

Apunte de Cátedra

**RESOLUCION
DE PROBLEMAS Y
ALGORITMOS**

**Analista de Sistemas y
Licenciatura en Sistemas**

Lic. Verónica L. Vanoli

Mg. Sandra I. Casas

**Universidad Nacional de la Patagonia Austral
Unidad Académica Río Gallegos**

Indice

La Programación	1
Análisis del Problema	3
Ejemplos	4
Diseño del Algoritmo	4
Ejemplos	5
Escritura inicial del algoritmo	7
Datos	8
Tipos de Datos Simples	9
Tipos de Datos Compuestos o Clases/Objetos	9
Operaciones	9
Asignación	9
Expresiones	10
Entrada	13
Salida	13
Ejemplos	14
Estructuras de Control	16
Secuenciación	16
Ejemplos	16
Selección	17
Estructura SI-SINO	18
Ejemplos	18
Estructura ALTERNAR-CASO	21
Ejemplos	23
Iteración	26
Estructura MIENTRAS	27
Bucles controlados por contador	28
Bucles controlados por sucesos	29
Bucles contadores	30
Bucles acumuladores/sumadores	30
¿Cómo diseñar bucles?	31
Ejemplos	35
Estructura HACER-MIENTRAS	37
Ejemplos	37
Estructura PARA	38
Ejemplos	39
Identación	41
Llaves	42
Estructura General de un Programa	45
Estructura General de un Programa en Pseudocódigo	46
Ejemplos	46
Declaración de constantes y variables	46
Uso de variables	48
Pruebas del programa	48
Arreglos	49
Arreglos Unidimensionales: Vectores	49
Declaración y creación	50
Índices	50
Manejo	51

Indice

Ejemplos	52
Ordenación y Búsqueda	55
Ordenación por Inserción	55
Ordenación por Selección	56
Ordenación por Intercambio	58
Búsqueda Secuencial	59
Búsqueda Binaria	60
Arreglos Bidimensionales: Matrices	62
Declaración y creación	63
Manejo	63
Ejemplos	64
Diseño de Algoritmos: Funciones	68
Funciones	70
Ejemplos	71
Aserciones: Precondiciones y Postcondiciones	75
Invocación a la función	75
Conceptos fundamentales de una función	78
Variables locales	79
Parámetros	79
Visibilidad de identificadores	83
Ámbito de identificadores	84
Clases/Objetos: Tipos de Datos Compuestos	85
Orientación a Objetos	85
Objetos	87
Clases	90
Mensajes	93
Constructor	93
Clase Vector	98
Ejemplos	101
Clase Matriz	103
Ejemplos	105
Lenguaje de Programación: JAVA	108
De Pseudocódigo a Java	112
Estructuras de Control en Java	115
Clases en Java	119
El método main	122
Referencias en Java	123
Agregación y Dependencia	126
Parámetros	127
Paquetes	128
Modificadores de acceso	128
Sobrecarga	131
Operador THIS	132
Arreglos en Java	132
Programas en Java	135
API	135
Clase String (Cadena)	136
Clases estáticas	138



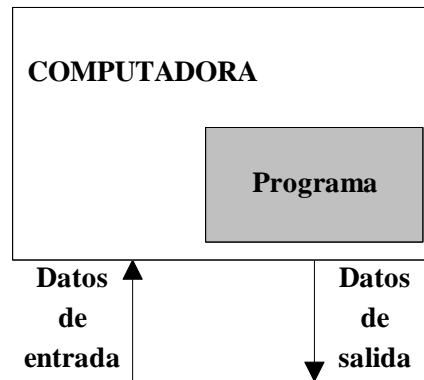
Carreras: Analista de Sistemas y Licenciatura en Sistemas
Asignatura: Resolución de Problemas y Algoritmos
Docente: Lic. Verónica L. Vanoli

Indice

La clase Random	146
Pilas	148
Implementación	149
Colas	153
Implementación	154
Recursividad	157
Ejemplos	161
ANEXO: Los Lenguajes de Programación	163
Bibliografía	165

Apunte de Cátedra

Una **computadora** es un dispositivo electrónico utilizado para procesar información y obtener resultados. Los datos y la información se pueden introducir en la computadora como **entrada** (input) y a continuación se procesan para producir una **salida** (output, resultados).



Los componentes físicos que constituyen la computadora, junto con los dispositivos que realizan las tareas de entrada y salida, se conocen con el término **Hardware**. El conjunto de instrucciones que hacen funcionar a la computadora se denomina **programa**; a la persona que escribe programas se llama **programador** y al conjunto de programas escritos para una computadora se llama **Software**.

LA PROGRAMACION

Muchas de las cosas que realizamos diariamente las hacemos en forma automática o inconsciente, sin pensar o analizar cada vez que las realizamos, como las hacemos. Por ejemplo, no es necesario pensar las cosas que hacemos para avanzar la página de un libro:

1. levantar la mano;
2. mover la mano a la parte derecha del libro;
3. agarrar la esquina de la página;
4. mover la mano de derecha a izquierda hasta que la página se posicione, de forma que pueda leerse lo que hay en la otra cara;
5. soltar la página.

Porque ya hemos aprendido a realizar esta tarea.

Otro ejemplo sería el conjunto de acciones que realizamos cuando arrancamos nuestro auto:

1. poner la llave;
2. asegurarse de que la marcha esté en punto muerto;
3. presionar el pedal del acelerador;
4. girar la llave hasta la posición de "arranque";
5. si el motor arranca antes de seis segundos, dejar la llave en posición "ignición";
6. si el motor no arranca antes de seis segundos, esperar diez segundos y repetir los pasos 3 al 6;
7. si el coche no arranca, llamar al mecánico.

Como vemos, estas simples tareas, se realizan mediante una secuencia lógica de pasos. La mayoría de las cosas que hacemos, se realizan siguiendo una secuencia ordenada de pasos, que alguna vez hemos aprendido y luego las realizamos sin pensar. Podríamos agregar más ejemplos como cepillarse los dientes, manejar un automóvil, llamar por teléfono, ir a cenar, etc. También los matemáticos han

Apunte de Cátedra

desarrollado secuencias lógicas de pasos para resolver los problemas o demostrar los teoremas. La producción o fabricación de productos (alimentos, electrodomésticos, autos, etc.) se realiza siguiendo determinadas secuencias lógicas de pasos. Es decir, casi todo esto basado en el orden de las cosas y las acciones (primero en forma consciente y luego en forma inconsciente). Esta ordenación se adquiere a través de un proceso que podemos llamar programación.

Programación: planificación, proyección o ejecución de una tarea o suceso.

Pero a nosotros, nos interesa en particular la programación de computadoras para resolver determinados problemas. Es decir, no usamos computadoras para arrancar un automóvil, pero sí para calcular el sueldo de un empleado.

Entonces podríamos decir que los pasos para calcular el salario semanal de un empleado son:

1. buscar el sueldo por hora del empleado.
2. determinar el número de horas trabajadas durante la semana.
3. sí el número de horas trabajadas no supera las 40 horas, multiplicar el número de horas trabajadas por el sueldo por horas, para obtener la paga regular.
4. sí el número de horas trabajadas supera las 40 horas, multiplicar el sueldo por hora por 40 para obtener la paga regular y multiplicar una vez y media el sueldo por hora por la diferencia, entre el número de horas trabajadas, y 40 para obtener la paga de las horas extras.
5. añadir la paga regular a la paga extra (si la hubiera) para obtener el sueldo total de la semana.

Programación de computadora: proceso de planificar una secuencia de instrucciones (pasos) que ha de seguir una computadora.

Programa de computadora: secuencia de instrucciones que indica las acciones que han de ser ejecutadas por una computadora.

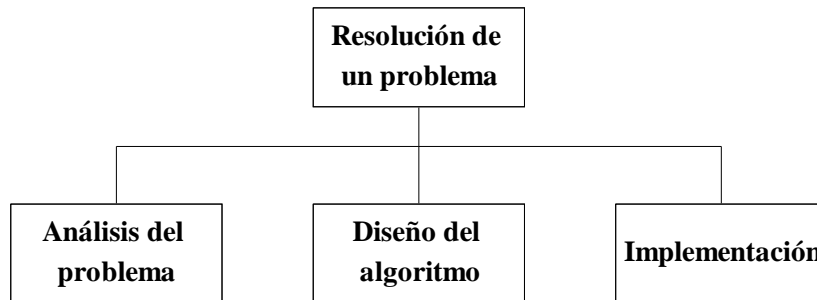
La principal razón por la cual las personas aprenden los lenguajes de programación y a programar, es para utilizar la computadora como una herramienta para la resolución de problemas. La computadora nos permite hacer tareas más eficientes, rápidas y precisas. Pero para poder utilizar esta poderosa herramienta, debemos especificar exactamente lo que queremos hacer y el orden en el que debe hacerse. Esto se logra mediante la programación.

Un programa tiene por objetivo resolver un problema. Para desarrollar un programa hemos de seguir un método, que se compone de las siguientes fases:

- 1°. Análisis del Problema: comprender (definir) el problema.
- 2°. Diseño del Algoritmo: desarrollar una solución determinada a partir de una secuencia lógica de pasos.
- 3°. Implementación o Codificación del Algoritmo: traducir el algoritmo a un lenguaje de programación.

Apunte de Cátedra

Las fases del proceso de resolución de un problema mediante computadora se indican en la siguiente figura:



La computadora no puede analizar un problema y proponer una solución. El programador debe encontrar la solución y comunicársela a la computadora. El programador resuelve el problema, no la computadora. La computadora es una herramienta que puede ejecutar la solución (programa) muchas veces rápidamente.

El primer paso (análisis del problema) requiere que el problema se defina y comprenda claramente para poder resolverlo correctamente. Una vez analizado el problema, se debe desarrollar el algoritmo (procedimiento paso a paso, para solucionar el problema dado). Por último, para resolver el algoritmo mediante una computadora se necesita codificar el algoritmo en un lenguaje de programación - Pascal, Cobol, Fortran, C, etc. -, es decir, convertir el algoritmo en programa, ejecutarlo y comprobar que el programa soluciona verdaderamente el problema.

Análisis del Problema

El propósito de esta fase es ayudar al programador para llegar a una comprensión de la naturaleza del problema. El problema debe estar bien definido si se desea llegar a una solución satisfactoria.

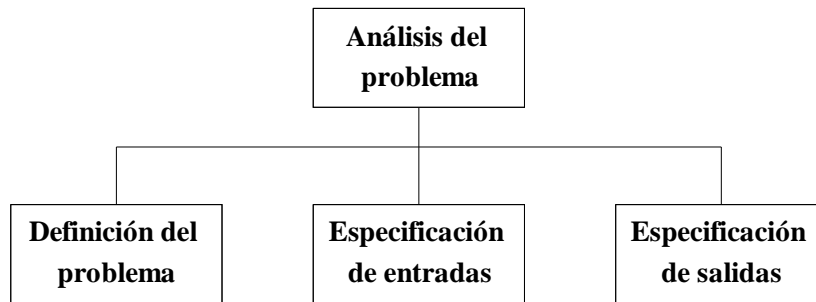
Para poder definir con precisión el problema se requiere que las especificaciones de entrada y de salida sean descriptas con detalle. Una buena definición del problema, junto con una descripción detallada de las especificaciones de entrada y de salida, son los requisitos más importantes para llegar a una solución eficaz.

El análisis del problema exige una lectura previa del problema a fin de obtener una idea general de lo que se solicita. La segunda lectura deberá servir para responder a las preguntas:

- ¿Qué datos se necesitan para resolver el problema? (¿qué necesito?)
- ¿Qué información debe proporcionar la resolución del problema? (¿qué se pide?)

La respuesta a la primera pregunta indicará qué datos se proporcionan o las **entradas** del problema. La respuesta a la segunda pregunta indicará los resultados deseados o las **salidas**.

Apunte de Cátedra



❖ Ejemplos.

Problema: Calcular la superficie y circunferencia de un círculo.

Análisis

Entradas: Radio del círculo (valor real)

Salidas: Superficie del círculo (valor real)
Circunferencia del círculo (valor real)

Problema: Obtener la calificación promedio de los resultados obtenidos en el último parcial de la asignatura Análisis Matemático I.

Análisis

Entradas: Las calificaciones de cada alumno (valores enteros)
Cantidad de calificaciones (valor entero)

Salidas: Promedio (valor real)

Problema: Hallar la cantidad de alumnos que aprobaron el examen de la asignatura Programación I.

Análisis

Entradas: Las calificaciones de cada alumno (valores enteros)

Salidas: Cantidad de aprobados (valor entero)

Diseño del Algoritmo

En esta fase el objetivo es obtener la secuencia ordenada de pasos - sin ambigüedades - que conducen a la solución de un problema dado.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan, como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.

Características de los algoritmos:

- Un algoritmo debe ser **preciso** e indicar el orden de realización de cada paso.
- Un algoritmo debe estar **definido**. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.

Apunte de Cátedra

- Un algoritmo debe ser **finito**. Si se sigue un algoritmo, se debe terminar en algún momento; o sea se debe tener un número finito de pasos.

La definición de un algoritmo debe describir tres partes: **Entradas, Proceso y Salidas**.

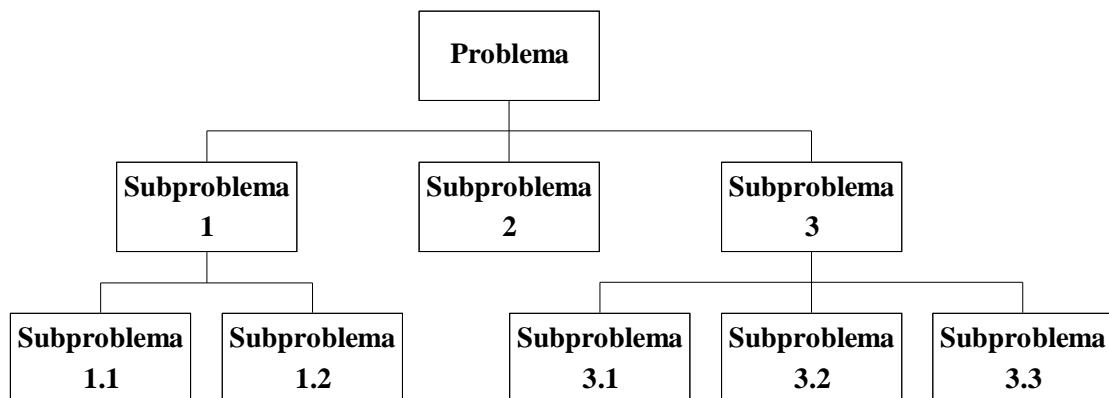
Por ejemplo, en un algoritmo de una receta de cocina se tendrá:

Entradas: ingredientes y utensilios empleados
Proceso: elaboración de la receta en la cocina
Salidas: terminación del plato

Una computadora no tiene capacidad para solucionar problemas más que cuando se le proporcionan los sucesivos pasos a realizar. Estos pasos sucesivos que indican las instrucciones a ejecutar por la máquina constituyen, como ya conocemos, el **algoritmo**.

La información proporcionada al algoritmo constituye su *entrada* y la información producida por el algoritmo constituye su *salida*.

Los problemas complejos se pueden resolver más eficazmente con la computadora cuando se descomponen en subproblemas que sean más fáciles de solucionar que el original. Este método se suele denominar **divide y vencerás**. Normalmente los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán sólo unos pocos pasos. Por lo tanto, luego de esta primera descripción, se realiza una ampliación, obteniendo una descripción más detallada con más pasos específicos. Este proceso se denomina **refinamiento del algoritmo**. Para problemas complejos se necesitan con frecuencia diferentes *niveles de refinamiento* antes de que se pueda obtener un algoritmo claro, preciso y completo.



❖ Ejemplos.

Problema: Preparar empanadas.

Análisis

Entradas: harina, agua, aceite, carne picada, huevos, aceitunas, pasas de uva, cebolla, condimentos.

Salidas: empanadas.

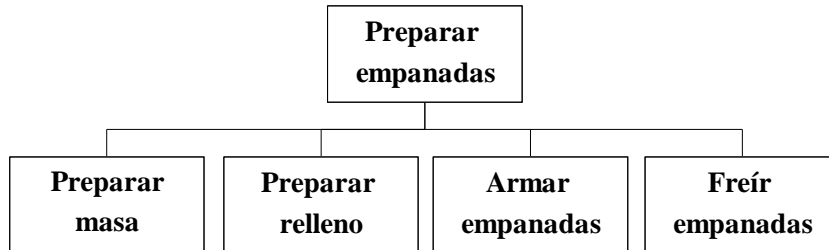
Diseño

Proceso:

- Preparar la masa de las tapas.

Apunte de Cátedra

- Preparar el relleno.
- Armar las empanadas.
- Freír las empanadas.



Pero como sabemos, cada uno de estos pasos se realiza a su vez mediante un conjunto de pasos. Por lo tanto, cada uno de estos pasos es un problema, o un subproblema del problema original.

Refinamiento:

- Preparar la masa de las tapas.
 - Mezclar harina y agua.
 - Amasar.
 - Dejar reposar 1 hora.
- Preparar el relleno.
 - Freír la cebolla.
 - Agregar la carne y dejar que se cocine.
 - Condimentar.
 - Agregar huevo, aceitunas y pasas de uva.
- Armar las empanadas.
 - Colocar relleno en el centro de cada tapa.
 - Doblar por la mitad.
 - Hacer repulgue.
- Freír las empanadas.

Como vemos algunos de estos pasos, podrían refinarse aún más.

Problema: Calcular la superficie y circunferencia de un círculo.

Análisis

Entradas: radio (valor real)

Salidas: superficie, circunferencia (valores reales)

Diseño

Proceso:

- Obtener el radio.
- Calcular la superficie.
- Calcular la circunferencia.
- Mostrar los resultados.

Refinamiento:

- Obtener el radio.
- Calcular la superficie.
 - superficie: $3.141592 * \text{radio}^2$

Apunte de Cátedra

- Calcular la circunferencia.
 - circunferencia : $2 * 3.141592 * \text{radio}$
- Mostrar los resultados.

Las **ventajas** más importantes del diseño descendente son:

- El problema se comprende más fácilmente al dividirse en partes más simples denominadas **módulos**.
- Las modificaciones en los módulos son más fáciles.
- La comprobación del problema se puede verificar fácilmente.

Tras los pasos anteriores (**diseño descendente y refinamiento por pasos**) es preciso representar el algoritmo mediante una determinada herramienta de programación: **diagrama de flujo, pseudocódigo o diagrama N-S**. En esta asignatura, los algoritmos los vamos a representar en pseudocódigo (casi-código).

Escritura inicial del algoritmo

El método para describir (escribir) un algoritmo consiste en realizar una descripción paso a paso con un lenguaje natural del citado algoritmo. Recordemos que un algoritmo es un método o conjunto de reglas para solucionar un problema. En cálculos elementales estas reglas tienen las siguientes propiedades:

- deben estar seguidas de alguna secuencia definida de pasos hasta que se obtenga un resultado coherente;
- sólo puede ejecutarse una operación a la vez.

Supongamos el siguiente problema: *¿Qué hacer para ver la película Titanic?*

La respuesta es muy sencilla y puede ser descrita en forma de algoritmo general de modo similar a:

- Ir al cine.
- Comprar una entrada.
- Ver la película.
- Regresar a casa.

El algoritmo consta de cuatro acciones básicas, cada una de las cuales debe ser ejecutada antes de realizar la siguiente. En términos de computadora, cada acción se codificará en una o varias sentencias que ejecutan una tarea particular.

El algoritmo descrito es muy sencillo; sin embargo, como ya se ha indicado en párrafos anteriores, el algoritmo general se descompondrá en pasos más simples en un procedimiento denominado **refinamiento sucesivo**, ya que cada acción puede descomponerse a su vez en otras acciones simples.

- Ver la cartelera de cines en el diario.
- Si no proyectan "Titanic"
 - Decidir otra actividad.
- Sino
 - Ir al cine.
 - Si hay cola
 - Ponerse en ella.
 - Mientras haya personas delante
 - Avanzar en la cola.
 - Si hay localidades
 - Comprar una entrada.

Apunte de Cátedra

- Pasar a la sala.
- Localizar la(s) butaca(s)
- Mientras proyectan la película
 - Ver la película.
- Salir del cine.
- Sino
- Refunfuñar.
- Volver a casa.

Podemos continuar refinando este algoritmo, para ello analicemos la acción: - localizar la(s) butacas(s)

- Caminar hasta llegar a la primera fila de butacas.
- Hacer
 - Comparar número de fila con número impreso en billete.
 - Si no son iguales
 - Pasar a la siguiente fila.
- Mientras que se localice la fila correcta
- Mientras número de butaca no coincida con número de billete
 - Avanzar a través de la fila a la siguiente butaca.
- Sentarse en la butaca.

Se considera que el número de la butaca y fila coincide con el número y fila rotulado en el billete.

Para describir un algoritmo hemos de utilizar tres componentes esenciales:

- **Datos.**
- **Operaciones.**
- **Estructuras de Control.**

Datos

El primer objetivo de toda computadora es el manejo de la información o datos. Las entradas y las salidas en un programa son datos. A veces son necesarios datos auxiliares para realizar cálculos intermedios que permiten obtener las salidas finales.

Un algoritmo maneja datos usando **variables** y **constantes**. La característica principal de una variable es que es un dato que puede cambiar o alterar su valor durante la ejecución del programa. Y una constante es un dato cuyo valor permanece inalterable durante la ejecución del algoritmo. Pero en ambos casos, el dato o valor *residirá en la memoria de la computadora*, y para poder acceder a ellas usamos un nombre o *identificador*, que nos permitirá referenciarlo en el programa.

Un dato será una variable o una constante, que utilizaremos mediante su identificador. Los siguientes pueden ser nombres de variables o constantes: NUM, MAX, EDAD, LETRA, ... Se recomienda, para mayor claridad y seguimiento de las mismas, colocarle nombres significativos dentro de un programa. No use espacios, en ese caso escriba de la siguiente manera: NUM_1. Tampoco use acentos.

Las variables son de algún tipo: simple o compuesto (Clases/Objetos). Cada variable y constante es de un tipo de datos determinado (entero, real, caracter, booleano, etc.). El tipo de datos determina el conjunto de valores al que pertenece una constante o sobre el cuál puede tomar valores una variable, y el conjunto de operaciones que se pueden realizar sobre el mismo.

Apunte de Cátedra

Tipos de Datos Simples

Los tipos de datos simples se clasifican en:

- **Numérico.**
- **Caracter.**
- **Lógico o Booleano.**

Datos de tipo numérico: es el conjunto de los valores numéricos. Se representan en dos formas distintas:

- **Enteros** → es un subconjunto finito de los números enteros. Son números completos, no tienen componentes fraccionarios o decimales y pueden ser negativos o positivos.
- **Reales** → es un subconjunto de los números reales. Un número real consta de una parte entera, un punto y una parte decimal; pueden ser negativos o positivos.

Datos de tipo caracter: es el conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato tipo caracter contiene un solo caracter.

- **Caracteres Alfabéticos** → pueden ser algunas de las siguientes letras: 'A', 'B', 'C', ..., 'Z', 'a', 'b', 'c', ..., 'z'.
- **Caracteres Numéricos** → pueden ser algunos de los siguientes dígitos: '0', '1', '2', ..., '9'.
- **Caracteres Especiales** → pueden ser alguno de los siguientes símbolos: '+', '-', '*', '/', '\$', ';', '<', '>', '#', ...

Datos de tipo lógico (booleano): es aquel dato que sólo puede tomar uno de dos valores: **Verdadero** (True) o **Falso** (False). Este tipo de dato se utiliza para representar las alternativas (si/no) a determinadas condiciones.

Tipos de Datos Compuestos o Clases/Objetos

Los tipos de datos compuestos también llamados clases/objetos, se clasifican en: arreglos (vectores y matrices), cadenas, archivos, pilas, colas, listas, árboles, etc., que se estudiarán mas adelante.

Operaciones

Para manipular datos, es decir, para transformar las entradas en salidas, para que la computadora se comunique con el exterior y para que podamos introducir datos en la computadora necesitamos operaciones. Estas operaciones pueden ser:

- **Asignación.**
- **Entrada.**
- **Salida.**

Operación de Asignación

La operación de asignación es el modo de darle valores a una variable. La operación de asignación se representa con el símbolo u operador ← (una flecha apuntando hacia la izquierda). La operación de asignación se conoce como *instrucción o sentencia* de asignación cuando se refiere a un lenguaje de programación.

Formato general:

nombre_de_la_variable ← expresión

Apunte de Cátedra

Por ejemplo, la operación de asignación: $A \leftarrow 5$. Significa que a la variable A se le ha asignado el valor entero 5.

La acción de asignar es destructiva, ya que el valor que tuviera la variable antes de la asignación se pierde y se reemplaza por el nuevo valor. Es decir que si a la variable A, le vuelvo a asignar otro valor, por ejemplo el 10, pierde su valor inicial (el 5) y pasa a tener un nuevo valor (el 10).

A una variable se le puede asignar:

- un valor, ejemplo: $X \leftarrow 3$.
- otra variable o una constante, ejemplo: $X \leftarrow Y$. Previo a esto la variable Y deberá contener algún valor.
- una expresión aritmética o lógica, ejemplo: $X \leftarrow Y + 3$.

En los dos últimos casos la computadora ejecuta la sentencia de asignación en dos pasos. En el primero de ellos se calcula o se obtiene el valor de la expresión del lado derecho del operador, obteniéndose un valor de un tipo específico. En el segundo paso, este valor se almacena en la variable cuyo nombre aparece a la izquierda del operador de asignación, sustituyendo al valor que tenía anteriormente. Así, en el ejemplo $X \leftarrow Y + 3$, el valor del resultado de la expresión $Y + 3$ se lo asigna a la variable X.

Veamos algunos ejemplos de asignaciones:

$AN \leftarrow 3 + 5 - 2$	se evalúa la expresión $3+5-2$ y se lo asigna a la variable AN, es decir, 6 será el valor que toma AN.
$A \leftarrow 0$	la variable A toma el valor 0.
$N \leftarrow 2$	la variable N toma el valor 2.
$A \leftarrow N + 1$	la variable A toma el valor $N + 1$, como N es igual a 2, la variable A toma el valor $2 + 1$, es decir, 3.

Expresiones

Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales. Las mismas ideas son utilizadas en notación matemática tradicional.

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas.

Una expresión consta de *operandos* y *operadores*. Según sea el tipo de objetos que manipulan, las expresiones se clasifican en:

- **Aritméticas.**
- **Booleanas.**

El resultado de la expresión aritmética es de tipo numérico y el resultado de la expresión booleana es de tipo lógico (verdadero o falso).

Expresiones Aritméticas

Son análogas a las fórmulas matemáticas. Las variables y constantes son numéricas (reales o enteras) y las operaciones son aritméticas.

Apunte de Cátedra

Operador	Significado
^	Exponenciación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto

Exponenciación (^)

Tipo de operando	Tipo de resultado
Entero	Entero
Real	Real

Tipo de operando1	Tipo de operador	Tipo de operando2	Tipo de resultado
Entero	+, -, *, /, %	Entero	Entero
Entero	+, -, *, /, %	Real	Real
Real	+, -, *, /, %	Entero	Real
Real	+, -, *, /, %	Real	Real

Reglas de Prioridad

Las expresiones que tienen dos o más operandos requieren reglas matemáticas que permitan determinar el orden de las operaciones, se denominan reglas de prioridad o precedencia y son:

1.- Las operaciones que están encerradas entre paréntesis se evalúan primero. Si existen diferentes paréntesis anidados (interiores unos a otros), las expresiones más internas se evalúan primero.

2.- Las operaciones aritméticas dentro de una expresión suelen seguir el siguiente orden de prioridad:

- Operador ^
- Operadores *, /
- Operadores +, -
- Operadores %

3.- En caso de coincidir varios operadores de igual prioridad en una expresión o subexpresión encerrada entre paréntesis, el orden de prioridad es de izquierda a derecha.

Algunos ejemplos de prioridad:

Fórmula matemática	Expresión algorítmica
$B^2 - 4AC$	$B \wedge 2 - 4 * A * C$
$A + B - C$	$A + B - C$
$\frac{A + B}{C + D}$	$(A + B) / (C + D)$
$\frac{1}{1 + X^2}$	$1 / (1 + X \wedge 2)$
$AX - (B + C)$	$A * X - (B + C)$

Apunte de Cátedra

Expresiones Booleanas

Una expresión booleana es una expresión que sólo puede tomar dos valores: *verdadero* o *falso*. Las expresiones booleanas se forman combinando operandos compatibles mediante operadores relacionales. Las expresiones booleanas se utilizan principalmente para conformar condiciones.

Operadores relacionales:

Operador	Significado
= =	Igualdad
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
!=	Desigualdad o Distinto

Los operadores relaciones o de relación permiten realizar evaluaciones o comparaciones de los operandos. El resultado de la operación será verdadero o falso.

Formato general:

operando1 operador_de_relación operando2

Por ejemplo:

Expresión	Resultado
1 = = 5	Falso
2 > 1.33	Verdadero
-5 != 5	Verdadero
1.7 <= -3	Falso

En estos ejemplos se han utilizado operandos constantes pero por lo general se utilizan variables.

Por ejemplo:

Suponemos que X: 5 e Y: -2.

Expresión	Resultado
X + 1 = = 7	Falso
Y > X - 10	Verdadero
Y * 2 != 4	Verdadero
7 <= X	Falso

Las expresiones booleanas pueden ser simples (como los ejemplos vistos) o compuestas. Es decir, los operandos de una expresión booleana podrían ser a su vez subexpresiones booleanas. Se utilizan los operadores lógicos para componer expresiones booleanas compuestas.

El formato de la expresión general en este caso es:

expresión1 operador_lógico expresión2

Apunte de Cátedra

Operadores lógicos o condicionales:

Operador	Significado	Resultado
&	Conjunción (y)	La expresión será cierta sólo si ambos operandos son ciertos. En cualquier otro caso será falsa.
	Disyunción (o)	La expresión será falsa sólo si ambos operandos son falsos. En cualquier otro caso será verdadera.

En estos casos, para evaluar la expresión booleana, se obtiene primero los resultados de las subexpresiones y con estos resultados se obtiene el resultado de aplicar los operadores lógicos.

Algunos ejemplos:

Suponemos que A: 3 y B: -5.

Expresión	Resultado
$(A \geq 7) \& (B == -5)$	Falso
$(3 != B) (B < -10)$	Verdadero
$(A == B + 8) (B < -4)$	Verdadero

Operación de Entrada

Es necesaria una operación que nos permita ingresar datos en la computadora, para que ésta los procese. Esta operación, se realizará a través de una sentencia denominada: **LEER**. Dicha sentencia permitirá que el dato ingresado por teclado se almacene en una variable. De esta manera el dato podrá ser diferente cada vez que se ejecute el programa.

Formato general:

leer lista_de_variables_de_entrada

Supongamos la sentencia: **leer** N. Si tipeamos (en el teclado de la computadora) el valor 30, la variable N valdrá 30 (suponemos que N es una variable entera). De esta forma, podemos suministrarle al programa datos desde su exterior, utilizando variables.

Operación de Salida

Los programas realizan cálculos para producir salidas. Y dichas salidas deben ser informadas a quien requiere estos resultados, entonces necesitamos una operación que permita que la computadora informe resultados y/o texto descriptivo. No tiene sentido que la computadora realice procesos si no podemos enterarnos de las salidas. Mediante una sentencia denominada: **ESCRIBIR**, se visualizarán en la pantalla los resultados de los cálculos o texto descriptivo.

Formato general:

escribir lista_de_expresiones_de_salida y/o "mensaje_de_salida"

Apunte de Cátedra

Supongamos la sentencia: **escribir** “Este es mi primer programa”. En la pantalla de nuestra computadora veremos el mensaje: “Este es mi primer programa”.

Ahora supongamos la sentencia: **escribir** N. En la pantalla de nuestra computadora veremos el valor de la variable N. Es decir: “30” (tomando el valor que se leyó en el ejemplo de LEER).

Como vemos, para diferenciar entre la escritura de un texto y una variable, en el caso del texto lo encerramos entre comillas simples. Si queremos combinar variables con texto, van separados con una coma (,).

Los cálculos que realizan las computadoras requieren para ser útiles la *entrada* de los datos necesarios para ejecutar las operaciones que posteriormente se convertirán en resultados, es decir, *salida*.

Las operaciones de entrada permiten leer determinados valores y asignarlos a determinadas variables. Esta entrada se conoce como operación de **lectura**. Los datos de entrada se introducen al procesador mediante dispositivos de entrada (teclado, mouse, scanner, tarjetas perforadas, unidades de disco, etc.). La salida puede aparecer en un dispositivo de salida (pantalla, impresora, unidades de disco, parlantes, etc.). La operación de salida se denomina **escritura**.

❖ Ejemplos.

Problema: Obtener dos números enteros y mostrar la suma de los mismos.

Análisis

Entradas: Dos números (enteros).

Salidas: Suma de los números (entera).

Diseño

Proceso:

- Leer los números.
- Sumar los números.
- Mostrar la suma de los números.

Algoritmo

escribir “Ingresar los valores a sumar”

leer NUM1, NUM2

SUMA ← NUM1 + NUM2

escribir “La suma de los valores ingresados es: ”, SUMA

Otra forma de escribirlo sería:

escribir “* Suma de dos números enteros *”

escribir “Ingrese el primer número”

leer NUM1

escribir “Ingrese el segundo número”

leer NUM2

SUMA ← NUM1 + NUM2

escribir “La suma de los valores ingresados es: ”, SUMA

Apunte de Cátedra

Problema: Leer una letra desde el teclado y mostrarla en pantalla tres veces separada por punto y coma.

Análisis

Entradas: Letra (caracter).

Salidas: (mensaje).

Diseño

Proceso:

- Leer la letra.
- Mostrar la letra tres veces, separada por punto y coma.

Algoritmo

escribir “Ingrese una letra”

leer LETRA

escribir LETRA, “;”, LETRA, “;”, LETRA

Nota: Como se aprecia en los ejemplos antes descriptos, las sentencias están resaltadas en negrita; esto se debe a que son **palabras reservadas**, que se diferencian del resto del algoritmo. En el caso de los algoritmos escritos a mano resalte las palabras reservadas mediante el subrayado de los mismos.

ESTRUCTURAS DE CONTROL

El orden de ejecución de las sentencias (enunciados o instrucciones) en un algoritmo se llama **flujo de control**. Aunque un flujo normal de un algoritmo es lineal, existen ocasiones en que es necesario salir del flujo lineal. La forma de manejar el flujo de control en un algoritmo se logra a través de las estructuras de control. Estas estructuras pueden ser:

- **Secuenciación.**
- **Selección.**
- **Iteración.**

Nota: En adelante se utilizan los términos enunciado, sentencia e instrucción en forma indistinta.

Secuenciación

La secuenciación se cumple cuando las sentencias se ejecutan siguiendo el orden físico, es decir, el orden en que están escritas.

Comenzamos con el primero paso, continuamos con el segundo, seguimos con el tercero y así hasta llegar al fin. Respetando siempre el orden.

Enunciado	1
Enunciado	2
Enunciado	3
.	
.	
.	
Enunciado	n

En ningún caso se ejecutará el enunciado 2 antes que el enunciado 1. El orden de ejecución será exactamente como están escritas.

❖ Ejemplos.

Problema: Calcular la cantidad de segundos que contiene una determinada cantidad de horas.

Análisis

Entradas: Horas (entero).
Salidas: Segundos (entero).

Diseño

Proceso:

- Leer la hora.
- Calcular los segundos que hay esa hora.
- Mostrar los segundos.

Algoritmo

escribir "Ingrese una hora para calcularle los segundos"

leer HORAS

SEGUNDOS ← HORAS * 60 * 60

escribir "La cantidad de segundos que tiene es: ", SEGUNDOS

Apunte de Cátedra

Problema: Escribir un algoritmo que dado el peso en libras de un objeto calcule y muestre el mismo en kilogramos y gramos (1 libra: 0.453592 kilogramos o 1 libra: 453.59237 gramos).

Análisis

Entradas: Peso en libras (real).

Salidas: Peso en kilogramos y peso en gramos (reales).

Diseño

Proceso:

- Leer el peso en libras.
- Calcular el peso en kilogramos del peso en libras.
- Calcular el peso en gramos del peso en libras.
- Mostrar los respectivos pesos.

Algoritmo

escribir “Ingrese un determinado peso expresado en libras”

leer PESO_LIBRAS

$PESO_KG \leftarrow PESO_LIBRAS * 0.453592$

$PESO_GR \leftarrow PESO_LIBRAS * 453.59237$

escribir “El peso: ”, PESO_LIBRAS, “ (en libras) equivale a: ”, PESO_KG, “ kilogramos”

escribir “El peso: ”, PESO_LIBRAS, “ (en libras) equivale a: ”, PESO_GR, “ gramos”

Problema: Convertir una temperatura de grados Celsius a grados Fahrenheit. (1 Fahrenheit: $9/5 \times 1$ Celsius + 32).

Análisis

Entradas: Temperatura en Celsius (entero).

Salidas: Temperatura en Fahrenheit (real).

Diseño

Proceso:

- Leer la temperatura en grados Celsius.
- Convertir la temperatura en Fahrenheit.
- Mostrar la temperatura resultante.

Algoritmo

escribir “Ingrese una temperatura en grados Celsius”

leer CELSIUS

$FAHRENHEIT \leftarrow 9/5 * CELSIUS + 32$

escribir “La temperatura en grados Celsius: ”, CELSIUS, “ equivale a: ”, FAHRENHEIT, “ grados Fahrenheit”

Selección

Esta estructura se utiliza cuando el algoritmo debe elegir (seleccionar) entre dos o más acciones posibles. Es decir, el algoritmo debe permitir que se tome una decisión y luego ejecutar una de varias acciones (camino) posibles. También se lo conoce con el nombre de Selección Dicotómica. Existen dos tipos de estructuras:

- **SI-SINO.**
- **ALTERNAR-CASO.**

Apunte de Cátedra

Estructura SI-SINO

Para poder tomar una decisión debe plantearse una condición (expresión booleana) que al ser evaluada dará un resultado que indicará las acciones (otros enunciados o sentencias) a seguir.

