Tipo de dato  nombre_Identificador _n

    Acceso *Tipo de dato nombre_del_método_1  ( lista de parámetros )
                    :
    Acceso *Tipo de dato nombre_del_método_n  ( lista de parámetros )
    }

Nota: * Tipo de dato a retornar por el método
```


3.2.1 Modos de acceso

Determinar la forma de acceso a los miembros de una clase permite mantener la característica de ocultamiento en POO. Normalmente, es buena práctica restringir el acceso a los datos de una clase y a otra información interna empleada para definir el objeto. Este ocultamiento de información es el proceso mediante el cual se hace disponible para el programa sólo la cantidad de información sobre la clase que éste requiere para usarla.

Para usar la clase, los programas deben conocer la información que la compone, es decir, sus datos y los métodos que los manipulan. Los programas no necesitan saber cómo trabajan los métodos sino, más bien, las tareas que cumplen.

Miembros públicos, privados y protegidos

Los miembros de una clase pueden ser *privados*, *públicos* y *protegidos*, y se pueden aplicar a los atributos (datos) y métodos de una clase. Los miembros públicos, privados o protegidos permiten ocultar la información dentro de los programas.

Los programas no pueden acceder, a través de los objetos, a los miembros privados de la clase. Si el objeto intenta acceder a un miembro privado (por ejemplo, obj_clase.x = 5, donde x es un dato privado) el compilador generará un mensaje de error. En vez de esto, el programa debe invocar un método de la clase para asignar los valores del miembro dato x.

Al evitar que los usuarios tengan acceso directo a los miembros datos, usted puede asegurarse de que el programa siempre valide los valores que intenta asignar a los datos.

Los objetos pueden acceder a los miembros públicos desde cualquier parte del programa. Por otra parte, sólo se puede acceder a los miembros privados y protegidos utilizando funciones (métodos) de la clase.

A continuación se especifican con más detalle cada uno de los modos de acceso:

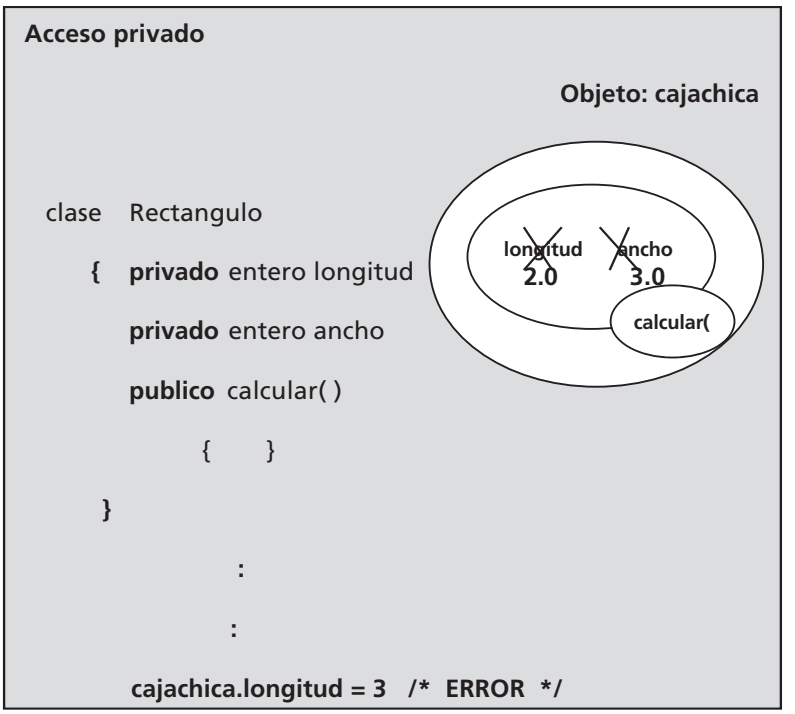
Miembros privados. Son aquellos a los que puede accederse sólo dentro de la clase en que están definidos. El objeto no puede tocar los miembros de la clase.

Miembros públicos. Son aquellos a los que puede accederse desde dentro de la clase y también por parte del objeto.

Miembros protegidos. Son aquellos a los que puede accederse sólo dentro de la clase en que están definidos. Es importante aclarar que este tipo de acceso se asocia directamente con la herencia.

Dependiendo del lenguaje de programación, al omitir la especificación de modo de acceso a los miembros de la clase, el compilador asumirá privado o publico. Por ejemplo, el lenguaje C++ asume privado, pero JAVA asume público.

Los siguientes esquemas ilustran los distintos tipos de acceso:



Acceso público:

clase Rectangulo

{

publico entero longitud

publico entero ancho

publico calcular( )

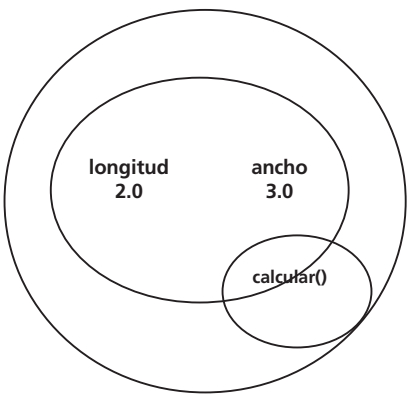
{ ... }

}

:

cajachica.calcular( )

Objeto: cajachica



En resumen:

Para controlar la forma en que los programas acceden a los miembros, es necesario definir a éstos como públicos, privados o protegidos.

Los miembros privados o protegidos permiten a la clase ocultar información que ésta contiene y que el programa no debe conocer o a la que no necesita acceder directamente. Las clases que usan miembros privados o protegidos proporcionan funciones (métodos) para acceder a ellos.

En el siguiente ejemplo se plantea una clase, Suma, que suma dos números enteros declarando miembros privados y públicos:

```
          clase Suma
{
                    privado entero num1
  privado entero num2
  publico Asignar ( )   {...}
  publico sumar( )     {.....}
  publico imprima( )   {....}
}
```

Suma
- entero num1 - entero num2
+ Asignar( ) + sumar( ) + imprima( )

Problemas propuestos

1. Cree una clase llamada **empleado** que contenga como atributos los datos: nombre y número de empleado, y como métodos asignar ( ) y verdatos ( ).
2. Defina una clase que contenga tres datos enteros correspondientes a las edades del padre, la madre y el hijo. Cree un método que calcule el promedio de las tres edades.
3. Una editorial desea crear fichas que almacenen título y precio de cada publicación. Cree la clase correspondiente y llámela **Publicación**; debe incluir los datos descritos.

3.3 Métodos

Un método (función) es un conjunto de instrucciones o sentencias que realiza una determinada tarea; se identifica con un nombre y puede o no devolver un valor.

Cuando se llama a un método, el flujo de control del programa se transfiere al método y se ejecutan una a una las instrucciones que lo integren; cuando se han ejecutado todas, el control regresa al punto desde donde se hizo la llamada y el programa continúa con la siguiente instrucción o sentencia. Todo método tiene que ser invocado (llamado) desde algún punto del programa (programa principal u otro método).

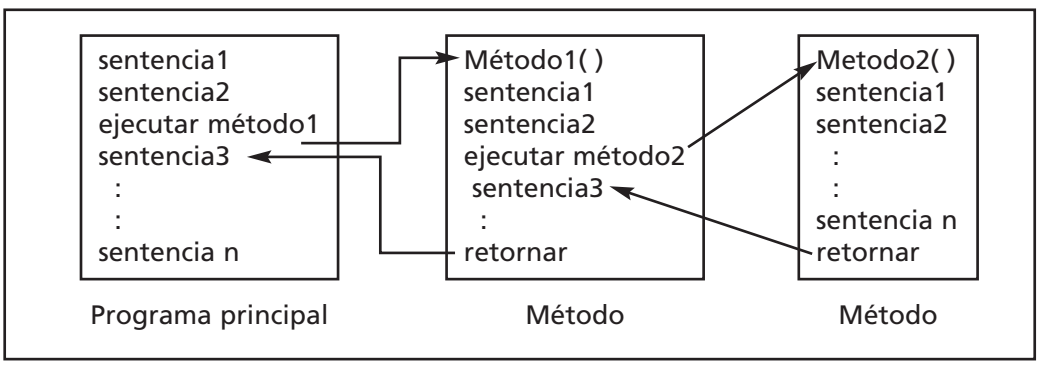


FIGURA 3.2
Flujo de control en una llamada en un método.
Muñoz, Niño, Vizcaíno (2002), Introducción a la Programación con Orientación a Objeto.

En los lenguajes orientados a objetos, los métodos son operaciones o servicios que describen un comportamiento asociado a un objeto. Representan las acciones (tareas concretas) que pueden realizarse por un objeto o sobre un objeto. Cada método tiene un nombre y un cuerpo que realiza la acción.

Los métodos tienen las siguientes características:

- Generan respuesta a un mensaje.
- Procesan datos.
- Corresponden a código ejecutable (instrucciones).
- Si devuelven un valor, deben retornar sólo uno.

La siguiente figura puede ayudar a entender los métodos:

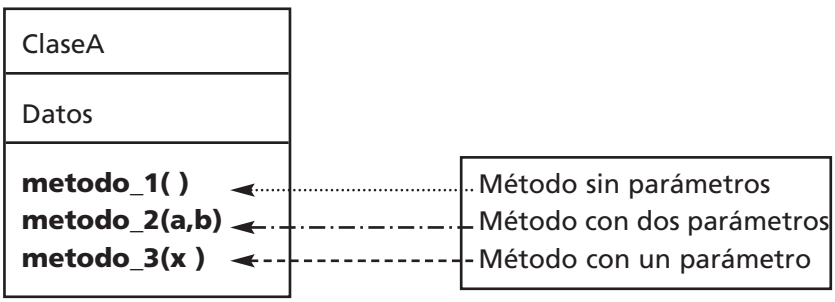


FIGURA 3.3
Métodos.

3.3.1 Definición de métodos

La definición de un método tiene tres componentes principales:

- El tipo de devolución y el nombre del método
- La declaración de los parámetros formales
- El cuerpo del método

Para definir un método se utiliza el siguiente formato:

```
Acceso Tipo_de_dato  nombre_del_método ( [lista de parámetros formales] )
    {
              /* bloque de declaración de variables locales   */

                    Tipo de dato  nombre_ variable_1
                         :
                    Tipo de dato  nombre_ variable_n

                       /* Bloque de Instrucciones */
                          ___________
                          ___________
                    retornar }
```

La definición del método implica los siguientes elementos fundamentales:

Acceso. Es el modificador que autoriza el acceso al método (público, privado o protegido).

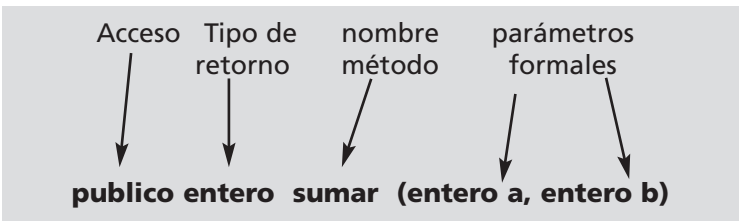
Tipo de dato. Es el tipo de respuesta que devuelve el método, si tiene algún valor para devolver.

Variable_1 .. n. Se consideran variables locales al método donde están definidas.

Parámetros formales. Son variables que reciben valores desde el punto de llamada que activa al método.

Los parámetros formales son valores que se suministran al método en forma de variables, objetos y/o constantes. Deben ser declarados con su tipo de dato correspondiente y separados con coma, y son locales al método, lo cual significa que serán reconocidos sólo en el método donde estén declarados.

El siguiente ejemplo ilustra la función de los parámetros formales. Se declara un método **sumar** que acepta dos parámetros formales, a y b, de tipo entero y que retorna un valor entero:



Cuerpo del método. Lo forman las sentencias que definen lo que hace la función o método. También pueden contener declaraciones de variables locales. El valor devuelto a la sentencia de llamada se hace utilizando la sentencia **retornar**. Revise el siguiente ejemplo donde la clase **Suma** suma dos números enteros:

```
clase Suma
       {  privado entero num1
   privado entero num2
   publico entero  sumar (entero a, entero b ) /* definición
   ➥del método */
        {   entero res          // variable local
            res =a + b
            retornar res        /* retorna la respuesta de la
            ➥suma   */
        }
   }
```


Problemas propuestos

Para cada una de las operaciones que se describen a continuación, construya una clase y elabore los métodos requeridos en cada caso:

1. Leer un número, elevar el número al cuadrado y al cubo, e imprimir el número junto con su cuadrado y cubo.
2. Calcular la ecuación $R = \frac{A + B}{C} . D$
3. Dados los tres lados de un triángulo A, B y C, encontrar el área.

   $$Area = \sqrt{S(S-A)(S-B)(S-C)} \qquad \text{donde: } S = (A+B+C)/2$$

4. Evaluar la ecuación $y = ax^2 + bx + c$. Imprimir el valor de y.

3.4 Objetos

Los objetos son sujetos o cosas tangibles de nuestro mundo real que tienen un estado y un comportamiento. En POO, un objeto está compuesto por datos y métodos.

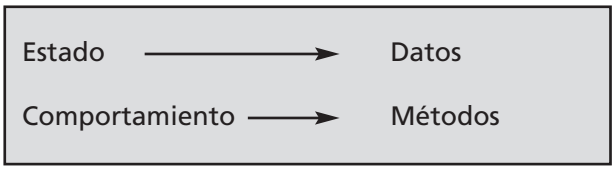


FIGURA 3.4
Introducción a la POO. Muñoz, Niño, Vizcaíno.

Los objetos son variables declaradas del tipo de una clase e instancias de una clase. Un objeto tiene la estructura y el comportamiento definido por la clase, y muestra el comportamiento reflejado por sus operaciones (métodos), que actúan sobre sus datos. Antes de emplear un objeto, primero es necesario declararlo. Un objeto se identifica por un nombre exclusivo y tiene un conjunto de atributos (datos) y operaciones (métodos). Además, tiene un ciclo de vida; es decir, se crea y se destruye.

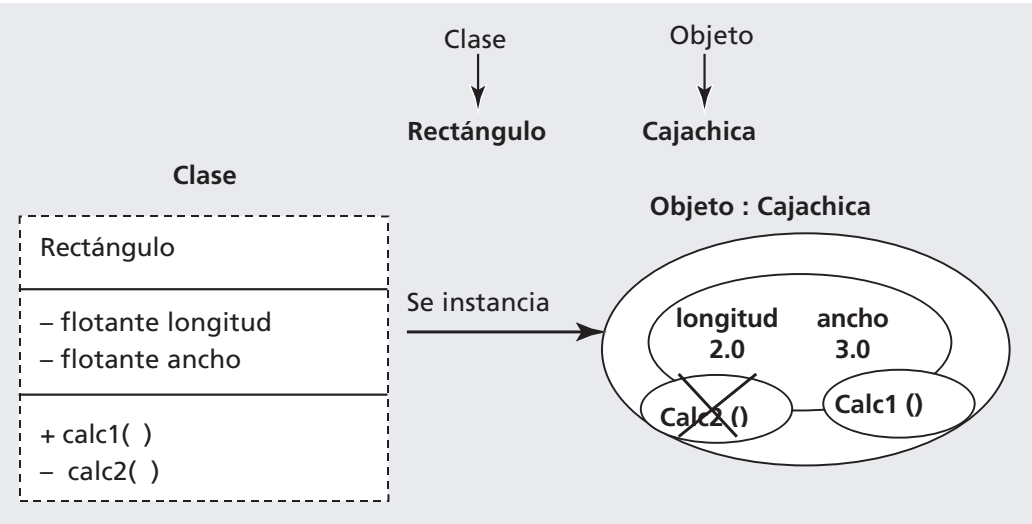
Los métodos son el único medio de acceder a los datos del objeto; no es posible hacerlo directamente. Los datos están ocultos y eso asegura que no puedan modificarse por accidente por métodos externos al objeto.

3.4.1 Declaración y creación de un objeto

Una vez definida la clase se pueden crear muchas instancias (objetos). El formato para declarar un objeto es el siguiente:

```
Nombre_clase   nombre_objeto_1... , nombre_objeto_n
```

El siguiente esquema ilustra el proceso descrito:



3.4.2 Acceso a los elementos de un objeto

Después de que se ha creado un objeto, se puede acceder a sus elementos con el formato siguiente:

```
nombre_objeto.nombre_variable
nombre_objeto.nombre_del_método ([lista argumentos] )

donde Lista de argumentos: son variables que llevan información al
                           método llamado
```

El operador punto (.) permite acceder a los miembros de un objeto; el siguiente ejemplo ilustra su empleo:

```
        Cajachica.calc2()  // Error: se tiene acceso

        Cajachica.calc1 ()
```

3.5 Mensajes

Es una llamada a una función o método, una orden que se envía a un objeto para instruirle que lleve a cabo una acción. Una llamada puede llevar argumentos o parámetros al método invocado y éste puede devolver un valor. Los mensajes o llamadas emplean el siguiente formato:

```
[Variable =] nombre_objeto.nombre_del_método ([lista parámetros actuales] )
```

Donde **lista de parámetros actuales** pueden ser identificadores y/o constantes que se envían al método llamado. Los argumentos deben corresponder en tipo, orden y cantidad con la lista de parámetros del método.

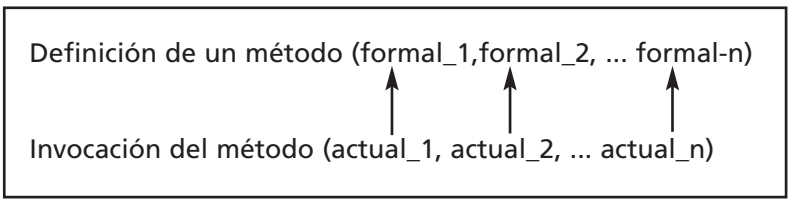


FIGURA 3.5
Correspondencia de parámetros actuales y formales.

El siguiente ejemplo de código ilustra una llamada o mensaje:

```
                clase Suma
{    ——————————————————.
      ——————————————————
                  publico entero   sumar(entero a, entero b )
                {  entero res
                  res =a + b
                  retornar res
                }
            }
                ——————————————————.
                ——————————————————.
                obj_sum . sumar( x,  y)     //llamada al método
```

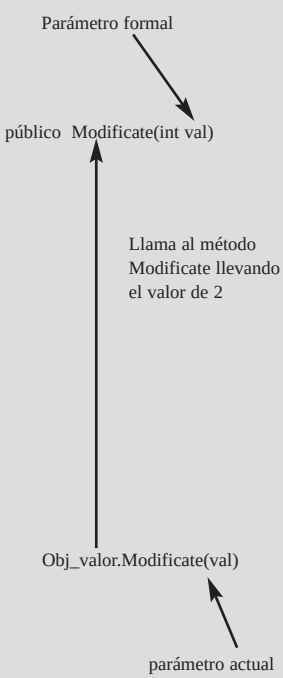
Parámetros por valor o paso por valor

El paso por valor copia el valor del parámetro actual al parámetro formal del método. Por este medio es posible modificar el parámetro formal dentro del método, pero el valor del parámetro actual no cambia una vez que se retorna al punto de llamada.

Las variables reales no se pasan al método; sólo se hacen copias de su contenido (valor).

El siguiente ejemplo ilustra el paso por valor:

```
clase Prueba
 {  privado entero valor
     publico  Modificate(int val) {
        Imprimir("Val dentro de Modificate", val)
        val = val + 5
        Imprimir("val en el Método = " , val)
    }
 }
INICIO.
entero val
val = 2
Imprimir ("Val en el programa principal =", val)
Prueba obj_valor
obj_valor.Modificate(val)
Imprimir("Val después del llamado al método = " val)
FIN.
```



En el siguiente ejemplo se define un algoritmo para leer tres números y calcular e imprimir la suma de los mismos:

```
Análisis y diseño
          Clase:  Csuma
          Atributos:  entero a, b, c
          Métodos:  Asignar()
          entero CalcularSuma()

Programación
            Algoritmo:
            /* Programa: Sumar   */
            /* Programa que suma tres números enteros */
            /* Declaración de la clase   */
               clase   Csuma {
               /* Declaración de  variables locales a la clase   */
                  privado  entero a, b, c
                  publico  Asignar( entero a1, entero b1, entero c1 )
            /* método de lectura de datos   */
                   { a = a1
                     b = b1
                     c = c1
                   }
               publico entero CalcularSuma( )
        /* método de calcular la suma   */
              { entero resultado
                resultado = a+b+c
                retornar resultado
              }
            }
        /*  CUERPO DEL PROGRAMA PRINCIPAL    */
        INICIO
            entero a, b, c, r
            Csuma  objsuma          /* Declaración del OBJETO   */
            Leer(a,b,c)  /* lectura de los datos*/
            objsuma.Asignar(a, b, c)  /* llamada al método por el objeto */
            r = objsuma.CalcularSuma ( )  /*  llamada al método
                                          ➥CalcularSuma */
          Imprimir("Resultado de la Suma es =", r)
        FIN
```

```
+-----------------------------+
|           Csuma             |
+-----------------------------+
| - entero a, b, c            |
+-----------------------------+
| + Asignar( )                |
| + entero CalcularSuma( )    |
+-----------------------------+
```

Declaración de la clase

Cuerpo del programa principal

Problemas propuestos

1. Escriba un algoritmo que lea dos números, y calcule e imprima el valor de su suma, resta, producto y división.
2. Escriba un algoritmo que lea las longitudes de los cuatro lados de un cuadrado, y calcule e imprima el área del mismo.

3.6 Métodos especiales

Son métodos cuyo objetivo es asignar espacio de memoria a un objeto o liberar espacio de memoria asignado al objeto, los cuales se implementan a través de los métodos llamados Constructores y Destructores.

Constructor. Es un método para construir un nuevo objeto y asignar valores iniciales a sus miembros datos. El propósito de un constructor es garantizar que el objeto permanezca en un estado consistente en el momento de su creación. Al no existir un constructor, el compilador genera uno automáticamente. Un constructor emplea el siguiente formato:

```
Nombre de la clase (Lista de Parámetros)
```


Los constructores se caracterizan por tener el mismo nombre de la clase que inicializan; no devuelven valores; admiten parámetros como cualquier método; puede existir más de uno o no existir, y deben aplicarse exclusivamente a la tarea de inicializar un objeto. El siguiente ejemplo ilustra el uso de un constructor:

> Suponga que se depositó un capital inicial de $5,000.00 en una cuenta de ahorros el primero de enero del año 2000 y que el banco paga anualmente el 7% de interés. Para determinar el capital al término de 5 años.