Formato general:

```
si (condición)
    enunciado1
[sino
    enunciado2]
```

La estructura SI-SINO evalúa cierta condición. Si el resultado es verdadero ejecuta el enunciado 1, caso contrario, si el resultado es falso, se ejecuta el enunciado 2. Es importante tener en cuenta que siempre se ejecutará sólo uno de los enunciados (el que le sigue al si o el que le sigue al sino), pero debemos escribir en el algoritmo ambos.

Las palabras **si** y **sino**, son palabras reservadas.

❖ Ejemplos.

Problema: Dado un número entero informar si el mismo es positivo o negativo.

Análisis

Entradas: Número (entero).

Salidas: (mensaje).

Diseño

Proceso: - Leer el número.
- Decir si es positivo o negativo.

Algoritmo

escribir "Ingrese un número entero"

leer NUM

si (NUM < 0)

escribir "El número ingresado es negativo"

sino

escribir "El número ingresado es positivo"

Problema: Dado un número entero determinar si es par o impar.

Análisis

Entradas: Número (entero).

Salidas: (mensaje).

Diseño

Proceso: - Leer el número.
- Decir si es par o impar.

Algoritmo

escribir "Ingrese un número entero"

leer NUMERO

Apunte de Cátedra

```
RESTO ← NUMERO % 2
si (RESTO == 0)
    escribir "El número ingresado es PAR"
sino
    escribir "El número ingresado es IMPAR"
```

Otra manera de escribir dicho algoritmo sería:

```
escribir "Ingrese un número entero"
leer NUMERO
si (NUMERO % 2 == 0)
    escribir "El número ingresado es PAR"
sino
    escribir "El número ingresado es IMPAR"
```

Como vemos en este último caso, con una sola variable se pudo armar el algoritmo. Cualquiera de las dos alternativas es válida.

Problema: Mostrar el valor absoluto de un número.

Análisis

Entradas: Número (real).

Salidas: Número en valor absoluto (real).

Diseño

Proceso:

- Leer el número.
- Pasar el número a valor absoluto.
- Mostrar el resultado de la transformación.

Algoritmo

```
escribir "Ingrese un número"
leer NUM
si (NUM < 0)
    ABSNUM ← (-1) * NUM
sino
    ABSNUM ← NUM
escribir "El número expresado en valor absoluto es: ", ABSNUM
```

Existen ocasiones, en las que el problema exige que "se ejecute una determinada acción si ocurre determinada condición, sino no ocurre nada". Como en los siguientes casos:

```
si (EDAD == 15)
    escribir "FELIZ ETAPA!!!"
```

```
si (DIA == 25 & MES == 12)
    escribir "FELIZ NAVIDAD"
```

Estos casos demuestran que no es necesario incorporar el SINO, ya que en el programa o proceso, la carencia de dicha instrucción no afecta al desarrollo o resultado del mismo.

Apunte de Cátedra

Problema: Dado un año informar si el mismo es bisiesto o no. Un año es bisiesto si es múltiplo de 4 (por ejemplo: 1984). Los múltiplos de 100 no son bisiestos, salvo si ellos son también múltiplos de 400. (2000 es bisiesto, 1800 no lo es).

Análisis

Entradas: Año (entero).
Salidas: (mensaje).

Diseño

Proceso:

- Leer el año.
- Verificar si es bisiesto o no.
- Mostrar el mensaje correspondiente.

Algoritmo

escribir "Ingrese un año determinado"

leer ANIO

BISIESTO ← falso

si (ANIO % 4 == 0)

BISIESTO ← verdadero

si (ANIO % 100 == 0 & ANIO % 400 != 0)

BISIESTO ← falso

si (BISIESTO == verdadero)

escribir "El año ingresado es bisiesto"

sino

escribir "El año ingresado NO es bisiesto"

Supongamos el siguiente problema: Ingresar una temperatura y mostrar qué deporte es apropiado para esa temperatura, usando los siguientes criterios:

Deporte	Temperatura (rangos)
Natación	> 85
Tenis	70 < TEMP <= 85
Golf	32 < TEMP <= 70
Esquí	10 < TEMP <= 32
Damas	<= 10

El algoritmo podría ser de la siguiente manera:

escribir "Ingrese una temperatura"

leer TEMP

si (TEMP > 85)

escribir "El deporte adecuado para dicha temperatura es la NATACIÓN"

si (TEMP > 70 & TEMP <= 85)

escribir "El deporte adecuado para dicha temperatura es el TENIS"

si (TEMP > 32 & TEMP <= 70)

escribir "El deporte adecuado para dicha temperatura es el GOLF"

si (TEMP > 10 & TEMP <= 32)

escribir "El deporte adecuado para dicha temperatura es el ESQUÍ"

si (TEMP <= 10)

escribir "El deporte adecuado para dicha temperatura es las DAMAS"

Apunte de Cátedra

El problema de este algoritmo es que cada vez que se ejecuta, se realizan las cinco evaluaciones, sabiendo que sólo será posible de cumplir una. Lo más apropiado es plantear este caso con lo que se conoce como: **SI ANIDADADO**. Si aplicamos este criterio, el algoritmo quedaría así:

escribir “Ingrese una temperatura”

leer TEMP

si (TEMP > 85)

escribir “El deporte adecuado para dicha temperatura es la NATACIÓN”

sino

si (TEMP > 70)

escribir “El deporte adecuado para dicha temperatura es el TENIS”

sino

si (TEMP > 32)

escribir “El deporte adecuado para dicha temperatura es el GOLF”

sino

si (TEMP > 10)

escribir “El deporte adecuado para dicha temperatura es el ESQUÍ”

sino

escribir “El deporte adecuado para dicha temperatura es las DAMAS”

Quizás, este algoritmo es más complejo de leer, pero es mejor que el anterior puesto que se ejecutarán a lo sumo cuatro evaluaciones de condiciones (la última acción sale por decantación de un sino).

Como se ve en el algoritmo, también se ha logrado acortar las condiciones de los SI, preguntando sólo por una de las partes de los rangos correspondientes.

Estructura ALTERNAR-CASO

Esta estructura permite hacer una selección (o alternación) de múltiples posibilidades. La estructura ALTERNAR-CASO es una abreviatura de estructuras SI anidadas. Tenemos las siguientes sentencias:

si (X == valor_constante_1)

enunciado1

sino

si (X == valor_constante_2)

enunciado2

sino

si (X == valor_constante_3)

enunciado3

sino

.
. .
.

En efecto esta construcción anidada no es más que una prueba de “opción múltiple” sobre el valor de la variable X. Otra forma más sencilla y mejorada de escribir dichas sentencias, es a través de la estructura ALTERNAR-CASO.

Apunte de Cátedra

Formato general:

```
alternar (expresión)
{
  caso valor_1 : {
    conjunto_de_sentencias_1
    corte
  }
  caso valor_2 : {
    conjunto_de_sentencias_2
    corte
  }
  .
  .
  .
  caso valor_n : {
    conjunto_de_sentencias_n
    corte
  }
  [caso contrario
    conjunto_de_sentencias_caso_contrario]
}
```

Las sentencias antes mencionadas, usando la estructura ALTERNAR-CASO, quedarán representadas de la siguiente manera:

```
alternar (X)
{
  caso valor_constante_1 : {
    enunciado_1
    corte
  }
  caso valor_constante_2 : {
    enunciado_2
    corte
  }
  caso valor_constante_3 : {
    enunciado_3
    corte
  }
}
```

Otras consideraciones a tener en cuenta:

- Las palabras **alternar**, **caso**, **caso contrario** y **corte** son palabras reservadas.
- La expresión puede ser cualquier expresión de tipo simple que no sea real.
- Cada uno de los valores constantes del CASO representa uno de los valores *permisibles* de la expresión. Así, si la expresión es de tipo entero, los valores constantes del CASO representarán valores enteros dentro de un rango determinado.
- Los valores constantes del CASO no tienen por qué aparecer en un orden particular, aunque cada uno de ellos debe ser único, es decir, no puede aparecer un mismo rótulo más de una vez.

Apunte de Cátedra

- Una sentencia se ejecutará si (y sólo sí) uno de sus valores constantes del CASO correspondiente coincide con el valor actual de la expresión. Por lo tanto, el valor actual de la expresión determinará cuál de las sentencias se va a ejecutar. Si el valor de la expresión no coincide con ninguno de los valores constantes, y no se encuentra definida la instrucción CASO CONTRARIO, la acción a realizar estará indefinida. Es decir, se ejecutará la estructura ALTERNAR-CASO, pero no ocurrirá nada.
- La estructura ALTERNAR-CASO, sólo se utiliza para comparaciones de igualdad. Es decir, no se pueden utilizar los operadores relacionales <, >, <=, y >=.
- La palabra CORTE produce que luego de ejecutarse los enunciados de un CASO (verdadero) el flujo de control salga de la estructura ALTERNAR-CASO, evitando así realizar evaluaciones innecesarias (casos restantes).

En el siguiente ejemplo se supone que la variable OPCION es de tipo caracter:

```
alternar (OPCION)
{
  caso 'A' : {
    escribir "Alfa"
    corte
  }
  caso 'B' : {
    escribir "Beta"
    corte
  }
  caso 'G' : {
    escribir "Gama"
    corte
  }
  caso contrario
    escribir "No es una letra del alfabeto griego"
}
```

En este ejemplo, si la variable caracter tiene el valor 'A' se mostrará el mensaje "Alfa", si tiene el valor 'B' se mostrará el mensaje "Beta" y si tiene el valor 'G' se mostrará el mensaje "Gama", sino (en cualquier otro caso) se mostrará el mensaje "No es una letra del alfabeto griego".

❖ Ejemplos.

Problema: Dado el número correspondiente a un mes, mostrar el nombre de dicho mes. Por ejemplo: si el número es 1 el nombre correspondiente es ENERO.

Análisis

Entradas: Número (entero).
Salidas: (mensaje).

Diseño

Proceso:

- Leer el número del mes.
- Verificar cuál es su nombre.
- Mostrar el nombre correspondiente.

Apunte de Cátedra

Algoritmo

escribir “Trabajando con los meses, ingrese un número entero”

leer MES

escribir “El nombre correspondiente al número del mes ingresado es: ”

alternar (MES)

```
{
  caso 1 :      {
                  escribir “ENERO”
                  corte
                }
  caso 2 :      {
                  escribir “FEBRERO”
                  corte
                }
  caso 3 :      {
                  escribir “MARZO”
                  corte
                }
  caso 4 :      {
                  escribir “ABRIL”
                  corte
                }
  caso 5 :      {
                  escribir “MAYO”
                  corte
                }
  caso 6 :      {
                  escribir “JUNIO”
                  corte
                }
  caso 7 :      {
                  escribir “JULIO”
                  corte
                }
  caso 8 :      {
                  escribir “AGOSTO”
                  corte
                }
  caso 9 :      {
                  escribir “SEPTIEMBRE”
                  corte
                }
  caso 10 :     {
                  escribir “OCTUBRE”
                  corte
                }
  caso 11 :     {
                  escribir “NOVIEMBRE”
                  corte
                }
}
```

Apunte de Cátedra

```

caso 12 :      {
                  escribir "DICIEMBRE"
                  corte
                }

caso contrario
    escribir "NUMERO DE MES INCORRECTO"
}

```

Problema: Leer dos números enteros y leer un símbolo, si dicho símbolo se encuentra entre los operadores +, -, *, /, y ^, por cada uno de ellos efectuar las operaciones correspondientes. En el caso de la suma, evaluar el resultado si es par o impar, en el caso de la resta, evaluar el resultado si es negativo o positivo y en el caso de la división, determinar si el resultado es igual a 12.00. Devolver los resultados y mensajes correspondientes.

Análisis

Entradas: Números (entero) y símbolo (carácter).
Salidas: Resultado (entero o real) y (mensaje).

Diseño

Proceso:

- Leer los números.
- Leer el símbolo.
- Determinar el tipo de símbolo.
- Realizar la operación y evaluación correspondiente.
- Mostrar los resultados y mensajes correspondientes.

Algoritmo

```

escribir "Trabajando con los operadores"
escribir "Ingrese el primer operador entero"
leer NUM1
escribir "Ingrese el segundo operador entero"
leer NUM2
escribir "Ingrese un símbolo cualquiera (+, -, *, /, ^, ...)"
leer SIMB
alternar (SIMB)
{
  caso '+' :      {
                  RESUL ← NUM1 + NUM2
                  si (RESUL % 2 == 0)
                    escribir "El resultado de la suma es PAR"
                  sino
                    escribir "El resultado de la suma es IMPAR"
                  corte
                }
  caso '-' :      {
                  RESUL ← NUM1 - NUM2
                  si (RESUL > 0)
                    escribir "El resultado de la resta es POSITIVO"
                  sino
                    escribir "El resultado de la resta es NEGATIVO"
                  corte
                }
}

```

Apunte de Cátedra

```
    }  
    caso '*' :  
    {  
        RESULT ← NUM1 * NUM2  
        corte  
    }  
    caso '/' :  
    {  
        RESULT ← NUM1 / NUM2  
        si (RESULT == 12.00)  
            escribir "El resultado de la división es igual al número 12.00"  
        sino  
            escribir "El resultado de la división NO es igual al número 12.00"  
        corte  
    }  
    caso '^' :  
    {  
        RESULT ← NUM1 ^ NUM2  
        corte  
    }  
    caso contrario  
        escribir "Operador incorrecto"  
    }  
escribir NUM1, " ", SIMB, " ", NUM2, " = ", RESULT
```

Como vemos en los dos problemas anteriores, las sentencias pueden ser simples o compuestas. En el caso de tratarse de una sentencia compuesta con varias operaciones, se incorporan la llave de apertura '{' y la llave de cierre '}', que delimitan el bloque correspondiente a la sentencia. No es necesario escribirlas cuando existe una sola sentencia, pero no es un error hacerlo y se recomienda para mayor claridad.

Nota: A partir de este momento, cada vez que escribamos un algoritmo, vamos a iniciarlo con una llave de apertura principal y vamos a concluirlo con una llave de cierre principal. Es decir, el algoritmo quedará delimitado entre llaves.

Iteración

Esta estructura de control, como su nombre lo indica, posibilita repetir la ejecución de una sentencia o un conjunto de sentencias.

Anteriormente hemos visto que en la secuencia el flujo de control sigue el orden físico de las sentencias, por lo tanto, diremos que el orden lógico (orden en que queremos que se ejecuten las sentencias) es igual que el orden físico. En cambio en la selección el flujo de control difiere del orden físico de las sentencias, en este caso el orden físico y el orden lógico son diferentes. En la iteración el orden lógico también difiere del orden físico de las instrucciones. Una iteración se conoce también como **bucle** o **ciclo**.

Existen tres tipos de estructuras que nos permiten iterar acciones:

- **MIENTRAS.**
- **HACER-MIENTRAS.**
- **PARA.**

Nota: En adelante se usan de manera indistinta los términos bucle, ciclo e iteración.

Apunte de Cátedra

Estructura MIENTRAS

Esta sentencia evalúa una condición (expresión booleana), si esta condición es verdadera ejecuta el enunciado, vuelve atrás (o arriba) y evalúa de nuevo la condición. Este proceso se repetirá hasta que la condición sea falsa. Cuando al evaluar la condición, esta da un resultado falso, ejecuta la próxima sentencia fuera de la iteración o bucle y no vuelve atrás.

Formato general:

mientras (condición)
enunciado1

Evalúa cierta condición booleana. Si el resultado es verdadero, ejecuta el enunciado 1 y continúa de esta manera mientras la condición siga siendo verdadera.

Es claro que el ciclo se romperá cuando la condición deje de ser verdadera, es decir, se vuelva falsa. Es posible incluso que el ciclo nunca se efectúe si de entrada la condición es falsa. Es decir, que la condición inicialmente sea falsa, por lo tanto, no entrará al bucle ni siquiera una vez. En el siguiente ejemplo, si la primera vez que se lee X, se lee -1, la condición $X \geq 0$ es inicialmente falsa, por lo tanto, no ejecutará ni siquiera una vez los enunciados que están dentro del bucle.

```
leer X  
mientras (X >= 0)  
  {  
    escribir X  
    leer X  
  }
```

A primera vista puede parecer que SI y MIENTRAS son parecidas. Tienen similitudes, pero un examen cuidadoso muestra sus diferencias fundamentales. En la estructura SI, el enunciado 1 puede ser saltado o ejecutado exactamente una vez. En la estructura MIENTRAS, el enunciado 1 puede ser saltado o ejecutado una y otra vez, varias veces.

si (condición)
enunciado1
enunciado 2

mientras (condición)
enunciado1
enunciado 2

Cada vez que se ejecuta el cuerpo del bucle (enunciados que deseamos que se repitan), decimos que se ha producido un paso a través del bucle. Este paso se llama una *iteración* del bucle. Cuando se ha realizado el último paso a través de un bucle y el flujo de control procede con la primera sentencia que va a continuación del bucle, decimos que se ha *salido del bucle*. La condición que produce que se salga del bucle se llama *condición de terminación*, que es simplemente la expresión booleana que al ser evaluada en la estructura MIENTRAS dio como resultado Falso.

Supongamos que vamos a pintar las paredes de nuestro dormitorio, y las instrucciones de la pintura que hemos comprado dicen: “aplicar tres manos de pintura”, entonces ejecutaremos esta tarea contando tres veces. Pero si las instrucciones dicen “aplicar tantas manos de pintura hasta obtener el color de la muestra” ejecutaremos esta tarea una cantidad de veces que al principio desconocemos.

De lo anterior podemos hacer una clasificación en dos tipos principales de bucles:

- **Controlados por contador.**
- **Controlados por sucesos.**

Apunte de Cátedra

Bucles controlados por contador

Un bucle controlado por contador es un bucle que se ejecuta o se repite un número especificado (conocido) de veces. Hace uso de una variable llamada **variable de control de bucle (VCB)**.

Hay tres partes en un bucle controlado por contador:

1. Inicializar la variable VCB.
2. Comparar la VCB.
3. Incrementar o decrementar la VCB.

Estas operaciones siempre deben estar acompañadas y son necesarias para que la estructura funcione.

Veamos un ejemplo:

```
CONT ← 1           * Inicialización *  
mientras (CONT <= 10) * Comparación *  
  {  
  .  
  .  
  .  
  CONT ← CONT + 1   * Incremento *  
  }
```

CONT es la variable de control de bucle. Se inicializa en 1 antes del bucle. La estructura MIENTRAS compara la expresión $CONT \leq 10$ y ejecuta las sentencias del cuerpo del bucle si es verdadera. La última sentencia del cuerpo del bucle incrementa CONT sumándole 1. Al utilizar enunciados de la forma: **variable** \leftarrow **variable** + 1, en el cuerpo del bucle hacemos que se incremente repetidamente en 1 el valor de la variable. Cada vez que se ejecuta el enunciado, se añade 1 al valor anterior de la variable y este resultado sustituye al valor anterior. La variable de control de un bucle controlado por contador siempre es un **contador**.

La estructura MIENTRAS siempre prueba primero la condición de terminación. Si la condición es Falsa al principio, nunca se ejecutará el cuerpo del bucle. Es responsabilidad del programador ver que la condición que ha de compararse se fije correctamente (inicialización) antes de que comience la sentencia MIENTRAS.

El programador debe también asegurarse de que la condición cambia dentro del bucle de forma que se pueda convertir en Falso en algún momento; si no ocurre así, el bucle nunca terminará. Un bucle cuya condición de terminación nunca se hace falsa se llama **bucle infinito**, esto significa que tampoco terminará el algoritmo y el programa se ejecutará erróneamente, puesto que no finaliza.

¿Cuántas veces se ejecuta el bucle del ejemplo anterior: 9 o 10?. Para determinar el número de veces, examinamos el valor al que se inicializó la variable de control del bucle y luego vemos cuál es el valor final que toma. En este caso, CONT (la variable de control de bucle) se inicializó en 1. La comparación indica que el cuerpo del bucle se ejecutará mientras que CONT sea menor o igual que 10. Esto es, el cuerpo del bucle se ejecutará por cada valor de CONT hasta 10 y saltará cuando CONT se haga 11. Si CONT comienza en 1 y llega hasta 10, esto significa que el cuerpo del bucle se ejecutará 10 veces. Si quisiéramos ejecutar el bucle 11 veces, podríamos haber inicializado la variable CONT en 0 o cambiando la comparación a “ $CONT \leq 11$ ”.

Apunte de Cátedra

Bucles controlados por sucesos

Hay varios tipos de bucles controlados por sucesos:

- controlados por centinelas;
- controlados por indicador;
- controlados por EOF y;
- controlados por EOLN.

Los dos últimos se verán mas adelante con Archivos. El factor común de todos estos bucles es que la condición de terminación depende de algún suceso que ocurre durante la ejecución del cuerpo del bucle que señala que el proceso de iteración debe terminar.

Bucles controlados por centinelas: Los bucles se utilizan frecuentemente para leer y procesar grandes listas de datos. Cada vez que se ejecuta el cuerpo del bucle, se lee un nuevo dato y luego se procesa. Sin embargo, hay un problema con este proceso: ¿Cómo decide el programa cuando parar el ciclo de proceso de lectura y salir del bucle?.

Una solución frecuente a este problema consiste en un utilizar un dato especial llamado **centinela**. Este valor, es una señal de que no hay que procesar mas datos. En otras palabras, la lectura de un centinela es el suceso que controla el proceso de iteración. El proceso continúa mientras que los valores leídos no sean el centinela. Se sale del bucle cuando se reconoce al centinela.

Un centinela debe elegirse con cuidado. No puede ser un valor normal. Un centinela debe ser algo que nunca sea un valor posible entre los datos utilizados. Por ejemplo, si un programa lee fechas de calendario, podríamos usar una variable día igual a 0 como centinela. Tal bucle podría tener la siguiente forma:

```
leer DIA * Obtención del primer valor de centinela *  
mientras (DIA != 0) * Comparación del centinela *  
{  
  .  
  .  
  .  
leer DIA * Obtención del nuevo valor de centinela *  
}
```

En este ejemplo cuando se lea 0 en la variable DIA, finalizará la iteración. Como podemos observar, la primera vez se lee el centinela antes de entrar al bucle, lo cual se conoce como *lectura adelantada*, de esta forma se inicializa el centinela con valores reales. Dentro del cuerpo del bucle, primero se procesan los datos y al final del cuerpo del bucle obtenemos el nuevo valor para el centinela. Si olvidamos hacer este último paso, tendremos un bucle infinito, el bucle que nunca termina.

Partes de un bucle controlado por centinela:

1. Obtener (leer o asignar) el valor de la variable centinela.
2. Comparar la variable centinela.
3. Procesar los datos.
4. Obtener el nuevo valor de la variable centinela.

Bucles controlados por indicador: Un indicador es una variable booleana que se utiliza para controlar el flujo lógico de un programa. Podemos colocar una variable booleana en Verdadero antes de un MIENTRAS y luego cuando queramos detener la ejecución del bucle, colocarla en Falso. Esto es, se utiliza una variable booleana para registrar si se ha producido o no el suceso que controla el proceso.

Apunte de Cátedra

Por ejemplo, a continuación se realiza una lectura de números y se muestran los valores hasta que el dato contenga un valor negativo.

```

POSITIVO ← Verdadero                                * Inicialización del indicador *
mientras (POSITIVO == Verdadero)
{
  leer NUMERO
  si (NUMERO < 0)                                    * Comparación del valor leído *
    POSITIVO ← Falso                                * Cambio del indicador *
  sino
    escribir NUMERO
}
  
```

Los bucles controlados por centinelas pueden codificarse con indicador. No estamos limitados a inicializar el indicador siempre en verdadero. Podemos inicializarlo en Falso y cuando ocurra el suceso cambiarlo a Verdadero para detener la iteración.

Hemos examinado de qué forma utilizar bucles para aceptar el flujo de control de nuestro programa. Pero la iteración por sí misma no hace nada. El cuerpo del bucle debe ejecutar una tarea a efectos de que el bucle sea útil. Algunas **subtareas iterativas** muy frecuentes son:

- **Bucles contadores.**
- **Bucles sumadores o acumuladores.**

Bucles contadores

El bucle del siguiente ejemplo incluye una variable contadora; sin embargo, el bucle **no** es un bucle controlado por contador. En un bucle controlado por contador se utiliza la variable contador y la condición de terminación del bucle depende del valor de dicho contador. Como demuestra el siguiente ejemplo, hay otros usos para un contador distintos a los de controlar la ejecución del bucle.

Una subtarea frecuente es llevar la cuenta de cuantas veces se ejecuta el bucle, aunque este contador no se utilice para controlar la ejecución del bucle. En el ejemplo propuesto, se desea saber cuantos caracteres distintos de “.” se han leído.

```

CONT ← 0                                             * Inicialización del contador *
leer CARACTER                                       * Lectura del primer caracter *
mientras (CARACTER != '.')
{
  CONT ← CONT + 1                                    * Incremento del contador *
  leer CARACTER                                       * Lectura del próximo caracter *
}
escribir CONT
  
```

El bucle continúa hasta que se lee un punto. En la terminación del bucle, la variable CONTADOR contiene la cantidad de caracteres leídos, distintos del caracter punto. Al salir del bucle se muestra dicho resultado. Se utiliza una lectura adelantada porque este bucle esta controlado por centinela. La variable CONTADOR de este ejemplo se llama **contador de iteración**, porque su valor es igual al número de iteraciones del bucle.

Bucles acumuladores o sumadores

Supongamos que deseamos obtener la suma de 10 valores.

Apunte de Cátedra

```

CONT ← 1                                * Inicialización de la variable de control *
SUMA ← 0                                  * Inicialización de la suma *
mientras (CONT <= 10)
{
  leer NUMERO                             * Lectura del valor *
  SUMA ← SUMA + NUMERO                     * Añadido del valor a suma *
  CONT ← CONT + 1                          * Incremento de la variable de control *
}
escribir SUMA                             * Muestra de la suma *

```

Este bucle está controlado por contador. Cuando se haya ejecutado este algoritmo, la variable SUMA contendrá la suma de los 10 valores leídos. CONT contendrá 11 y NUMERO contendrá el último valor leído.

Veamos otro ejemplo. Supongamos que queremos obtener la suma de los primeros 10 valores pares leídos.

```

CONT ← 1                                * Inicialización de la variable de control *
SUMA ← 0                                  * Inicialización de la suma *
mientras (CONT <= 10)
{
  leer NUMERO                             * Lectura del valor *
  si (NUMERO % 2 == 0)
  {
    SUMA ← SUMA + NUMERO                   * Añadido del valor a suma *
    CONT ← CONT + 1                       * Incremento de la variable de control *
  }
}
escribir SUMA                             * Muestra de la suma *

```

Se parece mucho al ejemplo anterior, sólo que incrementa a CONT y agrega a SUMA sólo si el valor leído es un número par. En este caso al finalizar la iteración, no sabemos cuantas veces se iteró. En este caso, CONT es un **contador de sucesos**, porque sólo se incrementa si ocurre determinado suceso.

¿Cómo diseñar bucles?

El diseño de un bucle es esencialmente una materia de determinación de qué condición debe existir al comienzo de cada paso del bucle para que funcione correctamente. La característica clave de estas condiciones es **qué** es verdad al comienzo de cada iteración del bucle. Debido a que son siempre verdad al comienzo de cada iteración, estas condiciones se llaman **invariantes del bucle**.

Invariante del bucle: son las condiciones que deben existir al comienzo de cada iteración de un bucle particular para que el bucle se ejecute correctamente.

La invariante del bucle no es simplemente la condición de la estructura MIENTRAS, que determina si el cuerpo del bucle se ejecuta o no. Las otras condiciones que forman la invariante, y que deben también considerarse, son cosas tales como: la actualización de contadores, sumadores, etc.

Apunte de Cátedra

Veamos un ejemplo de condición invariante. Si un problema nos pide sumar los primeros 10 enteros impares leídos, la variable que lleva el contador de los enteros impares debe comenzar en 0 y puede aumentar hasta 10. Si el contador va mas allá de este rango, hay algo erróneo en el diseño del bucle. Otra condición invariante de este problema nos exigirá que el valor del contador sea igual al número de impares que haya en la entrada. Esto puede parecer obvio, pero tiene sus implicaciones: el contador debe ser inicializado en 0 (ningún número impar ha sido leído al principio) y debe incrementarse cada vez que se introduce un número impar (no cada vez que se ejecuta el bucle). Ahora sabemos que el algoritmo para el bucle debe contener al menos los siguientes pasos:

```
Inicialización de CONTIMPARES
mientras (CONTIMPARES sea menor o igual que 10)
{
.
.
.
si (NUMERO es impar)
    Incremento de CONTIMPARES
.
.
.
}
```

Podemos entonces traducir este algoritmo en las siguientes sentencias:

```
CONTIMPARES ← 0
mientras (CONTIMPARES <= 10)
{
.
.
.
si (NUMERO % 2 != 0)
    CONTIMPARES ← CONTIMPARES + 1
.
.
.
}
```

Pero este algoritmo no es suficiente para que se ejecute el bucle correctamente. Aún debe añadirse otra condición al invariante y es que el nuevo valor debe estar disponible en la entrada al comienzo de cada iteración. Para asegurar que es un nuevo valor para cada iteración, debe haber una sentencia del bucle que lea un nuevo valor durante cada iteración. La sentencia LEER requerida debe insertarse antes de la sentencia SI, porque la sentencia SI compara el nuevo valor.

La invariante del bucle es:

1. El valor del contador está en el rango de 0 a 10, inclusive.
2. El valor del contador debe ser igual a la cantidad de números impares que se hayan leído.
3. Debe estar disponible un nuevo valor en la entrada al comienzo de cada iteración.

Apunte de Cátedra

El algoritmo revisado es:

```

CONTIMPARES ← 0
mientras (CONTIMPARES <= 10)
{
  leer NUMERO
  si (NUMERO % 2 != 0)
    CONTIMPARES ← CONTIMPARES + 1
}
  
```

Este algoritmo es suficiente para realizar lo que requiere el problema. Sin embargo, aún no está completo. Debemos añadir una sentencia **ESCRIBIR** antes de **LEER** para solicitar al usuario que introduzca un número y otra fuera del bucle para mostrar los resultados del proceso.

Determinación del invariante de un bucle: la escritura del invariante de un bucle es una buena forma de comenzar a alumbrar todas las cosas que deben hacerse para que un bucle funcione correctamente. Una vez que se hayan determinado las operaciones esenciales podemos volver atrás y añadir las operaciones no esenciales que refuerzan el algoritmo o ejecutan tareas extras.

¿Cómo determinar todas las condiciones que forman el invariante de un bucle?. Esto no es siempre fácil de hacer, e incluso los programadores con experiencia ocasionalmente dejan una o más de las condiciones invariantes cuando diseñan un bucle. Sin embargo, hay un conjunto de preguntas que nos pueden ayudar a comenzar en forma correcta:

1. ¿Cuál es la condición que termina el bucle?.
2. ¿Cómo debe inicializarse y actualizarse dicha condición?.
3. ¿Cuál es el proceso que se repite?.
4. ¿Cómo debe inicializarse y actualizarse el proceso?.

El diseño de un bucle puede dividirse en dos tareas: diseño del flujo de control y diseño del procesamiento que tiene lugar en el bucle. Las dos primeras preguntas nos ayudarían a diseñar las partes del bucle que controlan su ejecución. Las últimas dos preguntas nos ayudarían a diseñar el procesamiento dentro del cuerpo del bucle. Examinemos mas detalladamente cada una de estas tareas del diseño y sus correspondientes preguntas.

Diseño del flujo de control de un bucle

En esta parte debemos examinar la condición de terminación del bucle. El aspecto más importante de un bucle es decidir que hará que se detenga el bucle. Si la condición de terminación no está bien pensada, existen grandes posibilidades de caer en bucles infinitos y otros errores. Así pues, nuestra primera pregunta al comenzar un bucle debe ser: *¿Cuál es la condición de terminación del bucle?*.

La otra forma de ver esto es: “¿Qué condición debe ser *Verdad* para que el bucle *termine*?”. Esta pregunta puede ser respondida normalmente mediante un examen minucioso de la declaración del problema, por ejemplo:

Frase clave de la declaración del problema

Condición de terminación

“Suma de las 365 temperaturas diarias”	El bucle termina cuando un contador alcance 366 (controlado por contador).
“Hasta que se hayan leído 10 enteros impares”	El bucle termina cuando se hayan leído 10 números impares (controlado por sucesos).
“El final de los datos se indica por una puntuación negativa del examen”	El bucle termina cuando se haya encontrado un valor negativo de entrada (controlado por centinela).

Apunte de Cátedra

Una vez que hayamos determinado la condición de terminación del bucle, podemos escribir la parte del invariante que se convertirá en la condición de la sentencia MIENTRAS. Esencialmente, el invariante es justo lo opuesto de la condición de terminación. Esto es así debido a que la condición de la sentencia del MIENTRAS debe convertirse en Falsa para que *termine* el bucle. Escribimos la correspondiente condición en el invariante para expresar el hecho de que debe ser Verdadero para que el bucle sea *ejecutado*.

Por ejemplo,

CONTTEMP debe ser menor a 366.

CONTIMPAR debe ser menor o igual que 10.

Podemos ver que estos ejemplos son exactamente los opuestos de las condiciones de terminación. Todas las condiciones invariantes son sentencias sobre lo que debe ser Verdad al comienzo de una iteración para que la iteración se ejecute correctamente.

La simple determinación de la apropiada condición de terminación no es suficiente para asegurar que el bucle termina o que se ejecutará. Debemos también incluir sentencias que aseguren que el bucle comenzará correctamente y producirá que el bucle alcance la condición de terminación. La siguiente pregunta a responder es: *¿Cómo debe inicializarse y actualizarse la condición?*. La respuesta depende del tipo de condición de terminación.

Veamos algunos ejemplos:

- Controlado por centinela: una lectura adelantada puede ser la única inicialización requerida en un bucle controlado por centinela.

- Al comienzo de cada iteración debe leerse un nuevo valor (esto implica una lectura adelantada antes de la primera iteración y una lectura actualizada al final de cada iteración).

- Contador de iteraciones y contador de sucesos: a la variable contador debe dársele un valor inicial. En un bucle controlado por contador este valor inicial será normalmente 1. Si la variable contador es para contar sucesos dentro del bucle, normalmente será 0. Si el proceso requiere que el contador vaya en un rango específico de valores, el valor inicial debe ser el valor más bajo del rango.

La operación de actualización en un bucle contador requiere que el valor del contador se incremente en 1 en cada iteración o suceso (ocasionalmente encontraremos algún problema que requiere un contador que decremente o que incremente en más de una unidad).

- CONTTEMP debe ser igual al número de la iteración que se está ejecutando (esto implica inicializar en 1 e incrementar en 1 en cada iteración).

- CONTIMPAR debe ser igual al número de enteros impares que se hayan leído (esto implica inicializar en 1 e incrementar en 1 cada vez que se lea un entero impar).

- Controlado por indicador: la variable indicadora booleana debe inicializarse en Verdadero o Falso, según lo apropiado. Por ejemplo:

- ENCONTRADO debe ser igual a Falso al comienzo de cada iteración.

- POSITIVO debe ser igual a Verdadero al comienzo de cada iteración.

La operación de actualización es algo diferente en un bucle controlado por indicador, a otros bucles. La variable indicadora permanece esencialmente sin cambiar hasta que llega el momento de terminar el bucle. Alguna condición dentro del proceso que se repite detectará y producirá una sentencia de asignación para cambiar el valor del indicador. Puesto que la actualización depende del proceso, hemos de diseñar el proceso antes de diseñar la operación de actualización en un bucle controlado por indicador.

Apunte de Cátedra

Diseño del proceso interior del bucle

Este proceso debe primero decidir qué ha de realizarse en cada iteración. Ignoramos el bucle, suponiendo por un momento que el proceso se va a ejecutar solo una vez. ¿Qué tarea debe ejecutar el proceso?. En otras palabras, *¿Cuál es el proceso que se va a repetir?*. Para responder a esta pregunta debemos examinar la declaración del problema; por ejemplo:

“Hallar la suma de los primeros 100 números múltiplos de 3”. Este enunciado nos dice que el proceso a repetir es una operación de suma.

“Leer un sueldo bruto por cada empleado de una empresa y calcular el sueldo neto”. En este caso debemos repetir la lectura de un dato y luego el cálculo.

La lectura, conteo y suma son operaciones que se utilizan frecuentemente en los procesos iterativos. Otro proceso frecuente en los bucles implica la lectura de datos, la ejecución de un cálculo y la escritura de resultados. Hay muchas otras operaciones que pueden aparecer en los procesos iterativos, los mencionados aquí son sólo los ejemplos más sencillos.

Después que hallamos determinado las operaciones básicas que hay que ejecutar, podemos diseñar las partes del proceso que son necesarias para el bucle. Cuando se coloca un proceso en el cuerpo del bucle, frecuentemente tenemos que añadir algunos pasos para tener en cuenta el hecho de que se ejecutan más de una vez. Esta parte del diseño implica normalmente la inicialización de ciertas variables antes de entrar en el bucle y luego la reinicialización o actualización de las variables antes de que se vaya al siguiente paso del bucle. Por lo tanto, debemos preguntar, *¿Cómo debe inicializarse y actualizarse el proceso?*. Esto dependerá de que tipo de proceso se trate. Si el proceso es una operación de suma, implica la inicialización en 0 y la actualización añadiéndole datos a la suma. Esto puede ser expresado por un invariante de la forma:

- Al comienzo de cada iteración, SUMA debe ser igual al total de todos los múltiplos de 3 que se hayan leído.

Esta condición implica que antes de que se haya leído cualquier número, SUMA es igual a 0.

Un tipo similar de invariante puede usarse para describir las operaciones de conteo de sucesos:

- Al comienzo de cada iteración, CONTADOR debe ser igual al número de sucesos que haya ocurrido.

A veces el proceso de un bucle requerirá varias sumas diferentes y que se realicen varias cuentas. Cada uno de éstos tendrá su propia condición invariante, lo cual significa simplemente que habrá más sentencias para inicializar variables, más sentencias para realizar sumas y más sentencias para incrementar o decrementar variables contadores. Tratar cada operación de suma o de conteo por sí misma: primero escribir el invariante para la operación específica, luego escribir la sentencia de inicialización, seguida de la sentencia de suma o de incremento. Cuando se haya hecho esto para cada operación de suma o conteo particular, se puede ir a la siguiente.

Nota: A partir de este momento todos los problemas ejemplos no tendrán el Análisis, ni el Diseño (queda a cargo del lector).

❖ Ejemplos.

Problema: Ingresar una serie de números enteros y mostrar la sumatoria de los pares y la sumatoria de los impares.

Apunte de Cátedra

```
{  
escribir "Ingrese la cantidad de elementos de la serie"  
leer CANTIDAD  
CONTADOR ← 1  
SUMAPAR ← 0  
SUMAIMPAR ← 0  
mientras (CONTADOR <= CANTIDAD)  
  {  
    escribir "Ingrese un número entero"  
    leer NUMERO  
    si (NUMERO % 2 = 0)  
      SUMAPAR ← SUMAPAR + NUMERO  
    sino  
      SUMAIMPAR ← SUMAIMPAR + NUMERO  
    CONTADOR ← CONTADOR + 1  
  }  
escribir "La sumatoria de los números pares es: ", SUMAPAR  
escribir "La sumatoria de los números impares es: ", SUMAIMPAR  
}
```

Otra forma de resolver el problema sería:

```
{  
escribir "Ingrese la letra para cargar elementos"  
leer CONTINUA  
SUMAPAR ← 0  
SUMAIMPAR ← 0  
mientras (CONTINUA == 'S')  
  {  
    escribir "Ingrese un número entero"  
    leer NUMERO  
    si (NUMERO % 2 = 0)  
      SUMAPAR ← SUMAPAR + NUMERO  
    sino  
      SUMAIMPAR ← SUMAIMPAR + NUMERO  
    escribir "¿Continúa cargando elementos? (s/n)"  
    leer CONTINUA  
  }  
escribir "La sumatoria de los números pares es: ", SUMAPAR  
escribir "La sumatoria de los números impares es: ", SUMAIMPAR  
}
```

Problema: Ingresar una serie de caracteres y contar la cantidad de letras 'A' que aparecen. Termina la serie cuando se ingresa un '*'.

```
{  
escribir "Ingrese un caracter (con '*' termina)"  
leer LETRA  
CANTIDAD ← 0
```


Apunte de Cátedra

```

ientras (LETRA != '*')
  {
    si (LETRA == 'A' | LETRA == 'a')
      CANTIDAD ← CANTIDAD + 1
    escribir "Ingrese otro caracter (recuerde, con '*' termina)"
    leer LETRA
  }
escribir "La cantidad de letras A ingresadas es: ", CANTIDAD
  }
  
```

Estructura HACER-MIENTRAS

El bucle HACER-MIENTRAS es similar al MIENTRAS, con la diferencia que la condición (expresión booleana) se evalúa al final del bucle (en lugar del principio). Sin embargo, un bucle HACER-MIENTRAS al contrario de un MIENTRAS, se ejecuta siempre al menos una vez.

Formato general:

```

hacer
  enunciado1
ientras (condición)
  
```

El segmento significa "hacer el enunciado MIENTRAS que la condición sea verdadera".
 Veamos un ejemplo de dos segmentos de código que realizan lo mismo usando las dos estructuras.

<pre> CONT ← 1 hacer { escribir CONT CONT ← CONT + 1 } ientras (CONT <= 5) </pre>	<pre> CONT ← 1 ientras (CONT <= 5) { escribir CONT CONT ← CONT + 1 } </pre>
---	--

Debido a que la sentencia MIENTRAS compara la condición del bucle antes de ejecutar el cuerpo del mismo, se llama un bucle **pretest**, la sentencia HACER-MIENTRAS hace lo contrario, por lo tanto, es conocida como un bucle **posttest**.

HACER-MIENTRAS puede utilizarse para implementar un bucle controlado por contador si sabemos de antemano que el bucle siempre se ejecutará al menos una vez.

❖ Ejemplos.

Problema: Hallar la sumatoria de los números enteros pares comprendidos entre 1 y 100.

```

{
SUMA ← 0
NUMERO ← 1
hacer
  {
    si (NUMERO % 2 == 0)
      SUMA ← SUMA + NUMERO
    NUMERO ← NUMERO + 1
  }
  
```

Apunte de Cátedra

```
}  
mientras (NUMERO <= 100)  
escribir "La suma de los números pares de 1 a 100 es: ", SUMA  
}
```

Otra forma de solucionar este problema sería:

```
{  
SUMA ← 0  
NUMERO ← 2  
hacer  
  {  
    SUMA ← SUMA + NUMERO  
    NUMERO ← NUMERO + 2  
  }  
mientras (NUMERO <= 100)  
escribir "La suma de los primeros 100 pares es: ", SUMA  
}
```

Problema: Ingresar una serie de caracteres y contar la cantidad de letras 'A' que aparecen. Termina la serie cuando se ingresa un '*'. Una variante del ejercicio visto anteriormente con MIENTRAS. Compare los resultados.

```
{  
CANTIDAD ← 0  
hacer  
  {  
    escribir "Ingrese un caracter (con '*' termina)"  
    leer LETRA  
    si (LETRA == 'A' | LETRA == 'a')  
      CANTIDAD ← CANTIDAD + 1  
  }  
mientras (LETRA != '*')  
escribir "La cantidad de letras A ingresadas es: ", CANTIDAD  
}
```

Estructura PARA

El bucle PARA repite las sentencias (enunciados) de su interior un número fijo de veces, previamente establecido. No necesita una condición, el bucle PARA inicializa el valor de una variable e incrementa o decrementa *internamente* su valor hasta llegar al valor final. Esta estructura se utiliza cuando el o los enunciados necesitan un número específico de repeticiones y no requieren comprobación intermedia de ninguna condición.

Formato general:

```
para (valor_inicial; condición_de_fin; incremento)  
  enunciado1  
ó  
para (valor_inicial; condición_de_fin; decremento)  
  enunciado1
```

Apunte de Cátedra

Por ejemplo:

```
para (CONT ← 1; CONT <= 100; CONT ← CONT + 1)  
  escribir CONT
```

En este caso CONT incrementará de uno en uno, comenzado con el valor 1 hasta obtener el valor 100. El bucle PARA se ejecutará siempre que la condición final sea verdadera. Si utilizaremos la sentencia MIENTRAS, esto se escribiría:

```
CONT ← 1  
mientras (CONT <= 100)  
{  
  escribir CONT  
  CONT ← CONT + 1  
}
```

También puede utilizarse un bucle PARA que decremente, por ejemplo:

```
para (CONT ← 100; CONT >= 1; CONT ← CONT - 1)  
  escribir CONT
```

En este caso CONT decrementa de uno en uno, comenzado con el valor 100 hasta llegar al valor 1.

Como vemos la sentencia PARA simplifica la escritura de bucles controlados por contador. Pero debemos tener cuidado, estos bucles no son de propósito general, están diseñados exclusivamente como bucles controlados por contador. Para utilizarlos inteligentemente se deben conocer y recordar los siguientes hechos acerca de un bucle PARA:

1. La variable de control del bucle (VCB) no puede cambiarse desde dentro del bucle. Su valor puede utilizarse, pero no cambiarse. Esto es, la VCB puede aparecer en una expresión pero no en la parte izquierda de una sentencia de asignación.
2. La VCB se incrementa o decrementa automáticamente, de acuerdo al incremento o decremento establecido.
3. La VCB queda indefinida al final del bucle. Por lo tanto, no se recomienda utilizar la VCB en una expresión luego de ejecutado el bucle.
4. El bucle se ejecuta con la VCB en su valor inicial, todos los valores intermedios y el valor final. Si el valor inicial es mayor que el valor final (en el caso de un incremento) o el valor inicial es menor que el valor final (en el caso de un decremento), la sentencia PARA no se ejecuta ninguna vez. Inclusive, la cantidad de veces que se ejecute el bucle (0, 1, 2, ... de veces) dependerá también de la condición de fin.
5. No se puede poner una condición de terminación adicional en el bucle. La cabecera debe ser exactamente como el formato general lo indicó anteriormente. La variable es el nombre de la variable de control de bucle, y el valor inicial y la condición de fin pueden ser cualquier expresión válida (variables y/o constantes, y operadores).

❖ Ejemplos.

Problema: Mostrar la tabla de conversión de grados Celsius a grados Fahrenheit para temperaturas que comienzan en -5°C y terminan en 5°C . ($1^{\circ}\text{F} = 9/5 \times \text{C} + 32$)

Apunte de Cátedra

```
{
para (CELSIUS ← -5; CELSIUS <= 5; CELSIUS ← CELSIUS + 1)
    {
        FAHRENHEIT ← 9/5 * CELSIUS + 32
        escribir "La temperatura Celsius: ", CELSIUS, " equivale a Fahrenheit: ", FAHRENHEIT
    }
}
```

Problema: Mostrar la tabla de conversión de grados Celsius a grados Fahrenheit para temperaturas que comienzan en 5° C y terminan en -5° C. ($1^{\circ} F: 9/5 \times C + 32$)

```
{
para (CELSIUS ← 5; CELSIUS >= -5; CELSIUS ← CELSIUS - 1)
    {
        FAHRENHEIT ← 9/5 * CELSIUS + 32
        escribir "La temperatura Celsius: ", CELSIUS, " equivale a Fahrenheit: ", FAHRENHEIT
    }
}
```

Ambos algoritmos hacen lo mismo, sólo que el primero muestra la tabla en forma ascendente y el segundo muestra la tabla en forma descendente.

Criterios para la elección de una sentencia iterativa

1. Si el bucle es un bucle controlado por contador simple, utilizar una sentencia PARA. Si el bucle está controlado por suceso, entonces el bucle PARA no es apropiado. Utilizar en su lugar un bucle MIENTRAS o HACER-MIENTRAS.
2. Si el bucle es un bucle controlado por suceso y el cuerpo del bucle siempre se ejecuta al menos una vez, es apropiado utilizar la sentencia HACER-MIENTRAS.
3. Si el bucle es un bucle controlado por suceso y no se sabe nada acerca de la primera ejecución, utilizar la sentencia MIENTRAS.
4. Si son apropiados los dos bucles MIENTRAS y HACER-MIENTRAS, utilizar el que mejor refleje la semántica del bucle. Esto es, si el problema se ha establecido en términos de cuándo continuar la iteración, utilizar una sentencia MIENTRAS. Si el problema se ha establecido en términos de cuando parar una iteración, utilizar una sentencia HACER-MIENTRAS.
5. Cuando haya duda, utilizar una sentencia MIENTRAS.

Bucles Anidados

Supongamos que un problema requiere calcular la sumatoria de los sueldos que una empresa debe pagar a sus empleados en un mes.

El algoritmo para dicho problema sería:

```
{
escribir "Ingrese la cantidad de empleados"
leer CANTIDAD
CONT ← 1
SUMA ← 0
mientras (CONT <= CANTIDAD)
    {
        escribir "Ingrese un sueldo"
```

Apunte de Cátedra

```
leer SUELDO
SUMA ← SUMA + SUELDO
CONT ← CONT + 1
}
escribir "La sumatoria de los sueldos es: ", SUMA
}
```

Supongamos que el problema se extiende y nos solicitan calcular la sumatoria de los sueldos de los empleados de la empresa de todo el año, teniendo en cuenta que la cantidad de empleados cada mes puede ser diferente al igual que los importes de los sueldos. Básicamente el algoritmo será el mismo, sólo que el proceso debe repetirse 12 veces, ya que se solicita para todo el año.

```
{
MES ← 1
SUMATOTAL ← 0
mientras (MES <= 12)
{
escribir "Ingrese la cantidad de empleados del mes: ", MES
leer CANTIDAD
CONT ← 1
SUMA ← 0
mientras (CONT <= CANTIDAD)
{
escribir "Ingrese un sueldo"
leer SUELDO
SUMA ← SUMA + SUELDO
CONT ← CONT + 1
}
SUMATOTAL ← SUMATOTAL + SUMA
MES ← MES + 1
}
escribir "La sumatoria anual de los sueldos es: ", SUMATOTAL
}
```

Esto es lo que se denomina **bucle anidado**, es decir, que un bucle incluya en su cuerpo a otro bucle. Para diseñar una estructura iterativa anidada, comenzamos diseñando el bucle más exterior como de costumbre. Cuando llegamos al momento en el que es necesario diseñar el proceso que se repite, el bucle más exterior incluirá el bucle anidado. Podemos entonces diseñar el bucle anidado al igual que haríamos con cualquier otro bucle. Este proceso puede ser repetido para cualquier número de niveles de anidamiento.

Identación

El propósito de indentar las sentencias de un programa es dar ayudas visuales al lector y hacer al programa más fácil de corregir. Cuando un programa está adecuadamente indentado, la forma en que se agrupan las sentencias es obvia inmediatamente.

Comparemos los siguientes fragmentos:

Apunte de Cátedra

mientras (CONT <= 10)

```
{  
  leer NUM  
  si (NUM == 0)  
  {  
    CONT ← CONT + 1  
    NUM ← 1  
  }  
  escribir NUM  
  escribir CONT  
}
```

mientras (CONT <= 10)

```
{  
  leer NUM  
  si (NUM == 0)  
  {  
    CONT ← CONT + 1  
    NUM ← 1  
  }  
  escribir NUM  
  escribir CONT  
}
```

Como regla básica hemos de dejar algunos espacios (tabulaciones) como indentación, para delimitar los grupos de sentencias, a efectos de que sea más clara su identificación visual.

Importancia de las llaves ({, })

Observemos el siguiente ejemplo:

```
si (condicion1)  
    enunciado1  
sino  
    enunciado2  
    enunciado3
```

Sabemos que si condicion1 es verdadera se ejecutará el enunciado1, caso contrario se ejecutará el enunciado2, y en ambos casos luego (o en última instancia) se ejecutará el enunciado3.

Supongamos ahora lo siguiente:

```
si (condicion1)  
    enunciado1  
sino  
    enunciado2  
    enunciado3  
    enunciado4
```

En este caso, según como está indentado, creemos que si condicion1 es verdadera se ejecutará el enunciado1 y luego el enunciado4. Lo cual no es cierto. De la manera en que se encuentra escrito, si la condicion1 resulta ser verdadera se ejecutará el enunciado1, luego el enunciado3 y por último, el enunciado4.

Cuando necesitamos que de la rama SI o SINO se ejecuten más de un enunciado, es decir, un conjunto de enunciados (bloque), no nos tenemos que olvidar de abarcarlos o encerrarlos entre llaves ({, }), de esa manera logramos que se ejecute el bloque en completo.

El ejemplo anterior bien escrito sería:

Apunte de Cátedra

```
si (condicion1)
    enunciado1
sino
    {
        enunciado2
        enunciado3
    }
    enunciado4
```

De esta manera, queda bien claro, que el enunciado3 corresponde a la rama *sino*, ya que la indentación no siempre es clara y suficiente. Hay que tener en cuenta que la indentación se aplica a la estética y no al funcionamiento del programa.

Otro ejemplo con la estructura SI:

```
si (condicion1)
    enunciado1
    enunciado2
    enunciado3
```

En este caso, cuando condicion1 sea falsa se ejecutará enunciado2 y luego enunciado3. Para que esto no ocurra, debemos escribir:

```
si (condicion1)
    {
        enunciado1
        enunciado2
    }
    enunciado3
```

De esta manera, cuando condicion1 sea falsa, sólo se ejecutará enunciado3.

También puede ocurrir con la estructura MIENTRAS:

```
mientras (condicion1)
    enunciado1
    enunciado2
    enunciado3
es igual
mientras (condicion1)
    enunciado1
    enunciado2
    enunciado3
```

Aparentemente estos fragmentos de código son distintos pero en realidad hacen lo mismo ya que el alcance de la iteración es de **un** enunciado. Para que tome más de un enunciado debemos encerrarlo entre llaves.

```
mientras (condicion1)
    {
        enunciado1
        enunciado2
        enunciado3
    }
```

Apunte de Cátedra

Exactamente lo mismo puede ocurrir con la estructura PARA:

para (...)		para (...)
enunciado1		enunciado1
enunciado2	es igual	enunciado2
enunciado3		enunciado3

En estos ejemplos, enunciado1 es el único enunciado que se itera, al salir del bucle PARA se ejecutará una vez los otros enunciados. La forma correcta es:

```
para (...  
  {  
  enunciado1  
  enunciado2  
  enunciado3  
  }
```

También se puede hacer lo mismo con la estructura HACER-MIENTRAS.

ESTRUCTURA GENERAL DE UN PROGRAMA

Concepto de Programa

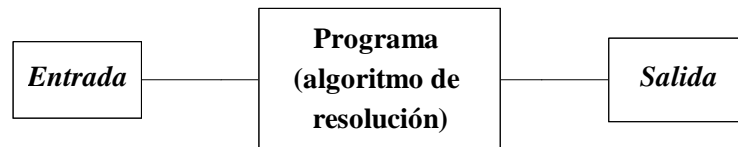
Un *programa de computadora* es un conjunto de instrucciones - órdenes dadas a la máquina - que producirán la ejecución de una determinada tarea. En esencia, *un programa es un medio para conseguir un fin*.

El *proceso de programación* es, por consiguiente, un proceso de solución de problemas y el desarrollo de un programa requiere de las siguientes fases:

- 1.- Definición y Análisis del problema.
- 2.- Diseño de algoritmos.
- 3.- Codificación del programa.
- 4.- Pruebas del programa.
- 5.- Depuración y Verificación del programa.
- 6.- Documentación.
- 7.- Mantenimiento.

Partes Constitutivas de un Programa

Tras la decisión de desarrollar un programa, el programador debe establecer el conjunto de especificaciones que debe contener el programa: *entrada, salida y algoritmos de resolución*, que incluirán las técnicas para obtener las salidas a partir de las entradas.



El algoritmo de resolución o caja negra, en realidad, es el conjunto de códigos que transforman las entradas del programa (datos) en salidas (resultados).

El programador debe establecer de dónde provienen las entradas al programa. Las entradas, en cualquier caso, procederán de dispositivos periféricos de entrada (teclado, mouse, disco, etc.). El proceso de introducir la información de entrada (datos) en la memoria de la computadora se denomina *entrada* de datos, operación de *lectura* o acción de LEER.

Las salidas de datos se deben presentar en dispositivos periféricos de salida (pantalla, impresora, discos, etc.). La operación de salida de datos se conoce también como *escritura* o acción de ESCRIBIR.

Elementos Básicos de un Programa

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para las que esos elementos se combinan. Estas reglas se denominan *sintaxis* del lenguaje. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazadas por la máquina.

Los elementos básicos constitutivos de un programa o algoritmo son:

1. Palabras Reservadas (**leer, escribir, si-sino, mientras**, etc.).
2. Identificadores (nombres de variables, constantes, etc.).
3. Operadores.

Apunte de Cátedra

Estructura General de un Programa en Pseudocódigo

Formato general:

```
principal nombre_del_programa
{
  declaración_de_constantes_y_variables
  ...
  cuerpo_del_programa
  ...
}
```

Declaración de constantes y variables

Antes de poder utilizarse constantes y/o variables en un programa, se las debe declarar en memoria. Ya que ambas no son más que posiciones en memoria y por eso, se debe reservar el lugar que se necesita para manipular el contenido de las mismas.

Formato general:

```
nombre_de_la_constante = valor
tipo_de_la_variable nombre_de_la_variable
```

Por ejemplo:

```
CANTDIAS = 30
PI = 'π'
entero NUMERO
caracter LETRA
```

❖ Ejemplos.

Problema: Suponemos que ya hicimos el análisis y diseño del algoritmo y ahora lo convertimos en un programa. Realizar la suma de diez números ingresados por pantalla.

```
principal SumoNumeros
{
  //declaración de constantes y variables
  CANT=10 //cantidad de números
  real NRO //números ingresados
  entero I //contador de la cantidad de números ingresados
  entero SUM //acumulador de la suma de los números

  //inicialización de variables
  I ← 0
  SUM ← 0

  //inicio del algoritmo
  mientras (I < CANT)
  {
    escribir "Ingrese un número"
    leer NRO
  }
}
```

Apunte de Cátedra

```

SUM ← SUM + NRO
I ← I + 1
}                                     //mientras

//muestra del resultado
escribir "La suma de los ", CANT, " números ingresados es: ", SUM
}                                     //principal

```

Nota: El texto que se encuentra inicializado con doble barra (//), representa a **comentarios** y sirve para describir o documentar ciertas partes del programa. Siempre son de mucha utilidad, ya que informan brevemente lo que está escrito. Y no tienen ningún efecto sobre la ejecución del programa.

Problema: calcular la sumatoria de los sueldos de los empleados de una empresa de todo el año, teniendo en cuenta que la cantidad de empleados cada mes puede ser diferente al igual que los importes de los sueldos.

principal Sueldos

```

{                                     //principal
//declaración de constantes y variables
ULTMES = 12
entero MES, CANTIDAD, CONT
real SUMATOTAL, SUMA, SUELDO

//inicialización de variables
MES ← 1
SUMATOTAL ← 0

//inicio del algoritmo
mientras (MES <= ULTMES)
{                                     //mientras de meses
escribir "Ingrese la cantidad de empleados del mes: ", MES
leer CANTIDAD
CONT ← 1
SUMA ← 0
mientras (CONT <= CANTIDAD)
{                                     //mientras de empleados
escribir "Ingrese un sueldo"
leer SUELDO
SUMA ← SUMA + SUELDO
CONT ← CONT + 1
}                                     //mientras de empleados
SUMATOTAL ← SUMATOTAL + SUMA
MES ← MES + 1
}                                     //mientras de meses

//muestra del resultado
escribir "La sumatoria anual de los sueldos es: ", SUMATOTAL
}                                     //principal

```

Apunte de Cátedra

Uso de variables

Una variable antes de ser utilizada (Lectura, Escritura o Asignación) debe ser declarada. Esto significa que debe darse a conocer el tipo de dato al que corresponde.

Por ejemplo:

principal PotenciaCuadrada

```
{  
  POT ← NUM ^ 2  
  escribir "El cuadrado de ", NUM, "es: ", POT  
}
```

principal PotenciaCuadrada

```
{  
  entero NUM, POT  
  POT ← NUM ^ 2  
  escribir "El cuadrado de ", NUM, "es: ", POT  
}
```

Como vemos, en el primer caso se usaron las variables NUM y POT, sin previa declaración. Lo cual es un error para el programa. La solución se encuentra en el segundo caso, donde las variables se declararon previamente. Pero el algoritmo no está completo:

Antes de utilizar una variable a la derecha de una operación de asignación u operación de escritura, la misma debe inicializarse mediante una operación de asignación u operación de lectura.

Por ejemplo:

principal PotenciaCuadrada

```
{  
  entero NUM, POT  
  POT ← NUM ^ 2  
  escribir "El cuadrado de ", NUM, "es: ", POT  
}
```

principal PotenciaCuadrada

```
{  
  entero NUM, POT  
  leer NUM  
  POT ← NUM ^ 2  
  escribir "El cuadrado de ", NUM, "es: ", POT  
}
```

Como vemos, el primer caso es incorrecto, puesto que para calcular POT es necesario que la variable NUM tenga algún valor, es decir, previo al cálculo, la variable no ha sido inicializada. Caso contrario ocurre en el segundo caso, donde la variable NUM, obtiene un valor a partir de la operación de lectura.

Pruebas del programa

Existen muchas técnicas para probar programas, que no se estudiaran en esta materia, pero sí es necesario comprobar de alguna manera el programa que se ha construido. Por el momento se utilizará lo que se conoce como **Prueba de Escritorio** o **Traza**. Consiste en hacer un seguimiento de las variables que están en juego en el programa, e ir determinando si el resultado de las mismas corresponde o no al objetivo del programa. La manera de representarlo es totalmente a elección del programador, por medio de una tabla, casilleros, etc. Es una manera efectiva de corroborar el funcionamiento del programa, desde ya que una traza bien hecha influye un 100% en determinar si el programa funciona o no.

Apunte de Cátedra

ARREGLOS

Hasta ahora hemos visto **datos de tipo simple**: entero, real, caracter y booleano. Todos estos datos de tipo simple tienen una característica común, que cada variable representa un dato individual.

Los **datos de tipo compuestos u objetos** son arreglos, cadenas, archivos, pilas, colas, listas, árboles, etc. Todos ellos tienen una característica en común: un único identificador puede representar a múltiples datos individuales, pudiendo cada uno de éstos ser referenciado separadamente. Así, los datos de tipo compuesto pueden ser manipulados colectiva o individualmente, dependiendo de las necesidades de cada aplicación particular.

Un **arreglo** es un conjunto finito, ordenado e indexado de elementos con las siguientes características:

- todos los elementos son del mismo tipo, esto hace que sea un tipo de dato **homogéneo**;
- los elementos pueden recuperarse en cualquier orden, simplemente indicando la posición que ocupa dentro de la estructura, por este motivo es una estructura **indexada**;
- la memoria ocupada a lo largo de la ejecución del programa es fija, por esto es una estructura de datos **estática**.

Existen varios tipos de arreglos (de acuerdo a su dimensión), en esta materia se estudiarán:

- **Arreglos Unidimensionales (Vectores).**
- **Arreglos Bidimensionales (Matrices).**

Arreglos Unidimensionales: Vectores

El tipo más simple de arreglo es el *arreglo unidimensional o arreglo lineal o vector* (arreglo de una dimensión). El *subíndice* o *índice* de un elemento [0, 1, 2, ..., n-1] designa su posición en la ordenación del vector.

Gráficamente, se puede representar de la siguiente forma:

elemento 1	elemento 2	elemento 3	...	elemento n-1	elemento n
posición 0	posición 1	posición 2	...	posición n-2	posición n-1

En un vector, la manera de hacer referencia a alguno de sus elementos es a través del nombre del arreglo y la posición del elemento (índice), de la forma:

nombre_del_vector[número_de_posición]

Por ejemplo: LISTA es el nombre de un arreglo que contiene 10 números enteros.

3	-2	1	10	2	5	1	6	7	-8
LISTA[0]	LISTA[1]	LISTA[2]	LISTA[3]	LISTA[4]	LISTA[5]	LISTA[6]	LISTA[7]	LISTA[8]	LISTA[9]

Esto significa que en la primer posición (0) de LISTA se halla almacenado el valor 3, en la segunda posición el valor -2, en la tercera posición el valor 1, ... y en la novena posición el valor -8.

Entonces, el elemento de cada posición se obtiene de la siguiente manera:

Apunte de Cátedra

LISTA[0]: 3
LISTA[1]: -2
LISTA[2]: 1
LISTA[3]: 10
LISTA[4]: 2
LISTA[5]: 5
LISTA[6]: 1
LISTA[7]: 6
LISTA[8]: 7
LISTA[9]: -8

Declaración y creación de un arreglo unidimensional (Vector)

Formato general:

Declaración:

tipo_de_dato [] nombre_del_vector

Creación:

nombre_del_vector = **nuevo** tipo_de_dato[dimensión_del_vector]

Por ejemplo:

entero [] LISTA

LISTA = **nuevo** entero[10]

real [] VECTOR

VECTOR = **nuevo** real[25]

En estos ejemplos hemos definido a la variable LISTA como un arreglo de 10 elementos enteros y a VECTOR como un arreglo de 25 elementos reales. Otra forma de declarar un vector, es por medio de constantes:

TOPE = 100

entero [] VECTOR

VECTOR = **nuevo** entero[TOPE]

Índices (posición o subíndice)

El índice de un arreglo es el valor que nos permite diferenciar los distintos elementos individuales del mismo. Como ya hemos visto el índice se indica entre corchetes luego del nombre de la variable del vector: VECTOR[1] o LISTA[5] (los índices son 1 y 5 respectivamente).

La primera posición de un vector siempre es 0. Por lo tanto, si el vector se ha declarado de dimensión 100, los índices están comprendidos entre 0 y 99. Si se hace una referencia a un valor mayor que 99, se produce un error en el programa.

En general, el índice puede ser una constante, una variable o una expresión aritmética. Si el índice es una expresión aritmética ésta se evaluará a efectos de obtener el valor del índice.

Veamos el siguiente ejemplo, según el vector LISTA (anteriormente mencionado) y la variable I cuyo valor es 5, podemos hacer las siguientes referencias:

LISTA[I]	Se refiere al 6to. elemento de LISTA cuyo valor es 5.
LISTA[I + 2]	Se refiere al 8vo. elemento de LISTA cuyo valor es 8.
LISTA[I - 1]	Se refiere al 5to. elemento de LISTA cuyo valor es 2.
LISTA[I / 2]	Se refiere al 3ro. elemento de LISTA cuyo valor es 1.

Todas estas referencias son válidas. Pero si hacemos: LISTA[I * 3], nos da un índice igual a 15, lo cual no es válido puesto que LISTA no posee un elemento LISTA[15]. Lo mismo ocurre si hacemos

Apunte de Cátedra

LISTA[I - 10], nos da un índice igual a -5, tampoco es válido, ya que LISTA no posee posiciones negativas.

Manejo de vectores

Una vez que hemos definido el vector, debemos guardar información en él, puesto que al principio el vector esta vacío o con “basura”. Hay varias formas de almacenar información en un vector. En este caso, veremos la carga para un vector.

1. A través de la asignación de datos:

```
LISTA[0] ← 3
LISTA[1] ← -2
LISTA[2] ← 1
LISTA[3] ← 10
LISTA[4] ← 2
LISTA[5] ← 5
LISTA[6] ← 1
LISTA[7] ← 6
LISTA[8] ← 7
LISTA[9] ← -8
```

Estamos asignando valores enteros a cada posición del vector.

2. A través de la lectura de los datos:

```
leer LISTA[0]
leer LISTA[1]
leer LISTA[2]
leer LISTA[3]
leer LISTA[4]
leer LISTA[5]
leer LISTA[6]
leer LISTA[7]
leer LISTA[8]
leer LISTA[9]
```

En este caso la lectura de los datos se almacena directamente en las distintas posiciones del vector. Podemos cargar un elemento al vector, algunos o cargar el vector de forma completo. El problema con esta última forma es que si nuestro vector tiene una dimensión grande, por ejemplo 1000 posiciones, escribir la lectura de cada posición del vector requiere una extensa cantidad de líneas que hacen lo mismo. Lo cual es innecesario. Generalmente, para solucionar este problema se utiliza un bucle que permite realizar la carga de un vector, de forma más rápida y sencilla.

Veamos:

```
para (POS ← 0; POS <10; POS ← POS + 1)
{
  escribir “Ingrese un valor en la posición ”, POS
  leer LISTA[POS]
}
```

Apunte de Cátedra

Donde POS es una variable que sirve para controlar el bucle y como índice del vector. Al utilizar una variable como índice logramos acceder a las distintas posiciones del vector.

De la misma manera que el vector ha sido cargado puede mostrarse uno, alguno o todos los elementos del mismo.

Veamos:

```
para (POS ← 0; POS <10; POS ← POS + 1)
  escribir LISTA[POS]
```

Tanto la carga como la muestra del vector, son actividades constantes en el uso de los mismos. Pero, existen otras operaciones que varían de acuerdo al problema a desarrollar. Por ejemplo, una vez cargado el vector, se puede hallar la suma de sus elementos. En este caso, procedemos de la misma manera, con un bucle cuya variable contador nos sirve además como índice del vector.

Veamos:

```
SUMA ← 0
para (POS ← 0; POS <10; POS ← POS + 1)
  SUMA ← SUMA + LISTA[POS]
escribir "La sumatoria de los elementos del vector es: ", SUMA
```

En este caso, con el bucle vamos recorriendo el vector, accediendo a cada elemento del mismo mediante el índice, e incrementado a la variable SUMA.

❖ Ejemplos.

Problema: Calcular la media de las estaturas de una clase de alumnos y deducir cuántos son más altos que la media y cuantos más bajos que dicha media.

```
principal EstaturasDeLaClase
{
  //declaración de variables
  entero ESTU //número de estudiantes de la clase
  real [ ] ESTATURAS
  ESTATURAS = nuevo real [100] //estatura de los n alumnos
  entero CONT //contador de alumnos
  real MEDIA //media de estaturas
  entero ALTOS //alumnos de estatura mayor que la media
  entero BAJOS //alumnos de estatura menor que la media
  real SUMA //totalizador de estaturas

  //carga de datos
  escribir "Ingrese la cantidad de estudiantes en la clase"
  leer ESTU

  //operación de suma
  CONT ← 0
  SUMA ← 0
  mientras (CONT < ESTU)
  {
```


Apunte de Cátedra

```
escribir "Ingrese la estatura del alumno ", CONT
leer ESTATURAS[CONT]
SUMA ← SUMA + ESTATURAS[CONT]
CONT ← CONT + 1
}
```

```
//calculo de la media
MEDIA ← SUMA / ESTU
ALTOS ← 0
BAJOS ← 0
para (CONT ← 0; CONT < ESTU; CONT ← CONT + 1)
{
  si (ESTATURAS[CONT] < MEDIA)
    BAJOS ← BAJOS + 1
  sino
    si (ESTATURAS[CONT] > MEDIA)
      ALTOS ← ALTOS + 1
}
```

```
//muestra de resultados
escribir "La media de las estaturas es: ", MEDIA
escribir "La cantidad de alumnos de estatura mayor que la media es de: ", ALTOS
escribir "La cantidad de alumnos de estatura menor que la media es de: ", BAJOS
} //principal
```

Problema: Generar una lista de enteros, reemplazar los múltiplos de 3 por 3 y mostrar la lista resultante.

principal MultiplosDeTres

```
{ //principal
//declaración de las variables
entero DIM //dimensión de la lista
entero [ ] LISTA
LISTA = nuevo entero[100] //Lista de enteros
entero CONT //contador de bucles

//carga de la dimensión y los datos
escribir "Ingrese la cantidad de elementos a cargar"
leer DIM
para (CONT ← 0; CONT < DIM; CONT ← CONT + 1)
{
  escribir "Ingrese el elemento", CONT
  leer LISTA[CONT]
}

//operación de múltiplo
CONT ← 0
hacer
{
```

Apunte de Cátedra

```
    si (LISTA[CONT] % 3 == 0)
        LISTA[CONT] ← 3
        CONT ← CONT + 1
    }
mientras (CONT < DIM)

//muestra de resultados
escribir "El vector resultante es"
CONT ← 0
mientras (CONT < DIM)
    {
        escribir LISTA[CONT]
        CONT ← CONT + 1
    }
} //principal
```

Nótese que en ambos ejemplos, no se usó como tamaño del vector el valor declarado al principio (100), sino que se pidió una cantidad y se trabajó con ella. La diferencia está en cuánto voy a reservar de memoria para guardar mi vector y cuánto puedo de esa cantidad, usar en realidad. Queda a cargo del Lector. Tenga en cuenta, qué sucedería si la dimensión ingresada es un valor negativo o mayor que el valor declarado como tamaño real del vector. Busque la solución para ello.

Recomendación: cada operación que se realice con un vector, escribirlo de forma separada. No mezclar en una misma iteración la carga, con alguna operación sobre el vector y la muestra del mismo.

Operaciones de Ordenación y de Búsqueda

Las computadoras emplean una gran parte de su tiempo en operaciones de búsqueda y clasificación de datos. Las operaciones de cálculo numérico, y sobre todo de gestión, requieren normalmente operaciones de clasificación de los datos. Por ejemplo, ordenar fichas de clientes por orden alfabético, por direcciones o por código postal.

Ordenación

Existen dos tipos de ordenación: **ordenación interna** (de arreglos), y **ordenación externa** (de archivos). Los *arreglos* se almacenan en memoria interna o central, de acceso aleatorio y directo, y por ello su gestión es rápida. Los *archivos* se sitúan en dispositivos de almacenamiento externo que son más lentos y basados en dispositivos mecánicos: cintas y discos.

La ordenación (clasificación) es una operación tan frecuente en programas de computadora que gran cantidad de algoritmos se ha diseñado para clasificar listas de elementos con eficacia y rapidez. La elección de un determinado algoritmo depende del tamaño del vector a clasificar, el tipo de datos y la cantidad de memoria disponible.

La **ordenación o clasificación** es un proceso de organizar datos en algún orden o secuencia específica, tal como creciente o decreciente para datos numéricos o para datos de caracteres, en orden alfabético.

Los algoritmos de ordenación se clasifican en directos o básicos y en avanzados. Estos últimos son más eficientes y más rápidos pero más complejos. En cambio los métodos directos son sencillos, aunque ineficientes para listas de gran tamaño.

Los métodos directos más populares son:

- **Ordenación por Inserción.**
- **Ordenación por Selección.**
- **Ordenación por Intercambio (Burbuja).**

Ordenación por Inserción

Este método consiste en insertar un elemento en una parte ya ordenada de este vector y comenzar de nuevo con los elementos restantes. Por ser utilizado generalmente por los jugadores de cartas se lo conoce también por el nombre de *método de la baraja*.