```
clase Ahorro {
    /* Declaración de datos de la clase */
       privado  flotante monto
    /* Implementación  de métodos */
    /*constructor  */
       publico     Ahorro ( ) {
       monto = 5000
          }

       publico flotante cal_capital ( )
          { monto = monto*.07 + monto
            retornar  monto
          }
        :
       /* En el cuerpo principal    */
        :
          /* creación del objeto y activación del constructor  */
          Ahorro  obj_ahorro
```

Destructor. Es un método que libera o limpia el almacenamiento asignado a los objetos cuando se crean. Para ello se emplea el siguiente formato:

```
Formato :   ~Nombre de la clase ()
```

Los destructores tienen el mismo nombre de la clase pero con una tilde (~) antepuesta; no puede tener argumentos; no pueden devolver valores, y cada clase puede tener sólo un destructor. El siguiente ejemplo ilustra el uso de un destructor:

```
clase Rectangulo {
            flotante longitud
            flotante ancho
    publico Rectangulo(flotante lon, flotante a)
       { longitud = lon
         ancho = a
       }
    ~Rectangulo( )
        { }
}
```

Problema resuelto

Definición o dominio del problema

Leer los datos de un triángulo, calcular la hipotenusa $\sqrt{\mathbf{h} = \quad \mathbf{a}^2 + \mathbf{b}^2}$ y presentar el resultado.

Metodología de trabajo

Actividades requeridas para construir un programa orientado a objetos.

Análisis

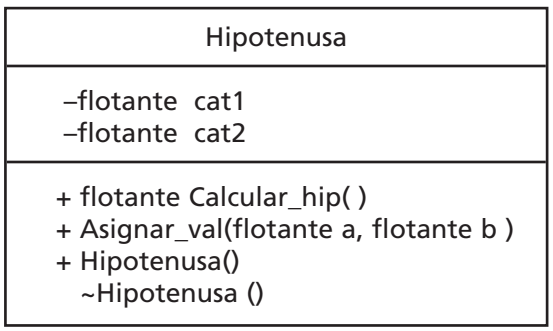
1. Identificar las abstracciones (clases).
2. Identificar los atributos (datos) de cada abstracción.
3. Identificar las operaciones (métodos) de cada abstracción.

4. Aplicar la herencia donde sea necesario.
5. Probar el diseño con escenario (hacer el programa).

1. Identificar la abstracción del Problema

```
Nombre de la clase:  Hipotenusa
Datos :    flotante cat1,  flotante cat2
Métodos:   flotante Calcular_ hip ( )
           Hipotenusa()
                     Asignar_val (flotante, flotante )
                     ~Hipotenusa()
```


Diseño



Programación

```
Algoritmo
√
   clase Hipotenusa {
   /* Declaración de datos de la
      ➥clase */
   privado  flotante cat1, cat2
   /* Implementación  de métodos */
   publico  asignar_val ( flotante a,
   ➥flotante b) {
       cat1 = a
       cat2 = b }
publico flotante calcular_hip ( )
   { flotante h
     h =    cat1 ** 2 + cat2 **2
      retornar h  }
}
```

```
Programa en C++
  #include <math.h>
  #include  <iostream.h>
  class Hipotenusa {
    float cat1, cat2;
  public:
   void asignar_val(float a ,
   ➥float b)
     { cat1 = a ;
       cat2 = b;
     }
     float calcular_hip( )
     { float h ;
       h = sqrt (pow(cat1,2) + pow
       ➥(cat2, 2));
       return(h); }
```

```
  INICIO.
  /* Declaración de variables */
  flotante a, b , res
  /* se crea el objeto de la
     ➥clase */
  Hipotenusa obj_hip

  IMPRIMIR ( "ENTRAR EL VALOR
  ➥DE A :")
  LEER (a)
  IMPRIMIR ( "ENTRAR EL VALOR
  ➥DE B :")
  LEER (b)
  /* Activación de mensajes      */
  obj_hip.asignar_val( a, b)
  res = obj_hip.calcular_hip()
/* Impresión de  salida         * /
  IMPRIMIR ("HIPOTENUSA ES :", res)
  FIN.
```

```
// Programa Principal
    void main( )       {
      float a, b;
      float res;
      /* se crea el objeto    */
      Hipotenusa obj_hip;
      clrscr( );
      cout << "Entrar el valor de
      ➥a : ";
      cin>> a;
      cout << "Entrar el valor de
      ➥b : ";
      cin>> b;
      // Activación de mensajes
      obj_hip.asignar_val(a, b);
      res = obj_hip.calcular_hip();
      cout<<"Hipotenusa es
      ➥= "<< res;
getch();   }
```

```
Algoritmo
√
  clase Hipotenusa {
  /* Declaración de datos de la
clase */
  privado  flotante cat1, cat2
  /* Implementación  de métodos */
  publico asignar_val ( flotante a,
  ➥flotante b) {
     cat1 = a
     cat2 = b }
  publico calcular_hip ( ) {
    flotante h

    h =    cat1 ** 2 + cat2 **2
    retornar h  }
}
  INICIO.
   /* Declaración de variables */
   flotante a, b , res
   /* se crea el objeto de la
      ➥clase */
   Hipotenusa obj_hip
   IMPRIMIR ( "ENTRAR EL VALOR
   ➥DE A :")
   LEER (a)
   IMPRIMIR ( "ENTRAR EL VALOR
   ➥DE B :")
```

```
Programa en Java
 import java.io.*;
class Hipotenusa
{ private float cat1,cat2;
   public void asignar_val(float a,
   ➥float b)
  { cat1=a;
    cat2=b; }
  public double calcular_hip()
  {double h;
   h=Math.sqrt(Math.pow(cat1,2) +
   ➥Math.pow(cat2,2));
   return (h);
   }  }
  //Programa Principal
  class Hipotenusap
{ public static void main (String
➥[] arg) throws IOException {
    BufferedReader br = new Buffered
    ➥Reader (new InputStreamReader
    ➥(System.in));
  float a,b;
  double res;
  /* se crea el objeto   */
  Hipotenusa obj_hip = new
  ➥Hipotenusa();
  System.out.println("Entrar el
  ➥valor de a:");
```

```
  LEER (b)
  /* Activación de mensajes     */
  obj_hip.asignar_val( a, b)
  res = obj_hip.calcular_hip();

  /* Impresión de  salida      * /
  IMPRIMIR ("HIPOTENUSA ES
  ➥:", res)
FIN.
```

```
a= Float.parseFloat(br.readLine());
System.out.println("Entrar el
➥valor de b:");
b= Float.parseFloat(br.readLine());
   //Activación de mensajes
obj_hip.asignar_val(a,b);
res=obj_hip.calcular_hip();
System.out.println("Hipotenusa
➥es = "+res);
 }  }
```

Problemas propuestos

1. Elabore la estructura de un programa orientado a objetos para cada una de las tareas descritas a continuación:
   a) Calcular el área de un triángulo (1/2*b*h). Imprimir los datos de entrada y el resultado.
   b) Calcular el área de un círculo dado el radio (π r^2). Imprimir el área.
   c) Calcular los grados Fahrenheit (F = 9/5 (C° + 32)) dados los centígrados. Imprimir el resultado.
   d) Calcular el salario de un empleado a partir de los siguientes datos de entrada: nombre del empleado, horas trabajadas, tarifa por hora, total de deducciones.
   e) Calcular salario bruto = tarifa por horas * horas trabajadas y el salario neto = salario bruto– deducciones. Imprimir salario bruto y salario neto.
   f) Leer el nombre del estudiante y tres notas, calcular e imprimir la nota promedio.
   g) Leer dos números, calcular e imprimir el valor de suma, resta, producto y su división.
   h) Leer el largo, ancho y altura de una habitación. Calcular e imprimir cuántos metros cuadrados se requieren comprar de alfombra (área = largo × ancho), y cuántos metros cuadrados de papel se requieren para tapizar la pared de la habitación (P = 2 * largo * ancho * altura).
   i) Calcular e imprimir el precio de venta de un artículo. Se tienen los datos descripción del artículo y costo de producción. El precio de ventas se calcula añadiendo al costo el 12% como utilidad y el 15% de impuesto.

2. Una niña deja caer su cartera desde el último piso de la torre Sears (1,454 pies de altura). Cree un algoritmo que calcule la velocidad de impacto al llegar al suelo.

   Utilizar la fórmula $v = \sqrt{2gh}$, donde h es la altura de la torre, g es la gravedad de 32 pies/seg ^{2}.

3. Elabore un algoritmo que lea una cantidad de horas e imprima su equivalencia en minutos, segundos y días.

CAPÍTULO 4

Estructuras de control

Objetivos:

- Definir los conceptos de las estructuras secuencial, de alternativa y de repetición.
- Definir los formatos de cada una de las estructuras de control.
- Resolver problemas que incluyan estructuras de control, utilizando seudocódigo como herramienta de desarrollo.

Contenido

Para construir los algoritmos existen estructuras que permiten a las aplicaciones ejecutar las instrucciones con el fin de obtener los resultados deseados.

La programación estructurada ha legado a la POO un conjunto de estructuras básicas que facilitan la construcción de programas, tales como las estructuras de secuencia, de alternativa y de repetición.

A continuación se explican cada una de ellas.

4.1 Definición de estructuras de control

Son las estructuras básicas necesarias para organizar el flujo de control en un algoritmo o programa. Tres de ellas —la secuencial, la de alternativa y la de repetición— constituyen el fundamento de la organización de un proceso sistemático de programación.

El programador encontrará que estas estructuras permiten desarrollar programas mejor organizados que, en consecuencia, se escribirán, leerán y modificarán más fácilmente.

4.2 Estructura secuencial

Es un conjunto de instrucciones, una debajo de la otra, que se ejecutan en forma secuencial o consecutiva. La figura 4.1 describe este tipo de estructuras:

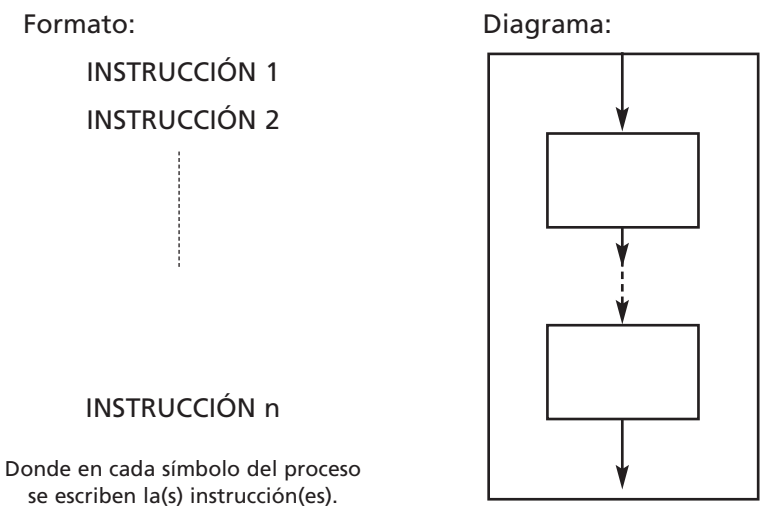


FIGURA 4.1
Estructuras secuenciales.

El siguiente ejemplo ilustra el empleo de una estructura secuencial:

```
Calcular la suma y el promedio de tres números.

sum = a + b + c

prom = sum/3
Imprimir ("Valor de la Suma", sum)
Imprimir ("Valor del promedio", prom)
```

4.3 Estructura de alternativa

Este tipo de estructura controla la ejecución de uno o varios bloques de instrucciones, dependiendo de si se cumple o no alguna condición, o del valor final de una expresión.

Una *condición* es una expresión lógica, es decir, una combinación de variables y/o constantes con operadores lógicos y relacionales que producen resultados ciertos o falsos.

4.3.1 Operadores relacionales y lógicos

Se utilizan para realizar comparaciones; generan un resultado de tipo booleano (cierto o falso).

Operadores relacionales. Relacionan un término con otro para definir su igualdad, jerarquía o cualquier otra relación que guarden. La siguiente tabla lista los operadores relacionales:

Operadores	*Descripción*	*Ejemplo*
<	Menor que	A < 3
>	Mayor que	A > 3
<=	Menor o igual que	A <= 3
>=	Mayor o igual que	A >= 3
= =	Igual que	A = = 3
<>	Distinto a	A <> 3

- *Operadores lógicos*. Un operador lógico permite asociar una o más expresiones relacionales. Si tenemos más de una expresión relacional, ésta se conoce como *condición compuesta*; sus partes estarán relacionadas a través de operadores lógicos.
- *Operador NOT*. Invierte el resultado de una expresión relacional. La siguiente tabla de verdad ilustra el funcionamiento de este operador:

A	***NOT A***
C	F
F	C

Los siguientes ejemplos ilustran la aplicación del operador NOT:

a) Si A = 2, entonces:

Expresión	
(A < 3)	NOT (A < 3)
C	F

b) Si A = 5 en la comparación (A < 3) el resultado es falso. Si se aplica el operador NOT al resultado de esa comparación, el resultado final de la expresión sería cierto (C):

Expresión	
(A < 3)	NOT (A < 3)
F	C

- *Operador Y*. Especifica que para que el resultado de una expresión sea cierto, todas las expresiones evaluadas también tienen que serlo. La siguiente tabla de verdad ilustra el funcionamiento de este operador:

A	*B*	***A Y B***
C	C	C
C	F	F
F	C	F
F	F	F

Los siguientes ejemplos ilustran la aplicación del operador Y:

a) Si A = −6, B = 0 entonces:

EXPRESIÓN 1	*EXPRESIÓN 2*	*RESPUESTA DE UNA EXPRESIÓN COMPUESTA*
(A < 3)	(B == 0)	(A < 3) Y (B == 0)
C	C	C

b) Si A= 2, B= 7 entonces:

En la comparación (A < 3) la respuesta es cierta, y en (B == 0), la respuesta es falsa; por consiguiente, en la expresión (A < 3) Y (B == 0) el resultado final es falso.

EXPRESIÓN 1	*EXPRESIÓN 2*	*RESPUESTA DE UNA EXPRESIÓN COMPUESTA*
(A < 3)	(B == 0)	(A < 3) Y (B == 0)
C	F	F

- *Operador O*. Especifica que para que el resultado sea cierto, basta con que una expresión sea cierta. La siguiente tabla de verdad ilustra el funcionamiento de este operador:

A	*B*	*A O B*
C	C	C
C	F	C
F	C	C
F	F	F

Los siguientes ejemplos ilustran la aplicación del operador Y:

a) Si A = −6, B = 0, entonces:

EXPRESIÓN 1	*EXPRESIÓN 2*	*RESPUESTA DE UNA EXPRESIÓN COMPUESTA*
(A < 3)	(B == 0)	(A < 3) O (B == 0)
F	C	C

b) Si A= 8, B= 7 entonces en la comparación (A < 3), la respuesta es falsa, y en (B == 0), el resultado es falso; por consiguiente, en la expresión (A < 3) Y (B == 0) el resultado final de la misma será falso.

Operadores	*Descripción*	*Ejemplo*
Not	Negación	Not(A < 3)
Y	AND	(A < 3) Y (B == 0)
O	OR	(A < 3) O (B == 0)

4.3.2 Jerarquía de operadores relacionales y lógicos

Cuando se tienen operaciones compuestas, es necesario encerrarlas entre paréntesis para evaluar la expresión correcta y determinar cuál operación se ejecuta primero. Cuando la expresión es muy compleja, es necesario que se conozca la prioridad de las expresiones para obtener el resultado correcto.

1) ( )
2) Not
3) <, >, <=, >=
4) = =, <>
5) Y
6) O

Los operadores con igual jerarquía se evalúan de izquierda a derecha.

Problemas propuestos:

1. Determine el resultado de cada una de las siguientes expresiones. Considere que:

   i = 8, j = 5, x = 0.005 , y = –0.01, c = ‘c’ , d = ‘d’ donde ‘c’= 99 y ‘d’= 100.

Expresión	*Resultado*
i <= j	Falso
c > d	
x >= 0	
j <> 6	
c == 99	
5*(i + j) > ‘c’	
Not (i <= j)	
Not (c == 99)	
(i > 0) Y (j < 5)	
(x > 0) Y (i > 0) O (j < 5)	
(x > 0) Y (i > 0) Y (j < 5)	
(3 * i – 2 * j) < 10	

4.3.3 Tipos de estructuras de alternativa

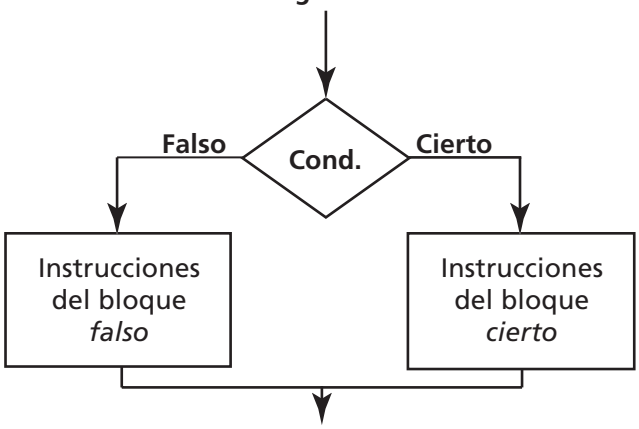
Existen tres estructuras de alternativa: doble, simple y múltiple.

Alternativa doble. Controla la ejecución de dos conjuntos de instrucciones mediante el cumplimiento o incumplimiento de una condición: si se cumple, se ejecutan las instrucciones que están en el bloque *cierto*; si no se cumple, se ejecutan las instrucciones que están en el bloque *falso*.

Formato:

```
Si (condición) entonces
  {       Sentencia 1
          :
          Sentencia n
   }
  De otro modo
   {      Sentencia 1
          :
          Sentencia n
  }
  Fin Si
```

Diagrama

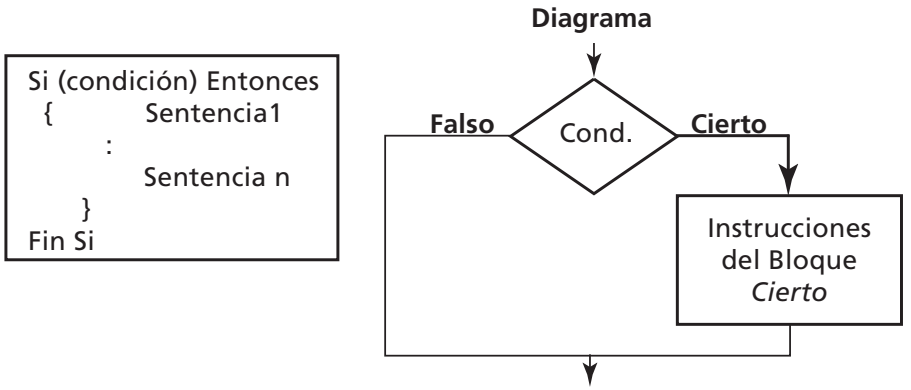


El siguiente ejemplo ilustra una estructura de alternativa doble:

```
Determinar el mayor de dos números:
Si (A > B)  Entonces
      Imprimir ( "GRANDE ", A)
De Otro Modo
      Imprimir ("GRANDE ", B )
  Fin Si
```


Alternativa simple. Hay situaciones en las cuales no es necesario ejecutar instrucciones a partir de la alternativa falsa de la condición, por lo cual existe el modelo de la alternativa simple.

Una alternativa simple controla la ejecución de un conjunto de instrucciones por el cumplimiento o no de una condición, de tal forma que, si se cumple, se ejecutan las instrucciones que se encuentran en el bloque *cierto*. Emplea el siguiente formato:



El siguiente ejemplo ilustra una estructura de alternativa simple:

```
Determinar si un número es menor que 2.1
  Si  ( X < 2.1 ) Entonces
           { Y  = 0.5 * X + 0.95
             Imprimir ( " Y = ", Y)
           }
               Fin Si
```

Alternativa múltiple. Son aquellas que permiten evaluar más de una condición, ya que, con frecuencia, en la práctica es necesario que existan más de dos elecciones posibles; según se elija uno de estos valores en la condición, se ejecutará uno de los *n* caminos o bloques de instrucciones (por ejemplo, al calcular la raíz cuadrada de un número, existen tres posibles opciones o caminos a seguir, según si el número es positivo, negativo o igual a cero). Se procede de acuerdo con el siguiente formato:

```
   Si (Condición) Entonces
{  Sentencia 1
 :
          Sentencia n
          }
   De Otro Modo Si (Condición) Entonces
          { Sentencia 1
                :
          Sentencia n
          }
            :
   De Otro Modo
          { Sentencia 1
      :
          Sentencia n
          }
   Fin Si
```


El siguiente ejemplo ilustra una estructura de alternativa múltiple:

```
Determinar la raíz cuadrada de un número

Si (num>0) Entonces
{
       Raiz =    num

       Imprimir("La Raíz = ", Raiz)
}              √‾‾‾‾‾
De Otro Modo Si (num < 0) Entonces
       Imprimir ("Raiz Imaginaria")
De Otro Modo
       Imprimir("El valor debe ser distinto de cero")
Fin Si
```

Problema resuelto

Definición o dominio del problema

Leer dos números —A , B— y determinar cuál es el mayor.

Análisis

Identificar la abstracción del problema

Nombre de la Clase: Mayor

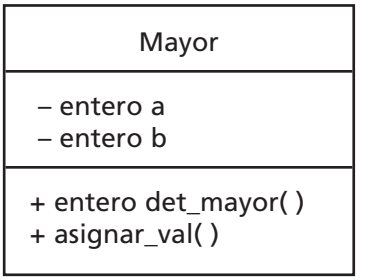
Identificar los atributos

Datos: entero a, entero b

Métodos: entero det_mayor ( )

asignar_val (entero, entero)

Diseño



Algoritmo

```
  clase Mayor {
  /* Declaración de datos de la
  ➥clase */
    privado entero a, b
  /* Implementación de métodos */
  publico asignar_val ( entero x,
  ➥entero y) {
      a = x
      b = y }
   publico entero det_mayor ( ) {
    Si (a>b) entonces
      retornar a
    De otro modo
       retornar b
    Fin Si
     }
}
 INICIO.
  /* Declaración de variables */
  entero a1, b1
  Imprimir ( "ENTRAR EL VALOR
  ➥DE A :")
  Leer (a1)
  Imprimir ( "ENTRAR EL VALOR
  ➥DE B :")
  Leer (b1)
  /* se crea el objeto de la
     ➥clase */
  Mayor obj_mayor
  /* Activación de mensajes    */
  Obj_mayor.asignar_val( a1, b1)
/* Impresión de salida      * /
   Imprimir ("Mayor es: ",
   ➥obj_mayor.det_mayor())
 FIN
```

Programa en C++

```
 #include <iostream.h>
  #include <conio.h>
  class Mayor{
    int a, b;
  public:
   void asignar_val(int x , int y)
    { a = x;
      b = y;
    }
   int det_mayor( )
    { if (a>b)
      return a;
     else
      return b;
    }
   };
// Programa Principal
   void main( )    {
     int a1, b1;
     clrscr( );
     cout << "Entrar el valor
     ➥de a : ";
     cin>> a1;
     cout << "Entrar el valor
     ➥de b : ";
     cin>> b1;
     /* se crea el objeto  */
     Mayor obj_mayor ;
   //Activación de mensajes
     obj_mayor.asignar_val(a1, b1);
     cout<<"Mayor es :
     ➥"<<obj_mayor.det_mayor();
     getch();
    }
```

Algoritmo

```
  clase Mayor {
  /* Declaración de datos de la
     ➥clase */
    privado entero a, b
  /* Implementación de métodos */
  publico asignar_val ( entero x,
  ➥entero y) {
      a = x
      b = y }
   publico entero det_mayor ( ) {
    Si (a>b) entonces
```

Programa en Java

```
import java.io.*;
class Mayor {
       private int a,b;
       ➥/* Declaración de atributos
       ➥de la clase*/
/* Implementación de los Métodos
➥de la clase Mayor*/
 public void asignar (int x, int y) {
        a= x;
        b = y;
 }
```

4

```
      retornar a
    De otro modo
       retornar b
    Fin Si
     }
}
 INICIO.
  /* Declaración de variables */
  entero a1, b1
  Imprimir ( "ENTRAR EL VALOR
  ➥DE A :")
  Leer (a1)
  Imprimir ( "ENTRAR EL VALOR
  ➥DE B :")
  Leer (b1)

  /* se crea el objeto de la
     ➥clase */
  Mayor obj_mayor

  /* Activación de mensajes    */
  Obj_mayor.asignar_val( a1, b1)

/* Impresión de salida      * /
   Imprimir ("Mayor es: ",
   ➥obj_mayor.det_mayor())
 FIN
```

```
 public int det_mayor () {
        if (a>b)
        return a;
        else
        return b;
 }
}
 /* Clase Principal */
 class Mayores { public static void
 ➥main (String arg[]) throws
 ➥IOException {
 int a1,b1;
 String cad;
/* Se crea al objeto*/
 Mayor obj = new Mayor();
 BufferedReader br=new
 ➥BufferedReader (new   Input-
 ➥StreamReader (System.in));
 System.out.println ("Introduzca un
 ➥entero");
 cad= br.readLine();
 a1=Integer.parseInt (cad);
 System.out.println ("Introduzca un
 ➥entero");
 cad= br.readLine();
 b1=Integer.parseInt (cad);
/* Activación del mensaje asignar*/
 obj.asignar (a1,b1);   System.out.
 ➥println ( "Mayor es:
 ➥"+ obj.det_mayor());
}
}
```

Problemas propuestos

Para cada una de las operaciones que se describe a continuación, construya un algoritmo empleando estructuras de alternativas:

1. Leer un número entero, encontrar su valor absoluto e imprimir número y valor absoluto.
2. Leer los valores de cuatro enteros: A, B, C y D. Imprimir el mensaje “SÍ”, si el valor de A/B = C/D; en caso contrario, imprimir el mensaje “NO”. Si alguno de los valores de B o D es cero, imprimir Indefinido.
3. Calcular el valor de CAT, el cual depende de las siguientes relaciones:

Si x > 0, CAT = (A+B)(C+D)

Si x = 0, CAT= (A+B)/(C+D)

Si x < 0, CAT = A+B-C+D

Imprimir CAT.

4. Calcular el valor de X1 y X2 a partir de la ecuación general cuadrática:

$$X= \frac{-b + b^2\sqrt{-4ac}}{2a}$$

Imprimir los valores de X1 y X2 o los mensajes de errores respectivos.

5. Leer un número: si es > 100, sumarle 20; si el número es = 100, sumarle uno; si el número es < 100, restarle 20. Imprimir el número después de hacer el cálculo.
6. Leer 3 números, y determinar cuál es el mayor y cuál el menor. Imprimir el mayor y el menor con sus respectivos mensajes.
7. Leer dos cantidades, una de monto adeudado y otra de pago de la deuda. Comparar ambas: si son iguales, el cliente no debe nada y ha pagado todas sus facturas; si el pago es mayor que la deuda, el cliente tiene un crédito a su favor y se calcula el monto de crédito; si el pago es menor que la deuda, se calcula la nueva deuda agregándole el cálculo del 3% de interés. Imprimir la información de entrada y la salida.
8. Leer un carácter y determinar si es una vocal o un número.
9. Leer los datos de un empleado: nombre, horas trabajadas y sueldo por hora. Calcular el sueldo del empleado. Si las horas trabajadas no pasan de 40, el sueldo se calcula de la siguiente forma: sueldo por hora * horas trabajadas. Si el empleado trabaja más de 40 horas, su sueldo se calcula así: 40 * sueldo por hora + horas extras * sueldo por hora * 1.5.

4.4 Estructuras repetitivas o de repetición

Son aquellas que controlan la repetición de un conjunto de instrucciones mediante la evaluación de una condición.

Toda estructura repetitiva está compuesta por los siguientes elementos:

- *Condición*. Es la expresión lógica o relacional a evaluar y que determina la entrada o no al ciclo de una sentencia repetitiva.
- *Bloque de instrucciones*. Conjunto de instrucciones a ejecutarse.
- *Ciclo o iteración.* El proceso de ejecución del bloque de instrucciones varias veces.

Estas estructuras permiten realizar tareas tales como:

- Procesar el promedio de calificaciones de 20 estudiantes.
- Calcular el salario de *n* grupo de empleados.
- Determinar cuántos estudiantes se leen.
- Determinar cuántos empleados se procesaron.

Para ello, es necesario definir procedimientos que permitan controlar la condición de un ciclo, se conozca o no el límite de iteraciones que éste debe realizar. Existen dos procedimientos básicos de repetición: ciclos definidos y ciclos indefinidos.

- *Ciclos definidos*. Son aquellos donde se conoce con anticipación el número de veces que se ejecutará el ciclo (por ejemplo, un ciclo controlado con contador).

  El ciclo controlado por contador es el más simple, porque se conoce de antemano el número de veces que el ciclo se repetirá; requiere:

  1. El nombre de la variable de control o contador.
  2. El valor inicial de la variable de control.
  3. El incremento o decremento con el cual, cada vez que se termine un ciclo, la variable de control será modificada.
  4. La condición que compruebe el valor final de la variable de control; es decir, si el ciclo finaliza o no.

- *Ciclos indefinidos*. Son aquellos donde no se conoce con anticipación el número de veces que se ejecutará el ciclo, como el ciclo controlado por valor centinela, el ciclo controlado por respuesta y otros.

  El *ciclo controlado por valor centinela* se utiliza en aquellas ocasiones en que no se conoce con anticipación el número de veces que el ciclo debe ejecutarse. Un *valor centinela* es uno que sirve para terminar el ciclo, dándole un valor diferente a un campo que forma parte del conjunto de datos de entrada. Es un método utilizado en los lenguajes de tercera generación

donde, al final del conjunto de datos de entrada, normalmente se cargaba un valor de fin de archivo. Requiere:

1. El nombre de la variable de control.
2. La condición que prueba que la variable de control toma el valor especificado que indica el fin del ciclo.

El *ciclo controlado por respuesta* se utiliza cuando se le quiere dar al usuario la opción de decidir la terminación del ciclo. Requiere los siguientes elementos:

1. El nombre de la variable de control o respuesta.
2. El valor inicial de la variable de control.
3. La condición que comprueba el valor final de la variable de control; es decir, si el ciclo termina o no.
4. Colocar dentro del bloque de instrucciones la solicitud de la respuesta del usuario, para la variable de control.

Para realizar procesos con ciclos definidos, las estructuras de repetición requieren de un contador, que debe ser inicializado.

Inicializar es el proceso de asignar un valor inicial, que puede ser cero o cualquiera otro dependiendo de los requerimientos del problema. Esta operación debe ser el primer paso antes de la ejecución del ciclo que requiere el uso de un contador o acumulador.

La inicialización se lleva a cabo de acuerdo con el siguiente ejemplo:

```
entero cont
cont = 0  ← inicialización
```

Un *contador* es una variable que se utiliza para contar cualquier evento que pueda ocurrir dentro de un programa; su valor se incrementa o disminuye en una cantidad constante en cada iteración a cada ocurrencia del evento. En general, se suele contar desde 0 y en incrementos o decrementos de 1, aunque éstos pueden ser de 2, de 5, etcétera. El siguiente ejemplo muestra cómo se define un incremento en un contador:

```
Contador = Contador + constante_numérica

ejemplo: cont = cont + 2
```

y el siguiente, ilustra una disminución (en estos casos, la variable contador debe ser inicializada con el valor tope o límite de veces que se ha de realizar el ciclo):

```
Contador = Contador - constante_numérica
cont = cont - 2
```

La condición en las estructuras de repetición pueden utilizar a los contadores como parte de expresión lógica o de relación que controla el número de veces que las instrucciones se ejecutan.

Entre las tareas que pueden llevarse a cabo con la ejecución de un ciclo se incluye determinar una cantidad total de dinero o cuánto acumuló una empresa, o determinar el total de ventas de un almacén, para lo cual se requiere utilizar un *acumulador* o *totalizador*.

Un *acumulador* o *totalizador* es una variable cuya función es almacenar cantidades variables, resultantes de sumas sucesivas.

Utiliza solamente incrementos variables en lugar de incrementos constantes, como los contadores. Su formato es el siguiente:

```
acumulador = acumulador +  variable
```

donde **variable**: es un valor a sumar que cambia en cada iteración, como el siguiente ejemplo:

```
acum = acum + salariobruto
```

Para realizar todos los procesos mencionados se emplean las estructuras de repetición: MIENTRAS, HASTA QUE y PARA.

4.4.1 Mientras (con condición inicial)

Esta instrucción evalúa una condición que será ejecutada mientras la condición sea cierta. Finalizará su ejecución cuando la condición evaluada sea falsa. La figura 4.2 ilustra este concepto:

Formato:

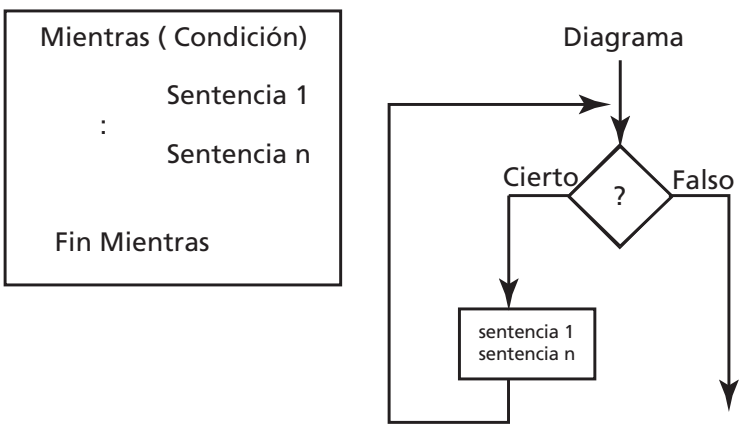


FIGURA 4.2
Instrucción Mientras.

El siguiente ejemplo describe el empleo de la instrucción Mientras:

Ciclo controlado por contador

Ejemplo 4.8 :
Imprimir los 10 primeros Números

```
c  = 0    /**contador */
Mientras (c < 10)
     c   =  c + 1
     Imprimir ( c)
Fin mientras
```

Ciclo controlado por contador, utilizando un acumulador

Ejemplo 4.9:
Calcular la suma de los 20 primeros números pares

```
     c = 0 /**contador */
     p = 0
   tot = 0 /*acumulador */
     Mientras (C < 20)
       p  = p + 2
       tot= tot + p
       c = c +1
       Fin mientras
  Imprimir ("La suma de los 20 primeros
números pares es : ", tot)
```

Nota: En ambos casos podemos observar que se utiliza la variable **c** como contador, la cual se incrementa de 1 en 1 y es la variable que controla el fin del ciclo. En el siguiente ejemplo se utiliza la variable **tot** como acumulador de la suma:

Ciclo controlado con respuesta

Ejemplo 4.10:

Se desean leer diferentes números y sumarlos. Darle al usuario la opción de terminar cuando lo desee.

```
Suma= o
Resp='s'
Mientras (resp == 's' o resp == 'S')
    Imprimir ("Entre un número")
    Leer(n)
    Suma= suma + n
    Imprimir ("Desea terminar s/n")
    Leer (resp)
Fin_mientras
Imprimir ("La suma =", suma)
```

Nota: Se puede observar que se utiliza la variable **resp** como variable de control para ejecutar el ciclo: mientras **resp** sea igual a 's', mayúscula o minúscula. El siguiente ejemplo ilustra el empleo de un valor centinela:

Ciclo controlado con valor centinela

Ejemplo: 4.11
Se desean leer una serie de velocidades mientras la velocidad sea diferente a –999 e imprima su promedio.

```
C=0
Suma=0
Imprimir ("Entre la velocidad:")
Leer(v)
Mientras (v <> –999)
     suma= suma + v
     c = c + 1
     imprimir ("Entre la velocidad")
     leer (v)
fin_mientras
prom = suma / c
imprimir ("Promedio de velocidades: ", prom)
```


Nota: Se puede observar que se utiliza la variable **v** como variable de control para ejecutar el ciclo: mientras **v** sea diferente a –999.

Problema resuelto

Definición o dominio del problema

Supongamos que se depositó un capital de $65,000.00 en una cuenta de ahorros el primero de enero del año 1998, y que el banco paga anualmente el 5% de interés; es decir, cada año el capital se incrementa en un 5%. Desarrolle un algoritmo que, a partir de los datos dados, imprima el año y el capital de la cuenta desde 1998 hasta 2003. Utilice una estructura Mientras.

Análisis

Identificar la abstracción del problema

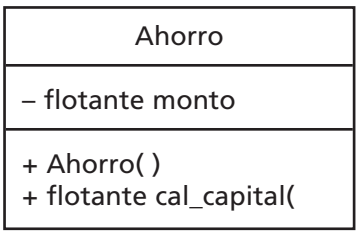
Nombre de la Clase: Ahorro

Datos: flotante monto

Métodos: Ahorro ( )

Flotante cal_capital( )

Diseño




```
Algoritmo
  clase Ahorro {
  /* Declaración de datos de la
     ➥clase */
   privado flotante monto
  /* Implementación de métodos */
/* Constructor  */
      Ahorro ( ) {
       Monto = 65000
    }
  publico flotante cal_capital ( )
   {
       monto = monto*.05 + monto
       retornar monto
     }
   }
```

```
Programa en C++
#include<iostream.h>
#include<conio.h>
class Ahorro{
 float monto;
public:
  float cal_capital()
  {
 monto = monto*0.05 + monto;
 return monto;
  }
// Constructor
  Ahorro(){
   monto=65000;
   }
};
```

```
INICIO.
 /* Declaración de variables */
 flotante capital
 entero anio
 /* se crea el objeto de la clase */
 Ahorro obj_ahorro
 // Imprimir Títulos
  Imprimir (" BANCO NOSOTROS")
  Imprimir("Año      Capital")
 // ciclo de cálculo e impresión
    ➥del capital
  anio = 1998
  mientras (anio<=2003)
 /* Activación de mensajes    */
  capital = obj_ahorro.cal_
  ➥capital( )
  Imprimir( anio, capital)
  Anio = anio + 1
 Fin mientras
FIN.
```

```
void main()
{ float capital;
 int anio;
 clrscr();
 Ahorro obj_ahorro;
 cout<<"\t \t"<<" BANCO
 ➥NOSOTROS\n";
 cout<<"A□o"<<"\t\t"<<"
 ➥Capital"<<endl;
 anio=1998;
 while (anio<=2003)
 { capital=obj_ahorro.cal_
   ➥capital();
  cout<<anio<<"\t\t"<<capital
  ➥<<endl;
  anio=anio+1;
 }
getch();
}
```

4

```
Algoritmo
  clase Ahorro {
  /* Declaración de datos de la
     ➥clase */
    privado flotante monto
  /* Implementación de métodos */
/* Constructor */
      Ahorro ( ) {
       Monto = 65000
    }
  publico flotante cal_capital ( )
   {
     monto = monto*.05 + monto
     retornar monto
    }
   }
 INICIO.
  /* Declaración de variables */
  flotante capital
  entero anio
  /* se crea el objeto de la
     ➥clase */
  Ahorro obj_ahorro
  // Imprimir Títulos
   Imprimir (" BANCO NOSOTROS")
   Imprimir("Año      Capital")
```

```
Programa en Java
class Ahorro {
 private double monto;
/* Constructor */
  Ahorro () {
  monto = 65000;
     }
  public double cal_capital ()
  {
   monto = monto *0.05 + monto;
   return monto ;
     }
  }
 class Banco { public static void
 ➥main (String arg[]) {
 double capital;
 int anio;
 Ahorro obj= new Ahorro ();
 System.out.println ("    BANCO
 ➥NOSOTROS");
 System.out.println ();
 System.out.println (" Año
 ➥Capital");
 System.out.println ();
 anio = 1998;
 while (anio<= 2003) {
```

```
// ciclo de cálculo e impresión
   ➥del capital
 anio = 1998
 mientras (anio<=2003)
/* Activación de mensajes    */
 capital = obj_ahorro.cal_
 ➥capital( )
 Imprimir( anio, capital)
 Anio = anio + 1
Fin mientras
FIN.
```

```
/* Inicio del    Mientras */
 capital = obj.cal_capital();
 System.out.println
 ➥(anio +"  "+ capital);
 anio = anio + 1;
  } /* Fin del Mientras */

}
}
```

4.4.2 Hasta que (con condición final)

Es una instrucción que evalúa una condición al final de la estructura; el ciclo se ejecutará hasta que la condición sea falsa. Esta instrucción permite por lo menos una iteración, dado que su condición es evaluada al final. Opera de acuerdo con el siguiente formato:

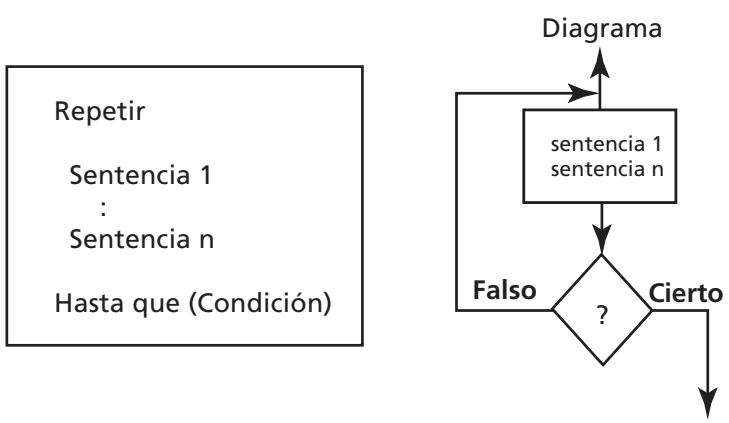


FIGURA 4.3
Instrucción Hasta que.

Los siguientes ejemplo describen el empleo de la instrucción Hasta que:

Ciclo controlado por contador

Ejemplo 4.12:
Imprimir los 10 primeros números

```
c = 0
 Repetir
   c = c + 1
   Imprimir (c)
 Hasta que (c == 10)
```

Ciclo controlado por contador, utilizando un acumulador

Ejemplo 4.13:
Calcular la suma de los 20 primeros números pares

```
   c = 0
 p = 0
   tot = 0
   Repetir
     p = p + 2
     tot= tot + p
     c=c+1
   Hasta que (c == 20)
  Imprimir ("La suma de los 20 primeros
numeros pares es : ", tot)
```

Nota: En ambos casos podemos observar que se utiliza la variable **c** como contador, la cual se incrementa de 1 en 1 y es la variable que controla el fin del ciclo.

Ciclo controlado con respuesta

Ejemplo 4.14:

Se desean leer diferentes números y sumarlos. Darle al usuario la opción de terminar cuando lo desee.