El método se basa en comparaciones y desplazamientos sucesivos. El algoritmo de clasificación de un vector X para N elementos, se realiza con un recorrido de todo el vector y la inserción del elemento correspondiente en el lugar adecuado. El recorrido se realiza desde el segundo elemento al n-ésimo elemento.

Algoritmo

```
para (I ← 1; I < N; I ← I + 1)
{
  AUX ← VECTOR[I]
  J ← I - 1                                     //apunta al primer elemento a comparar
  mientras (J >= 0 & VECTOR[J] > AUX)
  {
    VECTOR[J + 1] ← VECTOR[J]                 //se corre 1 lugar a la derecha
    J ← J - 1
  }
  VECTOR[J + 1] ← AUX                          //inserta el valor auxiliar
}
```

Apunte de Cátedra

Veamos el siguiente ejemplo, aplicando inserción:

	22	25	06	20	42	09	03	33
I: 1	22	25	06	20	42	09	03	33
I: 2	06	22	25	20	42	09	03	33
I: 3	06	20	22	25	42	09	03	33
I: 4	06	20	22	25	42	09	03	33
I: 5	06	09	20	22	25	42	03	33
I: 6	03	06	09	20	22	25	42	33
I: 7	03	06	09	20	22	25	33	42

El número de movimientos y comparaciones mínimo (mejor caso) se presenta cuando los elementos están ordenados inicialmente, el máximo (peor caso) se presenta si están inicialmente en orden inverso. Es evidente que el algoritmo no modifica el orden de los elementos con el mismo valor.

Ordenación por Selección

La idea esencial de este método es: si tuviéramos en una hoja de papel una columna con 10 números para ordenar en forma ascendente, probablemente haríamos lo siguiente:

1. Buscar el elemento más pequeño.
2. Escribirlo en el papel en una segunda columna.
3. Tachar el número de la lista original.
4. Repetir el proceso, siempre buscando el número más pequeño que queda de la lista original.
5. Parar cuando todos los números hubieran sido tachados de la lista original.

Pero esta idea tiene dos puntos complicados: hablar de una segunda columna implica utilizar un segundo arreglo, lo cual puede ser crítico (si el arreglo original es muy grande, entonces requiere mucha memoria - el doble -); tachar una componente de un arreglo es una tarea difícil. Pero estos dos inconvenientes, tienen solución haciendo una pequeña variación en la idea original; para no mover las componentes a un segundo arreglo y tacharlo de la lista original, podemos poner el valor en el lugar que le corresponda en el arreglo original intercambiando su lugar con la componente que hubiera allí. Finalmente lo que hará este método es buscar el elemento de menor valor del vector y colocarlo en la primera posición. Luego se busca el segundo elemento más pequeño y se coloca en la segunda posición, y así sucesivamente.

Los pasos sucesivos a dar son:

- 1.- Seleccionar el elemento menor del vector de n elementos.
- 2.- Intercambiar dicho elemento con el primero.
- 3.- Repetir estas operaciones con los $n-1$ elementos restantes, seleccionando el segundo elemento; continuar con los $n-2$ elementos restantes hasta que sólo quede el mayor.

Apunte de Cátedra

Algoritmo

```

para (I ← 0; I < N - 1; I ← I + 1)
{
  //busca el mínimo VECTOR[P] entre VECTOR[I], ..., VECTOR[N]
  P ← I
  para (J ← I + 1; J < N; J ← J + 1)
    si (VECTOR[J] < VECTOR[P])
      P ← J
  //intercambia VECTOR[I] y VECTOR[P]
  AUX ← VECTOR[P]
  VECTOR[P] ← VECTOR[I]
  VECTOR[I] ← AUX
  //ahora VECTOR[0] <= ... <= VECTOR[I] y además, VECTOR[I] <= VECTOR[J] con I < J < N
}

```

Es evidente que el número de comparaciones entre los elementos es independiente de la ordenación inicial de éstos.

En general el algoritmo de selección es preferible al de inserción, aunque en los casos en que los elementos están inicialmente ordenados o casi ordenados, éste último puede ser algo más rápido (mejor caso).

Veamos el siguiente ejemplo aplicando selección:

320	96	16	90	120	80	200	64
-----	----	----	----	-----	----	-----	----

El método comienza buscando el número más pequeño.

La lista nueva será:

16	96	320	90	120	80	200	64
-----------	----	-----	----	-----	----	-----	----

A continuación, se busca el siguiente número más pequeño, 64, y se realizan las operaciones 1 y 2.

La nueva lista sería:

16	64	320	90	120	80	200	96
----	-----------	-----	----	-----	----	-----	----

Si se siguen realizando las iteraciones se encontrarán las siguientes líneas:

16	64	80	90	120	320	200	96
----	----	-----------	----	-----	-----	-----	----

16	64	80	90	120	320	200	96
----	----	----	-----------	-----	-----	-----	----

16	64	80	90	96	320	200	120
----	----	----	----	-----------	-----	-----	-----

16	64	80	90	96	120	200	320
----	----	----	----	----	------------	-----	-----

16	64	80	90	96	120	200	320
----	----	----	----	----	-----	------------	-----

Apunte de Cátedra

Ordenación por Intercambio

El algoritmo de clasificación de *intercambio o burbuja* se basa en el principio de comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados.

Los pasos a dar son:

- 1.- Comparar los elementos del primer y segundo lugar (subíndice 0 y 1), si están en orden, se mantienen como están; en caso contrario se intercambian entre sí.
- 2.- A continuación se comparan los elementos del segundo y tercer lugar (subíndice 1 y 2); de nuevo se intercambian si es necesario.
- 3.- El proceso continúa hasta que cada elemento del vector ha sido comparado con sus elementos adyacentes y se han realizado los intercambios necesarios.

A este método se lo conoce como *burbuja* porque el elemento cuyo valor es más pequeño, sube posición a posición hacia el comienzo de la lista al igual que las burbujas de aire en un depósito o botella de agua. O podría verse también como que el elemento más grande se arrastra hacia el final del vector. Cualquier alternativa es válida. En el segundo recorrido, el segundo elemento más pequeño llegará a la segunda posición, o el segundo elemento más grande se arrastrará a la penúltima posición (n - 2), y así sucesivamente.

Algoritmo

```

para (I ← N - 1; I >= 1; I ← I - 1)
    //poner el mayor elemento de VECTOR[0], ..., VECTOR[I] en VECTOR[I]
    para (J ← 0; J <= I - 1; J ← J + 1)
        si (VECTOR[J] > VECTOR[J + 1])
            {
                //intercambiar elementos
                AUX ← VECTOR[J]
                VECTOR[J] ← VECTOR[J + 1]
                VECTOR[J + 1] ← AUX
            }
    //ahora VECTOR[I] <= ... <= VECTOR[N] y VECTOR[J] <= VECTOR[I] con 0 <= J < I

```

Veamos el siguiente ejemplo aplicando burbuja:

72	64	50	23	84	18	37	99	45	08
----	----	----	----	----	----	----	----	----	----

Se muestran los pasos necesarios de la ordenación por burbuja:

I: 9	64	50	23	72	18	37	84	45	08	99
------	----	----	----	----	----	----	----	----	-----------	----

I: 8	50	23	64	18	37	72	45	08	84	99
------	----	----	----	----	----	----	----	-----------	----	----

I: 7	23	50	18	37	64	45	08	72	84	99
------	----	----	----	----	----	----	-----------	----	----	----

I: 6	23	18	37	50	45	08	64	72	84	99
------	----	----	----	----	----	-----------	----	----	----	----

I: 5	18	23	37	45	08	50	64	72	84	99
------	----	----	----	----	-----------	----	----	----	----	----

I: 4	18	23	37	08	45	50	64	72	84	99
------	----	----	----	-----------	----	----	----	----	----	----

I: 3	18	23	08	37	45	50	64	72	84	99
------	----	----	-----------	----	----	----	----	----	----	----

Apunte de Cátedra

I : 2

18	08	23	37	45	50	64	72	84	99
----	-----------	----	----	----	----	----	----	----	----

I : 1

08	18	23	37	45	50	64	72	84	99
-----------	----	----	----	----	----	----	----	----	----

Se puede realizar una mejora en la velocidad de ejecución del algoritmo. Observemos que en el primer recorrido del vector (cuando I tiene el valor 9) el valor mayor del vector se mueve al último elemento VECTOR[N - 1]. Por consiguiente, en el próximo paso no es necesario comparar VECTOR[N - 2] y VECTOR[N - 1]. En otras palabras, el límite superior del bucle PARA puede ser N - 2. Después de cada paso se puede decrementar en uno el límite superior del bucle PARA.

A pesar de las mejoras que se puedan realizar, la operación de intercambio de dos elementos generalmente es mucho más costosa que una operación de comparación. Por lo tanto, la ordenación por intercambio es inferior a la ordenación por inserción o por selección.

Búsqueda

La búsqueda es un proceso de ubicar información particular en una colección de datos. La manera de buscar la información depende de la forma en que los datos de la colección se encuentren organizados. Los métodos de búsqueda consisten en buscar un dato que tiene una propiedad particular (por ejemplo una estructura con un componente igual a x) y en caso de encontrarlo, puede retornar afirmativo o retornar información relacionada, por ejemplo: su posición en la estructura.

Los métodos más conocidos son:

- **Búsqueda Secuencial o Lineal.**
- **Búsqueda Binaria.**

Búsqueda Secuencial

El método más sencillo de buscar un elemento en un vector es explorar secuencialmente el vector, o dicho en otras palabras, *recorrer el vector* desde el primer elemento hasta el último, linealmente. Se compara cada elemento del vector con un valor deseado (elemento a buscar) hasta que se encuentra o se termina de recorrer el vector completo. Un resultado posible es: si se encuentra el elemento buscado visualizar un mensaje similar a "Elemento encontrado", en caso contrario visualizar un mensaje similar a "Elemento no existente".

Algoritmo

```

I ← 0
ENCONTRADO ← Falso
mientras (ENCONTRADO == Falso & I < N)
{
    si (VECTOR[I] == BUSCADO)
    {
        ENCONTRADO ← Verdadero
        POSICION ← I
    }
    I ← I + 1
}
si (ENCONTRADO == Verdadero)
    escribir "El elemento buscado se encuentra en la posición ", POSICION
sino
    escribir "El elemento buscado NO se encuentra en el vector"

```

Apunte de Cátedra

El mejor caso en una búsqueda secuencial es cuando el elemento buscado se encuentra en la primera posición y el peor caso es cuando el elemento buscado no se encuentra en el vector o se encuentra en la última posición (es decir, es el último elemento). Aunque éste puede ser un método adecuado para pocos datos, se necesita una técnica más eficaz para conjuntos grandes de datos. Si el número de elementos en el vector es grande, el algoritmo se hace muy lento en tiempo, de un modo considerable. Por ende, la búsqueda secuencial o lineal no es el método más eficiente para vectores con un gran número de elementos.

Búsqueda Binaria

Para mejorar el tiempo de la búsqueda lineal, existe el método de búsqueda binaria, que accede al vector en mitades. Por ejemplo, si tuviéramos que consultar un abonado en la guía telefónica, normalmente no buscamos el nombre en orden secuencial, sino que buscamos en la primera o segunda mitad de la guía; una vez escogida esa mitad, volvemos a tantear una de sus dos submitades, y así sucesivamente repetimos el proceso hasta que se localiza la página correcta, con el nombre buscado. Naturalmente, las personas que realizan este tipo de búsquedas nunca utilizarían un método de búsqueda secuencial, sino un método que se basa en la división sucesiva del espacio ocupado por el vector en sucesivas mitades, hasta encontrar el elemento buscado. La búsqueda binaria utiliza el método de “*divide y vencerás*”, para localizar el valor deseado. Con este método se examina primero el elemento central de la lista; si éste es el elemento buscado, entonces la búsqueda ha terminado. En caso contrario, se determina si el elemento buscado está en la primera o en la segunda mitad de la lista y a continuación se repite este proceso, utilizando el elemento central de esa sublista. Para poder realizar esta operación la búsqueda binaria necesita como precondition que los elementos se encuentren clasificados en un determinado orden, caso contrario, no se podrá aplicar el método mencionado.

Algoritmo

```
ENCONTRADO ← Falso
PRIMERO ← 0 //límite inferior del intervalo de búsqueda
ULTIMO ← N - 1 //límite superior del intervalo de búsqueda
mientras (PRIMERO <= ULTIMO & ENCONTRADO == Falso)
{
  //límite central del intervalo de búsqueda
  PTOMEDIO ← (PRIMERO + ULTIMO) / 2
  si (VECTOR[PTOMEDIO] == BUSCADO)
    ENCONTRADO ← Verdadero
  sino
    si (VECTOR[PTOMEDIO] > BUSCADO)
      ULTIMO ← PTOMEDIO - 1
    sino
      PRIMERO ← PTOMEDIO + 1
}
si (ENCONTRADO == Verdadero)
  escribir “El elemento buscado se encuentra en la posición ”, PTOMEDIO
sino
  escribir “El elemento buscado NO se encuentra en el vector”
```


Apunte de Cátedra

Supongamos que se tiene la siguiente lista de elementos:

1	3	5	14	92	98	983	2005	2446	2685	3200
-----primera sublista-----					-----segunda sublista-----					

Si se está buscando el elemento 983, se examina el número central 98 en la quinta posición. Y como 983 es mayor que 98, se desprecia la primera sublista y nos centramos en la segunda:

1	3	5	14	92	98	983	2005	2446	2685	3200
						--1° sublista--		--2° sublista--		

El número central de esa sublista es 2446 y el elemento buscado es 983, por ende, es menor que 2446; entonces se elimina (se desprecia) la segunda sublista y nos queda:

1	3	5	14	92	98	983	2005	2446	2685	3200

Como no hay elemento central, elegimos el término inmediatamente anterior al término central, 983, que es el buscado. Se han necesitado tres comparaciones, mientras que la búsqueda secuencial hubiese necesitado siete.

La búsqueda binaria se utiliza en vectores ordenados (orden creciente si los datos son numéricos y orden alfabéticamente si son caracteres) y se basa en la constante división del espacio de búsqueda (acceso del vector). Como se ha comentado, se comienza comparando el elemento que se busca, no con el primer elemento, sino con el elemento central. Si el elemento buscado (x) es menor que el elemento central, entonces x deberá estar en la mitad izquierda o inferior del vector; si es mayor que el valor central, deberá estar en la mitad derecha o superior, y si es igual al valor central, se habrá encontrado el elemento buscado.

Si el elemento buscado se encuentra en la posición central en la primera vuelta, estamos en presencia del mejor caso. Y si el elemento no se encuentra o se encuentra en la última sublista generada se presenta el peor caso.

Apunte de Cátedra

la fila **I+1** y la columna **J+1**. Estos subíndices son tipos de datos ordinales, como lo son los *Enteros* o *Caracteres*.

Por ejemplo se podría referenciar al elemento que se encuentra en la posición [3][3] si se usa las variables I y J, cuyo contenido en ese momento es de 3 cada una, o sea I:3, J:3, entonces todas las siguientes referencias son iguales e indican el contenido de la celda [3][3], veamos:

T[I][J]: T[I][I]: T[J][I]: T[J][J]: T[3][I]: T[I][3]: T[J][3]: T[3][J]: T[3][3]

Ahora sí I: 3 y J: 6, entonces:

T[I][J]	: referencia al elemento ubicado en la fila <u>4</u> y en la columna <u>7</u> .
T[J][I]	: referencia al elemento ubicado en la fila <u>7</u> y en la columna <u>4</u> .
T[I][I + 7]	: referencia al elemento ubicado en la fila <u>4</u> y en la columna <u>11</u> .
T[J][8]	: referencia al elemento ubicado en la fila <u>7</u> y en la columna <u>9</u> .

La matriz T se dice que tiene M por N elementos. Existen M elementos en cada fila y N elementos en cada columna, por lo tanto, la matriz tiene a lo sumo (M * N) elementos. Si M es igual a N, se denomina Matriz Cuadrada, caso contrario, es una Matriz No Cuadrada.

Declaración y creación de un arreglo bidimensional (Matriz)

Formato general:

Declaración:

tipo_de_dato [] [] nombre_de_la_matriz

Creación:

nombre_de_la_matriz = **nuevo** tipo_de_dato[cantidad_de_filas][cantidad_de_columnas]

Por ejemplo:

entero [][] MATRIZ	real [][] TABLA
MATRIZ = nuevo entero[10][8]	TABLA = nuevo real[100][100]

Otra forma de declarar una matriz, es por medio de constantes:

FIL = 10

COL = 8

entero [][] MATRIZ

MATRIZ = **nuevo** entero [FIL][COL]

Manejo de matrices

Una matriz es un arreglo de dos dimensiones, por lo tanto, las operaciones son similares que las de un vector, sólo que la diferencia se encuentra en cómo manejar los dos índice.

Veamos como sería cargar una matriz de 10 x 10, en forma completa:

```
para (FILA ← 0; FILA <10; FILA ← FILA + 1)
  para (COLUMNA ← 0; COLUMNA <10; COLUMNA ← COLUMNA + 1)
  {
    escribir "Ingrese un valor en la posición ", FILA, " - ", COLUMNA
    leer MATRIZ[FILA][COLUMNA]
  }
```

Apunte de Cátedra

En este caso, ha sido necesario recorrer las filas y las columnas (de izquierda a derecha y de arriba hacia abajo) a través de dos bucles anidados. La variable FILA, controla el primer bucle y representa las filas, y la variable COLUMNA, controla el segundo bucle y representa las columnas.

❖ Ejemplos.

Problema: Generar e imprimir una matriz de números enteros.

principal ImprimirMatrizEnteros

```
{
//declaraciones
FILA = 50
COLUMNA = 50
entero [ ][ ] MATRIZ
MATRIZ = nuevo entero [FILA][COLUMNA]
entero FILAS, COLS, I, J //número de filas y número de columnas

//carga de dimensiones
escribir "Ingrese cantidad de filas"
leer FILAS
escribir "Ingrese cantidad de columnas"
leer COLS

//carga de datos
escribir "Carga de los elementos a la matriz"
para (I ← 0; I < FILAS; I ← I + 1)
{
escribir "Ingrese dato para la fila: ", I
para (J ← 0; J < COLS; J ← J + 1)
leer MATRIZ[I][J]
}

//muestra de datos
escribir "La matriz resultante es la siguiente"
para (I ← 0; I < FILAS; I ← I + 1)
para (J ← 0; J < COLS; J ← J + 1)
escribir MATRIZ[I][J]
} //principal
```

Problema: Generar dos matrices de enteros, calcular la suma de ambas (elemento a elemento) y escribir la matriz de sumas resultante. Para resolver este problema primero cargamos dos matrices, luego generamos la suma de éstas en una tercera matriz y por último mostramos los resultados.

principal DosMatrices

```
{
//declaraciones
entero [ ][ ] MAT1, MAT2, MAT3
MAT1 = nuevo entero [20][20]
MAT2 = nuevo entero [20][20]
MAT3 = nuevo entero [20][20] //MAT3 es la matriz de resultado
} //principal
```

Apunte de Cátedra

```

entero FILAS, COLS, I, J           //número de filas y número de columnas

//carga de dimensiones
escribir "Ingrese cantidad de filas"
leer FILAS
escribir "Ingrese cantidad de columnas"
leer COLS

//carga de la primera matriz
escribir "Carga de los elementos de la primera matriz"
para (I ← 0; I < FILAS; I ← I + 1)
  {
    escribir "Ingrese dato para la fila: ", I
    para (J ← 0; J < COLS; J ← J + 1)
      leer MAT1[I][J]
  }

//carga de la segunda matriz
escribir "Carga de los elementos de la segunda matriz"
para (I ← 0; I < FILAS; I ← I + 1)
  {
    escribir "Ingrese dato para la fila: ", I
    para (J ← 0; J < COLS; J ← J + 1)
      leer MAT2[I][J]
  }

//suma de MAT1 y MAT2
para (I ← 0; I < FILAS; I ← I + 1)
  para (J ← 0; J < COLS; J ← J + 1)
    MAT3[I][J] ← MAT1[I][J] + MAT2[I][J]

//muestra de los datos de MAT3
escribir "La suma de las dos primeras matrices es la siguiente"
para (I ← 0; I < FILAS; I ← I + 1)
  para (J ← 0; J < COLS; J ← J + 1)
    escribir MAT3[I][J]
  }
  //principal
  
```

Problema: Dada una matriz de números enteros imprimir los elementos de la 3er. columna.

```

principal ImprimirMatrizColumnaTres
  {
    //declaraciones
    TOPE = 50
    entero [ ][ ] MATRIZ
    MATRIZ = nuevo entero [TOPE][TOPE]
    entero FILAS, COLS, I, J

    //carga de dimensiones
    escribir "Ingrese cantidad de filas"
  }
  
```

Apunte de Cátedra

leer FILAS
escribir "Ingrese cantidad de columnas"
leer COLS

```
//carga de datos
escribir "Carga de los elementos a la matriz"
para (I ← 0; I < FILAS; I ← I + 1)
{
  escribir "Ingrese dato para la fila: ", I
  para (J ← 0; J < COLS; J ← J + 1)
    leer MATRIZ[I][J]
}

//muestra de datos
escribir "Los elementos de la tercera columna son los siguientes"
para (I ← 0; I < FILAS; I ← I + 1)
  para (J ← 0; J < COLS; J ← J + 1)
    si (J == 2)
      escribir MATRIZ[I][J]
} //principal
```

Otra posibilidad para mostrar los elementos de la tercera columna, sería:

```
para (I ← 0; I < FILAS; I ← I + 1)
  escribir MATRIZ[I][2]
```

Con esta última forma evitamos recorrer toda la matriz innecesariamente. A veces, encontrar formas de recorrido más simples hace a los algoritmos más rápidos.

Problema: Dada una matriz de números enteros generar un vector donde cada elemento del mismo es el resultado de la suma de los elementos de cada columna de la matriz.

Ejemplo:

Dada la siguiente Matriz:

3	6	4	8
0	4	1	3
4	0	7	8

El resultado sería el siguiente Vector:

7	10	12	19
---	----	----	----

El número 7 (VECTOR[0]) = 3 (MATRIZ[0][0]) + 0 (MATRIZ[1][0]) + 4 (MATRIZ[2][0]).
El número 10 (VECTOR[1]) = 6 (MATRIZ[0][1]) + 4 (MATRIZ[1][1]) + 0 (MATRIZ[2][1]).
El número 12 (VECTOR[2]) = 4 (MATRIZ[0][2]) + 1 (MATRIZ[1][2]) + 7 (MATRIZ[2][2]).
El número 19 (VECTOR[3]) = 8 (MATRIZ[0][3]) + 3 (MATRIZ[1][3]) + 8 (MATRIZ[2][3]).

Es muy importante ver el comportamiento de los índices, como se mostró anteriormente.

Apunte de Cátedra

```
principal CrearVector
{
//declaraciones
FILA = 50
COLUMNA = 50
entero [ ][ ] MATRIZ
MATRIZ = nuevo entero [FILA][COLUMNA]
entero [ ] VECTOR
VECTOR = nuevo entero [COLUMNA]
entero FILAS, COLS
entero SUMA, I, J

//carga de dimensiones
escribir "Ingrese cantidad de filas"
leer FILAS
escribir "Ingrese cantidad de columnas"
leer COLS

//carga de datos
escribir "Carga de los elementos a la matriz"
para (I ← 0; I < FILAS; I ← I + 1)
{
escribir "Ingrese dato para la fila: ", I
para (J ← 0; J < COLS; J ← J + 1)
leer MATRIZ[I][J]
}

//generación del vector
//el recorrido lo vamos a hacer por columnas
para (J ← 0; J < COLS; J ← J + 1)
{
SUMA ← 0
para (I ← 0; I < FILAS; I ← I + 1)
SUMA ← SUMA + MATRIZ[I][J]
VECTOR[J] ← SUMA
}

//muestra del vector
escribir "El vector resultante es la siguiente"
para (I ← 0; I < COLS; I ← I + 1)
escribir VECTOR[I]
}
principal
```

Los métodos de ordenación (clasificación) vistos anteriormente y aplicados a vectores se pueden extender a matrices o tablas considerando la ordenación respecto a una fila o columna. Lo mismo con los métodos de búsqueda para matrices, adaptándolos.

DISEÑO DE ALGORITMOS: FUNCIONES

Tras la fase de análisis, para poder solucionar problemas sobre la computadora, debe conocerse como diseñar algoritmos. En la práctica sería deseable disponer de un método para escribir algoritmos, pero, en la realidad no existe. El diseño de algoritmos es un proceso creativo. Sin embargo, existen una serie de pautas o líneas a seguir que ayudarán al diseño de algoritmos:

1. Formular una solución precisa del problema que debe solucionar el algoritmo.
2. Ver si existe ya algún algoritmo para resolver el problema o bien se puede adaptar uno ya conocido (algoritmos conocidos).
3. Buscar si existen técnicas estándar que se pueden utilizar para resolver el problema.
4. Elegir una estructura de datos adecuada.
5. Dividir el problema en subproblemas y aplicar el método a cada uno de los subproblemas (funciones).

De cualquier forma, antes de iniciar el diseño del algoritmo es preciso asegurarse que el programa está bien definido:

- Especificaciones precisas y completas de las entradas necesarias.
- Especificaciones precisas y completas de las salidas.
- ¿Cómo debe reaccionar el programa ante datos incorrectos?.
- ¿Se emiten mensajes de error?, ¿se detiene el proceso?, etc.
- Conocer cuándo y cómo debe terminar un programa.

Haciendo hincapié al punto 5 que se refiere a solucionar los siguientes problemas:

- Un programa realiza varias tareas, por lo tanto, se compone de muchos algoritmos, lo que lleva a una escritura extensa y confusa.
- La detección y corrección de errores en segmentos específicos de código en dicho programa se dificulta, dada la longitud del mismo.
- Un mismo algoritmo u operación se repite en varias partes dentro del mismo programa (redundancia o repetición de información).

Respecto al último problema, a menudo, determinadas tareas o algoritmos se realizan mas de una vez en el mismo programa en diferentes partes, por lo tanto, lo que hacemos es repetir secuencias de código iguales. Por ejemplo, si hablamos de una secuencia de 100 números cuyos valores deben mantenerse para su utilización posterior y queremos aplicar a cada uno de ellos la operación de factorial, deberíamos repetir dicha operación 100 veces. Por lo tanto, una forma de escribir programas más cortos y flexibles, sería dividir el programa en funciones y de esta manera, estaríamos solucionando el problema planteado hasta el momento. Es decir, que para este ejemplo se crearía una función factorial (se escribe el código de factorial una sola vez) y se la usaría y aprovecharía para los 100 números de la secuencia.

Veamos otro ejemplo de redundancia o repetición de información (código). Si un programa debe obtener el mayor promedio de dos vectores de diferentes dimensiones realizará las cargas, el cálculo del promedio y la muestra de los vectores por separado.

principal EjemploDeRedundancia

```
{ //principal
//declaración de variables
real PROM1, PROM2
```


Apunte de Cátedra

```
entero SUMA, CONT
entero [ ] VECTOR1
VECTOR1 = nuevo entero [100]
entero [ ] VECTOR2
VECTOR2 = nuevo entero [50]

//carga del primer vector
para (CONT ← 0; CONT < 100; CONT ← CONT + 1)
    leer VECTOR1[CONT]

//carga del segundo vector
para (CONT ← 0; CONT < 50; CONT ← CONT + 1)
    leer VECTOR2[CONT]

//sumatoria y promedio de los elementos del primer vector
SUMA ← 0
para (CONT ← 0; CONT < 100; CONT ← CONT + 1)
    SUMA ← SUMA + VECTOR1[CONT]
PROM1 ← SUMA / 100

//sumatoria y promedio de los elementos del segundo vector
SUMA ← 0
para (CONT ← 0; CONT < 50; CONT ← CONT + 1)
    SUMA ← SUMA + VECTOR2[CONT]
PROM2 ← SUMA / 50

//comparación de promedios
si (PROM1 > PROM2)
    escribir "El mayor promedio es: ", PROM1
sino
    escribir "El mayor promedio es: ", PROM2

//muestra del primer vector
para (CONT ← 0; CONT < 100; CONT ← CONT + 1)
    escribir VECTOR1[CONT]

//muestra del segundo vector
para (CONT ← 0; CONT < 50; CONT ← CONT + 1)
    escribir VECTOR2[CONT]
} //principal
```

Como vemos, el algoritmo para cargar un vector es el mismo independientemente de su dimensión. Lo mismo ocurre para calcular el promedio o mostrar el contenido de los vectores. Si en este programa se trabajara con 15 vectores, resultaría muy extenso y confuso. Además, siempre se estaría repitiendo lo mismo (carga, promedio y muestra).

Como se dijo anteriormente, lo más conveniente es dividir o estructurar el programa en *funciones*. Sería, realizar una función por cada algoritmo de nuestro programa. Una forma de plantear esta técnica es dividir un problema en subproblemas y para cada subproblema plantear un algoritmo y a cada

Apunte de Cátedra

algoritmo le corresponderá una función. El objetivo de cada función sería que tenga una tarea específica a realizar. Entonces, el programa se divide en funciones (partes independientes), cada una de las cuales ejecuta una única actividad o tarea y se codifican independientemente de otras funciones.

Cada programa contiene una función denominada *Principal* que es el punto de partida de ejecución del programa; se transfiere el control a funciones, de modo que ellas puedan ejecutar sus operaciones. Si la tarea asignada a cada función es demasiado compleja; ésta deberá romperse en otras funciones más pequeñas. El proceso sucesivo de subdivisión de funciones continúa hasta que cada función tenga solamente una tarea específica a ejecutar. Una función puede transferir temporalmente (bifurcación) el control a otra función; sin embargo, cada función debe devolver el control a la función del cual recibe originalmente el control.

Para las tareas que deben efectuarse más de una vez, las funciones *evitan la necesidad de programación redundante* (repetida) de un mismo conjunto de instrucciones, ya que una función puede ser definida una vez y llamada luego tantas veces como sea necesario. El uso de funciones puede reducir apreciablemente la longitud de un programa.

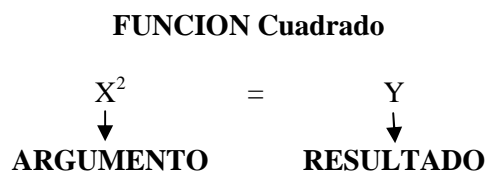
Luego de todo lo expuesto anteriormente, podemos decir que las ventajas de utilizar funciones son:

- Programas más legibles y claros.
- Programas más cortos.
- Disminuye la programación repetida (reuso de código).
- Mayor flexibilidad en el mantenimiento.
- Facilita la prueba y depuración de errores.

Funciones

Matemáticamente, una **función** es una operación que toma uno o más valores llamados **argumentos** y retorna ninguno, uno o varios valores, lo que se denomina **resultado**.

Por ejemplo:



De manera muy parecida diremos que una función es un subprograma que recibe uno o varios argumentos (parámetros), realiza algún proceso o cálculo y devuelve o entrega un resultado al programa que la llamó o invocó.

El formato general de la declaración de una función es:

```
Tipo_de_dato nombre_de_la_función(tipo_de_dato parámetro1, tipo_de_dato parámetro2,...)
{
    declaración_de_variables_locales
    ...
    cuerpo_de_la_función
    ...
    retornar [resultado]
}
```

Apunte de Cátedra

Donde:

Tipo_de_dato	es el tipo de dato del resultado que devuelve la función.
nombre_de_la_función	es el identificador o nombre que identifica a la función.
parámetro1, parámetro2, ...	es una lista de parámetros formales o argumentos, en la cual se indica el tipo de datos y nombre de los mismos (separados por coma). Puede estar compuesta de ninguno, uno o más parámetros.
cuerpo de la función	es el conjunto de declaraciones y sentencias que ejecuta la función.
retornar	es la instrucción que permite a la función devolver su resultado.

El cuerpo de la función debe ser una operación que no requiera una lectura de datos, dado que los datos que necesite la función serán entregados por los parámetros. Por lo tanto, no se puede utilizar el *leer* dentro de la función. La devolución de una función puede ser de tres tipos de resultado: un valor de tipo simple, un valor de tipo compuesto/objeto o nada. En este último caso, no se coloca tipo de dato de la función y la instrucción *retornar* va sin nada. Dado que una función es un subprograma, tiene prácticamente la misma forma que un programa: una cabecera, declaraciones de datos y un cuerpo (conjunto de sentencias).

❖ Ejemplos.

Problema: Escribir una función que devuelva el cubo de un valor.

```
entero Cubo(entero X)
{
  //declaración de variables
  entero AUXCUBO

  //cálculo del cubo
  AUXCUBO ← X * X * X

  //retorno del resultado
  retornar AUXCUBO
}
```

Entero es el tipo de dato que devuelve la función, *Cubo* es el nombre de la función, y el cálculo se realiza utilizando el parámetro *X* de tipo entero y una variable local de tipo entero llamada *AUXCUBO*. Luego, mediante *RETORNAR* se devuelve el resultado almacenado en *AUXCUBO*.

Problema: Crear una función que verifique si un número es par o no.

```
booleano esPar(entero NUM)
{
  //declaración de variables
  booleano RESULTADO

  //evaluación par o impar
  si (NUM % 2 = 0)
    RESULTADO ← Verdadero
```

Apunte de Cátedra

```
sino
    RESULT ← Falso

//retorno del resultado
retornar RESULT
}
```

Otra manera válida de escribir la función sin usar una variable local, es de la siguiente manera:

```
booleano esPar(entero NUM)
{
    //evaluación para par o impar y retorno del resultado
    si (NUM % 2 == 0)
        retornar Verdadero
    sino
        retornar Falso
}
```

Problema: Crear una función que a partir de un número entero retorne 1, cuando sea positivo, 0 cuando sea 0 y -1 cuando sea negativo.

```
entero positivoCeroNegativo(entero NUM)
{
    //evaluación y retorno del resultado
    si (NUM > 0)
        retornar 1
    sino
        si (NUM == 0)
            retornar 0
        sino
            retornar -1
}
```

Problema: Mostrar el nombre del mes correspondiente un valor entero ingresado. Ej.: 5 sería Mayo.

```
nombreMes(entero MES)
{
    //evaluación y muestreo
    alternar (MES)
    {
        caso 1 : {
            escribir "ENERO"
            corte
        }
        caso 2 : {
            escribir "FEBRERO"
            corte
        }
    }
```

Apunte de Cátedra

```
caso 3 :      {  
              escribir "MARZO"  
              corte  
              }  
caso 4 :      {  
              escribir "ABRIL"  
              corte  
              }  
caso 5 :      {  
              escribir "MAYO"  
              corte  
              }  
caso 6 :      {  
              escribir "JUNIO"  
              corte  
              }  
caso 7 :      {  
              escribir "JULIO"  
              corte  
              }  
caso 8 :      {  
              escribir "AGOSTO"  
              corte  
              }  
caso 9 :      {  
              escribir "SEPTIEMBRE"  
              corte  
              }  
caso 10 :     {  
              escribir "OCTUBRE"  
              corte  
              }  
caso 11 :     {  
              escribir "NOVIEMBRE"  
              corte  
              }  
caso 12 :     {  
              escribir "DICIEMBRE"  
              corte  
              }  
caso contrario  
              escribir "NUMERO DE MES INCORRECTO"  
    }  
retornar  
}
```

En este caso, se ilustra como una función no devolvería nada. Es decir, la salida es implícita, dado que estaría representada por los escribir.

Apunte de Cátedra

Problema: De acuerdo a dos caracteres retornar el menor de ellos.

```
caracter menorCaracter(caracter CAR1, caracter CAR2)
{
    //evaluación para menor y muestreo
    si (CAR1 < CAR2)
        retornar CAR1
    sino
        retornar CAR2
}
```

Problema: Hacer una función que devuelva la sumatoria de los elementos de un vector, otra función que “vacíe” el vector (todos valores 0) y otra función que muestre el contenido del vector.

```
entero sumatoriaElementos(entero [ ] VECTOR, entero DIM)
{
    //declaración de variables
    entero CONT, SUMA

    //cálculo de la sumatoria
    SUMA ← 0
    para (CONT ← 0; CONT < DIM; CONT ← CONT + 1)
        SUMA ← SUMA + VECTOR[CONT]

    retornar SUMA
}
```

```
entero [ ] vaciarVector(entero [ ] VECTOR, entero DIM)
{
    //declaración de variables
    entero CONT

    //vaciado del vector
    para (CONT ← 0; CONT < DIM; CONT ← CONT + 1)
        VECTOR[CONT] ← 0

    retornar VECTOR
}
```

```
muestraVector(entero [ ] VECTOR, entero DIM)
{
    //declaración de variables
    entero CONT

    //muestreo del vector
    para (CONT ← 0; CONT < DIM; CONT ← CONT + 1)
        escribir VECTOR[CONT]

    retornar
}
```

Apunte de Cátedra

En estos último ejemplo, se ilustra los tres tipos de resultados que puede devolver una función: un valor de tipo simple, un valor de tipo compuesto/objeto o nada (donde no se coloca el tipo de dato de la función y la instrucción *retornar* va sin nada).

Aserciones: Precondiciones y Postcondiciones

Un programa no puede decirse que sea correcto o incorrecto *per se*; es correcto o incorrecto con respecto a cierta especificación (descripción precisa de lo que se supone que el programa debe hacer). Estrictamente hablando, no se debería discutir sobre si un programa es o no *correcto*, sino sobre si es *consistente* con sus especificaciones. Esta discusión continuará utilizando el término bien aceptado de “corrección” pero siempre se debe recordar que la pregunta de si es correcto no se aplica a programas; se aplica a pares formados por un programa y una especificación.

Para expresar la especificación confiaremos en las aserciones. Una aserción es una expresión que involucra algunas entidades del software y que establece una propiedad que dichas entidades deben satisfacer en ciertas etapas de la ejecución de un programa. Una típica aserción puede ser la que expresa que un entero es positivo.

Se puede especificar la tarea que lleva a cabo una función mediante dos aserciones asociadas a la función: una *precondición* y una *postcondición*.

Una precondición es una información que se conoce como verdadera antes de iniciar la función.

La precondición establece las propiedades que se tienen que cumplir cada vez que se llame a la función. Por ejemplo: si utilizamos a la función *nombreMes(entero)* vista anteriormente, podemos decir que la precondición sería que el valor entero debe ser positivo y pertenecer al rango de valores entre el 1 y el 12.

Una postcondición es una información que debiera ser verdadera al concluir una función, si se cumple adecuadamente el requerimiento pedido.

La postcondición establece las propiedades que debe garantizar la función cuando retorne. Por ejemplo: en el caso de la función *menorCaracter(caracter, caracter)* debería devolver como postcondición un valor de tipo caracter igual a uno de los datos que ingresan por parámetro (que debe ser el menor de ellos).

Invocación a la función

La llamada a una función se realiza de la siguiente manera:

variable ← nombre_de_la_función(parámetro1, parámetro2, ...)
(si la función retorna un resultado simple o compuesto/objeto)
ó
nombre_de_la_función(parámetro1, parámetro2, ...)
(si la función no retorna ningún resultado)

Donde:

variable	es la variable del programa llamador.
nombre_de_la_función	es el identificador o nombre de la función que se llama.
parámetro1, parámetro2, ...	es una lista de parámetros actuales, en la cual se indica el nombre de los mismos (separados por coma). Pueden ser constantes, variables o expresiones. Puede estar compuesta de ninguno, uno o más parámetros.

Apunte de Cátedra

Por ejemplo, veamos cómo la función cubo se incorpora en un programa que la utiliza o invoca.

```
{  
principal CuboNumero  
  {                                     //principal  
    //declaración de variables  
    entero NUM, AUX  
    //carga del valor  
    escribir "Ingrese un valor"  
    leer NUM  
    //invocación a la función y asignación del resultado a una variable  
    AUX ← Cubo(NUM)  
    //muestra del resultado  
    escribir "El cubo de ", NUM, " es: ", AUX  
  }                                     //principal  
  
//función que calcula el cubo de un número entero  
//precondición: valor entero  
//postcondición: valor entero elevado a 3  
entero Cubo(entero X)  
  {  
    //declaración de variables  
    entero AUXCUBO  
    //cálculo del cubo  
    AUXCUBO ← X * X * X  
    //retorno del resultado  
    retornar AUXCUBO  
  }  
}
```

Cuando este programa se ejecute, luego de que se lea NUM, el principal llama a la función (pasa el control a la función), usando NUM como argumento, la función se ejecuta y devuelve su resultado asignándolo a la variable AUX.

Un programa se divide en funciones; y Principal es una función más que no devuelve nada y que tiene la característica particular de que en todo programa la ejecución comienza por ella misma.

Orden de ejecución

Cuando un programa se ejecute, siempre comienza por la primer sentencia del principal, el flujo de control se desplaza de acuerdo al tipo de estructuras de control (secuencia, selección, iteración), cuando ocurre una llamada a una función, se interrumpe la ejecución del principal y la función toma el control y se ejecuta por completo (comenzando en su primer línea, hasta llegar a la última). Cuando la función finaliza su ejecución, el control retorna al principal o al punto dónde fue interrumpido y continúa su ejecución.

A continuación veremos un ejemplo de un programa que trabaja con un arreglo de enteros, utilizando los tres casos vistos anteriormente del último ejemplo de funciones con vectores.

Apunte de Cátedra

```
{
principal TrabajoConArreglo
{
//principal
//declaración de variables
entero DIM, SUMATORIA, CONTADOR
entero [ ] VEC
VEC = nuevo entero [100]

//carga de la dimensión
escribir "Ingrese la dimensión del vector"
leer DIM

//carga del vector
para (CONTADOR ← 0; CONTADOR < DIM; CONTADOR ← CONTADOR + 1)
leer VEC[CONTADOR]

//invocación a la función y asignación del resultado a una variable
SUMATORIA ← sumatoriaElementos(VEC, DIM)

//muestra del resultado
escribir "La sumatoria del vector es: ", SUMATORIA

//muestra del vector
muestraVector(VEC, DIM)

//vaciado del vector
VEC ← vaciarVector(VEC, DIM)

//muestra del vector
muestraVector(VEC, DIM)
}
//principal

//función que calcula la sumatoria de los elementos del vector
//precondición: vector con elementos enteros y dimensión válida
//postcondición: resultado entero de la suma de los elementos del vector
entero sumatoriaElementos(entero [ ] VECTOR, entero D)
{
.
.
.
}

//función que vacía el contenido de un vector (colocando en 0 cada elemento)
//precondición: vector con elementos enteros y dimensión válida
//postcondición: vector vacío
entero [ ] vaciarVector(entero [ ] VECTOR, entero D)
{
.
.
.
}
```

Apunte de Cátedra

```
//función que muestra el contenido de un vector
//precondición: vector con elementos enteros y dimensión válida
//postcondición: muestreo de los elementos del vector
muestraVector(entero [ ] VECTOR, entero D)
{
    .
    .
    .
}
}
```

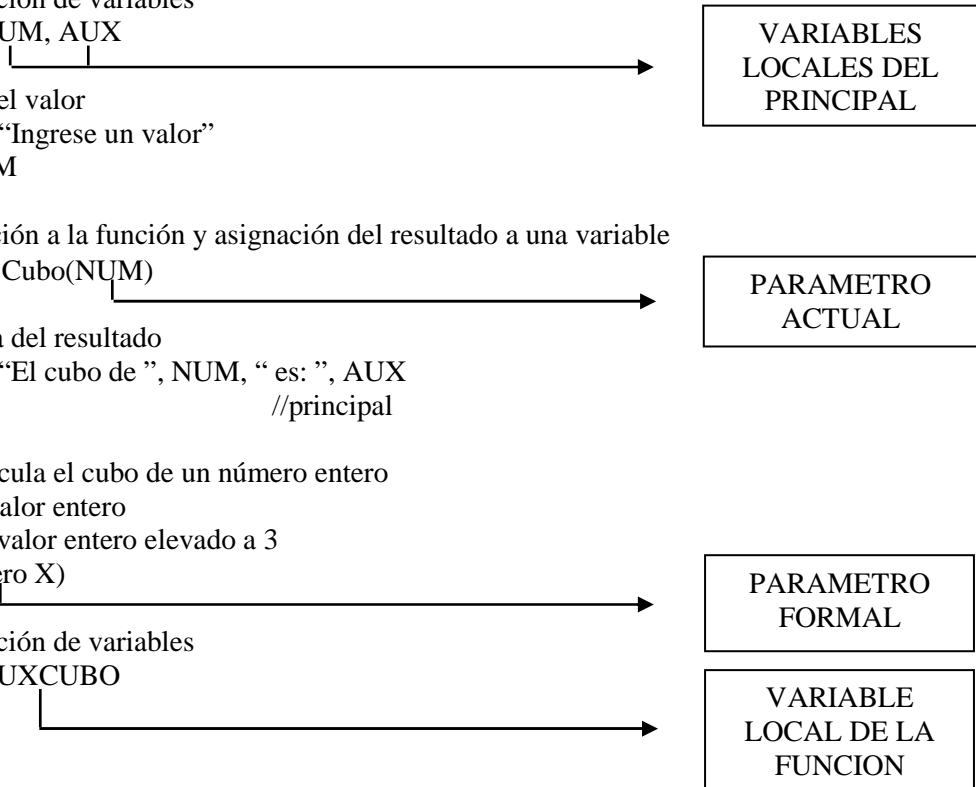
Conceptos fundamentales de una función

Veamos nuevamente el programa CuboNumero distinguiendo algunos conceptos fundamentales.

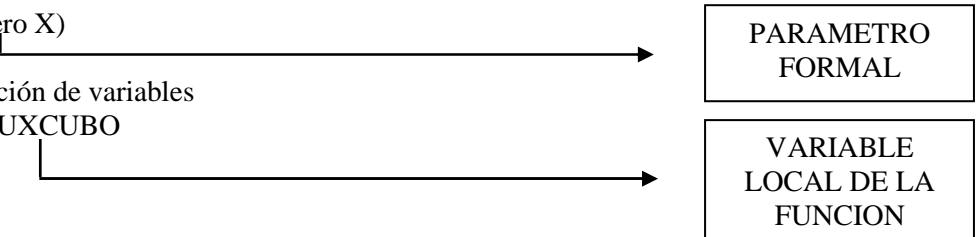
```
{
principal CuboNumero
{
    //declaración de variables //principal
    entero NUM, AUX
    //carga del valor
    escribir "Ingrese un valor"
    leer NUM

    //invocación a la función y asignación del resultado a una variable
    AUX ← Cubo(NUM)

    //muestra del resultado
    escribir "El cubo de ", NUM, " es: ", AUX
}
} //principal
```



```
//función que calcula el cubo de un número entero
//precondición: valor entero
//postcondición: valor entero elevado a 3
entero Cubo(entero X)
{
    //declaración de variables
    entero AUXCUBO
    .
    .
    .
}
}
```



Se han indicado aquí dos conceptos fundamentales de la programación modular:

- **Variables Locales.**
- **Parámetros (Actuales y Formales).**

Apunte de Cátedra

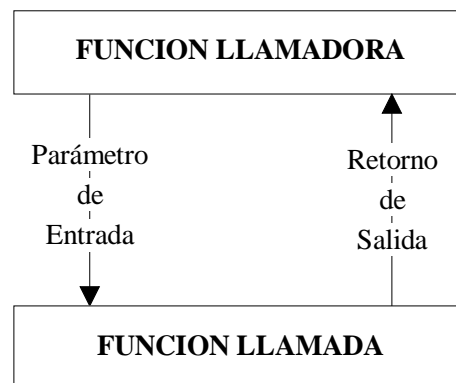
VARIABLES LOCALES

Una **variable local** es una variable que está declarada dentro de una función, y se dice que es local a la función. Una variable local sólo está disponible durante la ejecución de la misma. Su valor se pierde o se destruye una vez que la función finaliza su ejecución, es decir, cuando se retorna a la función llamadora.

El concepto subyacente es el de “*tiempo de vida de un identificador*”, cuando una variable es local, su existencia o vida se restringe a la ejecución de la función en que esta declarada. Entonces, en forma genérica recalcamos que si tenemos una variable local llamada X en la función P, no podremos usar a X fuera de P.

PARÁMETROS

Corresponde a la transferencia de información entre funciones, a través de una lista de parámetros. Un parámetro es una técnica para pasar información - valores a variables - de una función a otra y viceversa. En pocas ocasiones, podemos aplicar funciones sin parámetros, como el principal que es una función que no requiere parámetros de entrada.



Como se ilustra en el gráfico, un parámetro es una variable cuyo valor debe ser proporcionado desde la función llamadora a la función llamada. La función llamada, con el parámetro de entrada realiza ciertas operaciones, de la que obtiene un resultado y lo transmite como retorno de salida a la función llamadora. Como puede observarse en el programa CuboNumero, el parámetro de entrada de la función Cubo es un entero, que lo proporciona el principal.

Lista de parámetros actuales y formales

Los parámetros utilizados cuando se llama a una función se denominan **parámetros actuales**, éstos son variables de la función llamadora (ver el parámetro NUM del programa CuboNumero). Cuando en la función se incluye una lista de parámetros con sus correspondientes tipos, a éstos parámetros los denominamos **parámetros formales**.

El uso de parámetros ofrece una técnica para el intercambio de información entre funciones. Cada dato se transfiere entre un parámetro actual incluido dentro de la referencia (invocación a una función), y un parámetro formal correspondiente, definido dentro de la función llamada. Cuando se invoca a la función, los parámetros actuales son reemplazados por los parámetros formales, creando así un mecanismo de intercambio de información entre funciones.

Modifiquemos el programa TrabajoConArreglo para que realice todas las operaciones con dos arreglos.

Apunte de Cátedra

```
{
principal TrabajoConArreglo
{
//principal
//declaración de variables
entero DIM, SUMA1, SUMA2, CONTADOR
entero [ ] VEC1, VEC2
VEC1 = nuevo entero [100]
VEC2 = nuevo entero [100]

//carga de la dimensión
escribir "Ingrese la dimensión del vector"
leer DIM

//carga del primer vector
para (CONTADOR ← 0; CONTADOR < DIM; CONTADOR ← CONTADOR + 1)
leer VEC1[CONTADOR]

//carga del segundo vector
para (CONTADOR ← 0; CONTADOR < DIM; CONTADOR ← CONTADOR + 1)
leer VEC2[CONTADOR]

//invocación a la función con el primer vector y asignación del resultado a una variable
SUMA1 ← sumatoriaElementos(VEC1, DIM)

//invocación a la función con el segundo vector y asignación del resultado a una variable
SUMA2 ← sumatoriaElementos(VEC2, DIM)

//muestra de los resultados
escribir "La sumatoria del primer vector es: ", SUMA1
escribir "La sumatoria del segundo vector es: ", SUMA2

//muestra del primer vector
muestraVector(VEC1, DIM)

//muestra del segundo vector
muestraVector(VEC2, DIM)

//vaciado del primer vector
VEC1 ← vaciarVector(VEC1, DIM)

//vaciado del segundo vector
VEC2 ← vaciarVector(VEC2, DIM)

//muestra del primer vector
muestraVector(VEC1, DIM)

//muestra del segundo vector
muestraVector(VEC2, DIM)
}
//principal
```

Apunte de Cátedra

```
//función que calcula la sumatoria de los elementos del vector
//precondición: vector con elementos enteros y dimensión válida
//postcondición: resultado entero de la suma de los elementos del vector
entero sumatoriaElementos(entero [ ] VECTOR, entero D)
{
    .
    .
    .
}

//función que vacía el contenido de un vector (colocando en 0 cada elemento)
//precondición: vector con elementos enteros y dimensión válida
//postcondición: vector vacío
entero [ ] vaciarVector(entero [ ] VECTOR, entero D)
{
    .
    .
    .
}

//función que muestra el contenido de un vector
//precondición: vector con elementos enteros y dimensión válida
//postcondición: muestreo de los elementos del vector
muestraVector(entero [ ] VECTOR, entero D)
{
    .
    .
    .
}
}
```

En este ejemplo podemos ver que las funciones `sumatoriaElementos`, `vaciarVector` y `muestraVector` no se han tenido que modificar, sólo se tuvo que agregar al programa una llamada más a cada una de las funciones. Y en cada llamada lo que varía es el parámetro actual (`VEC1` y `VEC2`). El parámetro formal (`VECTOR`) se reemplazará por el parámetro actual cuando la función sea llamada. De esta manera, al utilizar parámetros podemos llamar a una función muchas veces para que realice el mismo cálculo con diferentes valores, ésta es una de las ventajas de trabajar con funciones.

En una función si usamos un determinado identificador, éste es o una variable local de la función o pertenece a la lista de parámetros formales a dicha función. CASO CONTRARIO, es incorrecto el uso de dicho identificador.

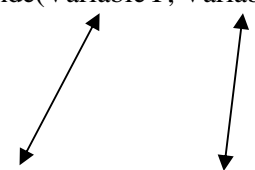
Correspondencia de parámetros

Los parámetros actuales en la invocación a una función deben coincidir en **número**, **orden** y **tipo** con los parámetros formales de la declaración de una función. Es decir debe existir una correspondencia de parámetros.

A continuación veremos un ejemplo genérico de correspondencia de parámetros, para explicar los tres puntos nombrados.

Apunte de Cátedra

```
{
principal Correspondencia
{
    entero VariableX, VariableY
    real VariableZ
    .
    .
    VariableX ← Corresponde(VariableY, VariableZ)
    .
    .
}
entero Corresponde (entero Parámetro1, real Parámetro2)
{
    .
    .
    .
}
}
```



- **Número:** la cantidad de parámetros actuales es 2 y la cantidad de parámetros formales es 2.
- **Orden:** al parámetro actual VariableY le corresponde el parámetro formal Parámetro1 (ambos son los primeros en la lista); al parámetro actual VariableZ le corresponde el parámetro formal Parámetro2 (ambos son los segundos en la lista).
- **Tipo:** VariableY y Parámetro1 son de tipo entero; VariableZ y Parámetro2 son de tipo real.

La función Corresponde entrega un valor entero, por lo tanto, la variable del principal que recibe el resultado de esta función, en este caso VariableX, también debe ser entero. Es decir, también debe coincidir el tipo de datos de la variable a la cual se le asigna el resultado de la función, con el tipo de datos de retorno de la función.

Veamos otro ejemplo de funciones, dónde se calcula el máximo común divisor de dos números enteros.

```
{
principal MaximoComunDivisor
{
    //declaración de variables
    entero MCD, NRO1, NRO2
    //carga de datos
    escribir "Ingrese dos números enteros"
    leer NRO1, NRO2
    //llamada a la función
    MCD ← MaxComDiv(NRO1, NRO2)
    //muestra de resultado
    escribir "El máximo común divisor entre ", NRO1, " y ", NRO2, " es: ", MCD
}
//principal
}
```

Apunte de Cátedra

```
//función que calcula el máximo común divisor de dos números enteros
//precondición: 2 números enteros
//postcondición: máximo común divisor de esos 2 números enteros
entero MaxComDiv(entero N1, entero N2)
{
    //declaración de variables
    entero COCIENTE, RESTO
    //cálculo del máximo común divisor
    hacer
        {
            COCIENTE ← N1 / N2
            RESTO ← N1 % N2
            si (RESTO != 0)
                {
                    N1 ← N2
                    N2 ← RESTO
                }
        }
    mientras (RESTO != 0)
        //retorno del resultado
    retornar N2
}
```

Visibilidad de identificadores

Un identificador es el nombre de una función, variable, constante o parámetro. Un programa se compone por lo general de varias funciones, estas funciones utilizan variables, constantes y parámetros. Puede ocurrir que dos funciones utilicen una variable con el mismo identificador.

```
{
principal P
{
    //declaración de variables locales a P
    .
    .
    .
}
```

Función F1 (lista de parámetros)

```
{
    //declaración de variables locales a F1
    .
    .
    .
}
```

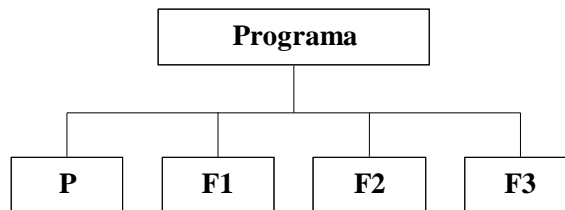
Función F2 (lista de parámetros)

```
{
    //declaración de variables locales a F2
    .
    .
    .
}
```

Apunte de Cátedra

```
.  
}  
  
Función F3 (lista de parámetros)  
{  
  //declaración de variables locales a F3  
  .  
  .  
  .  
}  
}
```

En este esquema tenemos el principal P y tres funciones (F1, F2 y F3). Todas estas funciones poseen su propia lista de parámetros y sus propias declaraciones locales. Gráficamente podemos ilustrar el esquema de la siguiente manera:



Con este ejemplo, se puede plantear la siguiente duda: ¿las variables declaradas en P pueden utilizarse en F1 y en F2?. Para responder a esta pregunta, debemos retomar el concepto de *tiempo de vida de un identificador* y agregar el concepto de *ámbito de un identificador*.

Ámbito de un identificador

Un *bloque* de función se compone de una cabecera (tipo de dato, nombre y lista de parámetros) y un cuerpo de sentencias (declaraciones de variables y constantes e instrucciones). Tanto el principal como el resto de las funciones, son bloques de función.

Los bloques en los que un identificador (variable y constante) puede ser utilizado se conoce como *ámbito del identificador*; dicho de otro modo, el *ámbito* es la sección de un programa en la que un identificador es válido.

Para saber concretamente la respuesta a la pregunta formulada en la sección anterior, hemos de guiarnos por las reglas de ámbito.

Primer Regla de Ámbito:

El ámbito de un identificador es el dominio en que está declarado. Por consiguiente, un identificador declarado en el bloque P puede ser representado en P.

De acuerdo al esquema que hemos realizado anteriormente, ahora estamos en condiciones de asegurar: Las variables declaradas en principal P, pueden sólo ser utilizadas en P; no pueden ser utilizadas fuera de él, es decir, en F1, F2 y F3.

Las variables declaradas en F1, pueden sólo ser utilizadas en F1. Lo mismo para las funciones F2 y F3.

Segunda Regla de Ámbito:

El ámbito de un parámetro formal es idéntico al ámbito de una variable local de la misma función.

Apunte de Cátedra

Clases/Objetos: Tipos de Datos Compuesto

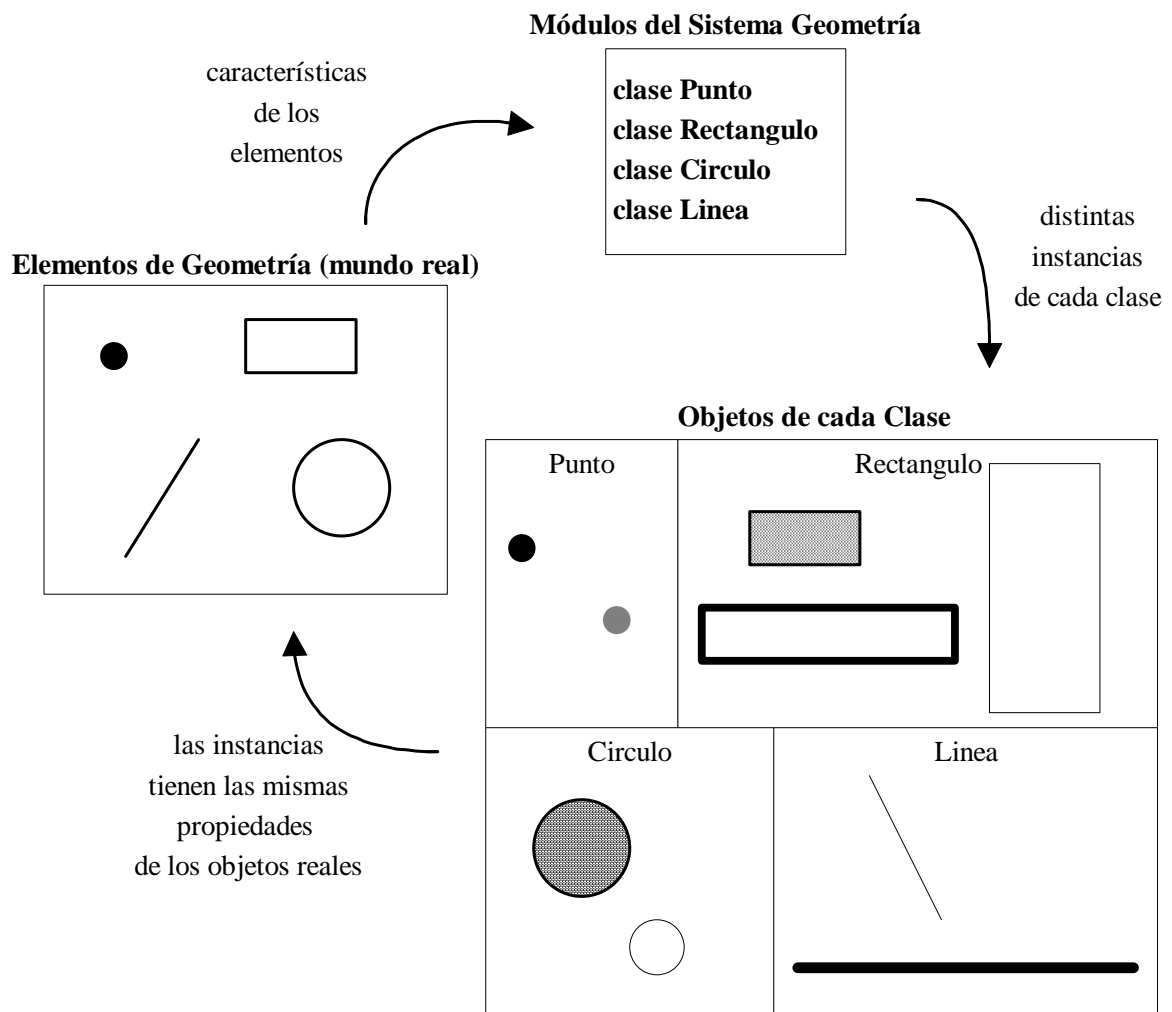
Antes de comenzar a hablar de las clases y los objetos, vamos a conocer algunos conceptos que nos llevarán al uso de estos tipos de datos.

Orientación a Objetos (OO)

La orientación a objetos puede describirse como el *conjunto de disciplinas que desarrollan y modelan software que facilitan la construcción de sistemas complejos a partir de componentes.*

El atractivo intuitivo de la OO es que proporciona conceptos y herramientas con las cuales se modela y representa el mundo real tan fielmente como sea posible. Las ventajas de la OO son muchas en programación y modelación de datos. Como apuntaban Ledbetter y Cox (1985):

La POO (Programación Orientada a Objetos) permite una representación más directa del modelo de mundo real en el código. El resultado es que la transformación radical normal de los requisitos del sistema (definidos en términos de usuario) a la especificación del sistema (definido en términos de computación) se reduce considerablemente.



Apunte de Cátedra

Los conceptos y herramientas OO son tecnologías que permiten que los problemas del mundo real sean expresados de modo fácil y natural. Las técnicas OO proporcionan mejoras metodológicas para construir sistemas de software más fiables a partir de unidades de software modularizado y reutilizable.

El enfoque OO es capaz de manipular tanto sistemas grandes como pequeños, y crear sistemas fiables que sean flexibles, mantenibles y capaces de evolucionar para cumplir las necesidades de cambio. Estas tareas se realizan mediante la modelización del mundo real. El soporte fundamental es el *modelo objeto*. Los cuatro elementos (propiedades) más importantes de este modelo son:

- Abstracción.
- Encapsulación.
- Modularidad.
- Jerarquía (tema que se verá más adelante).

Abstracción

La abstracción es uno de los medios más importantes mediante el cual nos enfrentamos con la complejidad inherente al software. La abstracción es la propiedad que permite representar las características esenciales de un objeto, sin preocuparse de las restantes características (no esenciales).

Una abstracción se centra en la vista externa de un objeto, de modo que sirva para separar el comportamiento esencial de un objeto, de su implementación. Definir una abstracción significa describir una entidad del mundo real, no importa lo compleja que pueda ser, y a continuación utilizar esta descripción en un programa.

El elemento clave de la POO es la clase. Una clase se puede definir como una descripción abstracta de un grupo de objetos, cada uno de los cuales se diferencia por su *estado* específico y por la posibilidad de realizar una serie de *operaciones*. Por ejemplo, una lapicera es un objeto que tiene un estado (llena de tinta o vacía) y sobre la cual se pueden realizar algunas operaciones (escribir, recargar la tinta, poner o quitar capuchón).

La idea de escribir programas definiendo una serie de abstracciones no es nueva, pero el uso de clases para gestionar dichas abstracciones en lenguajes de programación ha facilitado considerablemente su aplicación.

Encapsulación

La encapsulación o encapsulamiento es la propiedad que permite asegurar que el contenido de la información de un objeto está oculta al mundo exterior: el objeto A no conoce lo que hace el objeto B, y viceversa. La encapsulación (también se conoce como *ocultamiento de la información*), en esencia, es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales.

La encapsulación permite la división de un programa en módulos, estos módulos se implementan mediante clases, de forma que una clase representa encapsulación de una abstracción.

Modularidad

La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas *módulos*), cada una de las cuales deben ser tan independientes como sea posible de la aplicación en sí y de las restantes partes.

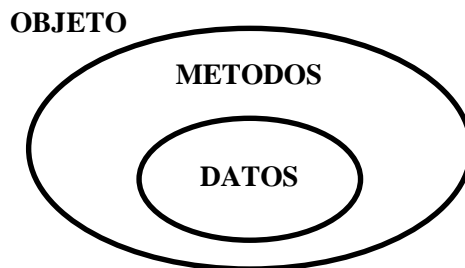
La modularización consiste en dividir un programa en módulos que se puedan compilar por separado, pero que tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularización de diferentes formas, en OO estos módulos son las clases.

Apunte de Cátedra

Beneficios de la Orientación a Objetos

La principal razón para desarrollar sistemas informáticos OO sin lugar a dudas, son los beneficios de esta tecnología: aumento de la fiabilidad y productividad del desarrollador. La fiabilidad se puede mejorar debido a que cada objeto es simplemente “*una caja negra*” con respecto a objetos externos con los que debe comunicarse. Las estructuras de datos internas y métodos se pueden refinar sin afectar a otras partes de un sistema.

La productividad del desarrollador se puede mejorar debido a que las clases de objetos se pueden hacer reutilizables de modo que en cada subclase o instancia de un objeto se puede utilizar el mismo código de programa para la clase. Por otra parte, esta productividad también aumenta debido a que existe una asociación más natural entre objetos del sistema y objetos el mundo real.



El objeto como caja negra

Resumiendo, podemos decir que los principales beneficios de la OO son:

- Reutilización.
- Sistemas más fiables.
- Desarrollo más rápido.
- Desarrollo más flexible.
- Modelos que reflejan mejor la realidad.
- Resultados de alta calidad.
- Fácil mantenimiento.

Importante:

Cuando estamos desarrollando un sistema, el sistema se divide en clases. Cuando estamos ejecutando un sistema, existe un conjunto de objetos que tienen la responsabilidad de realizar las diferentes funciones del sistema.

Objetos



Antena Parabólica



Objeto Profesor



Helicóptero

$a + bi$

Número Complejo



Cuenta Bancaria



Automóvil

Apunte de Cátedra

Casi todo puede ser considerado un objeto. El dinero, el helicóptero, una bicicleta, los perros, el auto. Los objetos representan cosas, simples o complejas, reales o imaginarias. Una antena parabólica es un objeto complejo y real. Un objeto Profesor, representa los detalles y actividades de una persona, no es esa persona en sí misma, es pues, imaginario. Una frase, un número complejo, una receta y una cuenta bancaria también son representaciones de cosas intangibles. Todas son objetos.

Algunas cosas no son objetos, sino atributos, valores o características de objetos. Es decir, no todas las cosas son objetos, ni son consideradas normalmente como objetos. Algunas de ellas son simplemente atributos de los objetos como el color, el tamaño y la velocidad. Los atributos reflejan el estado de un objeto, la velocidad del objeto avión, o el tamaño de un objeto edificio. Normalmente no tiene sentido considerar la velocidad como un objeto.

Algunas de las propiedades que se pueden aplicar a los objetos, tal como se han definido:

- Los objetos son cosas.
- Los objetos pueden ser simples o complejos.
- Los objetos pueden ser reales o imaginarios.

El objeto mi auto

Para la discusión sobre objetos, se va a retomar el ejemplo de la introducción y utilizar la definición de auto (automóvil - coche) para definir algunos conceptos.

Para definir el objeto mi auto, hay que ser capaces de abstraer las funciones y atributos de un auto; es decir, hay que ser capaz de definir auto en términos de qué puede hacer y qué características lo distinguen de otros objetos.

Abstracción funcional

Funcionalmente, un auto puede realizar las siguientes acciones:

- Ir.
- Parar.
- Girar a la derecha.
- Girar a la izquierda.

Hay cosas que se saben que los autos hacen, pero el cómo lo hacen, la implementación de *ir*, *parar*, *girar* (a la derecha, a la izquierda) es irrelevante desde el punto del diseño. Esto es lo que se conoce como *abstracción funcional*.

Abstracción de datos

De forma semejante a la anterior, un auto tiene las siguientes características o atributos:

- Color.
- Velocidad.
- Tamaño.
- Precio.

La manera en que se almacenan o definen esos atributos, también es irrelevante para el diseño del objeto. Por ejemplo, el color puede definirse como la palabra *rojo*, o como un entero que representa el *número* (1 para rojo, 2 para azul, etc.), o como un *caracter* ('R' para rojo, 'A' para azul, etc.). La forma en que el objeto almacena el atributo color es irrelevante para el programador. Este proceso de preocupación de cómo se almacena el color es lo que se llama *abstracción de datos*.

Apunte de Cátedra

Encapsulación de objetos

Los objetos encapsulan sus operaciones y su estado, son islas de estado y comportamiento.

- El *comportamiento* del objeto está definido por las operaciones
- El estado está definido por los datos o atributos del objeto



Encapsulación es el término de orientación a objetos que describe la vinculación de operaciones y el estado a un objeto particular. La encapsulación está íntimamente relacionada con la ocultación de la información, definiendo qué partes de un objeto son visibles y qué partes están ocultas.

La encapsulación abarca a la ocultación de la información:

- Algunas partes son visibles.
- Otras partes son ocultas.

Por ejemplo: el volante de un auto representa una parte visible hacia el mecanismo de giro de un auto. La implementación del volante es oculta y sobre ella sólo puede actuar el propio volante.

¿Qué ocurre si cambia la implementación?.

Repitiendo, la encapsulación es la agrupación del estado y comportamiento para formar objetos, donde algunas partes son visibles mientras que otras permanecen ocultas. La ventaja de la ocultación de los detalles de implementación es que el objeto puede cambiar, y la visibilidad proporcionada puede ser compatible con el original. Entonces los programas que utilizaban el objeto pueden seguir funcionando sin alteración alguna.

Esto es extremadamente útil al modificar código (ya que se restringe la propagación de cambios).

También se fomenta la reusabilidad, ya que el código puede ser utilizado como una tecnología de caja negra (igual que los circuitos integrados en la industria electrónica). Y si se vuelve a pensar desde el punto de vista económico, esto viene a representar una ventaja de indudable valor.

Podemos decir entonces, que los objetos tienen determinadas características, que denominamos propiedades. Estas propiedades son los atributos y los métodos. Los atributos son características o cualidades o datos del objeto. Los métodos son el comportamiento propio del objeto (operaciones que el objeto sabe hacer o hace). Además, los atributos de un objeto sólo pueden ser accedidos y modificados por métodos del mismo (encapsulación).

Gráficamente podemos representar al objeto mi auto de la siguiente manera:

Apunte de Cátedra

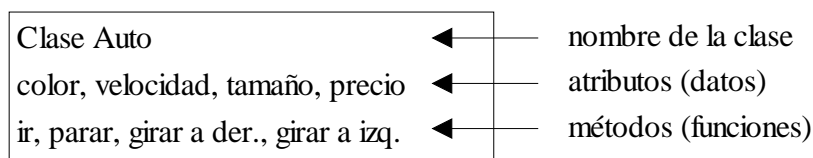
Objeto: mi auto
<p><u>Atributos:</u> color : <i>rojo</i> velocidad: <i>40</i> tamaño: <i>mediano</i> precio: <i>14000</i></p>
<p><u>Métodos:</u> ir parar girar a la derecha girar a la izquierda</p>

Algo último para rescatar, es que los principios de la definición de objetos ayudan a los programadores a hacer código más robusto, mantenible y seguro; porque se pueden aislar a cada uno de esos objetos y tratarlo como un ente único, con su propia personalidad, sin que haya cientos de características que tengamos que tener presentes. Lo cual, desde el punto de vista económico, que nunca hay que dejar de tener presente, sí resulta viable, ya que los objetos bien diseñados pueden ser utilizados en muy diversas aplicaciones, con lo cual el tiempo de desarrollo total se reduce.

Clases

Las clases representan el tipo de dato al que pertenecen los objetos, *ya que en un programa un objeto es una variable*. Una clase define todas las propiedades (atributos y métodos) que un objeto tendrá. Siguiendo el ejemplo del objeto mi auto, es obvio que la clase es Auto, ya que todos los autos tienen las mismas propiedades (color, velocidad, tamaño, precio, ... ir, parar, girar a la derecha o a la izquierda, ...).

Una clase define un miembro real o una entidad abstracta. Es el tipo o clasificación de datos. Una clase define el comportamiento y atributos de un grupo de objetos de características similares.



Un objeto es una instancia o variable de una clase. Se dice que pertenece a la clase. Un objeto se distingue de otros miembros de la clase por sus atributos.

Una clase describe un conjunto de objetos con un comportamiento común y atributos relacionados. Una clase define datos (atributos) y operaciones (métodos).

Apunte de Cátedra

Estructura de una clase

Formato general:

clase NombreDeLaClase

```
{
//declaración de atributos
tipo_de_dato nombre_del_atributo1
tipo_de_dato nombre_del_atributo2
tipo_de_dato nombre_del_atributo3
.
.
.
tipo_de_dato nombre_del_atributoN

//definición de métodos
tipo_de_dato nombre_del_metodo1(tipo_de_dato parámetro1, tipo_de_dato parámetro2, ...)
{
//cuerpo del método1
}

tipo_de_dato nombre_del_metodo2(tipo_de_dato parámetro1, tipo_de_dato parámetro2, ...)
{
//cuerpo del método2
}

tipo_de_dato nombre_del_metodo3(tipo_de_dato parámetro1, tipo_de_dato parámetro2, ...)
{
//cuerpo del método3
}
.
.
.

tipo_de_dato nombre_del_metodoN(tipo_de_dato parámetro1, tipo_de_dato parámetro2, ...)
{
//cuerpo del método N
}
}
```

Como puede apreciarse los métodos tienen el mismo formato que las funciones. De hecho, los métodos son funciones, es decir, tienen todas las propiedades de las mismas.

Convenciones:

1. El nombre de la clase va en forma completa y el comienzo de cada palabra en mayúscula. Ejemplo: NumerosEnteros.
2. El nombre de los métodos va en forma completa y el comienzo de cada palabra, menos la primera, en mayúscula. Ejemplo: sumarNumerosEnteros.
3. El tipo de dato va en minúscula. Ejemplo: entero.
4. El nombre de los atributos va en minúscula. Ejemplo: color.