```
Suma= o
Resp = 'n'
Repetir
    Imprimir ("Entre un número")
    Leer(n)
    Suma= suma + n
    Imprimir ("Desea terminar s/n")
    Leer (resp)
Hasta que (resp == 's' o resp == 'S')

Imprimir ("La suma =",suma)
```

Nota: Se puede observar que se utiliza la variable **resp**, con valor inicial de **n**, como variable de control para ejecutar el ciclo hasta que **resp** sea igual a **'s'**.

Ciclo controlado con valor centinela

Ejemplo: 4.15
Se desean leer una serie de velocidades mientras la velocidad sea diferente a –999 e imprima su promedio.

```
C=0
Suma=0
Imprimir ("Entre la velocidad:")
Leer(v)
Repetir
     suma= suma + v
     c = c + 1
     imprimir ("Entre la velocidad")
     leer (v)
Hasta que (v == –999)
prom = suma / c
imprimir ("Promedio de velocidades: ", prom)
```

Nota: Se puede observar que se utiliza la variable **v** como variable de control para ejecutar el ciclo, hasta que **v** sea igual a **–999**.

Algunos lenguajes de programación, como Visual Basic y Pascal, permiten la implementación de esta estructura; sin embargo, los dos lenguajes que estamos tomando como ejemplos, C++ y JAVA, no lo hacen.

4.4.3 Para

Es una instrucción condicional que ejecuta un conjunto de acciones un número específico de veces, de acuerdo con el siguiente formato:

```
Formato:

Para  Variable_ de_ control  =  Valor ,  Valor , Incremento o
                                Inicial  Final   Decremento
    Sentencia 1
     :
    Sentencia n
Fin Para
```

La figura 4.4 ilustra una estructura Para:

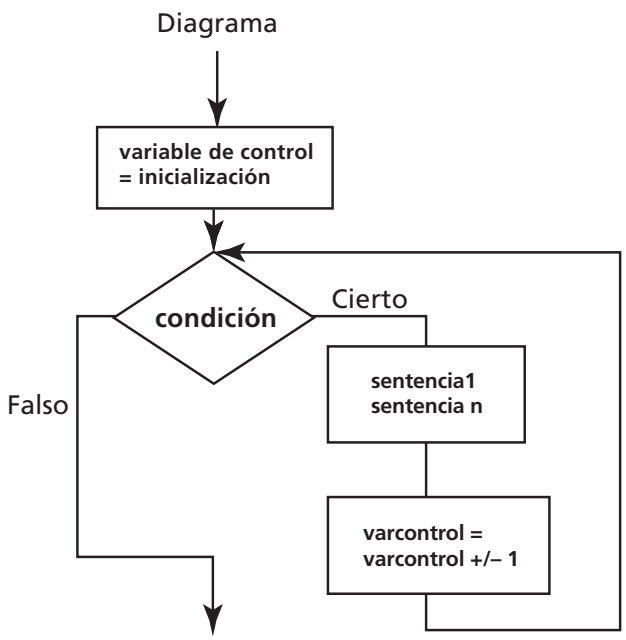


FIGURA 2.1
Instrucción Para.

La instrucción Para está compuesta por una variable de control, el valor inicial, el valor final y el incremento.

La *variable de control* se incrementa o disminuye al final del ciclo para evaluar internamente la condición que determina la terminación de éste.

El *valor inicial* establece el valor con el cual debe iniciar la variable de control. Puede ser un valor constante, variable o expresión.

El *valor final* es el valor de comparación, que se utiliza para determinar la condición de finalización del ciclo. Puede ser un valor constante o variable.

El *incremento* o *decremento* se expresa como un valor entero o carácter.

Esta estructura se utiliza sólo en los casos en que se conoce de antemano el valor inicial y el número de veces que se desea ejecutar las acciones de un ciclo o bucle.

El siguiente ejemplo ilustra el empleo de la instrucción Para:

Ciclo controlado por contador	**Ciclo controlado por contador usando acumulador**
Ejemplo 4.16: **Imprimir los 10 primeros números** Para c = 1,10,1 Imprimir (c) Fin_para	**Ejemplo 4.17:** **Calcular la suma de los 20 primeros números pares** tot = 0 Para c = 2, 40 , 2 tot = tot + c Fin_para Imprimir ("La suma de los 20 primeros números pares es : ", tot)

¿Cómo funciona el ciclo cuando es un incremento y cómo cuando es un decremento?

Si el valor inicial de la variable de control es menor que el valor final, los incrementos deben ser positivos, ya que en caso contrario la secuencia de instrucciones no se ejecutará. De igual modo, si el valor inicial es mayor que el valor final, se dará un decremento, que se expresa como un número negativo. Los siguientes ejemplos ilustran esta distinción:

```
Ejemplo: Con incremento.
     para i = 1,5,1
          imprimir (i)
          suma = suma + i
      fin_para

Ejemplo: Con decremento.

     para an = 2000, 1900, -10
          imprimir ("año = ", an)
      fin_para
```

Los valores inicial, final y de incremento no deben alterarse durante la ejecución del ciclo, aunque ello sea posible.

Problema resuelto

Definición o dominio del problema

Supongamos que se depositó un capital de $65,000.00 en una cuenta de ahorros el primero de enero de 1998 y que el banco paga anualmente el 5% de interés; es decir, cada año el capital se incrementa en un 5%. Desarrolle un algoritmo que a partir de los datos dados imprima el año y el capital de la cuenta desde 1998 hasta 2003. Utilice una estructura Para.

Análisis

Identificar la abstracción del problema

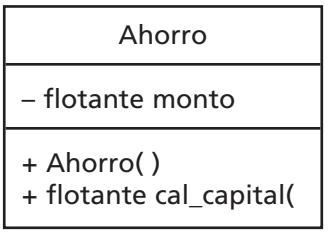
Nombre de la Clase: Ahorro

Datos: flotante monto

Métodos: Ahorro ( )

Flotante cal_capital( )

Diseño



Algoritmo

```
  clase Ahorro {
  /* Declaración de datos de la
     ➥clase */
    privado flotante monto
  /* Implementación de métodos */
/* Constructor  */
  Ahorro ( )
  {  Monto = 65000   }
  publico flotante cal_capital ( )
   { monto = monto*.05 + monto
    retornar monto
     }  }
 INICIO.
  /* Declaración de variables */
  flotante capital
  entero anio
  /* se crea el objeto de la
     ➥clase */
```

Programa en C++

```
 #include<iostream.h>
 #include<conio.h>
 class Ahorro{
 float monto;
 public:
  float cal_capital()
  {  monto = monto*0.05 + monto;
 return monto;
  }
/* Constructor */
  Ahorro(){
   monto=65000;
   }    };
// Programa Principal
void main()
{ float capital;
 int anio;
 clrscr();
```

```
  Ahorro obj_ahorro
// Imprimir Titulos
 Imprimir (" BANCO NOSOTROS")
 Imprimir("Año      Capital")
// ciclo de calculo e impresión del
➥capital
Para anio = 1998, 2003, 1
  /* Activación de mensajes    */
  capital = obj_ahorro.cal_capital()
 Imprimir( anio, capital) Fin para
FIN.
```

```
// declaración del objeto
 Ahorro obj_ahorro;
 cout<<"\ t \ t"<<" BANCO
 ➥NOSOTROS\n";
 cout<<"Año"<<"\ t \ t"<<"Capital"
 ➥<<endl;
// ciclo de calculo e impresión
   ➥del capital
 for(anio=1998;anio<=2003; anio++)
 { capital=obj_ahorro.cal_
   ➥capital();
  cout<<anio<<"\ t \ t"<<capital
  ➥<<endl;
 }
getch(); }
```

```
Algoritmo
  clase Ahorro {
  /* Declaración de datos de la
     ➥clase */
    privado flotante monto
  /* Implementación de métodos */
/* Constructor  */
  Ahorro ( ) { Monto = 65000   }
  publico flotante cal_capital ( )
   {  monto = monto*.05 + monto
    retornar monto
     }
  }
 INICIO.
  /* Declaración de variables */
  flotante capital
  entero anio
  /* se crea el objeto de la clase */
  Ahorro obj_ahorro
// Imprimir Titulos
 Imprimir (" BANCO NOSOTROS")
 Imprimir("Año      Capital")
// ciclo de cálculo e impresión
   ➥del capital
Para anio = 1998, 2003, 1
  /* Activación de mensajes    */
  capital = obj_ahorro.cal_capital( )
 Imprimir( anio, capital)
Fin para
FIN.
```

```
Programa en Java
class Ahorro {
 private double monto;
/* Constructor  */
 Ahorro () { monto = 65000;  }
 public double cal_capital ()
  {   monto = monto *0.05 + monto;
   return monto ;
    }
  }
 class BancoPara { public static
 ➥void main (String arg[])
{ double capital;
 int anio;
 Ahorro obj= new Ahorro ();
 System.out.println ("BANCO
 ➥NOSOTROS");
 System.out.println ();
 System.out.println (" Año
 ➥Capital");
 System.out.println ();
 /* Inicio del Para */
 for (anio= 1998; anio<= 2003;
 ➥anio++) {      capital =
 ➥obj.cal_capital();
    System.out.println
    ➥(anio +"          "+ capital);
    ➥} /* Fin del Para */
 }
}
```

4

4.4.4 Estructuras de repetición anidadas

Es posible insertar un ciclo dentro de otro si todas las instrucciones del ciclo interno están contenidas en el externo. El esquema 4.5 ilustra este concepto:

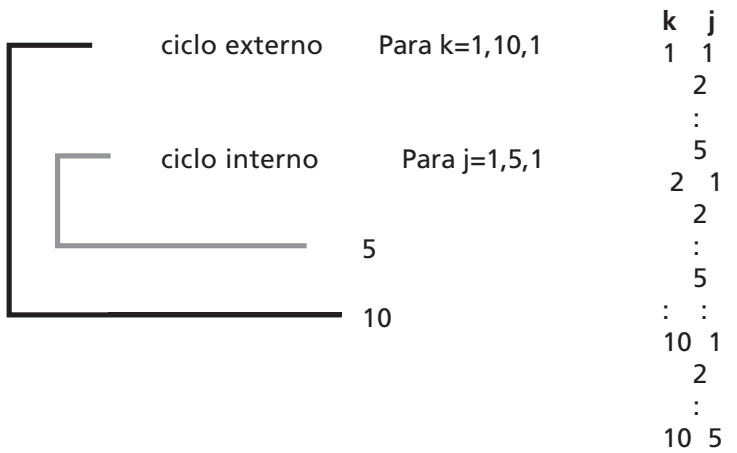


FIGURA 4.5
Estructura de repetición anidada.

En el esquema podemos observar que j varía más rápidamente que k; es decir, el ciclo interno varía más rápido que el ciclo externo. Es posible combinar estructuras como Para y Mientras, Mientras y Hasta Que, Para y Para, etcétera.

Se pueden anidar tantas estructuras como sea necesario, pero debemos recordar que se debe incluir un ciclo dentro del otro y no solaparlos (figura 4.6).

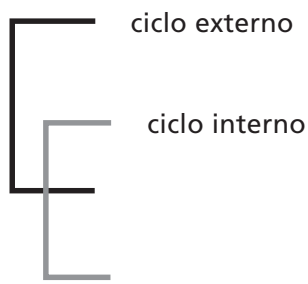


FIGURA 4.6
Ciclos solapados.

A simple vista podemos observar que ésta es una forma incorrecta, ya que los ciclos están solapados.

El siguiente ejemplo ilustra un caso de ciclos anidados:

Imprimir todas la tablas de multiplicar del 1 al 10 con 12 elementos.

```
imprimir ("Tablas de multiplicar")
para (i=1, i < 10, i=i+1)
    imprimir ("Tabla del", i)
    para (j=1, j < 12, j=j+1)
       p=i*j
       imprimir (i, 'x', j '=', p)
    fin_para
fin_para
```

```
i=1
imprimir ("Tablas de  multiplicar")
mientras ( i < 10 )
    imprimir ("Tabla del", i)
    para (j=1, j < 12, j=j+1)
       p=i*j
       imprimir (i, 'x', j '=', p)
    fin_para
    i=i+1
fin_mientras
```

4

Problemas propuestos

Para cada una de las operaciones que se describe a continuación, construya un algoritmo empleando estructuras de repetición:

1. Leer cinco números e imprimir la suma de ellos.
2. Sumar los primeros 5 enteros e imprimir la suma.
3. Leer diez (10) números. Cada número debe ser verificado: si es negativo, se suma; si es cero, se cuenta, y si es positivo, se suma. Imprimir la cantidad de ceros, la suma de positivos y la suma de negativos.
4. Se tiene un número indeterminado de valores para la ecuación Y= ax + bx +c. Evaluar e imprimir Y; cuando el valor de x sea igual a cero, imprimir la sumatoria de las Y calculadas.
5. Generar la tabla de multiplicar del 5 con 10 elementos. Imprimir el título y cada valor de la tabla, en la forma 5*1=5, 5*2=10... .
6. Generar las 12 tablas de multiplicar con 10 elementos cada una. Imprimir el título de cada tabla generada con sus elementos, en la forma 1*1=1, 1*2=2...

7. Leer un número *N*, determinar si el número es par o impar empleando el método de restas sucesivas. Imprimir el número con su mensaje respectivo. (si, por ejemplo, el número es 6, se imprimiría 6−2=4; 4−2=2; 2−2=0, entonces el número es par).
8. Leer un conjunto de datos de entrada, cada uno de los cuales contiene dos valores, N1 y N2. El valor N1 es un indicador que determina si N2 es par o impar, de acuerdo con el siguiente ejemplo: si N1= 3, N2 es par (imprimir su cuadrado), Si N1=2, N2 (imprimir su valor absoluto). Leer hasta que el usuario lo determine.
9. Al soltar una pelota desde una altura de 10 pies, ésta rebota y la altura a la que llega es, a cada salto sucesivo, 2/3 de su altura alcanzada en el rebote anterior. El algoritmo debe determinar qué distancia habrá recorrido la bola al momento de realizar su rebote número 40.
10. Imprimir los 500 primeros números de la siguiente progresión aritmética: 1, 5, 9, 13, 17....
11. En 1616, Peter compró la isla de Manhattan por el equivalente a 24 dólares en piedrecitas de fantasía. Determinar el capital que se tendría en 1976 si estos 24 dólares se hubiesen capitalizado a un interés anual del 12%.
12. Miguel pidió un préstamo de 200 dólares para comprar un perro. Un banco se lo está facilitando a un interés del 6% anual. Leer la letra mensual asignada por el banco e imprimir el monto pagado a interés, el monto pagado a capital y el saldo pendiente mensual en un periodo de 5 años.
13. Hacer un algoritmo que determine el valor mayor de 20 números leídos. Imprima dicho número.
14. Un programador está preocupado acerca de su rendimiento en clases de informática. En el primer programa comete una falta, en el segundo dos, en el tercero cuatro, y así sucesivamente. Parece que en cada programa comete el doble número de faltas que cometió en el programa anterior. Las clases transcurren durante trece semanas, a razón de 2 problemas de programación por semana. Hacer un algoritmo que calcule el número total de errores que puede esperar el programador, según su ritmo normal de rendimiento.
15. Escriba un algoritmo que simule el control de velocidad por radar de la policía. El problema debe leer la velocidad del automóvil e imprimir el mensaje “BIEN” si la velocidad es menor o igual a 90KM/H o “RÁPIDO”,

si es mayor. Detenga el proceso cuando la velocidad leída exceda los 160KM/H, e imprima el número de velocidades menores o iguales a 90 y el número mayores a 90.

16. La tasa de degradación radiactiva de un isótopo es dada habitualmente en términos de periodo de semidesintegración (PS: lapso necesario para que el isótopo se degrade hasta la mitad de su masa original). Para el isótopo de estroncio 90, la razón de degradación es aproximadamente de 0.60 PS. El periodo de semidesintegración del estroncio 90 es de 28 años. Calcular e imprimir, en forma tabular, el residuo después de cada año a partir de una cantidad inicial de gramos leída. Para cada año, la cantidad residual de isótopo puede calcularse de la siguiente manera:

$$R = \text{cantidad de gramos} * C^{(\text{año}/PS)}$$

$$\text{Donde } C = e^{-0.693} \qquad e = 2.71828$$

CAPÍTULO 5

Arreglos

Objetivos:

- Definir los conceptos de arreglos de una y dos dimensiones.
- Conocer los formatos de la declaración de los arreglos de una y dos dimensiones.
- Resolver problemas a través del empleo de arreglos.

Contenido

Hasta este punto, los ejercicios realizados han tenido como fin procesar cada dato por separado y reutilizar la celda en que estaba almacenado. Sin embargo, en muchas aplicaciones se requiere almacenar y manejar grandes cantidades de datos en la memoria; también existen casos en que se hace necesario guardar datos para un posterior uso.

Por ejemplo, si se desea escribir un programa que calcule e imprima el total de habitantes por provincia en un país de 50 provincias para luego ordenarlas de la menos poblada a la más poblada, se necesitaría procesar primero a todos los habitantes de cada provincia y luego hacer el ordenamiento de menos a más. Al procesar cada provincia, sería extremadamente tedioso hacer referencia a cada celda (50) con un nombre diferente (lista de variables) para almacenar cada total.

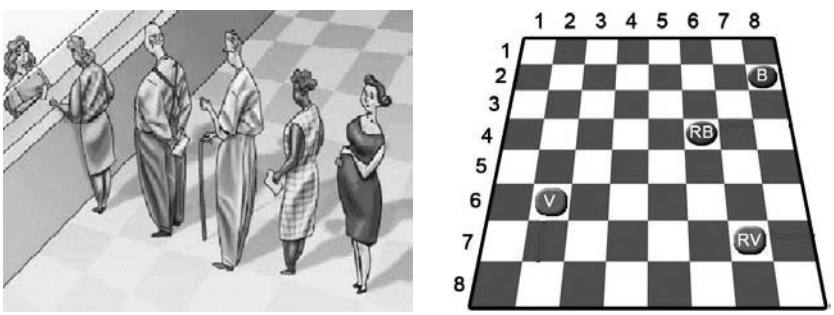
El uso de arreglos simplificaría la tarea antes mencionada al permitir referencias a los datos del conjunto agrupados en una sola estructura. Una vez almacenado el conjunto en la memoria, se podría aludir a cualquiera de los datos cuantas veces se deseara sin tener que volver a ingresarlos en la memoria.

En todos los programas desarrollados hasta ahora, cada variable empleada se utilizó para almacenar un solo dato; a continuación estudiaremos una nueva estructura que nos brinda la oportunidad de almacenar más de un dato con un solo nombre de variable.

5.1 Definición de arreglo

Es un conjunto de celdas de memoria consecutivas en el cual se guardan elementos del mismo tipo bajo un mismo nombre de variable. Los arreglos pueden alojar cualquier tipo de dato de los ya conocidos.

Una representación en el mundo real de lo que es un arreglo puede ser, por ejemplo, la fila para llegar a un cajero, o un tablero de damas con ocho filas y ocho columnas (figura 5.1).



a) Fila de clientes en el banco b) Tablero de damas

FIGURA 5.1
Ejemplos de conjuntos que pueden ser representados por arreglos de datos.

Se alude a un arreglo mediante un nombre de variable; a sus elementos se accede a través de un subíndice, es decir, un valor numérico que identifica una celda específica dentro de un arreglo.

Por ejemplo, `tabla (5)` y `arr (2,3)` son dos arreglos:

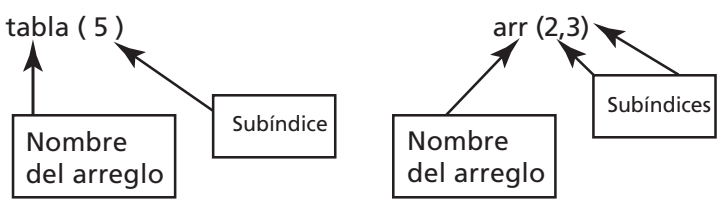


FIGURA 5.2
Nombre y subíndices.

Como estructuras de datos, los arreglos pueden ser manejados en cualquier tipo de operación que hasta el momento se haya requerido.

Los arreglos se clasifican de acuerdo con las siguientes dimensiones:

1. Arreglos unidimensionales, vectores o tablas.
2. Arreglos multidimensionales o matrices.

5.2 Tipos de arreglos

5.2.1 Arreglos unidimensionales

Un arreglo unidimensional es una lista finita de valores del mismo tipo, al cual se hace referencia con nombre común y a cuyos elementos, almacenados en forma secuencial, se alude con subíndices.

La dimensión de un vector es igual al número de elementos que lo componen.

Declaración de arreglo unidimensional

Declarar un arreglo es definir dentro del programa o algoritmo la estructura que tendrá el mismo, su nombre, su tipo de datos y la cantidad de elementos que puede contener. A un arreglo se le pueden dar valores iniciales al momento de declararlo, cuando así se requiera (figura 5.3).

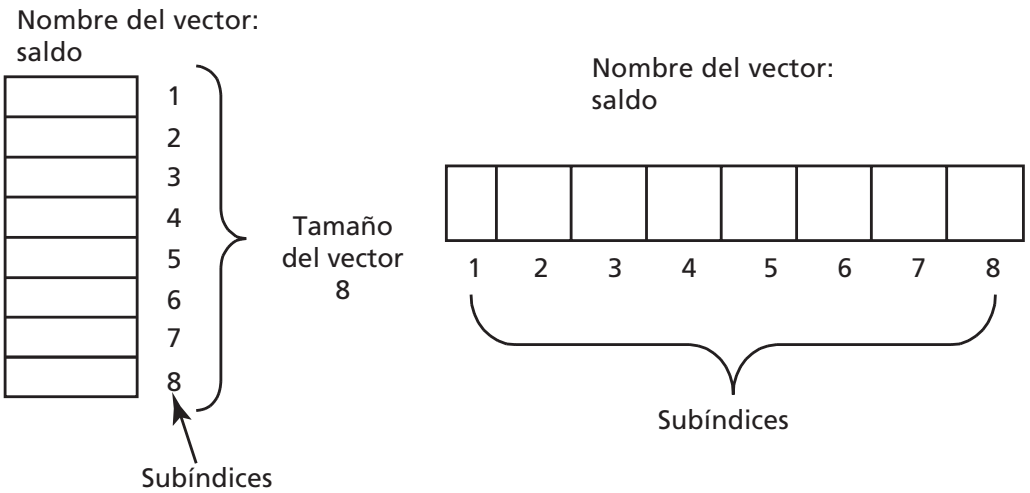


FIGURA 5.3
Representación gráfica de un vector.

Para declarar un arreglo se emplea el siguiente formato:

```
tipo de dato nombre _arreglo (tamaño) /* declaración de vectores */
/*Declaración e Inicialización de vectores */
tipo de dato nombre _arreglo (tamaño) = lista de valores separados por
➥comas
```

Donde:

```
Tipo de dato define el tipo de dato del vector.
nombre_arreglo es un nombre de identificador válido.
tamaño define la cantidad de elementos del vector.
```

El siguiente es un ejemplo de declaración de arreglo:

```
flotante vec(15), veca(10)
cadena nom(5), hombre(10)
entero canhab(9), total (100)
flotante x(4) = 1.1, 2.2, 8.3, 4.5
carácter cod (5) = 'a', 'e', 'i', 'o', 'u'
```

Cómo acceder a los elementos de un arreglo unidimensional

Para acceder a los elementos de un arreglo es necesario contar con un subíndice, el cual es cualquier constante, variable o expresión numérica positiva.

El siguiente es un ejemplo de empleo de subíndice:

```
vect(i) = vect(i) + 10
veca(5) = veca(9) * 2
nomp(i+2) = "Panamá"
suma = vect(15) + vect(10)
```

Supóngase que se tiene un vector entero llamado `vec` de cinco elementos y se quiere almacenar en él los siguientes valores: `15`, `7`, `21`, `4`, `63`. Su representación gráfica sería la siguiente:

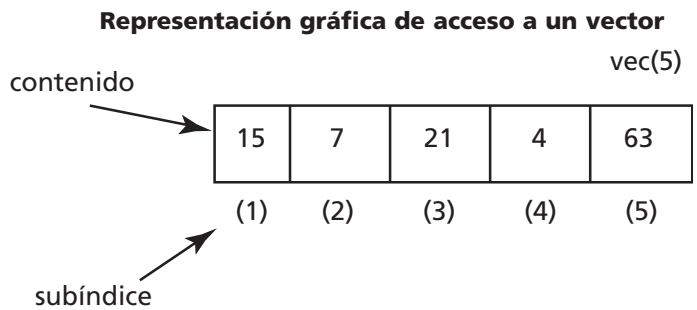


FIGURA 5.4
Almacenamiento de valores en un vector entero de cinco elementos.

Si se desea acceder al valor `21` del vector `vec`, entonces se escribe `vec(3)`, porque para acceder a un elemento del arreglo se requiere especificar la posición que tiene el elemento deseado.

Manipulación de vectores

Los vectores, al igual que cualquier variable, pueden ser procesados en cualquier operación de las que ya hemos visto: lectura, escritura y matemáticas entre otras.

A continuación veremos como se realizaría cada una de ellas:

- *Lectura de un vector.* Al igual que una variable simple, los vectores pueden ser leídos, pero como las variables de arreglo van acompañadas de un subíndice para identificar cada elemento, se requiere del uso de estructuras de repetición (más comúnmente Para). A esta operación se le denomina cargar un vector desde una lectura.

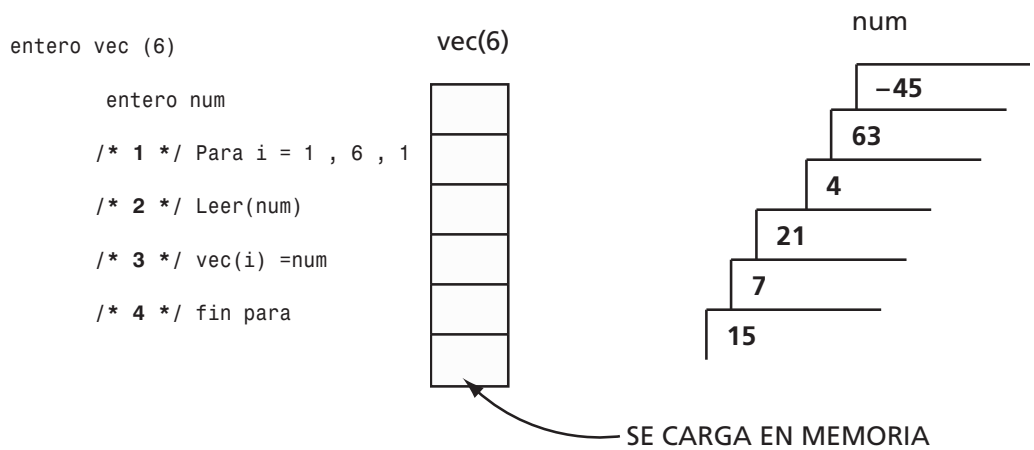


FIGURA 5.5
Representación gráfica de lectura y almacenamiento de datos en un vector a partir de carga desde lectura.

5

Siguiendo los pasos enumerados, el proceso sería:

1. Se inicializa `i =1` y pregunta si `i<=5`.
2. Se lee `num`, que toma el valor de `15`.
3. Se le asigna a `vec(i)`, cuando i =1 el valor de `num`, que es `15`.

num	I	Vec(i)
15	1	15

4. Se le suma `1` a `i`; `i=i+1` y regresa al punto 1.
   a) Pregunta si `i<=6`.

b) Se lee num, que toma el valor de `7`.

c) Se le asigna a `vec(i)` cuando `i=2`, el valor de `num`.

num	I	Vec(i)
7	2	15
		7

d) Se le suma `1` a `i`; y se regresa al punto 1, y así sucesivamente hasta que no se cumpla `i<=6`.

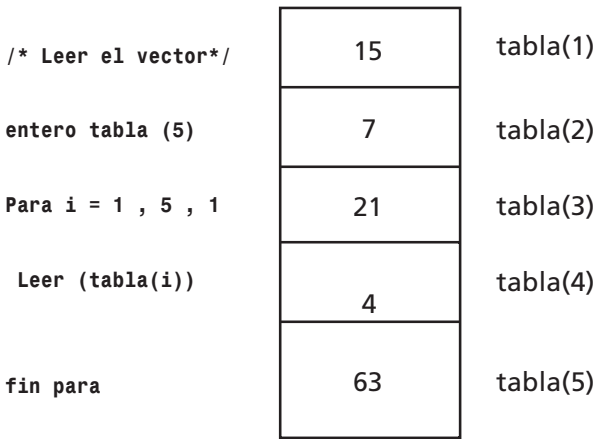


FIGURA 5.6
Representación gráfica de leer al mismo vector.

- *Impresión de un vector*. Para la salida de los resultados en pantalla o cualquier otro dispositivo de salida, se requiere trabajar con la sentencia de repetición, siempre y cuando se vaya a imprimir todo el vector; de otro modo, se puede utilizar las posiciones directas del vector.

```
entero tabla (5)
entero i
Para i = 1, 5, 1
Imprimir (tabla (i))
Fin Para
```

- *Operaciones matemáticas***.** Al igual que en los procesos de lectura e impresión, el nombre del arreglo debe ir acompañado del subíndice al momento de intervenir en cualquier operación matemática.

```
Ejemplo:
 Entero tabla (5)
 entero i
 vec (4) = 3 * 5 / 4
 :
 entero tabla (5)
 entero i = 2
 vec(i) = vec(i) + 10
```

Problema resuelto

Definición o dominio del problema

Una universidad procesa sus pruebas de admisión a los estudiantes que desean ingresar. Los estudiantes realizan sus exámenes y una vez terminados son corregidos por el sistema electrónico con que cuenta la institución. Este sistema tiene una clave de respuestas para la prueba aplicada, la cual consta de doce preguntas. Por cada estudiante que presentó la prueba se tienen los datos Identificación, nombre y respuestas del examen.

Por cada estudiante se leerán sus datos de la siguiente manera:

Una primera lectura obtiene su identificación y nombre; una segunda lectura hace lo propio con las doce respuestas dadas por el estudiante.

Se desea calcular e imprimir la calificación final obtenida por cada estudiante.

Análisis

Identificar la abstracción del problema

Nombre de la clase: Evaluacion

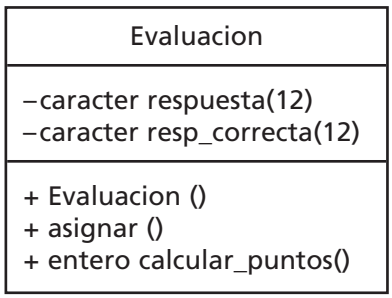
Datos: caracter respuesta(12), carácter clave(12)

Métodos: Evaluacion ()

asignar ()

entero calcular_calif ()

Diseño



Algoritmo

```
Clase Evaluacion {
 /* Declaración de datos de la
    ➥clase */
privado caracter respuesta(12)
privado caracter clave(12)
/* Implementación de métodos*/

Evaluacion (caracter correcta
➥( ) ) { /* constructor */
entero c
Para c = 1,12,1
clave ( c ) = correcta (c)
```

Programa en C++

```
#include <stdio.h>
 class Evaluacion {
 // Declaración de datos de la
clase
 private:
 char respuesta[12];
 char clave[12];
 public:
 Evaluacion(char correcta[])
 {int c;
 for(c=0;c<12;c++)
 clave[c]= correcta[c];
 }
```

```
Fin para
}
publico asignar ( caracter resp,
➥entero i ){
 respuesta ( i ) = resp
}
publico entero calcular_puntos () {
entero f
entero puntos = 0
Para f = 1, 12, 1
si (respuesta ( f ) = = clave (f )
➥entonces
 puntos = puntos + 1
fin si
Fin para
retornar puntos
}
}
INICIO
entero ident, i
entero puntos
caracter clave( 12 ) = 'b', 'c',
➥'a', 'a', 'b', 'b', 'a', 'a', 'b',
➥'c', 'a', 'a'
Evaluacion alumno ( clave )
caracter resp = 's'
caracter r
cadena nom
mientras ( resp = = 's' )
Imprimir ("Introduzca nombre e
➥identificación")
Leer ( nom, ident )
Para i = 1, 12,1
Imprimir (" teclee la respuesta de
➥la preg. # ", i )
Leer ( r )
alumno.asignar ( r, i)
Fin para
puntos = alumno.calcular_puntos( )
Imprimir ("nombre: ", nom,
➥"identificación : ", ident,
➥"Nota: ", puntos)
Imprimir ("Desea introducir otro
➥estudiante s/n")
Leer ( resp )
Fin Mientras
FIN
```

```
 void asignar(char resp, int i)
 { respuesta [ i ] = resp;
 }
int calcular_puntos( )
{int f, puntos=0;
 for(f=0;f<12;f++)
 if(respuesta[f]= =clave[f])
 puntos = puntos +1;
 return puntos
}
};
// Programa Principal
void main( ){
 int ident, i, puntos;
 static char clave[12]= {'b', 'c',
 ➥'a', 'a', 'b', 'b', 'a', 'a', 'b',
 ➥'c', 'a', 'a'};
 Evaluacion alumno(clave);
char resp = 's';
char r;
char nom[15] ;
while(resp = ='s')
{cout<<"Introduzca Nombre e
➥Identificación: ";
 gets(nom);
 cin>>ident;
 for(i=0;i<12;i++)
 {cout<<"teclee la respuesta de la
  ➥preg. #"<< i;
 cin>>r;
 alumno.asignar(r, i);
 }
 puntos = alumno.calcular_puntos ( );
 cout<< "\n Nombre:";
 puts(nom);
 cout<<"Identificación : "<<ident;
 cout<<"Nota : "<<puntos;
 cout<<" Desea continuar s/n";
 cin>>resp; }
}
```

Algoritmo

```
Clase Evaluacion {
 /* Declaración de datos de la
    ➥clase */
privado caracter respuesta(12)
privado caracter clave(12)
/* Implementación de métodos*/

Evaluacion (caracter correcta
➥( ) ) { /* constructor */
entero c
Para c = 1,12,1
clave ( c ) = correcta (c)
Fin para
}
publico asignar ( caracter resp,
➥entero i ){
 respuesta ( i ) = resp
}
publico entero calcular_puntos () {
entero f
entero puntos = 0
Para f = 1, 12, 1
si (respuesta ( f ) = = clave (f )
➥entonces
 puntos = puntos + 1
fin si
Fin para
retornar puntos
}
}
INICIO
entero ident, i
entero puntos
caracter clave( 12 ) = 'b', 'c',
➥'a', 'a', 'b', 'b', 'a', 'a', 'b',
➥'c', 'a', 'a'
Evaluacion alumno ( clave )
caracter resp = 's'
caracter r
cadena nom
mientras ( resp = = 's' )
Imprimir ("Introduzca nombre e
➥identificación")
Leer ( nom, ident )
Para i = 1, 12,1
Imprimir (" teclee la respuesta de
➥la preg. # ", i )
Leer ( r )
```

Programa en Java

```
import java.io.*;
class Evaluacion {
 // Declaración de datos de la
    ➥clase
 private char [] respuesta =
 ➥new char [12];
 private char [] clave =
 ➥new char [12];

 Evaluacion(char correcta[])
 {int c;
 for(c=0;c<=11;c++)
 clave[c]= correcta[c];
 }
 void asignar(char resp, int i)
 { respuesta [ i ] = resp;
 }
int calcular_puntos( )
{int f, puntos=0;
 for(f=0;f<=11;f++)
 if(respuesta[f] ==clave[f])
 puntos = puntos +1;
 return puntos;
}
}
// Programa Principal
class Evalua
{public static void main (String
➥arg [] )
 throws Exception {
 BufferedReader br = new
 BufferedReader (new InputStream-
 ➥Reader (System.in));
 int ident, i, puntos;
 char [] clave = {'b', 'c', 'a',
 ➥'a', 'b', 'b', 'a', 'a', 'b',
 ➥'c', 'a', 'a'};
 Evaluacion alumno =
 ➥new Evaluacion(clave);
 char resp = 's';
 char r;
 String nom;
 while(resp =='s')
 {System.out.println("Introduzca
 ➥Nombre e Identificación: ");
 nom=br.readLine();
 ident=Integer.parseInt
 ➥(br.readLine());
 for(i=0;i<12;i++)
```

```
alumno.asignar ( r, i)
Fin para
puntos = alumno.calcular_puntos( )
Imprimir ("nombre: ", nom,
➥"identificación : ", ident,
➥"Nota: ", puntos)
Imprimir ("Desea introducir otro
➥estudiante s/n")
Leer ( resp )
Fin Mientras
FIN
```

```
{System.out.println("teclee la
➥respuesta de la preg. #"+i);
r=(char)br.read();
br.skip(2);
alumno.asignar(r,i);
}
puntos = alumno.calcular_puntos ( );
System.out.println("\n Nombre:");
System.out.println(nom);
System.out.println("Identificación :
➥"+ident);
System.out.println("Nota :
➥"+puntos);
System.out.println(" Desea
➥continuar s/n");
resp=(char)br.read();
br.skip(2); }
}
}
```

5.2.2 Arreglos multidimensionales

Los arreglos multidimensionales son comúnmente llamados *matrices*. En este punto veremos solamente el manejo de arreglos *bidimensionales*, pero recordemos que se pueden manejar matrices de 3, 4 o más dimensiones, por ejemplo: `mata (2,2)` es un arreglo bidimensional, `ventas (3,4,2)` tridimensional, donde el primer arreglo hace referencia a fila-columna y el segundo ejemplo hace referencia a fila-columnas-profundidad.

Un arreglo bidimensional es aquel que se representa a través de dos dimensiones y se concibe como una matriz de filas y columnas. Llamamos *filas* a las líneas horizontales y *columnas* a las verticales. Para trabajar con arreglos de 2 dimensiones se requieren de 2 subíndices para poder acceder a cualquier elemento.

La siguiente es una representación gráfica de un arreglo bidimensional:

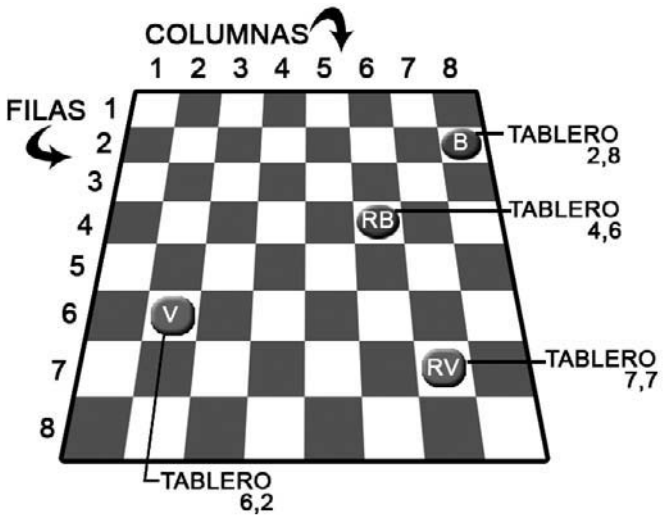


FIGURA 5.7
Representación gráfica de un arreglo bidimensional.

Un tablero de damas ilustra el concepto de una matriz de dos dimensiones. Equiparándolo, sus partes serían las siguientes: nombre de la matriz `tablero`, cantidad de filas `(8)`, cantidad de columnas `(8)` y cantidad de elementos de la matriz `(64)` [cifra que se obtiene de multiplicar las filas por las columnas `(8 * 8)`].

Declaración de un arreglo bidimensional

La declaración debe indicar la cantidad de elementos que tendrá la matriz, así como la cantidad de filas y columnas, y puede hacerse tanto en una clase como en el algoritmo principal, dependiendo de la necesidad de la aplicación.

Para declarar un arreglo bidimensional se emplea el siguiente formato:

```
tipo de dato nombre_del_arreglo (subind1, subind2)
tipo de dato nombre_del_arreglo (subind1, subind2) = lista de valores,
➥separados por coma
```

Donde:

tipo de dato define el tipo de valores que se almacenan en cada celda de la matriz.

nombre_de_arreglo es el identificador válido.

subíndice1 define la cantidad de filas de la matriz.

subíndice 2 define la cantidad de columnas de la matriz.

El siguiente es un ejemplo de declaración de arreglo bidimensional:

```
entero mat (4,4) /* mat tiene 4 filas y 4 columnas y maneja 16 elementos
flotante total (5,4) /* total tiene 5 filas y 4 columnas y maneja 20
➥elementos
booleano mat (3,8) /* mat tiene 3 filas y 8 columnas y maneja 24 elementos
```

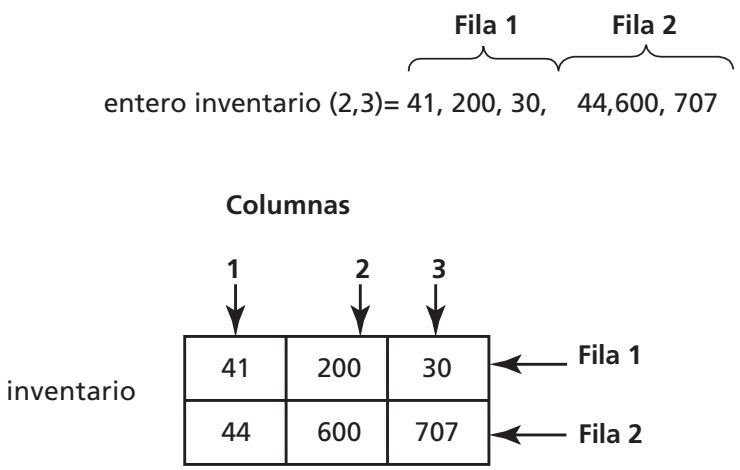


FIGURA 5.8
Representación gráfica del arreglo inventario inicializado con valores durante su declaración.

Cómo acceder a los elementos de un arreglo bidimensional

Para acceder a los elementos de una matriz se requiere de 2 variables subíndices, una para la fila y otra para la columna.

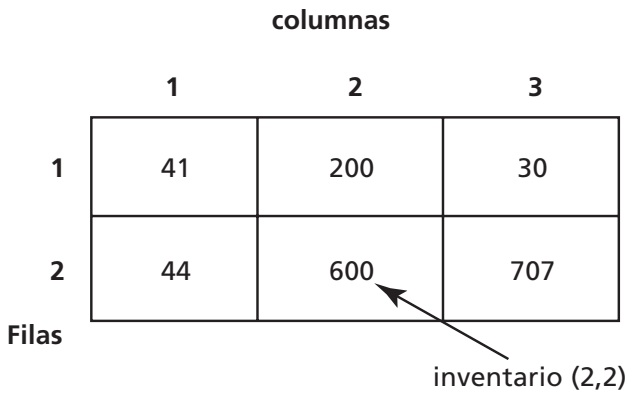


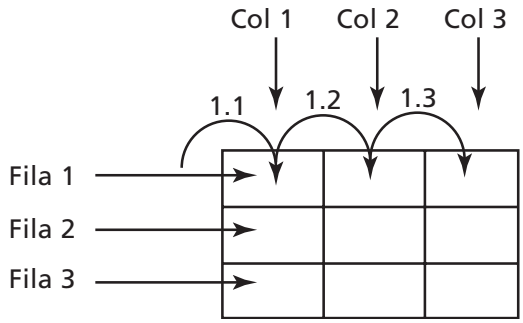
FIGURA 5.9
Representación gráfica del acceso a los elementos de una matriz.

Si se desea acceder al valor 600 de la matriz `inventario`, se debe indicar la fila y la columna en donde se encuentra almacenado el valor; esto sería: `inventario (2,2)`.

Manipulación de matrices

Para cualquier operación que se realice sobre una matriz, se debe indicar el tipo de recorrido que desea hacerse sobre ella. Para esto se requiere de estructuras repetitivas con el fin de manejar los subíndices del arreglo. Para una matriz se requiere de dos ciclos anidados: uno que controle las filas y otro que controle las columnas. Luego, el usuario debe establecer como desea recorrer a la matriz, por fila o por columna.

Recorrido por fila. Cuando se recorre a la matriz por fila, se accede a los elementos de la primera fila, luego los elementos de la segunda y así sucesivamente hasta recorrer todo el arreglo. Por ejemplo:




```
Para f = 1,3,1
 Para c = 1,3,1
 imprimir (mat (f,c))
 Fin para f
Fin para f
```

FIGURA 5.10
Representación gráfica del recorrido por fila.

En el diagrama se observa que en el recorrido por fila, el ciclo que controla las columnas varía mucho más rápido que las filas.

Recorrido por columnas. En este tipo de recorrido se accede al contenido de los elementos de la primera columna, luego a los elementos de la segunda y así sucesivamente hasta recorrer todo el arreglo.

En este recorrido, el ciclo que controla las filas varía mucho más rápido que el de las columnas.

Lectura e impresión de una matriz

Una vez definido el tipo de recorrido se procede a cargar el arreglo, a través de la lectura, para después imprimirlo.

El siguiente es un ejemplo de lectura e impresión de una matriz:

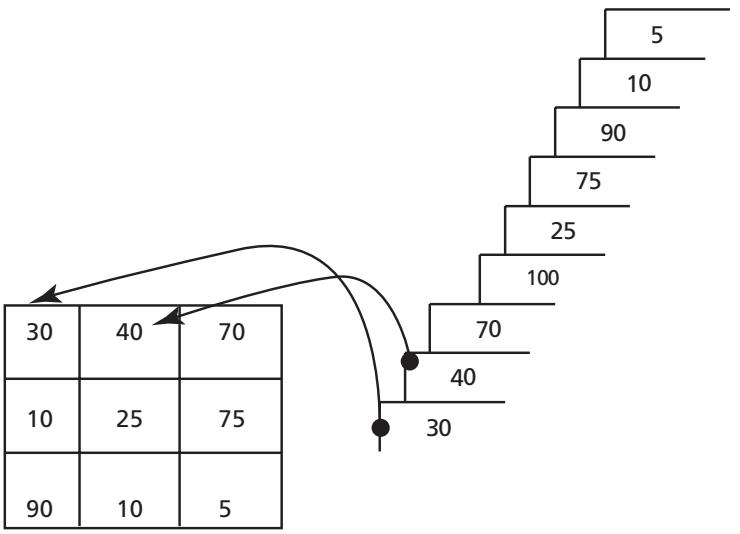


FIGURA 5.11
Representación gráfica de carga a una matriz desde registros de entrada.

```
/* Se carga por fila*/
 entero mat (3,3).
 entero f, c, num
 Para f = 1, 3,1
 Para c = 1,3,1
 mat (f,c) = num
 Fin para c
 Fin para f
```

```
/* Se imprime por columna*/
Para c = 1,3,1
Para f = 1,3,1
 imprimir ( mat ( f, c) )
Fin para f
Fin para c
```

Operaciones matemáticas:

```
/* Sumar todo los elementos de una matriz */
entero mat (3,3)
entero f, c, tot = 0
Para f = 1,3,1
 Para c = 1,3,1
 tot = mat (f,c) + tot
 Fin para c
Fin para f
```

Problema resuelto

Definición o dominio del problema

Elabore un algoritmo que cargue una **matriz (5,5)** y determine la suma de la diagonal principal; luego determine si la misma es simétrica o no.

Nota: Una matriz es simétrica si el elemento (i,j) es igual (j,i) para todos los valores de i y de j.

Análisis

Identificar la abstracción del problema

Nombre de la clase: Simetria

Datos: entero mat(5,5)

Métodos: asignar ()

booleano determinar()

entero suma ()

Diseño

Simetria
– entero mat(5,5)
+ asignar (entero, entero, entero) + booleano determinar () + entero suma()

Algoritmo

```
clase Simetria {
 privado entero mat(5,5)
 publico asignar( entero i,
 ➥entero j, entero valor)
{ mat(i,j) = valor}
publico entero determinar( ) {
 entero f, c
 entero sw = 1
 Para f = 1, 5, 1
 Para c = 1, 5, 1
 Si (( mat(f,c) <> mat(c,f)) entonces
 sw = 0
 c = 6
 f = 6
 fin si
 Fin Para
 Fin Para
retornar sw }

publico entero suma () {
 entero f
 entero sum = 0
 Para f = 1,5,1
 sum= sum + mat (f,f)
 Fin para f
 retornar sum
}
} }

INICIO
 entero f,c, num,sw,sum
 Simetría obj;
 Para f = 1, 5, 1
 Para c = 1, 5, 1
 imprimir ("Entre un numero entero")
 leer (num)
 obj.asignar (f,c,num)
 Fin para f
 Fin para c
 sw = obj.determinar()
 if (sw == 1) entonces
 Imprimir (" La Matriz es
 ➥Simétrica");
 sum = obj.suma ()
 Imprimir ("La suma de los
 ➥elementos de la diagonal
 ➥principal = " , sum)
FIN
```

C++

```
#include <iostream.h>
#include <conio.h>
class Simetria
{ private: int mat[2][2];

 public: void asignar (int f, int c,
 ➥int num)
 { mat[f][c]=num; }

 public: int determinar()
 { int f, c, sw= 1;
 for (f=0;f<=1;++f)
        for (c=1;c<=1;++c)
        if (mat[f][c] != mat[c][f])
        { sw = 0;
                c = 6;
                f = 6; }
        return sw;
 }
 public: int suma()
 { int f, sum=0;
 for (f=0;f<=1;++f)
        sum=sum + mat[f][f];

 return sum;
 }
 };
 main()
 { int f, c, num, sw,sum;
 Simetria obj;
 clrscr();
 for (f=0;f<=1;++f)
        for (c=0;c<=1;++c)
        { cout<<"Entre un numero
          ➥entero";
        cin>>num;
        obj.asignar(f,c,num); }
 sw =obj.determinar();
 if (sw)
        cout<<"\nLa matriz es
        ➥simetrica";
 sum=obj.suma();
 cout<<"\nLa suma de los elementos
 ➥de la diagonal principal="<<sum;
 getch();
 return 0;

 }
```

Algoritmo

```
clase simetria {
 privado entero mat(5,5)
 publico asignar( entero i, entero j,
 ➥entero valor)
{ mat(i,j) = valor}
publico entero determinar( ) {
 entero f, c
 entero sw = 1
 Para f = 1, 5, 1
 Para c = 1, 5, 1
 Si (( mat(f,c) <> mat(c,f))
 ➥entonces
 sw = 0
 c = 6
 f = 6
 fin si
 Fin Para
 Fin Para
retornar sw }

publico entero suma () {
 entero f
 entero sum = 0
 Para f = 1,5,1
 sum= sum + mat (f,f)
 Fin para f
 retornar sum
}
} }

INICIO
 entero f,c, num,sw,sum
 Simetría obj;
 Para f = 1, 5, 1
 Para c = 1, 5, 1
 imprimir ("Entre un numero entero")
 leer (num)
 obj.asignar (f,c,num)
 Fin para f
 Fin para c
 sw = obj.determinar()
 if (sw == 1) entonces
 Imprimir (" La Matriz es
 ➥Simetrica");
 sum = obj.suma ()
 Imprimir ("La suma de los elementos
 ➥de la diagonal principal = " , sum)
FIN
```

```
Java
import java.io.*;
class Simetria
{ private int [] []mat = new int
  ➥[2][2];


 public void asignar (int f, int c,
 ➥int num)
 { mat[f][c]=num; }

 public int determinar()
 { int f, c, sw= 1;
 for (f=0;f<=1;++f)
         for (c=1;c<=1;++c)
         if (mat[f][c] !=
         ➥mat[c][f])
         { sw = 0;
                 c = 6;
                 f = 6; }
         return sw;
 }
 public int suma()
 { int f, sum=0;
 for (f=0;f<=1;++f)
         sum=sum + mat[f][f];

 return sum;
 }
 }
 class Simetriap
 { public static void main (String
   ➥arg[])
 throws Exception
 { BufferedReader br = new
 BufferedReader (new InputStream-
 ➥Reader (System.in));
 int f, c, num, sw,sum;
 Simetria obj = new Simetria();

 for (f=0;f<=1;++f)
         for (c=0;c<=1;++c)
         { System.out.println
         ➥("Entre un numero");
         num= Integer.parseInt
         ➥(br.readLine());
         obj.asignar(f,c,num);
         }
 sw =obj.determinar();
 if (sw == 1)
```

```
        System.out.println
        ➥("\nLa matriz es
        simetrica");
sum=obj.suma();
System.out.println("\nLa suma de
➥los elementos de la diagonal
➥principal="+sum);
} }
```

5.2.3 Parámetros por referencia

Todos los ejemplos que hemos visto anteriormente han sido procesados con el concepto de paso por valor. Recordemos que cuando se pasa un parámetro por valor a un método, el parámetro será una copia del valor transferido.

Por ejemplo, si tenemos este segmento del algoritmo:

```
entero n
n = 25
Saludar (n) /* llamada al método, mandando el valor 25 */
Imprimir (n)
:
:
publico saludar( entero numero) /* parámetro por valor : numero que
➥recibe 25 */
{ Imprimir (numero) /* imprimirá 25 */
numero = 34 /* modifica el valor 25 que tenía numero */
Imprimir (numero) /* imprimirá 34 */
}
```

El parámetro `numero` del método `Saludar` recibe una copia del argumento `n` de la llamada (en el ejemplo, el valor `25`). Cualquier cambio que se realice en la variable `numero` afectará sólo a la copia (`numero`) y no al original (`n`). Así, cuando se imprima `n` una vez ejecutado el método `saludar`, esta variable mantendrá su valor de `25`.

Definición de paso por referencia

El paso por referencia consiste en enviar la dirección de memoria de una variable a un parámetro. Esto ocasiona que cualquier cambio al parámetro en el método afecte a la variable original.

Existen estructuras de datos con características para recibir una dirección de memoria, tales como los arreglos y los objetos.

Paso de arreglos a métodos

El paso de arreglos a métodos se puede llevar a cabo de dos formas:

- Un elemento del arreglo (paso por valor).
- El arreglo completo (paso por referencia).

Un elemento del arreglo. Se debe especificar el elemento del arreglo que se desea enviar al método. Esto se logra especificando el nombre del arreglo y los subíndices del elemento que se desea enviar.

Por ejemplo:

```
_____
_____
_____
entero vec(3)
entero r
r = proc(vec(2))
 _ _ _ ──────────► x recibe el valor del elemento
 _ _ _             2 del vector vec
publico int proc (entero x)
{
 entero suma = 5
 suma = suma + x
retornar suma
}
```

Un arreglo completo. Se debe enviar el nombre del arreglo al método.

Al pasar un arreglo como argumento a un método, el nombre del arreglo pasa la dirección de memoria (o referencia) del primer elemento del vector, lo cual permitirá acceder a ese arreglo en el método.

Para ello, en la llamada al método se especifica el nombre del arreglo sin índice y en el método se define el parámetro formal con paréntesis en blanco.

Por ejemplo: si se declara un vector de 3 elementos, `vec(3)`, se reserva memoria para 3 valores.

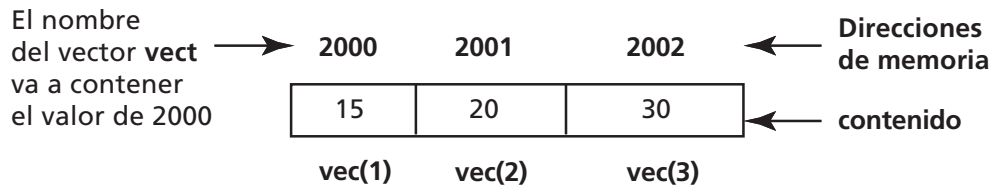


FIGURA 5.12
Representación gráfica.

El siguiente ejemplo ilustra esta operación:

```
_____
_____
_____
entero vect (3)

_______
proc(vec) /* Llamada al método proc */
 - - -
 - - -
publico proc (entero tab( ))
{
 entero i, suma
 para i = 1, 3
 tab(i) = tab(i) + 2
 fin para
}
```

tab recibe la dirección del primer elemento de **vec** , 2000

En este ejemplo, el nombre del vector `vec` guarda la dirección de memoria `de` su primer elemento (`2000`), tal como se observa en la figura 5.12. Por ello, `tab` recibe esa dirección o referencia y, a partir de ahí, se puede acceder a los elementos de `vec`. También se puede decir que la variable `tab` apunta a `vec(1)`.

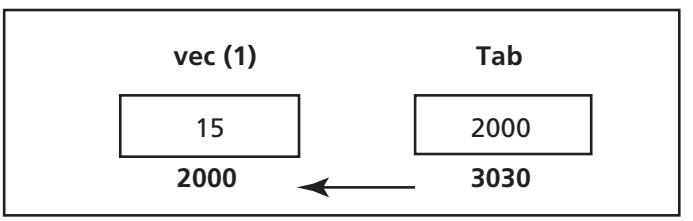


FIGURA 5.13
La variable tab apunta a vec (1).

Si imprimiéramos el vector `vec` después del llamado del método `proc`, veríamos que los elementos contienen su valor incrementado en `2`.

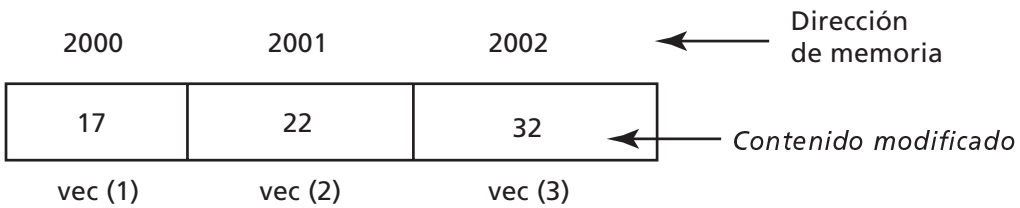


FIGURA 5.14
La memoria y su contenido modificado.

5.3 Procedimientos de búsqueda en arreglos lineales

5.3.1 Secuencial

Es la técnica más simple para buscar un elemento específico en una lista de datos. Este método consiste en recorrer al arreglo, elemento por elemento, comenzando con el primero de la lista; se utiliza en listas no ordenadas. En términos de eficiencia, es preciso considerar un contador que represente dicha posición.

Segmento de método de búsqueda secuencial

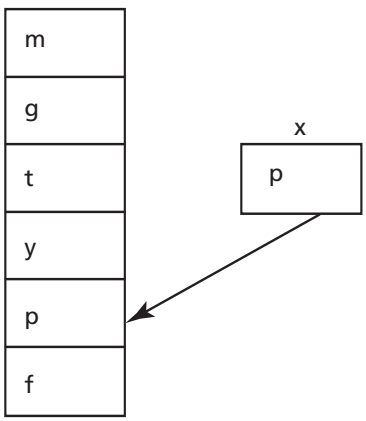


FIGURA 5.15
Búsqueda de un elemento en un vector.

```
publico entero secuencial (caracter x)
{ entero bandera = 0
entero indice = 0, i = 1
Mientras ((bandera == 0 ) O ( i <= 6))
si (vec (i ) == x) Entonces
         bandera = 1
         indice = i
Fin Si
         i = i + 1
Fin Mientras
retornar indice
}
```

Problema resuelto

Definición o dominio del problema

Se tiene un vector de tipo carácter de seis posiciones; debe leerse un carácter e indicar si el mismo se encuentra en el vector. El algoritmo debe finalizar cuando se encuentre el elemento buscado.

Análisis

Identificar la abstracción del problema

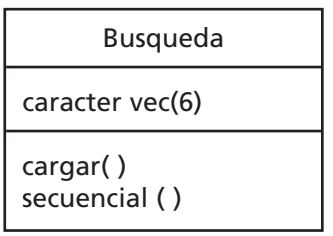
Nombre de la clase: Busqueda

Datos : caracter vec(6)

Métodos : cargar ( )

Secuencial( )

Diseño



Algoritmo

```
clase Busqueda
{ privado caracter vec(6)
 publico cargar(caracter valor,
 ➥entero i)
 { vec(i) = valor }

 publico entero secuencial
 ➥(caracter x)
 { entero bandera = 0
 entero indice = 0, i = 1
 Mientras ((bandera == 0 ) O
 ➥( i <= 6))
 si (vec (i ) == x) Entonces
           bandera = 1
           indice = i
 Fin Si
           i = i + 1
 Fin Mientras
 retornar indice
 }
}
INICIO
{ caracter valores
 entero veces, resp
 Busqueda obj

 Para veces=1, veces, 1
 Imprimir ("Teclee una letra:")
 Leer (valores)
 obj.cargar (valores,veces)
 Fin Para
 Imprimir ("Introduzca el caracter
 ➥a buscar:")
 Leer (valores)
 resp= obj.secuencial (valores)
 Si (resp == 0) Entonces
 Imprimir ("No se encontro",
 ➥valores, " en el vector")
 De Otro Modo
 Imprimir ("El caracter" , valores ,
 ➥" esta en la posición", resp)
Fin Si
FIN
```

Programa en C++

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
class Busqueda
{ private:
 char vec[6];
 public:
 void cargar(char valor, int i)
 { vec[i] = valor;}

 int secuencial (char x)
 { int bandera = 0;
 int indice = 0, i = 0;
 while (bandera == 0 ¦¦ i <= 5)
 { if (vec[i] == x)
             {bandera = 1;
             indice = i;
             }
             i = i + 1;
 }
 return indice;
 }
};
main()
{ char valores;
 int veces,resp;
 Busqueda obj;
 clrscr();
//los vectores en C comienzan en la
  ➥posición 0
 for (veces=0; veces<6; ++veces)
 { printf ("Teclee una letra:");
 scanf("%c",&valores);
 fflush(stdin);
 obj.cargar (valores,veces);
 }
 printf ("\nIntroduzca el caracter
 ➥a buscar:");
 scanf ("%c", &valores);
 resp= obj.secuencial (valores);
 if (resp == 0)
 printf("\nNo se encontro %c en el
 ➥vector",valores);
 else
 printf ("\nEl caracter %c esta en
 ➥la posicion %i",valores,resp);
getch();
return 0;
}
```

Algoritmo

```
clase Busqueda
{ privado caracter vec(6)
 publico cargar(caracter valor,
 ➥entero i)
 { vec(i) = valor }

 publico entero secuencial
 ➥(caracter x)
 { entero bandera = 0
 entero indice = 0, i = 1
 Mientras ((bandera == 0 ) O
 ➥( i <= 6))
 si (vec (i ) == x) Entonces
        bandera = 1
        indice = i
 Fin Si
        i = i + 1
 Fin Mientras
 retornar indice
 }
}
INICIO
{ caracter valores
 entero veces, resp
 Busqueda obj

 Para veces=1, veces, 1
 Imprimir ("Teclee una letra:")
 Leer (valores)
 obj.cargar (valores,veces)
 Fin Para
 Imprimir ("Introduzca el caracter
 ➥a buscar:")
 Leer (valores)
 resp= obj.secuencial (valores)
 Si (resp == 0) Entonces
 Imprimir ("No se encontro",
 ➥valores, " en el vector")
 De Otro Modo
 Imprimir ("El caracter" , valores ,
 ➥" está en la posición", resp)
Fin Si
FIN
```

Programa Java

```
import java.io.*;
class Busqueda
{ private char [] vec = new char [6];

 void cargar(char valor, int i)
 { vec[i] = valor;}

 int secuencial (char x)
 { int bandera = 0;
 int indice = 0, i = 0;
 while (bandera == 0 ¦¦ i <= 5)
 { if (vec[i] == x)
        {bandera = 1;
        indice = i;
        }
 i = i + 1;
 }
 return indice;
 }
}
class Busquedap
{ public static void main
➥(String arg[])
 throws java.io.IOException {
 BufferedReader br = new
 BufferedReader (new InputStream-
 ➥Reader (System.in));
 char valores;
 int veces,resp;
 Busqueda obj = new Busqueda();
 for (veces=0; veces<=5; ++veces)
 { System.out.println ("Teclee una
 ➥letra:");
 valores=(char) br.read();
 br.skip(2);
 obj.cargar (valores,veces);
 }
 System.out.print("\nIntroduzca el
 ➥caracter a buscar:");
 valores= (char)br.read();
 br.skip(2);
 resp= obj.secuencial (valores);
 if (resp == 0)
 System.out.print("\nNo se encontro
 ➥"+valores+" en el vector");
 else
 System.out.print ("\nEl caracter
 ➥"+valores+" está en la posicion
 ➥"+resp);
} }
```

5.4 Métodos de clasificación en arreglos lineales

La clasificación de un grupo en un orden secuencial, de acuerdo con ciertos criterios, puede llevarse a cabo en forma creciente en el ordenamiento numérico. En programación, a este ordenamiento se le denomina *descendente* o *ascendente* respectivamente.