Apunte de Cátedra

Veamos el ejemplo: Clase Auto.

```
clase Auto
{
    //declaración de atributos
    caracter color
    entero velocidad
    caracter tamaño
    real precio

    //definición de métodos
    ir()
        {
            escribir "Andando el auto"
            retornar
        }

    parar()
        {
            escribir "Deteniendo el auto"
            retornar
        }

    girarDerecha()
        {
            escribir "Girando a la derecha el auto"
            retornar
        }

    girarIzquierda()
        {
            escribir "Girando a la izquierda el auto"
            retornar
        }
}
```

Veamos ahora, un uso de la clase.

```
{
principal UsoClaseAuto
{
    //declaración de objetos
    Auto MI_AUTO, TU_AUTO

    //creación de los objetos
    MI_AUTO = nuevo Auto()
    TU_AUTO = nuevo Auto()

    //mensajes a los objetos
    MI_AUTO.ir()
}
```


Apunte de Cátedra

```
MI_AUTO.girarIzquierda()  
MI_AUTO.ir()  
MI_AUTO.girarDerecha()  
MI_AUTO.ir()  
MI_AUTO.parar()  
. .  
TU_AUTO.ir()  
TU_AUTO.parar()  
. .  
}  
}
```

En este ejemplo, se distingue en forma separada la declaración de la creación de los objetos.

En el caso de los objetos, a diferencia de los tipos de datos simples, el manejo de almacenamiento en memoria es distinto. Cuando se declara la variable, sólo se asigna un espacio en memoria para el identificador (MI_AUTO o TU_AUTO), y en el momento que se crea el objeto con la sentencia NUEVO se asigna el espacio en memoria necesario para almacenar los atributos (color, velocidad,...) del objeto que se está creando. En consecuencia, si un objeto no se crea, no se puede usar. En este caso, declaración y creación son diferentes.

Mensajes

Un mensaje es la forma disponible para solicitar a un objeto que ejecute o realice un método.

Formato general:

```
nombre_del_objeto.nombre_del_método(variable1, variable2, ...)
```

Por ejemplo:

```
MI_AUTO.ir()  
MI_AUTO.girarIzquierda()
```

En el caso de que el método al que se le está enviando un mensaje requiera parámetros, el mensaje debe cumplir con la correspondencia de parámetros (número, orden y tipo) con el método.

Constructor

El constructor es un método que tienen todas las clases, aunque no se implemente. Este método es el que se ejecuta, cuando se crea el objeto con la sentencia **nuevo**. Básicamente, la tarea del constructor es la asignación de espacio en memoria adecuado para el objeto que se está creando y la inicialización con valores a los atributos del objeto.

Si el constructor se implementa debe llevar *siempre* el mismo nombre de la clase, puede tener o no parámetros y *nunca* devuelve nada.

Apunte de Cátedra

Veamos un ejemplo con la clase Auto:

```
clase Auto
{
  //declaración de atributos
  caracter color
  entero velocidad
  caracter tamaño
  real precio

  //constructor
  Auto(caracter COL, entero VEL, caracter TAM, real PRE)
  {
    color ← COL
    velocidad ← VEL
    tamaño ← TAM
    precio ← PRE
  }
  //definición de métodos
  ir()
  {
    escribir "Andando el auto"
    retornar
  }

  parar()
  {
    escribir "Deteniendo el auto"
    retornar
  }

  girarDerecha()
  {
    escribir "Girando a la derecha el auto"
    retornar
  }

  girarIzquierda()
  {
    escribir "Girando a la izquierda el auto"
    retornar
  }
}
```

Como vemos, cuando se crea una instancia de la clase Auto, lo primero que se ejecuta es el método Constructor llamado Auto.

Si el método constructor no se implementa, sólo se asignará espacio en memoria adecuado para el objeto que se está creando y se asignan valores por defectos a los atributos (0 para los enteros, 0.0 para los reales, espacio vacío para los caracteres y falso para los booleanos). Por lo tanto, la creación del objeto ahora es diferente, veamos el siguiente ejemplo de constructor:

Apunte de Cátedra

```
{  
principal EjemploConstructor  
  {  
    //declaración de variables  
    caracter COLO, TAMA  
    entero VELO  
    real PREC  
  
    //declaración del objeto  
    Auto MI_AUTO, TU_AUTO  
  
    //carga de los datos  
    escribir "Ingrese el color del auto"  
    leer COLO  
    escribir "Ingrese la velocidad del auto"  
    leer VELO  
    escribir "Ingrese el tamaño del auto"  
    leer TAMA  
    escribir "Ingrese el precio del auto"  
    leer PREC  
  
    //creación del objeto  
    MI_AUTO = nuevo Auto(COLO, VELO, TAMA, PREC)  
  
    //mensajes a los objetos  
    MI_AUTO.ir()  
    MI_AUTO.girarIzquierda()  
    MI_AUTO.ir()  
    MI_AUTO.girarDerecha()  
    MI_AUTO.ir()  
    MI_AUTO.parar()  
    .  
    .  
    .  
  }  
}
```

Acceso a los atributos

Retomando el ejemplo de la clase Auto, cuyos atributos son color, velocidad, tamaño y precio, y cuyos métodos son ir(), parar(), girarDerecha() y girarIzquierda(), supongamos el caso en que un objeto auto en particular como MI_AUTO se desee vender, para ello será necesario conocer el precio. Puede ocurrir también que el auto aumente su valor, por consecuencia necesitaremos cambiar el precio del mismo. Si el auto adquiere otro color de pintura, entonces será necesario cambiarle el color. Y así, realizar ciertas operaciones que requieran acceder a los atributos. Pero con los métodos que tenemos no podemos hacerlo, necesitamos una forma que nos permita modificar los valores de los atributos, como así también obtener los valores.

Dado que los datos de un objeto no se pueden acceder directamente, es necesario tener métodos de acceso a los atributos.

Apunte de Cátedra

Por ejemplo:

```
real obtenerPrecio()  
{  
  retornar precio  
}  
  
modificarPrecio(real PREC)  
{  
  precio ← PREC  
  retornar  
}
```

Estos dos métodos aseguran que sólo el objeto accede a sus datos.

Por otra parte, si tenemos lo siguiente:

```
Real obtenerPrecio()  
{  
  retornar precio  
}  
  
mostrarPrecio()  
{  
  escribir precio  
  retornar  
}
```

La diferencia radica en que, a través del primer método, el objeto devuelve el valor del atributo precio y en el principal, mediante una variable (que toma ese valor), se puede realizar todas las operaciones que desee, como hacer cálculos, comparar con otros precios, e inclusive mostrar el valor. Esta última operación la realiza el segundo método, que sólo muestra el precio y no da la posibilidad de manipularlo.

Ámbito de un identificador

Si tenemos una clase A, que posee un identificador *ident* declarado de diferentes formas (atributo, variable o parámetro) en la clase, el identificador *ident* representará distintos datos según el lugar que esté referenciando dentro de la clase. Es decir, se está designando a diferentes datos, de acuerdo a las reglas de ámbito.

Así mismo, puesto que usar los mismos nombres de identificador para designar atributos, variables y parámetros, pueden ocasionar confusión; es importante tener en cuenta las reglas de ámbito de identificadores:

- Un atributo puede ser utilizado en cualquier método de la clase.
- Si en el método M existe declarada la variable *ident* y en la misma clase existe el atributo *ident*, al referenciar en M al identificador *ident*, se estará haciendo referencia a la variable y no al atributo.
- Si en el método M existe declarado un parámetro *ident* y en la misma clase existe el atributo *ident*, al referenciar en M al identificador *ident*, se estará haciendo referencia al parámetro y no al atributo.

Apunte de Cátedra

- Las variables y parámetros declarados en el método M son locales al método, por lo tanto, no pueden usarse fuera del mismo.
- En el método M no pueden declararse variables locales y parámetros con el mismo nombre de identificador.

clase A

```
{
  entero ident1
  real ident2

  A()
  {
    ident1... //se referencia al atributo ident1 de la clase
    ident2... //se referencia al atributo ident2 de la clase
  }

  metodoUno()
  {
    ident1... //se referencia al atributo ident1 de la clase
  }

  metodoDos(entero ident1)
  {
    ident1... //se referencia al parámetro ident1 del metodoDos
    ident2... //se referencia al atributo ident2 de la clase
  }

  metodoTres()
  {
    entero ident1
    ident1... //se referencia a la variable local ident1 del metodoTres
    ident2... //se referencia al atributo ident2 de la clase
  }

  metodoCuatro(real ident2)
  {
    entero ident1
    ident1... //se referencia a la variable local ident1 del metodoCuatro
    ident2... //se referencia al parámetro ident2 del metodoCuatro
  }
}
```

Reuso del Código

Un tipo de dato se compone de un conjunto de valores y una serie de operaciones permitidas que se pueden aplicar a esos datos. Así tenemos que cuando declaramos una variable del tipo de dato *entero*, esta podrá asumir un valor de un rango previamente establecido y sobre la misma se podrán aplicar determinadas operaciones (suma, resta, modulo, división, producto, asignación, comparaciones). A diferencia de una variable de tipo de dato *caracter*, que asume otros valores y otras operaciones.

Apunte de Cátedra

La gran ventaja de trabajar con tipos de datos, es que el programador solo debe saber:

- El conjunto de valores que puede asumir.
- Las operaciones permitidas.
- Y desconoce totalmente todo detalle de implementación interna. Es decir, como se representa internamente al entero o como realiza la adición, es algo que el programador no necesita saber para usar variables de tipo entero. Es decir, los detalles internos están encapsulados.
- Por otro lado, al tener el tipo de dato entero definido puede usarlo en tantos programas como desea y necesita.

Lo mismo ocurre con las clases. Las clases son tipos de datos que crea el programador. Una vez que la clase esta programada y probada, podrá ser usada en muchos programas sin necesidad que quién la usa, conozca sus detalles internos.

Clase Vector

Una clase define datos (atributos) y operaciones (métodos). Así podemos tener una clase Vector, que represente a un arreglo de una dimensión y ofrezca operaciones tales como: agregar un elemento, buscar un elemento, mostrar el vector, etc. La ventaja sería que los programas que usen esta clase, por ejemplo no deberán manejar índices, testeos, bucles y otras operaciones típicas de vectores.

Veamos el formato básico de la clase Vector.

clase Vector

```
{
    entero [ ] elementos           //elementos del vector
    entero dimension, actual       //tamaño asignado por el usuario y tamaño real del vector

    //constructor para la clase vector
    Vector(entero TAMANO)
    {
        si (TAMANO <= 0)
            dimension ← 100
        sino
            dimension ← TAMANO
        actual ← 0
        elementos = nuevo entero [dimension]
    }

    //método que agrega de un elemento
    //precondición: elemento entero
    //postcondición: valor agregado si hay lugar en el vector
    agregarElemento(entero VALOR)
    {
        si (actual < dimension)
        {
            elementos[actual] ← VALOR
            actual ← actual + 1
        }
        retornar
    }
}
```

Apunte de Cátedra

```
//método que muestra el contenido del vector
//precondición: vector con elementos
//postcondición: muestreo de los elementos del vector
mostrarVector()
{
    entero CONT
    para (CONT ← 0; CONT < actual; CONT ← CONT + 1)
        escribir elementos[CONT]
    retornar
}
```

La diferencia entre los atributos *dimension* y *actual*, es cuánto se va a reservar de memoria para guardar el vector (*dimension*) y cuánto se puede de esa cantidad usar en realidad (*actual*). Esto se debe a la flexibilidad del método de agregado de elementos, que va de uno en uno.

Como se ha dicho anteriormente, una función (método) no puede realizar lecturas dentro de ella. Por lo tanto, es necesario contar con un método que incorpore los elementos de a uno. De allí surge el *agregarElementos(entero)*. Una variante del método *agregarElementos(entero)* es para los casos donde el atributo actual supere el tamaño del vector. Sería incorporar una variable de tipo booleano que avise si se ha cargado o no el elemento. Si bien este método puede ser útil, el mismo está realizando más de una acción, desventaja para el método.

```
booleano agregarElementos(entero VALOR)
{
    booleano AGREGO
    si (actual < dimension)
        {
            elementos[actual] ← VALOR
            actual ← actual + 1
            AGREGO ← verdadero
        }
    sino
        AGREGO ← falso
    retornar AGREGO
}
```

Hay que tener en cuenta que la clase presentada sólo se aplica a los vectores cuyos elementos son enteros, habría que adaptarla al resto de los tipos de datos posibles. En consecuencia, debería ser clases distintas y los nombres ayudarían para ver la diferencia, por ejemplo: *VectorEnteros*, *VectorCaracteres*, etc.

La ventaja de crear la clase *Vector* sería que los programas que la usen, no deberán manejar operaciones inherentes a los vectores, sólo usarlos. A continuación se muestra un programa que hace uso de la clase *Vector*, creando dos instancias de la misma, de igual dimensión.

```
{
principal UsaClaseVector
{
    //declaración de variables
```

Apunte de Cátedra

Vector OBJECT1, OBJECT2
entero DIM, CONT, ELEM

//carga de la dimensión

escribir "Ingrese la dimensión de los vectores"

leer DIM

//creación del primer vector

OBJECT1 = **nuevo** Vector(DIM)

//creación del segundo vector

OBJECT2 = **nuevo** Vector(DIM)

//carga del primer vector en su totalidad

escribir "Carga de elementos del primer vector"

para (CONT ← 0; CONT < DIM; CONT ← CONT + 1)

{

escribir "Ingresar un elemento"

leer ELEM

OBJECT1.agregarElementos(ELEM)

}

//carga del segundo vector en su totalidad

escribir "Carga de elementos del segundo vector"

para (CONT ← 0; CONT < DIM; CONT ← CONT + 1)

{

escribir "Ingresar un elemento"

leer ELEM

OBJECT2.agregarElementos(ELEM)

}

//muestra del primer vector

escribir "Primer vector resultante"

OBJECT1.mostrarVector()

//muestra del segundo vector

escribir "Segundo vector resultante"

OBJECT2.mostrarVector()

}

}

En este programa *UsaClaseVector* hemos usado la clase Vector como un tipo de datos mas, como el tipo entero. La carga y muestra de los elementos no se codifican en cada programa donde se usan vectores, sino que estas operaciones al estar incluidas como métodos en la clase Vector y estos programas al trabajar con objetos de la misma clase, reusan el código ya escrito en la misma. Esta es una de las formas de Reusar Código, principal característica de la Programación Orientada a Objetos.

Tanto la carga como la muestra del vector, son actividades constantes en el uso de los mismos. Pero, existen otras operaciones que varían de acuerdo al problema a desarrollar. Por ejemplo, una vez cargado el vector se puede hallar la suma de sus elementos. Veamos el método con dicha operación que deberá agregado a la clase Vector.

Apunte de Cátedra

```
//suma de los elementos del vector
//precondición: vector con elementos enteros
//postcondición: suma entera de los elementos del vector
entero sumarElementos()
{
    entero SUMA, IND
    SUMA ← 0
    para (IND ← 0; IND < actual; IND ← IND + 1)
        SUMA ← SUMA + elementos[IND]
    retornar SUMA
}
```

En este caso, con el bucle vamos recorriendo el vector, accediendo a cada elemento del mismo mediante el índice (IND), y acumulando los elementos en la variable SUMA.

Podemos incluir en el programa *UsaClaseVector* como un objeto vector suma sus elementos.

```
SUMATORIA ← OBJECT1.sumarElementos()
escribir "El resultado de la suma de los elementos del primer vector es", SUMATORIA
```

❖ Ejemplos.

Problema: Agregar a la clase vector un método que cuente cuántos elementos del vector son múltiplos de 5.

```
clase Vector
{
    .
    .
    .

    //cantidad de múltiplos por 5
    //precondición: vector con elementos enteros
    //postcondición: cantidad entera de los elementos múltiplos de 5 del vector
    entero contarMultiplosCinco()
    {
        entero CANT, POS
        CANT ← 0
        POS ← 0
        hacer
        {
            si (elementos[POS] % 5 == 0)
                CANT ← CANT + 1
        }
        mientras (POS < actual)
        retornar CANT
    }
}
```

Apunte de Cátedra

Problema: A partir de un vector de caracteres ya cargado, reemplazar las vocales por el caracter '*'. También a una posición determinada asignarle un espacio vacío. Probar la clase.

clase VectorCaracteres

```
{
    caracter [ ] elementos
    entero dimension, actual

    VectorCaracteres(entero DIM)
    {
        .
        .
        .
        elementos = nuevo caracter [dimension]
    }

    agregarElementos(caracter ELEM)
    {
        .
        .
        .
    }

    //reemplazo de vocales
    //precondición: vector con elementos caracteres
    //postcondición: vector modificado con las vocales reemplazadas por '*'
    reemplazarVocales()
    {
        entero IND
        para (IND ← 0; IND < actual; IND ← IND + 1)
            si ((elementos[IND] >= 'a' & elementos[IND] <= 'z') |
                (elementos[IND] >= 'A' & elementos[IND] <= 'Z'))
                elementos[IND] ← '*'
        retornar
    }

    //reemplazo de una posición determinada
    //precondición: vector con elementos caracteres y posición entera válida
    //postcondición: vector modificado en una posición determinada por espacio vacío
    reemplazarPosicion(entero POS)
    {
        si (POS >= 0 & POS < actual)
            elementos[POS] ← ' '
        retornar
    }
}

principal UsaVectorCaracteres
{
```

Apunte de Cátedra

```
//declaración de variables
Vector OBJECT
entero DIM, CONT, ELEM, POS

//carga de la dimensión
escribir "Ingrese la dimensión del vector"
leer DIM

//creación del vector
OBJECT = nuevo VectorCaracteres(DIM)

//carga del vector en su totalidad
escribir "Carga del vector"
para (CONT ← 0; CONT < DIM; CONT ← CONT + 1)
{
    escribir "Ingrese un elemento al vector"
    leer ELEM
    OBJECT.agregarElementos(ELEM)
}

//reemplazo de vocales por '*'
OBJECT.reemplazarVocales()

//muestra del vector
escribir "Vector resultante luego de reemplazar las vocales por '*'"
OBJECT.mostrarVector()

//reemplazo de una determinada posición
escribir "Ingrese una posición que se reemplazará por un espacio vacío"
leer POS
OBJECT.reemplazarPosicion(POS)

//muestra del vector
escribir "Vector resultante luego de reemplazar una determinada posición por espacio vacío"
OBJECT.mostrarVector()
}
}
```

Dos detalles importantes que se están descuidando son: ¿qué ocurre si se ingresa un valor incorrecto como dimensión desde el principal? ¿qué ocurre con el vector original luego de realizar el primer reemplazo?. *¡¡¡Analizarlo y actualizar si es necesario!!!*

Es importante aclarar que la clase debe proteger a sus atributos y sus métodos no deben fallar, por lo tanto, se debe realizar la mayor cantidad de testeos o validaciones posibles dentro de la clase. Esto no quiere decir que en el principal no puedan realizarse testeos. Pero la realidad es que existe una sola clase y varios principales que la usan, por lo tanto, es muy difícil asegurarnos de que todos los principales hagan lo mismo.

Clase Matriz

De la misma manera que podemos crear la clase Vector, también podemos contar con una clase Matriz que contenga todo lo referido a un arreglo de dos dimensiones.

Apunte de Cátedra

Veamos el formato básico de la clase Matriz.

clase Matriz

```
{
    entero [ ][ ] elementos //elementos de la matriz
    entero dimensionfila, dimensioncolumna //tamaño asignado por el usuario
    entero actualfila, actualcolumna //tamaño real de la matriz

    //constructor para la clase matriz
    Matriz(entero DIMF, entero DIMC)
    {
        si (DIMF <= 0 | DIMC <= 0)
        {
            dimensionfila ← 9
            dimensioncolumna ← 11
        }
        sino
        {
            dimensionfila ← DIMF
            dimensioncolumna ← DIMC
        }
        actualfila ← 0
        actualcolumna ← 0
        elementos = nuevo entero [dimensionfila][dimensioncolumna]
    }

    //método que agrega de un elemento
    //precondición: elemento entero
    //postcondición: valor agregado si hay lugar en la matriz
    agregarElementos(entero VALOR)
    {
        elementos[actualfila][actualcolumna] ← VALOR
        si (actualcolumna == dimensioncolumna - 1)
        {
            actualcolumna ← 0
            actualfila ← actualfila + 1
        }
        sino
        actualcolumna ← actualcolumna + 1
        si (actualfila == dimensionfila)
        actualcolumna ← dimensioncolumna
        retornar
    }

    //método que muestra el contenido de la matriz
    //precondición: matriz con elementos
    //postcondición: muestreo de los elementos de la matriz
    mostrarMatriz()
    {
```

Apunte de Cátedra

```

entero FIL, COL
para (FIL ← 0; FIL < actualfila; FIL ← FIL + 1)
    para (COL ← 0; COL < actualcolumna; COL ← COL + 1)
        escribir elementos[FIL][COL]
    retornar
}
}

```

Lo mismo que se ha visto en la clase Vector, este formato es para matrices con elementos enteros, por lo tanto, si los elementos fueran de otro tipo, deberíamos adaptar la matriz, como por ejemplo: *MatrizEnteros*, *MatrizCaracteres*, etc. Además, como se puede apreciar, la clase trabaja con una matriz no cuadrada. Sabemos muy bien que existen operaciones que sólo pueden ser aplicadas a matrices cuadradas, por lo tanto, para qué tener que controlar esto en cada método. Entonces, sería recomendable contar con otra clase que sólo trabaje con este tipo de matrices. De esta manera, no es necesario contar con dos dimensiones como atributos de la matriz. Así, nos quedarían ejemplos como: *MatrizNoCuadradaEnteros*, *MatrizNoCuadradaCaracteres*, *MatrizCuadradaEnteros*, *MatrizCuadradaCaracteres*, etc.

❖ Ejemplos.

Problema: Multiplicar los elementos de una matriz.

```

clase Matriz
{
    entero [ ][ ] elementos
    entero dimensionfila, dimensioncolumna
    entero actualfila, actualcolumna
    .
    .
    .

    //producto de los elementos de la matriz
    //precondición: matriz con elementos enteros
    //postcondición: producto entero de los elementos de la matriz
    entero multiplicarElementos()
    {
        entero PRODUCTO, POSF, POSC
        PRODUCTO ← 1
        para (POSF ← 0; POSF < actualfila; POSF ← POSF + 1)
            para (POSC ← 0; POSC < actualcolumna; POSC ← POSC + 1)
                PRODUCTO ← PRODUCTO * elementos[POSF][POSC]
            retornar PRODUCTO
        }
    }
}

```

Problema: Hacer un método que imprima los elementos de una columna determinada.

```

//muestra de una determinada columna de la matriz
//precondición: matriz con elementos y número de columna válido

```

Apunte de Cátedra

```
//postcondición: muestreo de los elementos enteros de la columna ingresada
mostrarColumna(entero COLMOS)
{
    entero FIL
    si (COLMOS >= 0 & COLMOS < actualcolumna)
        para (FIL ← 0; FIL < actualfila; FIL ← FIL + 1)
            escribir elementos[FIL][COLMOS]
    retornar
}
```

Problema: Dada una matriz de números reales generar un vector donde cada elemento del mismo es el elemento más grande de cada columna de la matriz. Realizar el programa que pruebe la clase.

Veamos como ejemplo la siguiente Matriz:

3.4	6.0	4.6	8.8
0.9	4.9	1.1	3.9
4.3	0.1	7.4	8.0

El resultado sería el siguiente Vector:

4.3	6.0	7.4	8.8
-----	-----	-----	-----

clase Matriz

```
{
    real [ ][ ] elementos
    entero dimensionfila, dimensioncolumna
    entero actualfila, actualcolumna

    Matriz(entero DIMF, entero DIMC)
    {
        .
        .
        .
        elementos = nuevo real [dimensionfila][dimensioncolumna]
    }

    agregarElementos(real VALOR)
    {
        .
        .
        .
    }

    //generación del vector con los elementos mayores de cada columna
    //precondición: matriz con elementos reales
    //postcondición: vector con los elementos mayores de cada columna de la matriz
    real [ ] elementosMayoresColumnas()
    {
        entero POSF, POSC
        real MAYOR
        real [ ] VECTOR = nuevo real [actualcolumna]
```

Apunte de Cátedra

```

    para (POSC ← 0; POSC < actualcolumna; POSC ← POSC + 1)
    {
        MAYOR ← elementos[0][POSC]
        para (POSF ← 1; POSF < actualfila; POSF ← POSF + 1)
            si (elementos[POSF][POSC] > MAYOR)
                MAYOR ← elementos[POSF][POSC]
        VECTOR[POSC] ← MAYOR
    }
    retornar VECTOR
}

{
principal UsaClaseMatriz
{
//declaración de variables
Matriz OBJMAT
entero FILAS, COLUMNAS
real [ ] VECTOR
entero CONTF, CONTC, CONT
real ELEM

//carga de las dimensiones
escribir “Ingrese la cantidad de filas para la matriz”
leer FILAS
escribir “Ingrese la cantidad de columnas para la matriz”
leer COLUMNAS

//creación de la matriz
OBJMAT = nuevo Matriz(FILAS, COLUMNAS)

//carga de la matriz en su totalidad
escribir “Carga de los elementos de la matriz”
para (CONTF ← 0; CONTF < FILAS; CONTF ← CONTF + 1)
    para (CONTC ← 0; CONTC < COLUMNAS; CONTC ← CONTC + 1)
        {
            escribir “Ingrese un elemento para la posición”, CONTF, “ y ”, CONTC
            leer ELEM
            OBJMAT.agregarElementos(ELEM)
        }

//muestra de la matriz
escribir “La matriz resultante es”
OBJMAT.mostrarMatriz()

//creación y carga del vector para mostrar los elementos mayores de cada columna de la matriz
VECTOR = nuevo real [COLUMNAS]
VECTOR ← OBJMAT.elementosMayoresColumnas()

```

Apunte de Cátedra

```
//muestra del vector
escribir "Vector con los elementos más grandes de cada columna de la matriz"
para (CONT ← 0; CONT < COLUMNAS; CONT ← CONT + 1)
    escribir VECTOR[CONT]
}
```

En este caso, la clase *Matriz* contiene el método *elementosMayoresColumnas()* que devuelve un vector. Esto no quiere decir que la clase *matriz* deba contener métodos que manipulen dicho vector, como por ejemplo, mostrar el vector. La clase *Matriz*, sólo maneja cuestiones relacionadas a matrices. Es por eso, que en este caso, el principal se hace cargo de crear un vector vacío, obtener el vector resultante y mostrarlo. Este proceso se puede mejorar haciendo un buen uso de los objetos. Es decir, el principal, crearía un objeto (instancia) de la clase *Matriz* y un objeto de la clase *Vector*, entonces la tarea de mostrar el vector, la tendría la clase *Vector* y no el principal.

Recordemos que hasta el momento estamos trabajando con una sola clase (es decir, el principal sólo crea objetos de una sola clase), pero cuando el problema a resolver sea aún mayor, seguramente nos veremos obligados a trabajar con más de una clase. Esto ocurre normalmente con los *Sistemas*.

LENGUAJE DE PROGRAMACION: JAVA

Java fue creado por la empresa Sun Microsystem en el año 1995. En sus comienzos fue pensado para pequeños procesadores incorporados en electrodomésticos (lavarropas, televisores, etc.) pero dicho proyecto fracasó. Con el auge de Internet rápidamente se noto un importante problema: las diferentes plataformas o arquitecturas (hardware y sistemas operativos). Por lo tanto, Sun se propone crear un lenguaje de programación cuyos programas se ejecutaran sin problemas en diferentes plataformas y de propósito general pero con especial énfasis en el uso de redes.

Java es un lenguaje que tiene muchísimas ventajas pero la más importante es que esta en permanente evolución. Las características principales que nos ofrece Java respecto a cualquier otro lenguaje de programación, son:

Es SIMPLE

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ es un lenguaje que adolece de falta de seguridad, pero C y C++ son lenguajes más difundidos, por ello, Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos, entre las que destacan:

- aritmética de punteros.
- no existen referencias.
- registros (struct).
- definición de tipos (typedef).
- macros (#define).
- necesidad de liberar memoria (free).

Además, el intérprete completo de Java que hay en este momento es muy pequeño, solamente ocupa 215 Kb de RAM.

Es ORIENTADO A OBJETOS

Java implementa la tecnología OO con algunas mejoras y elimina algunas cosas para mantener el objetivo de la simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo. Las plantillas de objetos son llamadas clases y sus copias, instancias. Estas instancias, necesitan ser construidas y destruidas en espacios de memoria.

Es DISTRIBUIDO

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como http y ftp. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

Es ROBUSTO

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria. También implementa los arrays (arreglos) auténticos, en vez de listas enlazadas de punteros, con comprobación de límites, para evitar la posibilidad de sobrescribir o

Apunte de Cátedra

corromper memoria, resultado de punteros que señalan a zonas equivocadas. Estas características reducen drásticamente el tiempo de desarrollo de aplicaciones en Java.

Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los byte-codes, que son el resultado de la compilación de un programa Java. Es un código de máquina virtual que es interpretado por el intérprete Java. No es el código máquina directamente entendible por el hardware, pero ya ha pasado todas las fases del compilador: análisis de instrucciones, orden de operadores, etc., y ya tiene generada la pila de ejecución de órdenes.

Entonces Java proporciona:

- Comprobación de punteros.
- Comprobación de límites de arrays.
- Excepciones.
- Verificación de byte-codes.

Es PORTABLE

Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.

Es SEGURO

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el casting implícito que hacen los compiladores de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un navegador, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El código Java pasa muchos tests antes de ejecutarse en una máquina. El código se pasa a través de un verificador de byte-codes que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal (código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto).

Si los byte-codes pasan la verificación sin generar ningún mensaje de error, entonces sabemos que:

- El código no produce desbordamiento de operandos en la pila.
- El tipo de los parámetros de todos los códigos de operación son conocidos y correctos.
- No ha ocurrido ninguna conversión ilegal de datos, tal como convertir enteros en punteros.
- El acceso a los campos de un objeto se sabe que es legal: public, private, protected.
- No hay ningún intento de violar las reglas de acceso y seguridad establecidas.

El Cargador de Clases también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de ficheros local, del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya, ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.

En resumen, las aplicaciones de Java resultan extremadamente seguras, ya que no acceden a zonas delicadas de memoria o del sistema, con lo cual evitan la interacción de ciertos virus. Java no posee una semántica específica para modificar la pila de programa, la memoria libre o utilizar objetos y métodos de un programa sin los privilegios del kernel del sistema operativo. Además, para evitar modificaciones por parte de los crackers de la red, implementa un método ultraseguro de autenticación por clave pública. El Cargador de Clases puede verificar una firma digital antes de

Apunte de Cátedra

realizar una instancia de un objeto. Por lo tanto, ningún objeto se crea y almacena en memoria, sin que se validen los privilegios de acceso. Es decir, la seguridad se integra en el momento de compilación, con el nivel de detalle y de privilegio que sea necesario.

Dada la concepción del lenguaje y si todos los elementos se mantienen dentro del estándar marcado por Sun, no hay peligro. Java imposibilita, también, abrir ningún fichero de la máquina local (siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el applet), no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar nuestra máquina para hacer peticiones o realizar operaciones con otra. Además, los intérpretes que incorporan los navegadores de la Web son aún más restrictivos. Bajo estas condiciones (y dentro de la filosofía de que el único ordenador seguro es el que está apagado, desenchufado, dentro de una cámara acorazada en un bunker y rodeado por mil soldados de los cuerpos especiales del ejército), se puede considerar que Java es un lenguaje seguro y que los applets están libres de virus.

Respecto a la seguridad del código fuente, no ya del lenguaje, JDK (Java Developers Kit: Paquete de Desarrollo para Java) proporciona un desensamblador de byte-code, que permite que cualquier programa pueda ser convertido a código fuente, lo que para el programador significa una vulnerabilidad total a su código. Utilizando *javap* no se obtiene el código fuente original, pero sí desmonta el programa mostrando el algoritmo que se utiliza, que es lo realmente interesante. La protección de los programadores ante esto es utilizar llamadas a programas nativos, externos (incluso en C o C++) de forma que no sea descompilable todo el código; aunque así se pierda portabilidad. Esta es otra de las cuestiones que Java tiene pendientes.

Es INTERPRETADO

El intérprete Java (sistema run-time) puede ejecutar directamente el código objeto. Enlazar (linkar) un programa, normalmente, consume menos recursos que compilarlo, por lo que los desarrolladores con Java pasarán más tiempo desarrollando y menos esperando por el ordenador. No obstante, el compilador actual del JDK es bastante lento. Por ahora, que todavía no hay compiladores específicos de Java para las diversas plataformas, Java es más lento que otros lenguajes de programación, como C++, ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional.

La verdad es que Java para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado como compilado. Y esto no es ningún contrasentido; el código fuente escrito con cualquier editor se compila generando el byte-code. Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones máquinas propias de cada plataforma y no tiene nada que ver con el p-code de Visual Basic. El byte-code corresponde al 80% de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma. Para ello, hace falta el run-time, que sí es completamente dependiente de la máquina y del sistema operativo, que interpreta dinámicamente el byte-code y añade el 20% de instrucciones que faltaban para su ejecución. Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el run-time correspondiente al sistema operativo utilizado.

Es de ARQUITECTURA NEUTRAL

Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado. Actualmente existen sistemas run-time para Solaris 2.x, SunOs 4.1.x, Windows 95, Windows NT, Windows 2000, Windows XP, Linux, Irix, Aix, Mac, Apple y probablemente haya grupos de desarrollo trabajando en el porting a otras plataformas.

Apunte de Cátedra

El código fuente Java se “compila” a un código de bytes de alto nivel independiente de la máquina. Este código (byte-codes) está diseñado para ejecutarse en una máquina hipotética que es implementada por un sistema run-time, que sí es dependiente de la máquina.

Es MULTITHREADED

Al ser multithreaded (multihilo), Java permite muchas actividades simultáneas en un programa. Los threads (a veces llamados, procesos ligeros) son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los threads construidos en el lenguaje, son más fáciles de usar y más robustos.

El beneficio de ser multithreaded consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.) aún supera a los entornos de flujo único de programa (single-threaded) tanto en facilidad de desarrollo como en rendimiento.

Cualquiera que haya utilizado la tecnología de navegación concurrente, sabe lo frustrante que puede ser esperar por una gran imagen que se está abriendo. En Java, las imágenes se pueden ir abriendo en un thread independiente, permitiendo que el usuario pueda acceder a la información en la página sin tener que esperar por el navegador.

Es DINAMICO

Java se beneficia todo lo posible de la tecnología orientada a objetos. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librerías API (Application Programming Interface: Interfaz de Programación de Aplicaciones) nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que mantengan el API anterior).

Java también simplifica el uso de protocolos nuevos o actualizados. Si su sistema ejecuta una aplicación Java sobre la red y encuentra una pieza de la aplicación que no sabe manejar, tal como se ha explicado en párrafos anteriores, Java es capaz de bajar automáticamente cualquiera de esas piezas que el sistema necesita para funcionar.

Java, para evitar que los módulos de byte-codes o los objetos o nuevas clases, haya que estar bajándolos de la red cada vez que se necesiten, implementa las opciones de persistencia, para que no se eliminen cuando se limpie la caché de la máquina.

De Pseudocódigo a Java

A continuación veremos como sería el traspaso de todo lo visto hasta el momento, hecho en pseudocódigo, al Lenguaje de Programación Java.

Tipos de Datos Simples

Pseudocódigo	Java	Rango	Descripción
entero	int	32-bit complemento a 2	Entero
entero	byte	8-bit complemento a 2	Entero de 1 byte
entero	short	16-bit complemento a 2	Entero corto
entero	long	64-bit complemento a 2	Entero largo
real	float	32-bit IEEE 754	Coma flotante de precisión simple
real	double	64-bit IEEE 754	Coma flotante de precisión doble
caracter	char	16-bit Caracter	Un sólo caracter
booleano	boolean	true o false	Un valor booleano (verdadero o falso)

Declaración de variables

Formato general:

Apunte de Cátedra

tipo_de_dato nombre_variable;

Por ejemplo:

```
char LETRA;  
double MONTO;  
int NUM, CONT;  
boolean BANDERA;
```

El punto y coma (;) es un separador de sentencias. En Java, como en otros lenguajes, hay que indicar explícitamente cuando termina una sentencia. Esto es así, porque no trata como especiales los cambios de línea.

Asignación de variables

Formato general:

nombre_variable = valor;

Por ejemplo:

```
NUM=12;  
CONT=CONT + 1;  
LETRA='S'
```

Declaración de constantes

Formato general:

final tipo_de_dato nombre_constante = valor;

Por ejemplo:

```
final int TOPE=100;  
final long PI=3.14;
```

Entrada de datos

Formato general:

nombre_variable = **Console**.readNombre_del_tipo_de_dato();

Por ejemplo:

```
De entero:   VAR1=Console.readInt(); //VAR1 es una variable entera  
De double:  VAR2=Console.readDouble(); //VAR2 es una variable double  
De cadenas: VAR3=Console.readString(); //VAR3 es una variable cadena de caracteres
```

Salida de datos

Formato general:

```
System.out.print(lista_de_expresiones_de_salida y/o "mensaje_de_salida");  
System.out.println(lista_de_expresiones_de_salida y/o "mensaje_de_salida");
```

Por ejemplo:

```
System.out.println("Vector resultante");           //se verá el mensaje escrito  
System.out.println(RESULTADO);                   //se verá el contenido de la variable RESULTADO  
System.out.println("El resultado de la suma es: "+SUMA); //para mostrar un mensaje y una  
System.out.print("El resultado de la suma es: ");  
System.out.print(SUMA);                           //produce el mismo resultado anterior
```

Apunte de Cátedra

//sacando *ln* para el salto de línea

Tipo de dato de los resultados

Se ha visto anteriormente los resultados de aplicar los distintos operadores aritméticos con los tipos de datos simples.

Así por ejemplo:

```
int A, B, C;
double D;
A=5;
B=2;
C=A / B;    //a la variable C se le asigna 2.
D=A / B;    //a la variable D se le asigna 2.0 y no 2.5.
```

En todos estos casos, el lenguaje automáticamente ha realizado una conversión de datos, por lo general en el resultado de la operación. Esta operación se conoce como Cast Implícito.

Para que D sea 2.5 antes hay que realizar una conversión para A y/o B en double, esto se llama Cast Explícito y lo realiza el programador usando (*tipo_de_dato*). Veamos como:

```
D=(double) A / (double) B;    //los valores de A y B se convierten en double antes de efectuar
                               //la división, a D se le asigna 2.5
D=(double) A / B;            //se logra el mismo efecto anterior, según la tabla de conversión
D=(double) (A / B);          //OJO!, en este caso, el resultado de la división se convierte en double,
                               //a D se le asigna 2.0.
```

Reglas:

- Si los operandos de una operación aritmética son todos del mismo tipo, el resultado es del mismo tipo.
- Si los operandos de una operación aritmética son de diferentes tipos, se convierte el de tipo más pequeño al de tipo más grande. Como el caso: D=(**double**) A / B ó D=5.0 / 2.

Conversiones:

N: no se pueden convertir.

A: conversión automática.

C: conversión que requiere un cast.

A*: conversión automática pero se pueden perder valores significativos.

Léase de izquierda a derecha (fila-columna)

	boolean	byte	short	char	int	long	float	double
boolean	-	N	N	N	N	N	N	N
byte	N	-	A	C	A	A	A	A
short	N	C	-	C	A	A	A	A
char	N	C	C	-	A	A	A	A
int	N	C	C	C	-	A	A*	A
long	N	C	C	C	C	-	A*	A*
float	N	C	C	C	C	C	-	A
double	N	C	C	C	C	C	C	-

Operaciones con contadores

A++; //A=A + 1;

A--; //A=A - 1;

Apunte de Cátedra

Pero si hacemos C=A++, dos variables modifican su valor, C y A.

¿Qué valor recibe C, el de A antes de incrementarse en 1 o el de A después de incrementarse en 1?.
Probarlo.

Operaciones con acumuladores y otros

```
SUMA += NUM;           //SUMA=SUMA + NUM;
PRODUCTO *= NUM;      //PRODUCTO=PRODUCTO * NUM;
```

Lo mismo se puede aplicar para el resto de los operadores.

Operaciones lógicas

Operador	Operación	Notas
!	Negación lógica	Convierte un valor booleano en su contrario.
&	Y lógico	Es verdadero, si y sólo sí, ambos operandos son verdaderos. Se fuerza la evaluación de los operandos.
&&	Y lógico	Igual que el anterior, pero permitiendo la evaluación incompleta de los operandos.
	O lógico	Es falso, si y sólo sí, ambos operandos son falsos.
	O lógico	Igual que el anterior, pero permitiendo la evaluación incompleta de los operandos.

Comentarios

```
// una línea
/*...*/ un bloque
```

Como ejemplo del último caso, podemos ver que todo lo que se encuentra encerrado en el bloque es un comentario, por lo tanto, no se ejecuta.

```
/* int A, B, C;
A=5;
B=2;
C=A / B; */
```

Estructuras de Control en Java

El lenguaje Java soporta varias sentencias de control de flujo, incluyendo:

Selección

Pseudocódigo	Java
si-sino	if-else
alternar-caso	switch-case

Veamos algunos ejemplos:

1. Para la sentencia **if-else**.

```
char RESPUESTA;
if(RESPUESTA == 'S')
{
    //código para la acción 'S'
}
```

Apunte de Cátedra

```
else
{
//código para la acción 'N'
}
```

2. Para la sentencia **switch-case**.

```
char COLOR; //colores primarios
switch(COLOR)
{
case 'R':      {
System.out.println("Rojo");
break;
}
case 'A':      {
System.out.println("Azul");
break;
}
case 'M':      {
System.out.println("Amarillo");
break;
}
}
```

La sentencia `switch` evalúa la expresión (`COLOR`) y ejecuta la sentencia `case` apropiada. Decidir cuando utilizar las sentencias `if` o `switch` depende del juicio personal. Se puede decidir cual utilizar basándose en la buena lectura del código o en otros factores. Cada sentencia `case` debe ser única y el valor proporcionado a cada sentencia `case` debe ser del mismo tipo que el tipo de dato devuelto por la expresión proporcionada a la sentencia `switch`. La sentencia `break` hace que el control salga de la sentencia `switch` y continúe con la siguiente línea. La sentencia `break` es necesaria porque las sentencias `case` se siguen ejecutando hacia abajo. Esto es, sin un `break` explícito, el flujo de control seguiría secuencialmente a través de las sentencias `case` siguientes. En el ejemplo anterior, no se quiere que el flujo vaya de una sentencia `case` a otra, por eso se han tenido que poner las sentencias `break`. Sin embargo, hay ciertos escenarios en los que querrás que el control proceda secuencialmente a través de las sentencias `case`.

Se puede utilizar la sentencia `default` al final de la sentencia `switch` para manejar los valores que no se han manejado explícitamente por una de las sentencias `case`.

```
switch(COLOR)
{
case 'R':      {
System.out.println("Rojo");
break;
}
case 'A':      {
System.out.println("Azul");
break;
}
case 'M':      {
System.out.println("Amarillo");
}
```


Apunte de Cátedra

```

        break;
    }
    default:
    {
        System.out.println("¡No es un color primario!");
    }
}

```

Iteración

Pseudocódigo	Java
mientras	while
hacer-mientras	do-while
para	for

Veamos algunos ejemplos:

1. Bucle **while**.

```

int NUM;
NUM=Console.readInt("Ingrese un número: ");
int CONT=0; //al mismo tiempo que se declara se le puede dar un valor de inicio
while(NUM != -1)
{
    CONT++;
    NUM=Console.readInt("Ingrese un número: ");
}
System.out.println("La cantidad de números leídos es: "+CONT);

```

2. Bucle **for**.

```

//VEC es un arreglo de cualquier tipo
...
final int LONG=10;
for(int IND=0; IND < LONG; IND++)
{
    //se realiza algo con cada elemento del arreglo VEC
}

```

3. Bucle **do-while**.

```

do
{
    //se realiza algo que itere, teniendo en cuenta que al menos una vez se cumple
}
while (expresión de tipo booleano);

```

Reglas de sintaxis:

- Todas las palabras reservadas van en minúscula.
- Se diferencian las minúsculas de las mayúsculas: la variable a no es la variable A.
- Se utilizan {} para delimitar bloques de sentencias (if, while, etc.).
- Cada línea termina con ; (punto y coma) excepto if, while, for.
- Las condiciones van entre paréntesis.

Apunte de Cátedra

Veamos como sería el Ejercicio 8.e del Práctico 4, en Java:

```
{
//declaración de variables e inicialización de contadores
int CONTB, CONTA, CONTM, CONTP, CONT;
char ELEMENTO;
CONTB=0;
CONTA=0;
CONTM=0;
CONTP=0;

//carteles e inicio del inventario
System.out.println("Inventario de una escuela");
System.out.println("Datos: B: borradores; A: bancos;
                    M: mesas; P: pizarrones");
for(CONT=1; CONT<=2500; CONT++)
{
//carga y control de carga
do
    ELEMENTO=Console.readChar("Ingrese un elemento: ");
while(ELEMENTO != 'A' & ELEMENTO != 'B' &
        ELEMENTO != 'M' & ELEMENTO != 'P');

//verificación y conteo del elemento cargado
switch(ELEMENTO)
{
case 'B': {
    CONTB=CONTB + 1;
    break;
}
case 'A': {
    CONTA=CONTA + 1;
    break;
}
case 'M': {
    CONTM=CONTM + 1;
    break;
}
case 'P': {
    CONTP=CONTP + 1;
    break;
}
}
}

//resultados
System.out.println("Resultado del inventario");
System.out.println("La cantidad de borradores (B) es: "+CONTB);
System.out.println("La cantidad de bancos (A) es: "+CONTA);
System.out.println("La cantidad de mesas (M) es: "+CONTM);
System.out.println("La cantidad de pizarrones (P) es: "+CONTP);
}
```

Apunte de Cátedra

Clases en Java

Estructura de una clase:

```
class NombreClase
{
    //declaración de atributos
    tipo_de_dato nombreAtributo;
    .
    .
    .

    //declaración de métodos
    tipo_de_dato nombreDelMetodo(lista de parámetros)
    {
        //declaración_de_variables_locales
        ...
        //cuerpo_del_método
        ...
        return [resultado];
    }
    .
    .
    .
}
```

Veamos como sería el Ejercicio 5 del Práctico 8, en Java:

```
class Hora
{
    //atributos de la clase Hora
    int hora, minuto, segundo;

    //constructor de la clase Hora
    Hora(int HH, int MM, int SS)
    {
        if(validarHora(HH) && validarMinutoSegundo(MM) &&
            validarMinutoSegundo(SS))
        {
            hora=HH;
            minuto=MM;
            segundo=SS;
        }
        else
        {
            hora=0;
            minuto=0;
            segundo=0;
        }
    }
}
```

Apunte de Cátedra

```
//verifica si es o no válida la hora
boolean validarHora(int HH)
{
    if(HH >= 0 && HH < 24)
        return true;
    else
        return false;
}

//verifica si es o no válida tanto los minutos como los segundos
boolean validarMinutoSegundo(int MS)
{
    if(MS >= 0 && MS < 60)
        return true;
    else
        return false;
}

//permite cambiar el valor de la hora
//void es la palabra reservada a utilizar en los casos
//dónde el método no devuelve nada
void setHora(int HH)
{
    if(validarHora(HH))
        hora=HH;
    return;
}

//devuelve el valor de la hora
int getHora()
{
    return hora;
}

//permite cambiar el valor de los minutos
void setMinuto(int MM)
{
    if(validarMinutoSegundo(MM))
        minuto=MM;
    return;
}

//devuelve el valor de los minutos
int getMinuto()
{
    return minuto;
}

//permite cambiar el valor de los segundos
void setSegundo(int SS)
{
    if(validarMinutoSegundo(SS))
        segundo=SS;
    return;
}
```

Apunte de Cátedra

```
//devuelve el valor de los segundos
int getSegundo()
{
    return segundo;
}

//devuelve la hora pasada en minutos
int devolverHoraEnMinutos()
{
    int MIN;
    MIN=minuto + hora * 60;
    return MIN;
}

//devuelve la hora pasada en segundos
int devolverHoraEnSegundos()
{
    int MIN, SEG;
    MIN=devolverHoraEnMinutos();
    SEG=segundo + MIN * 60;
    return SEG;
}

//modifica la hora aumentándola en la cantidad de minutos ingresados
//si el valor es negativo, lo restaría
void cambiarHoraConMinutos(int MM)
{
    minuto=minuto + MM;
    if(minuto >= 60)
        do
        {
            if(minuto == 60)
                minuto=0;
            else
                minuto=minuto - 60;
            hora=hora + 1;
        }
        while(minuto >= 60);
    if(hora > 24)
        do
            hora=hora - 24;
        while(hora > 24);
    return;
}

//muestra la hora en el formato hh:mm:ss
void mostrarFormatoHora()
{
    System.out.println(hora+":"+minuto+":"+segundo);
    return;
}
}
```

Convenciones:

1. El nombre de la clase va en forma completa y el comienzo de cada palabra en mayúscula.
Ejemplo: HoraDigital.

Apunte de Cátedra

2. El nombre de los métodos va en forma completa y el comienzo de cada palabra, menos la primera, en mayúscula. Ejemplo: cambiarHoraConMinutos.
3. El nombre de los atributos va en minúscula. Ejemplo: minuto.

Método de acceso a lo atributos

Un atributo implica dos métodos, uno para actualizar su estado (método comando) y otro para devolverlo (método consulta). Dado que los datos de un objeto no se pueden acceder directamente, es necesario contar con métodos de acceso a los atributos. En el ejemplo anterior podemos ver como se puede actualizar o modificar el valor de los minutos (*set*) y como se puede devolver u obtener dicho valor (*get*).

```
void setMinuto(int MM) | int getMinuto()
{                       | {
    minuto=MM;          |     return minuto;
    return;             | }
}                       |
```

Estos dos métodos aseguran que sólo el objeto accede a sus datos. Por otra parte, si tenemos:

```
int getMinuto() | void mostrarMinuto()
{               | {
    return minuto; |     System.out.print(minuto);
}               |     return;
}               | }
```

La diferencia radica en que a través del primer método el objeto devuelve el valor del atributo minuto y quien ha enviado el mensaje puede con él realizar todas las operaciones que desee, como hacer cálculos, comparar con otras horas, e inclusive mostrar el valor. Esto último lo realiza el segundo método, que sólo muestra el valor del minuto y no da la posibilidad de manipularlo.

El método main

En una aplicación en Java existirán, como ya se indicó, cientos de objetos interactuando. Estos objetos podrán corresponder a la misma clase o no. Es decir, la solución de un problema podría requerir de objetos de tipo Animal, Auto, Fecha, Hora, etc. estas clases representan una entidad. La aplicación o programa en la cual los objetos se crean e interactúan, es una clase especial, ya que no se corresponde con una entidad. Esta clase posee el método *main* (principal en inglés). El método *main* tiene el objetivo de posibilitar la ejecución de un programa. Toda ejecución de un programa en Java comienza por ese método, si no estuviera daría un error. Entonces, en una aplicación tendremos muchas clases correspondientes a diferentes entidades, y solo una clase con el método *main*.

Veamos como sería el principal para probar la clase Hora:

```
class PruebaHora
{
    public static void main(String[] args)
    {
        int HH, MM, SS, MIN;
        System.out.println("*** Primer ejemplo ***");
        Hora HOR1=new Hora(14, 15, 20);
        System.out.print("La hora cargada es la siguiente: ");
        HOR1.mostrarFormatoHora();
        System.out.println("La hora pasada a minutos es: "+
            HOR1.devolverHoraEnMinutos()+" minutos");
    }
}
```

Apunte de Cátedra

```
System.out.println("La hora pasada a segundos es: "+
    HOR1.devolverHoraEnSegundos()+" segundos");
MIN=Console.readInt("Ingrese cantidad de minutos para
    aumentar a la hora: ");
HOR1.cambiarHoraConMinutos(MIN);
System.out.print("La hora modificada es: ");
HOR1.mostrarFormatoHora();

System.out.println("** Segundo ejemplo **");
Hora HOR2=new Hora(33, 50, 20);
System.out.print("La hora cargada es la siguiente: ");
HOR2.mostrarFormatoHora();
if(HOR2.getHora() == 0 && HOR2.getMinuto() == 0 &&
    HOR2.getSegundo() == 0)
    {
        HOR2.setHora(23);
        HOR2.setMinuto(50);
        HOR2.setSegundo(79);
    }
System.out.print("La hora modificada es la siguiente: ");
HOR2.mostrarFormatoHora();
System.out.println("La hora pasada a minutos es: "+
    HOR2.devolverHoraEnMinutos()+" minutos");
System.out.println("La hora pasada a segundos es: "+
    HOR2.devolverHoraEnSegundos()+" segundos");
MIN=Console.readInt("Ingrese cantidad de minutos para
    aumentar a la hora: ");
HOR2.cambiarHoraConMinutos(MIN);
System.out.print("La hora modificada es: ");
HOR2.mostrarFormatoHora();
}
}
```

En este ejemplo, Hora es una clase que representa a una entidad bien definida del mundo real, y de la cual se podrán crear Objetos. En cambio PruebaHora, no representa ninguna entidad, su único propósito es contener el método main, el cual permite la ejecución de un programa. Probar dicho ejemplo y entender que realiza cada línea de código.

Referencias en Java

En un programa un objeto es una variable. Esta variable corresponde a una instancia de alguna clase, es decir su tipo, al igual que una variable de tipo simple (int, float, etc.). Pero existe una importante diferencia, las variables utilizadas para identificar objetos, son *referencias o punteros*.

En el siguiente código:

```
int contador; // 1
contador=0; // 2
```

En la línea 1, se reserva un espacio en memoria identificado con el identificador contador para almacenar un dato entero. En la línea 2, en el espacio de memoria identificado como contador se asigna (almacena) el valor entero 0. Es decir, en el espacio de memoria identificado como contador se almacena un valor. Este mecanismo para manejar memoria es simple y se conoce como “direccionamiento directo”. Para las variables correspondientes a objetos, el mecanismo es diferente y se conoce como “direccionamiento indirecto”.

Apunte de Cátedra

Supongamos el siguiente código:

```
Hora objHora;           // 1  
objHora=new Hora(12, 20, 10); // 2
```

En la línea 1, se reserva un espacio en memoria identificado con el identificador `objHora` para almacenar una dirección de memoria (puntero). Al momento de la declaración, se asigna automáticamente un valor nulo. En la línea 2, cuando se crea el objeto (mediante la sentencia `new`), se obtiene de memoria un espacio diferente que contendrá todos los datos del objeto `objHora`. Es decir, en el espacio de memoria identificado como `objHora`, no se almacenan los datos del objeto `objHora`. En `objHora`, se almacena una dirección memoria que apunta al espacio en memoria que contiene los datos de `objHora` (hora, minuto y segundo).

La sentencia `new` busca espacio para una instancia de un objeto de una clase determinada e inicializa la memoria a los valores adecuados. Luego invoca al método constructor de la clase, proporcionándole los argumentos adecuados. La sentencia `new` devuelve una referencia a sí mismo, que es inmediatamente asignada a la variable referencia.

Debemos ser cuidadosos en nuestros programas y tener en cuenta que los objetos son variables, especialmente cuando hacemos asignaciones.

Ilustramos con un el siguiente ejemplo:

Pensemos en una referencia como si se tratase de la llave electrónica de la habitación de un hotel. Vamos a utilizar precisamente este ejemplo del hotel para demostrar el uso y la utilización que podemos hacer de las referencias en Java. Primero crearemos la clase *Habitacion*, implementada en el archivo *Habitacion.java*, mediante instancias de la cual construiremos nuestro hotel:

```
class Habitacion  
{  
    int numHabitacion;  
    int numCamas;  
  
    Habitacion(int numeroHab, int camas)  
    {  
        setHabitacion(numeroHab);  
        setCamas(camas);  
    }  
  
    int getHabitacion()  
    {  
        return numHabitacion;  
    }  
  
    void setHabitacion(int numeroHab)  
    {  
        numHabitacion=numeroHab;  
        return;  
    }  
  
    int getCamas()  
    {  
        return numCamas;  
    }  
}
```


Apunte de Cátedra

```
void setCamas(int camas)
{
    numCamas=camas;
    return;
}
```

Vamos a construir nuestro hotel creando habitaciones y asignándole a cada una de ellas su llave electrónica; tal como muestra el código siguiente del archivo *Hotel.java*:

```
class Hotel
{
    public static void main(String[] args)
    {
        //paso 1
        Habitacion llaveHab1;
        Habitacion llaveHab2;

        //paso 2
        llaveHab1=new Habitacion(222, 1);
        llaveHab2=new Habitacion(1144,3);
    }
}
```

El primer paso es la creación de las llaves, es decir, definir la variable referencia, por defecto nula. En el segundo paso se crean las habitaciones. Podemos tener múltiples llaves para una misma habitación:

```
Habitacion llaveHab3, llaveHab4;
llaveHab3=llaveHab1;
llaveHab4=llaveHab2;
```

De este modo conseguimos copias de las llaves. Las habitaciones, en sí mismas, no se han tocado en este proceso. Así, ya tenemos dos llaves para la habitación 222 y otras dos para la habitación 1144.

Una llave puede ser programada para que funcione solamente con una habitación en cualquier momento, pero podemos cambiar su código electrónico para que funcione con alguna otra habitación; por ejemplo, para cambiar una habitación anteriormente utilizada por un empedernido fumador por otra limpia de olores y con vistas al mar. Entonces cambiemos la llave duplicada de la habitación del fumador (la 222) por la habitación con olor a sal marina, 1144:

```
llaveHab3=llaveHab2;
```

Ahora tenemos una llave para la habitación 222 y tres para la habitación 1144. Mantendremos una llave para cada habitación en la conserjería, para poder utilizarla como llave maestra, en el caso de que alguien pierda su llave propia.

Alguien con la llave de una habitación puede hacer cambios en ella, y los compañeros que tengan llave de esa misma habitación, no tendrán conocimiento de esos cambios hasta que vuelvan a entrar en la habitación. Por ejemplo, vamos a quitar una de las camas de la habitación, entrando en ella con la llave maestra:

```
llaveHab2.setCamas(2);
```

Ahora cuando los inquilinos entren en la habitación podrán comprobar el cambio realizado:

Apunte de Cátedra

```
System.out.println(llaveHab4.getCamas());
```

Este último punto es muy importante, ya que más de una puntero puede referenciar a un mismo objeto. En consecuencia, las modificaciones al objeto producidas desde una particular referencia, se reflejarán para todas las referencias.

En conclusión, una asignación de variables simples es absolutamente diferente a una asignación de variables punteros.

Agregación y Dependencia

La POO intenta promover la reusabilidad de objetos. No sólo se trata de crear clases, sino de usar o adaptar clases ya hechas. Hay tres formas fundamentales de reutilizar clases, mediante: composición/agregación, herencia y uso/dependencia.

Agregación: consiste en formar objetos a partir de otros. Cuando entre dos objetos de distintas clases se produce la relación “se compone de” o “es parte de” o “tienen un”, estamos en presencia de una relación de agregación o composición. Por ejemplo, un auto tiene un precio, el precio es un dato real, simple; pero un auto tiene un motor, y un motor tiene a su vez una serie de particularidades y no todos los autos tienen el mismo motor. Además, los motores no sólo se usan en los autos, también en barcos, aviones, electrodomésticos, etc. Es decir, *Motor* debería ser una clase. La relación se puede describir como: un auto “se compone de” un motor, o un auto “tiene un” motor, o un motor “es parte de” un auto.

Por ejemplo:

```
class Auto
{
    Cadena marca;
    Motor mot;
    .
    .
}
```

La agregación implica que en la clase *Auto* hay al menos un atributo de tipo *Motor*. En algún método de la clase *Auto* el atributo *mot* debe ser creado, ya que se encuentra solamente declarado como atributo. Esto significa que desde cualquier método de la clase *Auto* se pueden enviar mensajes al objeto *mot*.

Dependencia: cuando entre dos objetos de distintas clases se produce la relación “usa” o “depende”, estamos en presencia de una relación de dependencia. Por ejemplo, un auto tiene 4 ruedas y si es necesario cambiar una de ellas, la clase *Auto* deberá tener un método *cambiarRueda()*. Para poder cambiar una rueda es necesario usar un críquet, que no es un componente del auto, pero se usa para cambiar una rueda. Es decir, un objeto de la clase *Auto*, usará un objeto de la clase *Criquet*, cada vez que se ejecute el método *cambiarRueda()*. La relación se puede describir como: un auto “usa” un críquet.

Por ejemplo:

```
class Auto
{
    .
    .
}
```

Apunte de Cátedra

```
void cambiarRueda(Criquet c)
{
    //código
}

void cambiarRueda( )
{
    Criquet c=new Criquet();
    //código
}
.
.
.
}
```

La dependencia significa que la clase *Auto* tiene la posibilidad de enviar mensajes a un objeto de la clase *Criquet*, que recibe por parámetro o crea como variable local.

Parámetros

Los parámetros son el mecanismo por el cual se transmiten datos a los métodos. Existen dos tipos de parámetros: por valor o copia y por referencia. En Java cuando un parámetro corresponde a un tipo simple (int, char, float, double, etc.) es siempre por valor, y cuando el parámetro es un objeto es siempre por referencia.

Un parámetro por valor significa que el método trabaja con una “copia” del valor original pero no con el valor en cuestión. En consecuencia, si en el método el valor del parámetro sufriera modificaciones, éstas no se verían reflejadas en el valor original al finalizar el método, puesto que se trabajó con una copia.

Por ejemplo:

```
void ejemplo(int num)
{
    num=num * 5;           //num es una copia del parámetro actual
    return;
}
```

En otra clase se envía un mensaje:

```
int numero=23;
objeto.ejemplo(numero)    //envía una copia de 23
System.out.println(numero) //imprime 23
```

Al contrario, los parámetros por referencia trabajan con el objeto real enviado. Por lo tanto, todas las modificaciones que el objeto sufra en el método se reflejarán en el mismo, aun después de la ejecución del método.

Por ejemplo:

```
void ejemplo(Persona p)
{
    p.setEdad(5); //p es una referencia al objeto real
    return;
}
```

Apunte de Cátedra

En otra clase si se hace:

```
Persona per=new Persona();  
per.setEdad(44);  
System.out.println(per.getEdad()) //imprime 44  
objeto.ejemplo(per);  
System.out.println(per.getEdad()) //imprime 5
```

Paquetes

Las clases compiladas (.class) se organizan en paquetes. Un paquete contiene un conjunto de clases. A su vez, un paquete puede contener a otros paquetes. La estructura es similar a la de los directorios (carpetas) y archivos en el Explorador de Windows. Los archivos cumplen el papel de las clases Java y los directorios cumplen el papel de paquetes. De hecho, la estructura de directorios en la que se organizan los archivos .class (estructura física del sistema operativo) debe corresponderse con la estructura de paquetes definida en los archivos fuente .java.

La palabra clave *package* permite agrupar clases. Los nombres de los paquetes son palabras separadas por puntos y se almacenan en directorios que coinciden con esos nombres.

Por ejemplo, si al comienzo de un archivo .java llamado *Persona.java* se escribe:

```
package misclases.negocios;
```

estamos declarando que la clase *Persona* (el archivo *Persona.class*) deberá residir en una estructura de directorios *misclases\negocios*.

El lenguaje Java proporciona una serie de paquetes que incluyen ventanas, utilidades, un sistema de entrada/salida general, herramientas y comunicaciones. Algunos de los paquetes que se incluyen son: *java.applet*, *java.awt*, *java.io*, *java.lang*, *java.net*, *java.util*, etc.

Para poder cargar una clase de un paquete determinado se utiliza la palabra clave *import*, especificando el nombre del paquete como ruta y nombre de clase.

Por ejemplo,

```
import java.util.Date; //se importa la clase Date del paquete java.util  
import java.awt.*; //se importan todas las clases que contenga el paquete java.awt
```

El único paquete que no es necesario importar, ya que Java lo toma por defecto, es java.lang.

Modificadores de acceso

Uno de los beneficios de las clases es que pueden proteger sus atributos y métodos frente al acceso de otros objetos. ¿Por qué es esto importante?. Ciertas informaciones y peticiones contenidas en la clase, las soportadas por los métodos y atributos accesibles públicamente en su objeto son correctas para el consumo de cualquier otro objeto del sistema. Otras peticiones contenidas en la clase son sólo para el uso personal de la clase. Estas otras soportadas por la operación de la clase no deberían ser utilizadas por objetos de otros tipos. Se querría proteger esos atributos y métodos personales a nivel del lenguaje y prohibir el acceso desde objetos de otros tipos.

En Java se puede utilizar los especificadores de acceso para proteger tanto los atributos como los métodos de la clase cuando se declaran. El lenguaje Java soporta cuatro niveles de acceso para los atributos y métodos: *public*, *private*, *protected* y todavía no especificado, acceso de paquete.

Acceso público: es el especificador de acceso más sencillo. Todas las clases, en todos los paquetes tienen acceso a los miembros públicos de la clase. Los miembros públicos se declaran sólo si su acceso no produce resultados indeseados si un extraño los utiliza. Para declarar un miembro público se utiliza la palabra clave *public*. Por ejemplo,

Apunte de Cátedra

```
package Griego;
public class Alpha
{
    public int soyPublico;
    public void metodoPublico()
    {
        System.out.println("metodoPublico");
        return;
    }
}

import Griego.*;
package Romano;
class Beta
{
    void metodoAccesor()
    {
        Alpha a = new Alpha();
        a.soyPublico = 10; // LEGAL
        a.metodoPublico(); // LEGAL
        return;
    }
}
```

Como se puede ver en el ejemplo anterior, *Beta* puede crear objetos de la clase *Alpha*, inspeccionar y modificar legalmente el atributo *soyPublico* de la clase *Alpha* y puede llamar legalmente al método *metodoPublico()*. Si no se coloca ningún modificador de acceso, Java asume por defecto que el miembro es público.

Acceso privado: Es el nivel de acceso más restringido. Un miembro privado es accesible sólo para la clase en la que está definido. Se utiliza este acceso para declarar miembros que sólo deben ser utilizados por la clase. Esto incluye los atributos que contienen información que si se accede a ella desde el exterior podría colocar al objeto en un estado de inconsistencia, o los métodos que llamados desde el exterior pueden poner en peligro el estado del objeto o del programa donde se está ejecutando. Los miembros privados son como secretos. Para declarar un miembro privado se utiliza la palabra clave *private* en su declaración. La clase siguiente contiene un atributo y un método privados:

```
class Alpha
{
    private int soyPrivado;
    private void metodoPrivado()
    {
        System.out.println("metodoPrivado");
        return;
    }
}
```

Los objetos del tipo *Alpha* pueden inspeccionar y modificar el atributo *soyPrivado* y pueden invocar el método *metodoPrivado()*, pero los objetos de otros tipos no pueden acceder. Por ejemplo, la clase *Beta* definida aquí:

Apunte de Cátedra

```
class Beta
{
    void metodoAccesor()
    {
        Alpha a = new Alpha();
        a.soyPrivado = 10; // ILEGAL
        a.metodoPrivado(); // ILEGAL
        return;
    }
}
```

La clase *Beta* no puede acceder al atributo *soyPrivado* ni al método *metodoPrivado()* de un objeto del tipo *Alpha* porque *Beta* no es del tipo *Alpha*. Si una clase está intentando acceder a una variable miembro a la que no tiene acceso, el compilador mostrará el siguiente mensaje de error y no compilará su programa:

```
Beta.java:6: soyPrivado has private access in Alpha
a.soyPrivado = 10; // ILEGAL
^
```

Y si un programa intenta acceder a un método al que no tiene acceso, generará el siguiente mensaje de error de compilación:

```
Beta.java:7: metodoPrivado() has private access in Alpha
a.metodoPrivado(); // ILEGAL
^
```

Acceso de Paquete: El último nivel de acceso es el que se obtiene si no se especifica ningún otro nivel de acceso a los miembros. Este nivel de acceso permite que las clases del mismo paquete que la clase tengan acceso a los miembros. Este nivel de acceso asume que las clases del mismo paquete son amigas de confianza. Este nivel de confianza es como la que extiende a sus mejores amigos y que incluso no la tiene con su familia. Por ejemplo, esta versión de la clase *Alpha* declara un atributo y un método con acceso de paquete. *Alpha* reside en el paquete *Griego*:

```
package Griego;
class Alpha
{
    int estoyEmpaquetado;
    void metodoEmpaquetado()
    {
        System.out.println("metodoEmpaquetado");
        return;
    }
}
```

```
package Griego;
class Beta
{
    void metodoAccesor()
    {
```

Apunte de Cátedra

```
Alpha a = new Alpha();
a.estoyEmpaquetado = 10; // LEGAL
a.metodoEmpaquetado(); // LEGAL
return;
}
}
```

Entonces, *Beta* puede acceder legalmente a *estoyEmpaquetado* y *metodoEmpaquetado()* de la clase *Alpha* dado que ambas clases se encuentran en el mismo paquete.

Sobrecarga

Se produce cuando existen en la misma clase dos o más métodos (incluido el constructor) con el mismo identificador (nombre) pero con diferentes listas de parámetros (en número y/o en tipo). Cada método realiza una operación en función de los argumentos que recibe. Semánticamente la sobrecarga resuelve el problema de que una determinada función se puede realizar de diferentes formas (algoritmos) con distintos datos (parámetros). La sobrecarga se resuelve en tiempo de compilación, dado que los métodos se deben diferenciar en el tipo o en el número de parámetros.

Por ejemplo:

```
class VectorEnteros
{
    int[] elementos;
    int dimension, actual;

    VectorEnteros(int DIM)
    {
        if(DIM > 0)
            dimension=DIM;
        else
            dimension=10;
        actual=0;
        elementos=new int[dimension];
    }

    VectorEnteros( )
    {
        dimension=10;
        actual=0;
        elementos=new int[dimension];
    }
    .
    .
    .
}
```

El método constructor de la clase *VectorEnteros* esta sobrecargado, es decir un vector de enteros se puede construir de 2 maneras diferentes. La diferencia se encuentra en los parámetros, en uno se permite determinar la dimensión del vector y en otro no. Se podría agregar un constructor más que además de la dimensión reciba por parámetro un valor entero cuyos dígitos serán cargados al vector, esto permitiría que el vector no se cargue vacío, por ejemplo.

Apunte de Cátedra

Operador THIS

El operador *this* se utiliza para hacer referencia a los miembros de la propia clase. Uno de los usos más comunes es cuando existen otros elementos (identificadores) en la clase, con el mismo nombre para distinguir.

Por ejemplo,

```
public class MiClase
{
    private int i;
    public MiClase(int i)
    {
        this.i = i;    //al atributo i de la clase se le asigna el parámetro i
    }
}
```

Nota: recuerde que el uso de identificadores iguales puede ocasionar inestabilidad en la clase, si no se utilizan correctamente.

Arreglos en Java

Dado que los arreglos (arrays) son tipos de datos muy utilizados en programación, en Java son objetos predefinidos. Se pueden declarar y crear arrays de cualquier tipo:

```
char[] palabra=new char[10];
int[] numeros=new int[100];
```

Los corchetes se pueden colocar de dos formas en la declaración:

```
tipo[] identificador;    //indica que todos los identificadores son arrays del tipo
tipo identificador[];    //es array sólo el identificador al que le siguen los [ ].
```

Veamos en estos ejemplos como se escriben distintas declaraciones de arrays.

```
char cad[], p;    //cad es un array de tipo char; p es una variable de tipo char.
```

```
int[] v, w;    //tanto v como w son declarados arrays unidimensionales de tipo int.
```

```
double[] m, t[], x;    //m y x son array de tipo double; t es un array de array de tipo double.
```

También, se pueden construir arrays de arrays (matrices):

```
int tabla[][]=new int[4][5];
```

Los límites de los arrays se comprueban en tiempo de ejecución para evitar desbordamientos y la corrupción de memoria.

En Java un array es realmente un objeto, porque tiene redefinido el operador []. Tiene un método miembro *length*, que se puede utilizar cómo método para conocer la longitud de cualquier array.

Algunos ejemplos más completos:

```
//de vector
```

```
char vector[]=new char[100];
```

```
System.out.println("La longitud del vector es: "+vector.length); //La longitud del vector es: 100
```

```
//de matriz
```

```
int matriz[][]=new int[10][3];
```

```
//La cantidad de filas de la matriz es: 10
```


Apunte de Cátedra

```
System.out.println("La cantidad de filas de la matriz es: "+matriz.length);  
//La cantidad de columnas de la matriz es: 3  
System.out.println("La cantidad de columnas de la matriz es: "+matriz[0].length);
```

Para crear un array hay dos métodos básicos:

1. Crear un array vacío: **int** lista[]=**new int**[50];
 2. Crear un array con sus valores iniciales: **int** lista[]={5, 4, 3, 2};
- Esto que es equivalente a:

```
int lista[]=new int[4];  
lista[0]=5;  
lista[1]=4;  
lista[2]=3;  
lista[3]=2;
```

En Java, al ser un tipo de datos verdadero, se dispone de comprobaciones exhaustivas del correcto manejo del array; por ejemplo, de la comprobación de sobrepasar los límites definidos para el array, para evitar desbordamientos o corrupción de memoria.

Al igual que los demás objetos en Java, la declaración del array no localiza, o reserva, memoria para contener los datos. En su lugar, simplemente localiza memoria para almacenar una referencia al array.

La memoria necesaria para almacenar los datos que componen el array se buscará en memoria dinámica a la hora de instanciar y crear realmente el array.

Los índices de un array siempre empiezan en 0. Además, en Java no se puede rellenar un array sin declarar el tamaño con el operador *new*. Como se ilustra en la última parte del siguiente código. Este ejemplo intenta ilustrar un aspecto interesante del uso de arrays en Java. Se trata de que Java puede crear arrays multidimensionales, que se verían como arrays de arrays. Aunque esta es una característica común a muchos lenguajes de programación, en Java, sin embargo, los arrays secundarios no necesitan ser todos del mismo tamaño. En el ejemplo, se declara e instancia un array de enteros, de dos dimensiones, con un tamaño inicial (tamaño de la primera dimensión) de 3. Los tamaños de las dimensiones secundarias (tamaño de cada uno de los subarrays) son 2, 3 y 4, respectivamente.

Veamos un ejemplo para ilustrar la creación y manipulación de un array de dos dimensiones en donde los subarrays tienen diferente tamaño.

```
class EjemploMatriz  
{  
    public static void main(String[] args)  
    {  
        //Declaramos un array de dos dimensiones con un tamaño de 3 en  
        //la primera dimensión y diferentes tamaños en la segunda  
        int miArray[][]=new int[3][]; //No especificamos el tamaño  
        //de las columnas  
        miArray[0]=new int[2]; //El tamaño de la segunda dimensión es 2  
        miArray[1]=new int[3]; //El tamaño de la segunda dimensión es 3  
        miArray[2]=new int[4]; //El tamaño de la segunda dimensión es 4  
  
        //Rellenamos el array con datos  
        for(int i=0; i < 3; i++)  
            for(int j=0; j < miArray[i].length; j++)  
                miArray[i][j]=i * j;
```

Apunte de Cátedra

```
//Visualizamos los datos que contiene el array
for(int i=0; i < 3; i++)
{
    for(int j=0; j < miArray[i].length; j++)
        System.out.print(miArray[i][j]);
    System.out.println();
}

//Hacemos un intento de acceder a un elemento que se encuentre
//fuera de los límites del array
System.out.println("Acceso a un elemento fuera de limites");
miArray[4][0]=7; //Esa sentencia originará el lanzamiento
                //de una excepción de tipo
                //ArrayIndexOutOfBoundsException
}
}
```

En este ejemplo, también se pueden observar alguna otra cosa, como es el hecho de que cuando se declara un array de dos dimensiones, no es necesario indicar el tamaño de la dimensión secundaria a la hora de la declaración del array, sino que puede declararse posteriormente el tamaño de cada uno de los subarrays. También se puede observar el resultado de acceder a elementos que se encuentran fuera de los límites del array; Java protege la aplicación contra este tipo de errores de programación.