Existen dos métodos: *Push Down* y *Exchange*.

5.4.1 Método Push Down

En este método se trabaja con una tabla de *n* elementos que requiere de un ordenamiento ascendente. Esto se lleva a cabo comparando el primer elemento de la tabla para determinar si es más grande que el segundo; si lo es, se intercambian los contenidos. A continuación se comparan el primero con el tercero y así sucesivamente hasta hacer la comparación n-1.

```
/* Método Push Down */
publico push_Down ( entero veca ( ), entero n ) {

entero i, j, temp
i = 0
Para i = 1, n-1, 1
       Para j = i + 1 , n, 1
       Si veca (i) > veca( j ) Entonces
                     Temp = veca( i )
                     Veca ( i ) = veca ( j )
                     Veca ( j ) = temp
              Fin Si
       Fin Para
Fin Para
}
```

5.4.2 Método Exchange

En este método se compara el primer elemento con el segundo; si el primero es mayor, se intercambian los contenidos. A continuación se compara el segundo con el tercero y el proceso continúa hasta que se compara el penúltimo de la tabla

(n-1 comparaciones). El proceso se repite para el segundo elemento y así sucesivamente hasta ordenar la tabla completamente.

```
/* Método Intercambio */
 publico Exchange ( entero veca( ), entero n )

entero i, j, temp
cadena orden
i = 1
Repetir
       orden = "cierto"
       Para j = 1, n, 1
              Si vec(j) > vec(j+1) Entonces
                     temp = vec(j)
                     vec(j) = vec(j+1)
                     vec(j+1) = temp
                     orden = "falso"
              Fin Si
       Fin Para
       i = i + 1
Hasta ( i > n O orden = = "cierto" )
}
```

Problema resuelto

Definición o dominio del problema

Cargar un vector de 5 elementos enteros y ordenar el vector en forma ascendente.

Imprimir el vector antes y después de ordenado.

Análisis

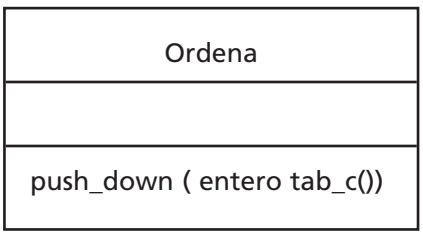
Identificar la abstracción del problema

Nombre de la clase: Ordena

Datos:

métodos : push_down (entero tab_c ( ))

Diseño




```
Algoritmo
class Ordena
{
 //método de ordenamiento PUSH DOWN
 publico push_down(entero tab_c( ))
 {entero i,j,temp
 Para i=1,4,1
 Para j=i+1, 5,1
        Si (tab_c(i) > tab_c(j)
        temp = tab_c[i]
        tab_c(i) = tab_c(j)
        tab_c(j) = temp
        Fin si
        Fin Para
 Fin Para
 }
INICIO
 entero tab(5)
 enetro i
 Ordena obj
```

```
Programa C++
#include <stdio.h>
#include <conio.H>
class Ordena
{
 public:
 //método de ordenamiento PUSH DOWN
 void push_down(int tab_c[])
 {int i,j,temp;
//los vectores en C empiezan en 0
 for (i=0;i<4;i++)
 for (j=i+1;j<5;j++)
         {if (tab_c[i] > tab_c[j])
         {temp = tab_c[i];
         tab_c[i] = tab_c[j];
         tab_c[j] = temp;
         }
         }
 }
};
void main()
```

```
 Imprimir ("Entre los elementos")
 Para i=1,5,1
 Imprimir("tab(%i) : ",i)
 Leer(tab(i))
 Fin Para
 obj.push_down(tab)
 Imprimir("Tabla ordenada");
 Para i=1,5,1
 Imprimir("tab(",i,"]")
 Fin Para
FIN
```

```
{ int tab[5];
 int i;
 Ordena obj;
 clrscr();
 printf("\nEntre los elementos");
 for (i=0;i<=4;i++)
 {printf("\n tab[%i] : ",i);
 scanf("%i",&tab[i]);
 }
 obj.push_down(tab);
 printf("\n\nTabla ordenada\n");
 for (i=0;i<=4;i++)
 printf("\ntab[%i] : %i",i,tab[i]);

getch();
}
```

Algoritmo

```
class Ordena
{
//método de ordenamiento PUSH DOWN
 publico push_down(entero tab_c( ))
 {entero i,j,temp
 Para i=0,4,1
 Para j=i+1, 5,1
        Si (tab_c(i) > tab_c(j)
        temp = tab_c[i]
        tab_c(i) = tab_c(j)
        tab_c(j) = temp
        Fin si
        Fin Para
 Fin Para
 }
INICIO
 entero tab(5)
 enetro i
 Ordena obj

 Imprimir ("Entre los elementos")
 Para i=1,5,1
 Imprimir("tab(%i) : ",i)
 Leer(tab(i))
 Fin Para
 obj.push_down(tab)
 Imprimir("Tabla ordenada");
```

Programa Java

```
import java.io.*;
class Ordena
{
 //método de ordenamiento PUSH DOWN
 void push_down(int tab_c[])
 {int i,j,temp;
 for (i=0;i<=3;i++)
 for (j=i+1;j<=4;j++)
         {if (tab_c[i] > tab_c[j])
         {temp = tab_c[i];
         tab_c[i] = tab_c[j];
         tab_c[j] = temp;
         }
         }
 }
}
class Ordena_p
{ public static void main(String
  ➥args[])
 throws java.io.IOException {
 BufferedReader br = new
 BufferedReader (new InputStream-
 ➥Reader (System.in));
 int tab [];
 tab = new int [5];
 int i;
 Ordena obj = new Ordena();
 System.out.print("\nEntre los
 ➥elementos\n");
```

```
 Para i=1,5,1
 Imprimir("tab(",i,"]")
 Fin Para
FIN
```

```
 for (i=0;i<=4;i++)
 {System.out.print(" tab["+i+"] : ");
 tab[i]= Integer.parseInt(br.read-
 ➥Line());
 }
 obj.push_down(tab);
 System.out.println("\n\nTabla
 ➥ordenada\n");
 for (i=0;i<=4;i++)
 System.out.println("\ntab["+i+"] :
 ➥"+i+tab[i]);
}
}
```

Problemas propuestos

Problemas para manejo de vectores

1. Escriba un algoritmo que cargue un vector de 5 elementos; luego sume todos los elementos del vector. Imprima al vector y la suma.
2. Dado un conjunto de 50 números enteros, desarrolle los siguientes procesos:
   - Cargue en el vector los datos de entrada.
   - Obtenga una sumatoria de todos los números pares almacenados en las posiciones pares del vector.
   - Obtener una sumatoria de todos los números impares almacenados en el vector.
   - Imprimir el vector, ambas sumatorias y un mensaje que defina si las sumatorias son iguales o no.
3. Una clase de 35 estudiantes hace un examen; sus calificaciones se almacenan en un vector R. Elabore un algoritmo que encuentre la cantidad de estudiantes con notas perfectas, es decir, calificaciones de 100. Imprima el resultado.
4. A partir del planteamiento del problema 3, calcule el promedio de las notas y el número de estudiantes con notas menores que 60. Imprima los resultados.
5. Se tienen las calificaciones de 50 estudiantes. Elabore un algoritmo que calcule el promedio de las calificaciones e imprima todas aquellas notas que sean superiores al promedio.

6. Diseñe un algoritmo que lea un conjunto de 50 datos de entrada y cargue una tabla de 100 elementos de manera que cada valor leído quede en la posición par del vector.
7. Cree un algoritmo que lea un vector sin ordenamiento con las estaturas de 20 estudiantes. Imprima la estatura del más alto y la posición en que se encuentra.
8. Elabore un algoritmo que cargue un vector de 5 elementos y determine la suma y el producto de todos los elementos del vector. Imprima la suma y el producto.
9. Cree un algoritmo que considere un vector de 100 elementos y luego lea un conjunto de 51 valores y los almacene en el vector. A continuación, debe leerse un valor numérico e investigar el número de veces que el valor leído aparece en el vector. Imprima el valor leído y el número de veces que se repite en el vector.
10. Sean nombre y sexo dos vectores que contienen el nombre y el sexo de cada uno de los miembros de un club juvenil de 25 miembros cada uno. Exprese Masculino y Femenino mediante ('`m`', '`f`'), respectivamente. Escriba un algoritmo que pueda generar dos nuevos vectores, llamados `ma` y `fe`, de tal forma que `ma` contenga el nombre de todos los hombres en orden alfabético y `fe` contenga el nombre de todas las mujeres en el mismo orden.

Problemas para manejo de matrices

1. Escriba un algoritmo que lea una matriz de 36 elementos por columnas y visualice las mismas por filas. A continuación encuentre los elementos mayor y menor y sus posiciones.
2. Cargue una matriz 4*4 con números, luego sume los elementos de cada fila y guárdelos en un vector `fila`; luego sume todos los elementos de cada columna y guárdelos en un vector `columna`.
3. Cargue dos matrices cuadradas y realice las siguientes operaciones: `SUMA`, `RESTA`. Deberá imprimir cada matriz al lado de la otra.
4. Una agencia de ventas de vehículos distribuye 15 modelos diferentes y tiene en su planilla a 10 vendedores. Elabore un algoritmo que muestre la siguiente información:
    - Un informe mensual de las ventas por vendedor y modelo.

- Un informe mensual del total de autos vendidos por vendedor y modelo.
- Y por último se desea entregar un premio al mejor vendedor, para ello considere como criterio de selección el que más autos vendió.

5. Industria Moderna S. A. le encarga diseñar un algoritmo que refleje las ventas por departamento (totales por departamento); para ello se cuenta con los siguientes datos de entrada: código de departamento, nombre del departamento, código de artículo, monto de la venta.

   El almacén cuenta con 5 departamentos y cada departamento con 10 artículos.

   Salida deseada:

   INDUSTRIAS MODERNAS S.A.
   VENTAS POR DEPARTAMENTO

   DEPARTAMENTO: x - - - - - - - -x

COD. DE ARTÍCULO	MONTO DE LA VENTA
99	999.99

   TOTAL DEL DEPARTAMENTO: 9999.99

6. Con base en el anterior planteamiento, elabore un algoritmo que realice los siguientes procesos:
   - Determinar el departamento de mayor venta.
   - Calcular el promedio de ventas por departamento.
   - Determinar el artículo de mayor venta en el almacén.
7. Cargue una matriz A (8,8) de la siguiente manera:
   - Llene con ceros (0) la diagonal principal.
   - Llene con unos (1) la que quede encima de la diagonal principal.
   - Llene con números (2) la que quede debajo de la diagonal principal.
   - Imprima la matriz.

CAPÍTULO 6

Herencia

Objetivos:

- Definir el concepto de herencia.
- Identificar los modificadores de acceso implicados en la herencia.
- Demostrar la diferencia entre constructores con y sin argumentos en la herencia.
- Resolver problemas que apliquen los conceptos de herencia en algoritmos.

Contenido

Otra característica básica de la orientación a objetos es la herencia. En los capítulos anteriores hemos visto la utilidad de la abstracción, expresada a través de la encapsulación de información e implementada por medio de los conceptos de clase y objeto.

La herencia es un mecanismo de abstracción consistente en la capacidad de derivar nuevas clases a partir de otras ya existentes. La herencia permite reutilizar código y mantener la complejidad de las nuevas clases dentro de límites manejables.

6.1 ¿Qué es herencia?

La herencia es una técnica de desarrollo de software muy potente que relaciona los datos y métodos de clases nuevas con los de clases ya existentes, de forma que la nueva clase se puede entender como una extensión de la antigua. Mediante el uso de componentes de software ya existentes para crear otros, puede sacarse todo el partido del esfuerzo realizado durante el diseño, la implementación y las pruebas del software existente.

Se entiende por herencia el proceso de crear clases, llamadas *clases derivadas* o *subclases*, a partir de una existente, la *clase base* o *superclase*. La nueva clase contendrá automáticamente algunos o todos los elementos (variables y métodos) de la clase original (y de todos sus ascendentes) dependiendo de los modificadores de acceso.

Las clases derivadas pueden heredar el código y los datos de la clase base, añadiendo su propio código y datos, e incluso cambiar aquellos elementos de la clase base que necesita sean diferentes.

Por ejemplo, todos los vertebrados comparten la característica de tener huesos.

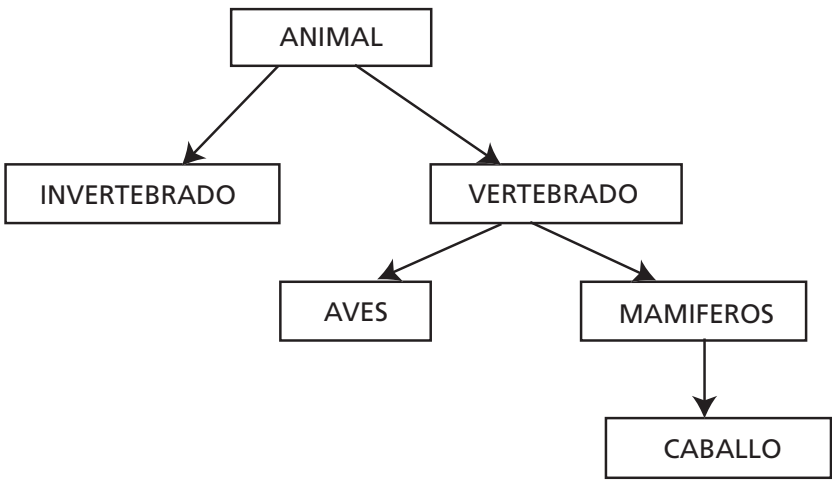


FIGURA 6.1
Parte de la organización jerárquica del reino animal. (Muñoz, Niño, Vizcaíno; Introducción a la POO.)

Para declarar una clase aplicando la herencia se emplea el siguiente formato:

```
clase   Clase_derivada  hereda  Clase_base
{               :
                .
}
```

Nota: La palabra **hereda** establece la relación jerárquica

6.2 Modificadores de acceso en la herencia

Una clase derivada no puede acceder a los miembros `privados` de su clase base; permitirlo violaría el encapsulado de la clase base. Una clase derivada puede, sin embargo, acceder a los miembros `públicos` y `protegidos` de su clase base. El acceso protegido (#) sirve como nivel intermedio de protección entre el acceso público y el acceso privado.

Una clase derivada puede tener acceso a los miembros privados de la clase base mediante funciones de acceso provistas en la interfaz pública de la clase base. Una interfaz es un método público que sirve como puente para tener acceso a un miembro privado. La siguiente figura ilustra lo anterior:

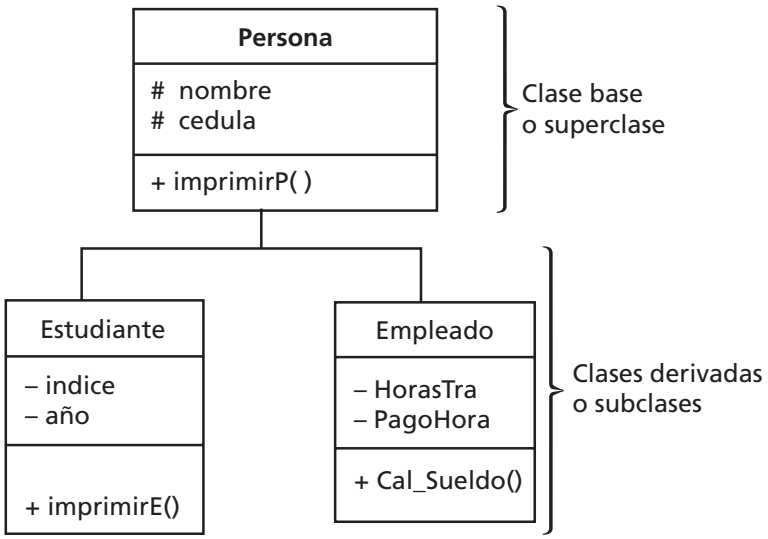


FIGURA 6.2
Acceso de la subclase a miembros de la superclase.

El siguiente ejemplo de clases permite entender mejor el seudocódigo correspondiente:

```
 clase Persona
 { protegido cadena nombre
    protegido cadena cedula
    publico imprimirP() {...}
 }
clase Estudiante hereda Persona
{privado flotante indice
  privado entero año
  publico imprimirE(){...}
}

clase Empleado hereda Persona
{ privado flotante horasTra
   privado flotante pagoHora
   publico flotante cal_Sueldo{...}
}
```

En esta jerarquía de `Persona`, `Estudiante` y `Empleado`, la clase `Estudiante` y `Empleado` heredan, de la clase `Persona`, los miembros `nombre` y `cedula`, y el método `imprimirP()`. La clase `Estudiante`, además de los miembros heredados, tiene sus propios miembros: `indice`, `año` y el metodo `imprimirE()`. Así que al crear un objeto llamado `obj_Est`, a partir de la clase `Estudiante`, éste tiene los datos `nombre`, `cedula`, `indice` y `año`, y los métodos `imprimirP()` e `imprimirE()`. De igual forma sucede al instanciar un objeto de la clase `Empleado`.

6.3 Qué no se hereda

Ya sea por restricciones de acceso o efectos del comportamiento de la metodología de orientación a objetos, no se heredan algunos elementos de las clases que serán utilizadas como superclases tales como:

- Los miembros privados (datos y métodos).
- Los constructores.
- Los destructores.

6.4 Constructores en la herencia

Un constructor definido en la superclase debe estar coordinado con los de la subclase. En particular, se debe considerar la `forma` en que el constructor de la superclase recibe valores de la subclase para crear el objeto completo.

La subclase debe definir un constructor que llama al constructor de la superclase, proporcionándole los argumentos (valores) requeridos.

Si un constructor se define para la superclase y otro para la subclase, la llamada del constructor de la superclase se hace en el constructor de la subclase y, por lo tanto, el mismo se ejecutará primero; una vez que el constructor de la superclase termine sus tareas, se ejecuta el constructor de la subclase.

Para ello se emplea el siguiente formato:

publico Subclase (lista parámetros) : Superclase (lista de argumentos)
{.....
}

Nota: El operador : denota el llamado al constructor de la superclase

El siguiente ejemplo ilustra el uso de constructores con parámetros:

```
clase Principal{
    privado entero campo1
     publico Principal (entero n )
        { campo1 = n  }
   }

clase Deriva hereda Principal {
    privado  entero valor
    publico Deriva (entero x, entero t ): Principal  (x)
    { valor = t }
 }

         :
         :

     Deriva   Obj_p1(5,6) /*  declaración del objeto de la subclase */
```

En el ejemplo, el constructor de la subclase debe recibir dos valores: uno para la variable de la superclase (variable `x`) y otro para la variable de la subclase (variable `t`).

Al declarar un objeto

```
Deriva   Obj_p1(5,6)
```

el valor 5 será mandado a la variable `x`; en consecuencia, la variable de la clase Principal, `campo1`, recibirá dicho valor. El valor 6, será mandado a la variable `t`, y asignado a la variable de la clase `Deriva`.

Sin embargo, pueden existir constructores `sin` parámetros, por lo cual no es necesario hacer un llamado al constructor de la superclase. El siguiente *Problema resuelto* presenta un caso de este tipo.

Problema resuelto

Definición o dominio del problema

Diseñar una clase `Calculo` que sume n números enteros leídos del teclado. Con base en la clase `Cálculo`, diseñar una clase `Promedio`, que obtenga el promedio de n números enteros e imprima la suma y el promedio.

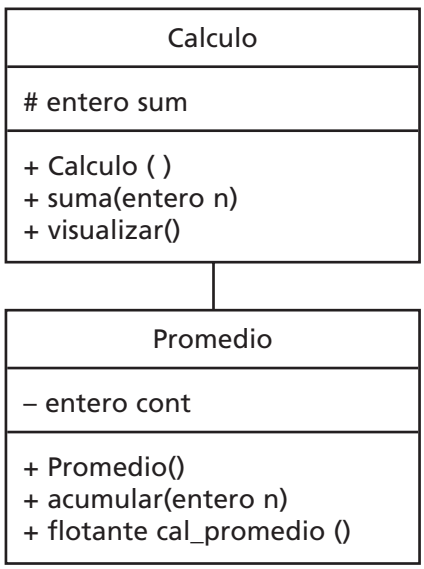
Metodología de trabajo

Análisis

Identificar la abstracción del problema

Nombre de la Clase: Calculo	Nombre de Clase: Promedio
Datos: entero sum	Datos: entero cont
Métodos: Calculo ( )	Métodos: Promedio()
suma(entero n)	acumular (entero n)
visualizar()	flotante cal_promedio()

Diseño



Algoritmo

```
   clase Calculo{
/* Declaración de datos de la
   ➥clase */
       protegido entero sum

  /* Implementación de métodos */
       publico Calculo( ) {
           sum = 0
      }
     publico suma (entero n) {
        sum = sum + n
         }
     publico visualizar() {
        Imprimir("La suma es: ", sum)
      }
   }
clase Promedio hereda Calculo {
        privado entero cont

       publico Promedio()
       {  cont = 0           }

         publico acumular(entero n){
             suma(n)
             cont = cont +1
         }
       publico flotante
       ➥cal_promedio() {
         flotante p
         Si (cont > 0) entonces
           p= sum/cont
         De otro modo
               p=0
          Fin si
       retornar p
    }
}
  INICIO.
    /* Declaración de variables */
    entero  numero, cant, i
  /* se crea el objeto de la
     ➥clase */
  Promedio obj_prom
  Imprimir ( "CUANTOS NUMEROS VA
  ➥A SUMAR:")
  Leer (cant)
  Para ( i = 1, i<=cant, i=i+1)
```

Programa en C ++

```
  #include  <iostream.h>
   #include  <conio.h>
 class Calculo{
/* Declaración de datos de la
   ➥clase */
       protected:   int sum;
/* Implementación de métodos */
       public:
        Calculo( ) {
           sum = 0;
         }
         void suma ( int n) {
                   sum = sum + n;
            }
         void visualizar() {
             cout<<"La suma es:
             ➥"<< sum;   }
};
class Promedio : public Calculo {
        private:  int cont;

       public:
        Promedio()
           {  cont = 0;           }

        void acumular(int n){
             suma(n);
             cont = cont +1;
         }
      float cal_promedio() {
         float p;
         if (cont > 0)
           p= sum/cont;
         else
               p=0;
    return p;
    }       };
  void main( ){
    /* Declaración de variables */
    int  numero, cant, i;
  /* se crea el objeto de la
     ➥clase */
    Promedio obj_prom;
  cout<< "CUANTOS NUMEROS VA
  ➥A SUMAR:";
  cin>>cant;
  for ( i = 1; i<=cant; i=i+1)
```

```
      Imprimir ( "ENTRAR EL VALOR DEL
      ➥NUMERO :")
      Leer (numero)
      obj_prom.acumular(numero)
     Fin para
      obj_prom.visualizar()
      Imprimir ( "Promedio=",
      obj_prom.cal_promedio())

FIN
```

```
      {cout<< "ENTRAR EL VALOR DEL
      ➥NUMERO :";
        cin>>numero;
      obj_prom.acumular(numero);
     }
    obj_prom.visualizar();
    cout<< "Promedio="<<
         obj_prom.cal_promedio();
getch();
  }
```

```
Algoritmo
   clase Calculo{
/* Declaración de datos de la
   ➥clase */
       protegido entero sum

  /* Implementación de métodos */
       publico Calculo( ) {
           sum = 0
      }
     publico suma ( entero n) {
        sum = sum + n
         }
     publico visualizar() {
        Imprimir("La suma es: ", sum)
      }
   }
clase Promedio hereda Calculo {
        privado entero cont

       publico Promedio()
       {  cont = 0           }

         publico acumular(entero n){
             suma(n)
             cont = cont +1
         }
       publico flotante
       ➥cal_promedio() {
         flotante p
         Si (cont > 0) entonces
           p= sum/cont
         De otro modo
               p=0
```

```
 Programa en Java
 import java.io.*;
class Calculo {
 protected int sum;
public Calculo () {
    sum = 0;
       }
  public void  suma ( int n) {
   sum = sum + n;
       }
  public void visualizar () {
  System.out.println (" La suma
  ➥es : " +sum);
  } }
  class Promedio extends Calculo {
   private int cont;
   public Promedio ()
   {  cont = 0;
      }
    public void acumular ( int n)
{
       suma (n);
       cont = cont + 1;
                }
   public double cal_promedio () {
      double p;
      if ( cont>0)
          p = sum / cont;
       else
          p = 0;
       return p;
  }   }
  class Herencia  { public static
  ➥void main (String arg[]) throws
  ➥IOException {
```

```
          Fin si
        retornar p
    }
}
  INICIO.
    /* Declaración de variables */
    entero numero, cant, i
  /* se crea el objeto de la
     ➥clase */
  Promedio obj_prom
  Imprimir ( "CUANTOS NUMEROS VA
  ➥A SUMAR:")
  Leer (cant)
  Para ( i = 1, i<=cant, i=i+1)
      Imprimir ( "ENTRAR EL VALOR
      ➥DEL             NUMERO :")
      Leer (numero)
      Obj_prom.acumular(numero)
   Fin para
   obj_prom.visualizar()
   Imprimir ( "Promedio=",
      obj_prom.cal_promedio())

FIN
```

```
/* Declaración de variables*/
int numero, cant, i;
String cad;
/* Se crea el objeto de la
   ➥clase*/
Promedio obj = new Promedio ();
BufferedReader br=new Buffered
➥Reader (new InputStreamReader
(System.in));
System.out.println (" Cuantos
➥numeros va a sumar");
 cad= br.readLine();
 cant=Integer.parseInt (cad);
 for (i= 1; i<= cant; i++)  {
  System.out.println (" ENTRA EL
  ➥VALOR DEL NUMERO: ");
  cad= br.readLine();
  numero=Integer.parseInt (cad);
  obj.acumular(numero);
    }
   obj.visualizar();
   System.out.println
   ➥(" Promedio = " +
   obj.cal_promedio());
  }}
```

Problemas propuestos

1. Una empresa de ventas y servicios tiene empleados que forman parte de un equipo de desarrollo. Estos empleados pueden ser contratados, cobrando por horas, o bien como trabajadores de planta con un sueldo fijo. Desarrolle un programa que cree objetos de empleados por hora y objetos de empleados de planta; lea datos y presente el cálculo del sueldo del empleado.
2. Una universidad está formada por un conjunto de departamentos a los que están asignados los profesores y estudiantes. Los profesores pueden ser titulares o asociados; los alumnos que asisten a uno o más cursos pueden ser de dos tipos: de curso regular o de maestría. Desarrolle un programa que cree objetos de profesores y objetos de estudiantes; lea datos y presente la información.
3. Una empresa de equipo pesado vende bombas, intercambiadores de calor, tanques, etcétera. Cada equipo lleva registrado nombre, fabricante, peso y

costo. La información requerida para las bombas es presión de succión, presión de descarga y medida de flujo. Dependiendo del tipo de bomba, necesitamos conocer:

- Bomba centrífuga: diámetro del impulsor, número de hojas y eje de rotación.
- Bomba de diafragma: material del diafragma.
- Bomba de pistón: longitud del pistón, diámetro, número de cilindros.

La información requerida para los tanques es: volumen y presión. Dependiendo del tipo de tanque, necesitamos conocer:

- Tanque esférico: diámetro.
- Tanque a presión: diámetro y altura.

Haga un programa que establezca la herencia, declare objetos, asigne información e imprima la información.

4. En 2003, cuatro autores de libros de computadoras crearon una base de datos que almacenaba los precios base de 3 libros (*Programación, C* y *Java*). Los precios de los libros eran informados a los clientes por medio de un método: se calculan multiplicando el precio de cada libro por el 15%. En el 2004, publican un libro, *Programación orientada a objetos*, y se desea que se incluya su precio base y su respectivo cálculo del precio al cliente en la base de datos, pero sin alterar lo que ya existe.

ANEXO

Tipos de datos básicos

ALGORITMO	*LENGUAJE C++*	*LENGUAJE JAVA*
entero	int variables enteras	int variable entera
flotante	float variables flotantes	float variable flotante
carácter	char un carácter	char un carácter
cadena	char cadena de caracteres	String cadena de caracteres

Ejemplos:

/ Declaración de variables */*	*/* Declaración de variables*/*	*/* Declaración de variables*/*
entero a	int a	int a
flotante promedio	float promedio;	float promedio;
carácter c	char c	char c
cadena nombre	char nombre[6];	String nombre;

*/ * Asignación de valores */*	** Asignación de valores */*	*/ * Asignación de valores */*
a = 45	a = 45;	a = 45;
promedio = 543.23	promedio = 543.23;	promedio = 543.23;
c = ´r´	c = ‘r’;	c = ‘r’;
nombre = “María”	nombre[6] ={“Maria”};	nombre =“Maria”;

Entrada/Salida

Algoritmo	*LENGUAJE C++*	*LENGUAJE JAVA*
1. Sentencia de lectura: **Formato:** Leer (iden1, iden2, ... idenn)	**C++ requiere incluir la biblioteca** # include iostream.h para poder realizar E/S por teclado y pantalla, al inicio del programa a) Función cin: Permite teclear un valor de entrada y almacenarlo en una variable >> iden1>> iden2 **Ejemplos:**	Java requiere incluir al paquete import java.io.*; al inicio del programa para la lectura de datos por flujo de caracteres. a) Método readLine Lee siempre una cadena; si el usuario desea emplear otro tipo de dato debe aplicar un método de conversión al tipo deseado (entero, flotante..).

Ejemplos:
Lleer (nombre)
Leer (nombre,salario)
	cin>>nombre;
	cin >> nombre >> salario;
		ident1 =br. readLine. ( );
		
		Ejemplos:
		nombre= br.readLine ( ) ;
		salario = Float.parseFloat (br.readLine( ));

	b) Función cin.get(): Permite teclear un carácter y almacenarlo en variable. char iden; cin.get (iden);	b) Método read ( ) Lee un solo carácter ident1 = (char) br.read( ) ; br.skip(2);
Leer (cod_sexo)	cin >> cod_sexo; c) Función cin.getline(): Permite teclear una cadena y almacenarla en una variable char iden [tamaño]; cin.getline(iden,tamaño):	cod_sexo = (char) br.read( ); skip(2);
Leer (apellido)	char apellido[20]; cin.getline(apellido,20);	

2. Sentencia de escritura:

	Función cout: Permite desplegar cadenas y múltiples valores	Método print y println Permite enviar a salida la información requerida, sea esta una cadena o variable
Formato:	Controles de impresión (tienen	
Imprimir (“cadena constante”)	funciones especiales para la salida)	Controles de impresión
Imprimir( )	endl o \n salto de línea	\n salto de línea
Imprimir (“cadena constante”, iden)	\ t 6 espacios en blanco /* salida simple * / cout<<”cadena”; cout <<iden;	\ t 6 espacios en blanco System.out.println (“Cadena constante”); System.out.println (); System.out.println (“Cadena constante” + var);
Imprimir (iden1, iden2, ... idenn)	/* salida encadenada */ cout<<iden<<iden<<iden;	System.out.println (var1 + var2 + … + varn); + = separador de valores (constantes o variables) en la salida

Ejemplos:

Imprimir (“Introduzca su nombre:”)
Leer (nombre)
Imprimir (“Introduzca su estatura y peso:”)
Leer(estatura,peso)
	cout<<”Introduzca su nombre”;
	cin<<nombre;
	cout<<”Introduzca su estatura y pes:”
	cin>>estatura>>peso;
		System.out.println (“Introduzca su nombre”)
		nom = br.readLine( ) ;
		System.out.println (“Introduzca su estatura y peso”);
		long = Float.parseFloat(br.readLine( ));
		peso = Float.parseFloat(br.readLine( ));

Imprimir(“U. T. P”) Imprimir(“F. I. S. C.”) Imprimir(“Desarrollo de Software”)	cout<<”U. T. P.”<<endl; cout<<” F. I. S. C.\n”; cout<<”Desarrollo de Softwa-re\n”;	System.out.println (”U.T.P. ”) ; System.out.print(”F.I.S.C. \n”) ; System.out.print (”Desarrollo de Software \n”);

Ejemplo:
 	
Imprimir (“HELLO, WORLD”)
	# include < iostream.h>
	# include <conio.h>
	void main()
	{ cout << “HELLO, WORLD”;
	getch( );
	}
		class Principal {
		public static void main (String args [ ])
		{
		System.out.println (“ HELLO, WORLD”);
		}
		}

Ejemplo: Imprimir la suma de dos valores constantes	**Ejemplo: Imprimir la suma de dos valores constantes**	**Ejemplo: Imprimir la suma de dos valores constantes**

Imprimir (“50 + 25 = “, (50+25))	# include < iostream.h>
	# include <conio.h>
	void main()
	cout << “50 más 25 es ” << (50+25);
	getch( );
	}
		class Sumas {
		public static void main (Strting args[ ] {
		System.out.println (“50 más 25 es “ + (50 + 25)) ;
		}
		}

Ejemplo: Imprimir el resultado de la multiplicación de dos números, solo en C++ y Java

LENGUAJE C++

```
#include < iostream.h>
#include <conio.h>
void main( )
{  /* declaraciones  */
int a, b, resultado;
   b = 5;
resultado = 0;

cout<<"introduzca valor de a =";
cin<< a;
resultado =  a * b;
cout<<"RESULTADO ES  =
      "<< resultado);
getch();
}
```

LENGUAJE JAVA

```
import java.io.*;
class Multiplica { public static void main
       (String args [ ] )
{
/* Declaraciones */
int a, b, resultado;
BufferedReader br = new BufferedReader  (new
InputStreamReader (System.in));
 b= 5;
 resultado = 0 ;
 System.out.println (" Introduzca valor de a = ");
 a = Integer.parseInt(br.readLine( ));
 resultado = a* b;
 System.out.println ("RESULTADO ES = " +
                        resultado);
}
}
```

Sentencias de Alternativas

Algoritmo	*LENGUAJE C++*	*LENGUAJE JAVA*

1. Alternativa simple
Formatos:
Si (condición) Entonces
{
Sentencia 1
:
Sentencia n
}
Fin Si
	if (condición)
	{ Sentencia 1;
	:
	Sentencia n;
	}
		if (condición)
		{ Sentencia 1;
		:
		Sentencia n;
		}

Nota: Cuando se escribe una sentencia no se requiere paréntesis de llave { }.	**Nota:** Cuando se escribe una sentencia no se requiere paréntesis de llave { }.	**Nota:** Cuando se escribe una sentencia no se requiere paréntesis de llave { }.

A

Ejemplos:

```
si   ( × < 2.1 )  entonces
      {
         y   =  0.5 * × + 0.95
        Imprimir ( " Y = ", y)
      }
     fin si
```

```
if ( × < 2.1 )
    {
      y   =  0.5 * × + 0.95;
      cout<< " Y = "<< y;
      }
```

```
if ( × < 2.1 )
  {
     y   =  0.5 * × + 0.95;
      System.out.println
      (" Y = "+ y);
     }
```

2. Alternativa doble:

Formatos:

```
Si  (condición)  Entonces
    {     Sentencia 1
                :
       Sentencia n
    }
   De Otro Modo
    {   Sentencia 1
                :
       Sentencia n
    }
   Fin Si
```

```
if  (condición)
    {  Sentencia 1;
             :
     Sentencia n;
    }
else
    {  Sentencia 1;
             :
      Sentencia n;
     }
```

```
if  (condición)
    {  Sentencia 1;
          :
      Sentencia n;
    }
else
    {  Sentencia 1;
          :
      Sentencia n;
     }
```

Ejemplos:

```
  Si  (A > B)    Entonces
  Imprimir ( "GRANDE ", A )
  De Otro Modo
  Imprimir ("GRANDE ", B )

  Fin Si
```

```
if  (A > B)
 cout<< "GRANDE "<< A ;
else
cout<<"GRANDE "<< B ;
```

```
if  (A > B)
System.out.println (
 "GRANDE " +  A)  ;
else
System.out.println (
 "GRANDE " +  B)  ;
```

3. Múltiple:

Formatos:

```
Si (Condición) Entonces
        { Sentencia 1;
                :
          Sentencia n;
         }
De Otro Modo Si (Condición)
Entonces
        { Sentencia 1;
                :
          Sentencia n;
         }
De Otro Modo  Si (Condición)
Entonces
        { Sentencia 1;
                :
          Sentencia n;
         }
De Otro Modo
         { Sentencia 1;
                :
           Sentencia n;
         }
 Fin Si
```

```
if  (Condición)
    { Sentencia 1;
            :
      Sentencia n;
    }
else  if  (Condición
    { Sentencia 1;
            :
      Sentencia n;
    }
 else  if (Condición)
    { Sentencia 1;
            :
      Sentencia n;
    }
 else
    { Sentencia 1;
            :
      Sentencia n;
     }
```

```
if  (Condición)
    { Sentencia 1;
            :
      Sentencia n;
    }
else  if  (Condición
    { Sentencia 1;
            :
      Sentencia n;
    }
 else  if (Condición)
    { Sentencia 1;
            :
      Sentencia n;
    }
 else
    { Sentencia 1;
            :
      Sentencia n;
    }
```

Ejemplos:

```
Si (num>0) Entonces
{
  raiz =   √num
 Imprimir("La Raíz = ", raiz)
 }
De Otro Modo Si(num < 0)
Entonces
   Imprimir ("Raiz Imaginaria")
De Otro Modo
    Imprimir("num  no debe
ser cero")
Fin Si
```

```
if  (num > 0)
{
raiz =  sqrt (num);
cout <<"La Raíz = "<< raiz;
}
else  if (num  <  0)
cout<<"Raiz Imaginaria";
else
cout<<"num no debe ser
cero";
```

```
if  (num > 0)
{
raiz = Math.sqrt (num);
System.out.println
      ("La Raíz = " + raiz);
}
else  if   (num  <  0)
System.out.println("Raiz
                  Imaginaria");
else
System.out.println("num no
          debe ser cero");
```

2. Estructuras repetitivas:
a) Instrucción Mientras (con condición inicial):

Formato:

```
Mientras ( Condición)
        Sentencia 1
            :
        Sentencia n
 fin mientras
```

```
while ( Condición)
 {
        Sentencia 1;
            :
        Sentencia n;
 }
```

Nota: Cuando se escribe una sentencia, no se requiere paréntesis de llave { }

```
while ( Condición)
  {
      Sentencia 1;
          :
      Sentencia n;
  }
```

Nota: Cuando se escribe una sentencia, no se requiere paréntesis de llave { }

Ejemplos:

```
c    = 0
Mientras (c < 10)
c     =   c + 1
Imprimir ( c)

fin mientras
```

```
c    = 0;
while (c < 10)
{
c     =   c + 1;
cout<< c;
}
```

```
c    = 0;
while (c < 10)
{
c     =   c + 1;
System.out.println  ( c) ;
}
```

Ejemplo:
Calcular la suma de los 20 primeros números pares.

Algoritmo	*LENGUAJE C++*	*LENGUAJE JAVA*
c=0	c=0;	c=0;
p = 0	p = 0;	p = 0;
tot = 0	tot = 0;	tot = 0;
Mientras (c < 20)	while (c < 20)	while (c < 20)
p = p + 2	{	{
tot = tot + p	p = p + 2;	p = p + 2;
c = c + 1	tot = tot + p;	tot = tot + p;
fin mientras	c = c + 1;	c = c + 1;
Imprimir(“ Suma = “, tot)	}	}
	cout<<“ Suma = “ <<tot;	System.out.println (“ Suma = “ + tot);

b) Instrucción Hasta que (con condición final):
Formatos:

Repetir Sentencia 1 : Sentencia n Hasta que (condición**)**	**La estructura repetitiva Hasta que no es manejada en C++**	**La estructura repetitiva Hasta que no es manejada en Java**.

Ejemplos en algoritmo

Ejemplo # 1	*Ejemplo #2.*
Imprimir los 10 primeros números	**Calcular la suma de Los 20 primeros números impares**
entero c = 0 Repetir c = c + 1 Imprimir (c) Hasta que (c == 10)	entero c =0 entero p = 1 entero tot = 0 Repetir tot = tot + p p = p + 2 c = c + 1 Hasta que (c == 20) Imprimir (“Suma = “, tot)

c) Instrucción Para

Formato en algoritmo:	*Formato en C++*	*Formato en Java*

Para var_control = val_ini, val_fin, incre/decre
 	
Sentencia 1
:
Sentencia n
fin para
	for (var_control =val_ini ; condición ;incre/decre)
	{
	Sentencia 1;
	:
	Sentencia n;
	}
		for (var_control =val_ini ; condición ;incre/decre)
		{
		Sentencia 1;
		:
		Sentencia n;
		}

A

Ejemplos:
Imprimir los 10 primeros números

para Para c = 1, 10, 1 Imprimir (c) fin para	for (c = 1; c < =10; c = c + 1) cout <<c;	for (c = 1; c < =10; c = c + 1) System.out.println (c) ;

Calcular la suma de los 20 primeros números pares

tot = 0 Para c= 2, 20, 2 tot = tot + c Fin para Imprimir (“ La suma es”, tot)	for (c = 1; c < =10; c = c + 1) cout <<c;	for (c = 1; c < =10; c = c + 1) System.out.println (c) ;

Calcular la suma de los 20 primeros números pares

tot = 0
 	
para c= 2, 20, 2
 	
tot = tot + c
 	
Fin para
Imprimir
(“ La suma es”, tot)
	tot = 0;
	
	for (c = 2; c < = 20 ; c = c + 2)
	tot = tot + c;
	
	cout << “La sumas es” <<tot;
		tot = 0;
		
		for (c = 2; c < = 20 ; c = c + 2)
		tot = tot + c;
		System.out.println (“La sumas es”
		+ tot);

BIBLIOGRAFÍA

James L. Antonakos, Kenneth C. Mansfield. *Programación estructurada en C*, Prentice Hall, 1997.

Oldemar Rodríguez. *C++ para ambiente Windows*, Editorial Tecnológica de Costa Rica, 1997.

Andrew C. Staugaard, Jr: *Técnicas estructuradas y orientadas a objetos*, Prentice Hall, 1998.

H.M. Deitel, P.J. F. Deitel. *Cómo programar C++*, Pearson Prentice Hall, 2a. Edición, 1999.

Craig Larman. *UML y patrones. Introducción al análisis y diseño orientado a objetos*, Pearson-Prentice Hall, 1a. Edición, *1999.*

Camelia Muñoz Caro; Alfonso Niño Ramos; Aurora Vizcaíno. *Introducción a la programación orientada a objetos*, Prentice Hall, 2002.