En Java, si se intenta hacer esto, el Sistema generará una excepción, tal como se muestra en el ejemplo. En él, la excepción simplemente hace que el programa termine, pero se podría recoger esa excepción e implementar un controlador de excepciones (*exception handler*) para corregir automáticamente el error, sin necesidad de abortar la ejecución del programa.

La salida por pantalla de la ejecución del ejemplo sería:

```
00
012
0246
Acceso a un elemento fuera de límites
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at EjemploMatriz.main(EjemploMatriz.java:25)
```

En el siguiente código se ilustra otro aspecto del uso de arrays en Java. Se trata, en este caso, de que Java permite la asignación de un array a otro. Sin embargo, hay que extremar las precauciones cuando se hace esto, porque lo que en realidad está pasando es que se está haciendo una copia de la referencia a los mismos datos en memoria. Y, tener dos referencias a los mismos datos no parece ser una buena idea, porque los resultados pueden despistar.

```
int[] primerArray=new int[3];
int[] segundoArray;
for(int i=0; i < 3; i++)
    primerArray[i]=i;
segundoArray=new int[3];
segundoArray=primerArray;
```

En un array se pueden almacenar elementos de tipos básicos (int, char, double, etc.) y objetos. En este último caso, el arreglo debe verse como un arreglo de punteros.

```
Animal[] vectAnimal=new Animal[12];
Animal objAnimal;
```

Apunte de Cátedra

```
for(int i=0; i <12; i++)
{
    objAnimal=new Animal();
    vectAnimal[i]=objAnimal;
}
```

La clase Vector y Matriz

Dado que los arreglos son objetos, lo más conveniente es codificar una clase que permita un uso simple de los arreglos (vectores y matrices) y totalmente transparente para los programas que lo utilicen, además de facilitar la reutilización de los mismos. Las clases vistas en unidades anteriores sobre Arreglos, pueden ser pasadas al Lenguaje Java, para su correcta utilización. Podemos realizar nuestra clase personalizada de acuerdo al tipo de arreglo que implementemos, pero también podemos hacer uso de la clase Vector que proporciona Java en su paquete *java.util*.

Programas en Java

Un programa codificado en el Lenguaje Java debe grabarse en un archivo de texto, cuyo nombre debe coincidir totalmente con el nombre de la clase y la extensión debe ser *.java*. Es decir, el archivo fuente sería por ejemplo: *Hora.java*, *PruebaHora.java*, *Habitacion.java*, *Hotel.java*. Luego de compilar el archivo fuente se obtiene un archivo ejecutable con el mismo nombre pero con extensión *.class*.

JDK proporciona en su carpeta `\bin`, las aplicaciones respectivas para compilar y ejecutar un archivo con Java.

Para compilar

```
javac Hora.java
javac PruebaHora.java
```

Para ejecutar

```
java PruebaHora
```

Estos comandos deberían utilizarse bajo el Símbolo del Sistema `c:\`. Hay que tener en cuenta que para poder realizar estas tareas se necesita añadir la carpeta a la variable de entorno `PATH`.

Java facilita la creación de entornos de desarrollo integrados (IDE) de modo flexible, poderoso y efectivo. Los programadores ahora disponen de herramientas y aplicaciones de programación gratuitas (Eclipse, NetBeans, JCreator LE, DrJava, BlueJ, FreeJava, entre otras), open-source (Eclipse, NetBeans, entre otras), entornos comerciales como las corporaciones Borland (JBuilder), Oracle (JDeveloper), Symantec (Visual Café), Microsoft (Visual J++ o Visual Java), IBM (Visual Age), JetBrains (IntelliJ IDEA), Allaire de Adobe (Kawa), entre otros. Algunas herramientas son más sofisticadas que otras, dado que proporcionan manejos más complejos como lo son los entornos visuales, la documentación, los debuggers y los visualizadores de jerarquía de clases. Esto proporciona una gran progresión a los entornos de desarrollo Java.

API

Una API (del inglés Application Programming Interface - Interfaz de Programación de Aplicaciones) es un conjunto de especificaciones de comunicación entre componentes software. Representa un método para conseguir abstracción en la programación, generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del software. Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, para dibujar ventanas o iconos en la pantalla. De esta forma, los programadores se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio. Las APIs asimismo son abstractas: el software que proporciona una cierta API generalmente es llamado la implementación de esa API.

Apunte de Cátedra

Una herramienta muy útil son las páginas HTML con la documentación del API de Java. A continuación veremos alguna de las clases que forman parte del conjunto de clases del Lenguaje Java.

La clase String (Cadenas)

Una secuencia de datos del tipo caracter se conoce como string (cadena) y en el entorno Java está implementada por la clase String (un miembro del paquete *java.lang*).

Formato general:

```
String nombre_variable;
```

Por ejemplo:

```
String cadena;
```

```
String nombre;
```

```
String palabra="Programar";
```

El segundo uso de String es el uso de cadenas literales (una cadena de caracteres entre comillas “ ”): “¡Hola mundo!”. El compilador asigna implícitamente espacio para un objeto String cuando encuentra una cadena literal.

Los objetos String son inmutables, es decir, no se pueden modificar una vez que han sido creados. El paquete *java.lang* proporciona una clase diferente, StringBuffer, que se podrá utilizar para crear y manipular caracteres al vuelo.

Concatenación de Cadenas

Java permite concatenar cadenas fácilmente utilizando el operador +. Como lo hemos visto al utilizar el método *System.out.println()*. El siguiente fragmento de código concatena tres cadenas para producir su salida.

```
int contador=3;
```

```
String salida="La entrada tiene "+contador+" cadenas.";
```

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

Dos de las cadenas concatenadas son cadenas literales: “La entrada tiene” y “cadenas.”. La tercera cadena (la del medio) es un entero que primero se convierte a cadena y luego se concatena con las otras. Java posee gran capacidad para el manejo de cadenas dentro de sus clases String y StringBuffer. Un objeto String representa una cadena alfanumérica de un valor constante que no puede ser cambiada después de haber sido creada. Un objeto StringBuffer representa una cadena cuyo tamaño puede variar, o puede ser modificada por programa.

Los Strings son objetos constantes y, por lo tanto, muy baratos para el sistema. La mayoría de los métodos relacionados con cadenas esperan valores String como argumentos y devuelven valores String.

Existen varios constructores (sobrecargados) para crear nuevas cadenas, entre ellos:

```
String();
```

```
String(String original);
```

```
String(char[] valor);
```

```
String(char[] valor, int inicio, int cantidad);
```

```
String(byte[] bytes);
```

```
String(byte bytes[], int inicio, int longitud);
```

```
String(StringBuffer buffer);
```

Apunte de Cátedra

Por ejemplo:

```
String materia=new String("Resolución de Problemas y Algoritmos");
char[] palabra={'C', 'L', 'A', 'S', 'E'};
String cadena=new String(palabra); //crea la cadena a partir de un vector de caracteres
```

Una reflexión especial merecen los arrays de Strings. La siguiente sentencia declara e instancia un array de referencias a cinco objetos de tipo String:

```
String[] miArray=new String[5];
```

este array no contiene los datos de las cadenas. Solamente reserva una zona de memoria para almacenar las cinco referencias a cadenas. No se guarda memoria para almacenar los caracteres que van a componer esas cinco cadenas, por lo que hay que reservar expresamente esa memoria a la hora de almacenar los datos concretos de las cadenas, tal como se hace en la siguiente línea de código:

```
miArray[0]=new String("Esta es la primera cadena");
miArray[1]=new String("Esta es la segunda cadena");
miArray[2]=new String("Esta es la tercera cadena");
miArray[3]=new String("Esta es la cuarta cadena");
miArray[4]=new String("Esta es la quinta cadena");
```

Métodos básicos

int	length();	Devuelve la longitud de la cadena.
char	charAt(int indice);	Devuelve el caracter que se encuentra en la posición que indica el índice.

Métodos de Comparación de Strings

boolean	equals(Object unObjeto);	Compara la cadena con el objeto especificado.
boolean	equalsIgnoreCase(String otraCadena);	Compara la cadena con otraCadena, ignorando las mayúsculas o minúsculas.
int	compareTo(String otraCadena);	Compara dos cadenas lexicográficamente. Es decir, retorna un entero menor que cero si la cadena es léxicamente menor que otraCadena. Devuelve cero si las dos cadenas son léxicamente iguales y un entero mayor que cero si la cadena es léxicamente mayor que otraCadena.

Métodos de Comparación de Subcadenas

Prueba si dos regiones de la cadena son iguales.

boolean	regionMatches(int inicioCadena, String otraCadena, int inicioOtraCadena, int long);
boolean	regionMatches(boolean mayMin, int inicioCadena, String otraCadena, int inicioOtraCadena, int long);

Determina si la cadena comienza o termina con un cierto prefijo o sufijo comenzando en un determinado desplazamiento.

boolean	startsWith(String prefijo);
boolean	startsWith(String prefijo, int inicio);
boolean	endsWith(String sufijo);

Apunte de Cátedra

Devuelve el primer/último índice de un caracter/cadena empezando la búsqueda a partir de un determinado desplazamiento.

int	indexOf(int caracter);
int	indexOf(int caracter, int inicio);
int	lastIndexOf(int caracter);
int	lastIndexOf(int caracter, int inicio);
int	indexOf(String cadena);
int	indexOf(String cadena, int inicio);
int	lastIndexOf(String cadena);
int	lastIndexOf(String cadena, int inicio);

Modificaciones sobre la cadena, obteniendo subcadenas, concatenando cadenas, reemplazando caracteres, conversiones a minúsculas/mayúsculas y ajustando los espacios en blanco tanto al comienzo y como al final de la cadena.

String	substring(int inicio);
String	substring(int inicio, int fin);
String	concat(String cadena);
String	replace(char caracterViejo, char caracterNuevo);
String	toLowerCase();
String	toUpperCase();
String	trim();

Métodos de Conversión para otros tipos de datos.

void	getChars(int inicio, int fin, char destino[], int inicioDestino);
void	getBytes(int inicio, int fin, byte destino[], int inicioDestino);
String	toString();
char	toCharArray();
int	hashCode();

La clase String posee numerosos métodos para transformar valores de otros tipos de datos a su representación como cadena. Todos estos métodos tienen el nombre de *valueOf*, estando el método sobrecargado para todos los tipos de datos básicos.

Clases estáticas (STATIC)

Se llama así coloquialmente a las clases que tienen todos sus atributos y métodos estáticos y que no permiten ser instanciadas. Ese tipo de clases suelen ser repositorios de atributos y/o métodos y suelen alejarse un poco de la idea de programación orientada a objetos.

Atributos Static

La declaración de un atributo de clase se hace usando *static*. El significado de un atributo static es que todas las instancias de la clase (todos los objetos instanciados de la clase) contienen la mismas variables de clase o estáticas.

En otras palabras, en un momento determinado se puede querer crear una clase en la que el valor de un atributo sea el mismo (y de hecho sea el mismo atributo) para todos los objetos instanciados a partir de esa clase. Es decir, que exista una única copia del atributo, entonces es cuando debe usarse la palabra clave *static*.

Apunte de Cátedra

```
class Documento
{
    static int version=10;
    static String extension="txt";
    .
    .
    .
}
```

El valor de los atributos *version* y *extension* será el mismo para cualquier objeto instanciado de la clase *Documento*. Siempre que un objeto instanciado de *Documento* cambie el atributo *version* o *extension*, éste cambiará para todos los objetos. Si se examinara en este caso la zona de memoria reservada por el sistema para cada objeto, se encontraría con que todos los objetos comparten la misma zona de memoria para cada una de los atributos estáticos, por ello, se llaman también atributos de clase, porque son comunes a la clase, a todos los objetos instanciados de la clase.

En Java se puede acceder a los atributos de clase utilizando el nombre de la clase y el nombre de la variable, no es necesario instanciar ningún objeto de la clase para acceder a las variables de clase. En Java, se accede a los atributos de clase utilizando el nombre de la clase, el operador punto (.) y el nombre de la variable. La siguiente línea de código, ya archivista, se utiliza para acceder a la variable *out* de la clase *System*. En el proceso, se accede al método *println()* del atributo de clase que presenta una cadena en el dispositivo estándar de salida:

```
System.out.println("¡Hola mundo!");
```

Es importante recordar que todos los objetos de la clase comparten el mismo atributo de clase, porque si alguno de ellos modifica alguno de esos atributos de clase, quedarán modificados para todos los objetos de la clase. Esto puede utilizarse como una forma de comunicación entre objetos.

Métodos Estáticos

Cuando un método de una clase Java se utiliza la palabra *static*, se obtiene un método estático o método de clase.

Lo más significativo de los métodos de clase es que pueden ser invocados sin necesidad de que haya que instanciar ningún objeto de la clase. En Java, se puede invocar un método de clase utilizando el nombre de la clase, el operador punto y el nombre del método:

```
MiClase.miMetodoDeClase();
```

En Java, los métodos de clase operan solamente como los atributos de clase; no tienen acceso a variables de instancia de la clase, a no ser que se cree un nuevo objeto y se acceda a los atributos de instancia a través de ese objeto.

Si se observa el siguiente trozo de código de ejemplo:

```
class Documento
{
    static int version=10;
    int capitulos;

    static void agregarCapitulo()
    {
        capitulos++; //ESTO NO FUNCIONA
    }
    return;
}
```

Apunte de Cátedra

```

    }

    static void modificarVersion(int i)
    {
        version++;    //ESTO SI FUNCIONA
        return;
    }
}

```

La modificación del atributo *capitulos* no funciona porque se está violando una de las reglas de acceso al intentar acceder desde un método estático a un atributo no estático.

Java proporciona clases como: Math, Character, Integer, Float, Double, etc. que disponen de atributos y métodos estáticos.

La clase Math

Esta clase pertenece al paquete *java.lang*. La clase Math representa la librería matemática de Java. Es una clase estática, por lo tanto, tiene un constructor privado, lo que significa que no podemos hacer instancias de ella. Pero como todos sus miembros son estáticos, sí que podemos acceder a ellos. Para hacerlo simplemente ponemos el nombre de la clase (en este caso Math), un punto y el nombre de su atributo/método. Math contiene:

Atributos estáticos	Descripción
double Math.PI	Devuelve el número PI (3.1415...).
double Math.E	Devuelve la base del logaritmo natural.

Métodos estáticos	Descripción
int Math.abs(int) long Math.abs(long) float Math.abs(float) double Math.abs(double)	Calcula el valor absoluto de su parámetro (le cambia el signo si es negativo).
double Math.sin(double) double Math.cos(double) double Math.tan(double)	Calcula el seno, coseno o tangente del ángulo (expresado en radianes) que recibe como argumento.
double Math.asin(double) double Math.acos(double) double Math.atan(double)	Devuelve el ángulo (expresado en radianes) correspondiente al seno, coseno o tangente que reciben.
double Math.exp(double) double Math.sqrt(double)	Devuelve el elevado al parámetro. Devuelve la raíz cuadrada del parámetro.
double Math.ceil(double)	Redondea al entero menor de los que son mayores que el parámetro (redondeo hacia arriba).
double Math.floor(double)	Redondea al entero mayor de los que son menores que el parámetro (redondeo hacia abajo).
double Math.pow(double a, double b)	a elevado a b.
int Math rint(double) long Math.round(double) int Math.round(float)	Redondea al entero más cercano.

Apunte de Cátedra

int Math.max(int, int) long Math.max(long, long) float Math.max(float, float) double Math.max(double, double)	Devuelve el máximo de los dos valores que recibe.
int Math.min(int, int) long Math.min(long, long) float Math.min(float, float) double Math.min(double, double)	Devuelve el mínimo de los dos valores que recibe.

Veamos un ejemplo de la clase Math:

```
class UsoClaseMath
```

```
{
    public static void main(String[] args)
    {
        double num1, num2;
        float max;
        num1=Console.readDouble("Ingrese un numero real: ");
        System.out.println("El primer valor ingresado es: "+num1);
        System.out.println("El valor absoluto de "+num1+" es: "+
            Math.abs(num1));
        System.out.println("El cubo de "+num1+" es: "+Math.pow(num1, 3));
        num2=Console.readDouble("Ingrese otro numero real: ");
        System.out.println("El segundo valor ingresado es: "+num2);
        System.out.println("El redondeo de "+num2+" es: "+Math.round(num2));
        System.out.println("El máximo valor de los dos es: "+
            Math.max(num1, num2));
    }
}
```

La clase Character

Esta clase pertenece al paquete *java.lang*. Los métodos necesarios al trabajar con caracteres están contenidos en la clase *Character*. Esta clase no es estática, por lo tanto, se puede instanciar, aunque la mayoría de los métodos son estáticos.

Métodos estáticos	Descripción
char Character.isLowerCase(char c) char Character.isUpperCase(char c)	Devuelven el caracter c convertido a minúsculas o mayúsculas, respectivamente
boolean Character.isDigit(char c) boolean Character.isLetter(char c) boolean Character.isSpace(char c)	Nos devuelven verdadero si el caracter c es un dígito, una letra o un espacio.
int Character.digit(char c, int base)	Devuelve el número correspondiente al caracter c que recibe como primer parámetro suponiendo que está expresado en la base que recibe como segundo argumento.
char Character.forDigit(int x, int base)	Parecido al anterior, convierte a caracter el int x que recibe como primer parámetro suponiendo que está expresado en la base que recibe como segundo argumento.

Apunte de Cátedra

Suponiendo que tenemos un variable *carac* con la cual crearemos una instancia de la clase *Character*:

```
carac=new Character('J');
```

disponemos de los siguientes métodos para *carac*:

Métodos	Descripción
char carac.charValue()	Devuelve el caracter que almacena carac, en este caso 'J'.
String carac.toString()	Convierte el caracter que hay en carac en cadena.
int carac.compareTo(Character c)	Compara dos caracteres numéricamente, el caracter carac con c.

Veamos un ejemplo de la clase *Character*:

```
class UsoClaseCharacter
{
    public static void main(String[] args)
    {
        char character;
        int cantDig=0;
        int cantLet=0;
        System.out.println ("Serie de caracteres (finalizar con punto)");
        character=Console.readChar("Ingrese un caracter: ");
        while(character != '.')
        {
            if(Character.isDigit(character))
                cantDig++; //cuenta los dígitos
            else
                if(Character.isLetter(character))
                    cantLet++; //cuenta las letras
            character=Console.readChar("Ingrese otro caracter: ");
        }
        System.out.println("La Cantidad de Dígitos ingresados es: "+cantDig);
        System.out.println("La Cantidad de Letras ingresadas es: "+cantLet);
    }
}
```

La clase Integer

Cada tipo numérico tiene su propia clase de objetos. Así el tipo *int* tiene el objeto *Integer*. Se han codificado muchas funciones útiles dentro de los métodos de la clase *Integer*. La misma se encuentra en el paquete *java.lang*.

Veamos la diferencia:

```
int entero; //declara una variable entera
Integer entero=new Integer(); //declara y crea un objeto entero
```

Los siguientes son los atributos y métodos de la clase *Integer*:

Atributos estáticos	Descripción
Integer .MAX_VALUE;	Máximo valor de un objeto entero.
Integer .MIN_VALUE;	Mínimo valor de un objeto entero.

Apunte de Cátedra

Métodos estáticos	Descripción
String Integer.toString(int x)	Convierte el parámetro x en un String.
String Integer.toString(int x, int base)	Convierte el primer parámetro x en string con la base del segundo argumento.
Integer Integer.valueOf(String s)	Convierte en Integer el String s.
String Integer.toBinaryString(int x) String Integer.toOctalString(int x) String Integer.toHexaString(int x)	Convierte a String el int x según la base

Métodos	Descripción
int entero.parseInt(String s)	Convierte el String s en un entero.
String entero.toString()	Retorna el valor del objeto Integer entero en un String.
int entero.intValue()	Retorna el valor del objeto Integer entero en un int.
long entero.longValue()	Retorna el valor del objeto Integer entero en un long.
float entero.floatValue()	Retorna el valor del objeto Integer entero en un float.
double entero.doubleValue()	Retorna el valor del objeto Integer entero en un double.

Veamos ejemplos de la clase Integer:

Integer.parseInt() se usa para convertir un String en un entero. Podemos especificar la base cuando llamamos a este método. Aquí tenemos un ejemplo sencillo del uso de este método:

```
class UsoClaseInteger1
{
    public static void main(String[] args)
    {
        int ival=Integer.parseInt("10101", 2);
        System.out.println("El Valor 10101 en decimal es: "+ival);
    }
}
```

Se puede invertir el proceso con el método *Integer.toString()*:

```
class UsoClaseInteger2
{
    public static void main(String[] args)
    {
        int ival=12345;
        String s=Integer.toString(ival, 2);
        System.out.println("123456 en base 2: "+s);
        s=Integer.toString(ival, 8);
        System.out.println("123456 en base 8: "+s);
        s=Integer.toString(ival,20).toUpperCase();
        System.out.println("123456 en base 20: "+s);
    }
}
```

Cuando usamos *parseInt* para convertir números, no se acepta un prefijo como "0x" para hexadecimal, veamos un ejemplo.

Apunte de Cátedra

```
class UsoClaseInteger3
{
  public static void main(String[] args)
  {
    Integer in=Integer.decode("0123456");
    int ival=in.intValue();
    System.out.println("En decimal: "+Integer.toString(ival, 10));
    System.out.println("En binario: "+Integer.toString(ival, 2));
    System.out.println("En hexadecimal: "+
      Integer.toString(ival, 16).toUpperCase());
  }
}
```

La clase Float

El tipo float tiene el objeto Float. Se han codificado muchas funciones útiles dentro de los métodos de la clase Float. La misma se encuentra en el paquete *java.lang*.

Veamos la diferencia:

```
float flotante; //declara una variable de tipo float
Float flotante=new Float(); //declara y crea un objeto de tipo float
```

Los siguientes son los atributos y métodos de la clase Float:

Atributos estáticos	Descripción
Float.POSITIVE_INFINITY	Infinito negativo del tipo float.
Float.NEGATIVE_INFINITY	Infinito positivo del tipo float.
Float.NaN	Valor no-número.
Float.MAX_VALUE	Máximo valor.
Float.MIN_VALUE	Mínimo valor.

Métodos estáticos	Descripción
String Float.toString(float f)	Convierte el parámetro f a un String.
float Float.valueOf("3.14")	Asigna el literal a un tipo float
boolean Float.isNaN(float f)	La función isNaN() comprueba si el float f es un No-Número. Un ejemplo de no-número es raíz cuadrada de -2.
boolean Float.isInfinite(float f)	Comprueba si el float f es infinito.
String Float.toString(float f)	Convierte el parámetro f a un String.

Métodos	Descripción
String flotante.toString()	Convierte el objeto Float en un String.
int flotante.intValue()	Convierte el objeto Float en un Int.
long flotante.longValue()	Convierte el objeto Float en un long.
float flotante.floatValue()	Convierte el objeto Float en un float.
double flotante.doubleValue()	Convierte el objeto Float en un double

Apunte de Cátedra

Veamos un ejemplo de la clase Float:

```
class UsoClaseFloat
{
    public static void main(String[] args)
    {
        float f1=Console.readFloat("Ingrese un valor real: ");
        Float F=new Float(2.30);
        if(f1 < F.floatValue())
            System.out.println(f1+" es menor que "+F.floatValue());
        else
            if(f1 > F.floatValue())
                System.out.println(f1+" es mayor "+F.floatValue());
            else
                System.out.println(f1+" es igual a "+
                    F.floatValue());
    }
}
```

La clase Double

El tipo double tiene el objeto Double. Se han codificado muchas funciones útiles dentro de los métodos de la clase Double. La misma se encuentra en el paquete *java.lang*.

Veamos la diferencia:

```
double real; //declara una variable de tipo double
Double real=new Double(); //declara y crea un objeto de tipo double
```

Los siguientes son los atributos y métodos de la clase Double:

Atributos estáticos	Descripción
Double.POSITIVE_INFINITY	Infinito positivo del tipo Double
Double.NEGATIVE_INFINITY	Infinito negativo del tipo Double
Double.NaN	No-número
Double.MAX_VALUE	Máximo valor del tipo Double
Double.MIN_VALUE	Mínimo valor del tipo Double

Métodos estáticos	Descripción
boolean Double.isInfinite(double d) boolean Double.isNaN(double d)	Retorna verdadero si es Infinito/no-número.
Double Double.parseDouble(String s)	Convierte a Double el String s.
String Double.toString(double d)	Convierte a String el double d.

Métodos	Descripción
boolean real.isInfinite() boolean real.isNaN()	Retorna verdadero si es Infinito/no-número.
String real.toString() int real.intValue() float real.floatValue() Long real.longValue()	Retorna un Double real según el tipo especificado.
boolean real.equals(Object O)	Compara el objeto real con el O.

Apunte de Cátedra

Veamos un ejemplo de la clase Double:

```
class UsoClaseDouble
{
    public static void main(String[] args)
    {
        Double d1=new Double(3.14159);
        Double d2=new Double("314159E-5");
        if(d1.equals(d2))
            System.out.println("Son Iguales");
        else
            System.out.println("No son Iguales");
    }
}
```

La clase Random

Esta clase pertenece al paquete *java.util*. Esta clase se usa para generar una serie de números aleatorios. La clase usa una semilla de 48 bits que se modifica usando una fórmula de congruencia lineal. Si se crean dos casos de muestras al azar con la misma semilla, la sucesión de llamadas del método se hace para cada uno; ellos generarán y devolverán sucesiones idénticas de números. Para que la serie sean diferentes se deberán utilizar semillas diferentes para cada una.

La semilla se inicializa cuando se crea el objeto de la clase Random. Existen dos constructores: *Random()* que crea un nuevo generador de números aleatorios y *Random(long)* que crea un nuevo generador con una semilla para la generación.

Suponiendo que tenemos una variable *valAleatorio* con la cual crearemos una instancia de la clase Random:

```
valAleatorio=new Random();
```

entonces:

Métodos	Descripción
valAleatorio.nextInt()	Extrae un int, distribuido uniformemente a lo largo del rango de los enteros.
valAleatorio.nextLong()	Extrae un long, distribuido uniformemente a lo largo del rango de long.
valAleatorio.nextFloat()	Extrae un float, distribuido uniformemente entre 0.0 y 1.0.
valAleatorio.nextDouble()	Extrae un double, distribuido uniformemente entre 0.0 y 1.0.

Veamos un ejemplo de la clase Random:

```
//es necesario importar el paquete donde se encuentra la clase
import java.util.Random;
```

```
class UsoClaseRandom
{
    public static void main(String[] args)
    {
        Random alea=new Random();
        System.out.println("Diez números Random enteros");
        for(int i=1; i<=10; i++)
            System.out.println(alea.nextInt());
        System.out.println("Cinco números Random long");
    }
}
```

Apunte de Cátedra

```
for(int i=1; i<=5; i++)  
    System.out.println(alea.nextLong());  
}
```

Aclaración: Tener en cuenta que si se ejecuta varias veces este código, la serie de resultados serán “diferentes” en cada caso. Ahora bien, si cuando se crea el objeto *alea* se utiliza el otro constructor, por ejemplo:

Random alea=new Random(3000);

en cada ejecución del código la serie de resultados serán los mismos ya que se inicializa siempre con la semilla 3000.

Apunte de Cátedra

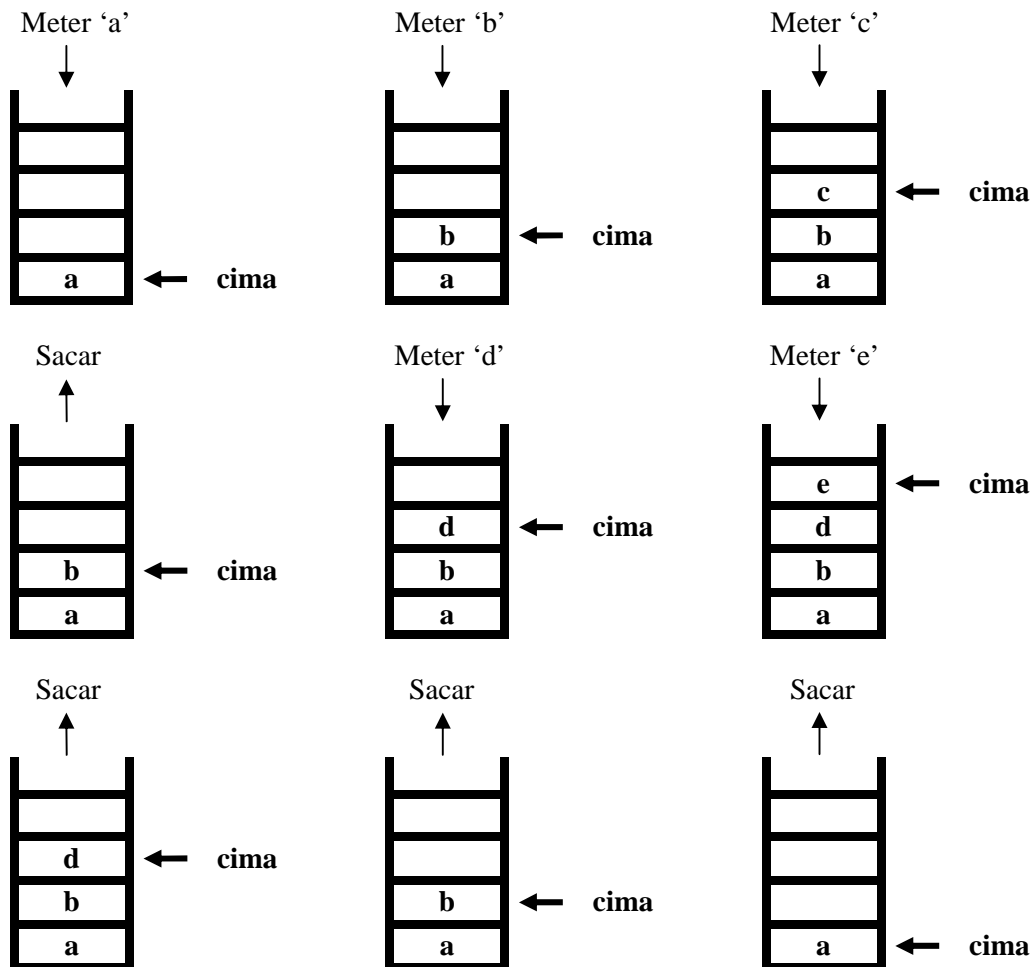
PILAS (LIFO)

Una pila (stack) es una estructura de datos (objeto), que consta de una serie de elementos (datos), en la cual las inserciones y eliminaciones se realizan por un extremo llamado cima (top) de la pila. La estructura pila se conoce también como LIFO (Last-In First-Out: último en entrar primero en salir), que significa “el último elemento insertado es el primer elemento en ser sacado”.

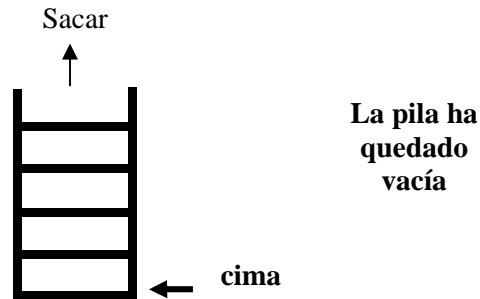
Existen numerosos casos prácticos en los que se utiliza el concepto de pila: por ejemplo, una pila de platos, una pila de latas en un supermercado, una pila de libros que se exhiben en una librería, etc. Es de suponer, para el primer caso, que si el cocinero necesita un plato limpio, tomará el que está encima de todos, que es el último que se colocó en la pila.

Las operaciones fundamentales de meter (push) y sacar (pop) elementos de la pila se hacen por un extremo de la pila llamado cima o cabeza. Es decir, el acceso a la información de la pila tiene restricciones.

Veamos la evolución a partir de una pila vacía luego de diferentes operaciones de entrada y salida:



Apunte de Cátedra



Las operaciones básicas con las pilas son:

- crear/inicializar: inicializar una pila vacía.
- pila vacía: determinar si una pila está vacía, es decir, no tiene elementos.
- pila llena: determinar si una pila está llena, es decir, si no caben más elementos.
- limpiar pila: quita todos los elementos y deja la pila vacía.
- meter: inserta un elemento en la cima de la pila.
- sacar: recupera y elimina el último elemento en la cima de la pila.

Estas operaciones se pueden ver como los siguientes métodos para la clase *Pila*:

```
Pila()           //crea-inicializa-limpia una pila
estaVacía()     //determina si la pila está o no vacía
estaLlena()     //determina si la pila está o no llena
limpiar()       //quita todos los elementos de la pila
meter(elem)     //mete el elemento elem en la pila
sacar()         //saca y devuelve un elemento de la pila
```

Posibles situaciones de error que se pueden presentar con las pilas:

- Intentar meter un elemento en una pila llena (*error de desbordamiento “overflow” o rebosamiento*). Forma parte de una precondición. Entonces, antes de meter un elemento en la pila:

```
if(!estaLlena())
    //meter el elemento
else
    //mensaje
```

- Intentar sacar un elemento en una pila vacía (*error de desbordamiento negativo “underflow”*). Forma parte de una precondición. Entonces antes de sacar un elemento de la pila:

```
if(!estaVacía())
    //sacar el elemento
else
    //mensaje
```

Implementación de una Pila

Una pila se puede implementar mediante arreglos unidimensionales (pila estática) o mediante listas enlazadas (pila dinámica). Veremos el primer caso según los siguientes niveles:

Apunte de Cátedra

1. Arreglos y un índice cima: en el arreglo guardaremos los elementos de la pila; usaremos un atributo cima para determinar el último elemento, como índice del arreglo y una constante con el máximo número de elementos.

```
class Pila
{
    private final int maxpila=100;           //cantidad de elementos de la pila
    private int[] elementos;                 //arreglo que representa a la pila
    private int cima;                        //la cima: último elemento

    //constructor de la pila
    public Pila()
    {
        elementos=new int[maxpila];
        cima=-1;
    }

    //verifica si la pila esta o no vacía
    public boolean estaVacia()
    {
        return (cima == -1);
    }

    //verifica si la pila esta o no llena
    public boolean estaLlena()
    {
        return (cima == maxpila - 1);
    }

    //agrega un elemento en la pila
    //precondición: pila no llena y elemento entero
    //postcondición: elemento entero cargado a la pila si no está llena
    public void meter(int elem)
    {
        if(!estaLlena())
        {
            cima++;
            elementos[cima]=elem;
        }
        return;
    }

    //saca un elemento del tope de la pila
    //precondición: pila no vacía
    //postcondición: elemento entero sacado de la pila si no está vacía
    public int sacar()
    {
        int aux=Integer.MIN_VALUE; //-2147483648
        if(!estaVacia())
        {
            aux=elementos[cima];
            cima--;
        }
        return aux;
    }
}
```

Apunte de Cátedra

La desventaja de esta implementación es que la pila solo puede ser de tipo entero.

2. Arreglo únicamente: en esta implementación se utiliza un arreglo que tiene de 0 a maxpila elementos, aprovechando la posición 0 del arreglo para llevar la cima.

```
class Pila
{
    private final int maxpila=100;        //cantidad de elementos de la pila
    private int[] elementos;              //arreglo que representa a la pila
    private final int cima=0;             //la cima: último elemento

    //constructor de la pila
    public Pila()
    {
        elementos=new int[maxpila];
        elementos[cima]=0;
    }

    //verifica si la pila esta o no vacía
    public boolean estaVacia()
    {
        return (elementos[cima] == 0);
    }

    //verifica si la pila esta o no llena
    public boolean estaLlena()
    {
        return (elementos[cima] == maxpila - 1);
    }

    //mete un elemento en la pila
    //precondición: pila no llena y elemento entero
    //postcondición: elemento entero metido en la pila si no está llena
    public void meter(int elem)
    {
        if(!estaLlena())
        {
            elementos[cima]++;
            elementos[elementos[cima]]=elem;
        }
        return;
    }

    //saca un elemento del tope de la pila
    //precondición: pila no vacía
    //postcondición: elemento entero sacado de la pila si no está vacía
    public int sacar()
    {
        int aux=Integer.MIN_VALUE; //-2147483648
        if(!estaVacia())
        {
            aux=elementos[elementos[cima]];
            elementos[cima]--;
        }
        return aux;
    }
}
```

Apunte de Cátedra

}
}

Las desventajas de esta implementación es que la pila solo puede ser de tipo entero y de la cantidad máxima de elementos se “roba” un espacio para la cima.

IMPORTANTE: En teoría, el tamaño de una pila no esta limitado más que por el espacio en memoria, en estas implementaciones como hemos utilizado arreglo (asignación estática de memoria) estamos restringidos al límite predefinido de los arreglos, más adelante veremos como implementar una pila y no preocuparnos por esto.

Pese a la similitud, es preciso notar que la pila no es igual que un vector. Una pila es, por esencia, un objeto dinámico mientras que un vector tiene un número máximo de elementos, determinado por su definición.

Apunte de Cátedra

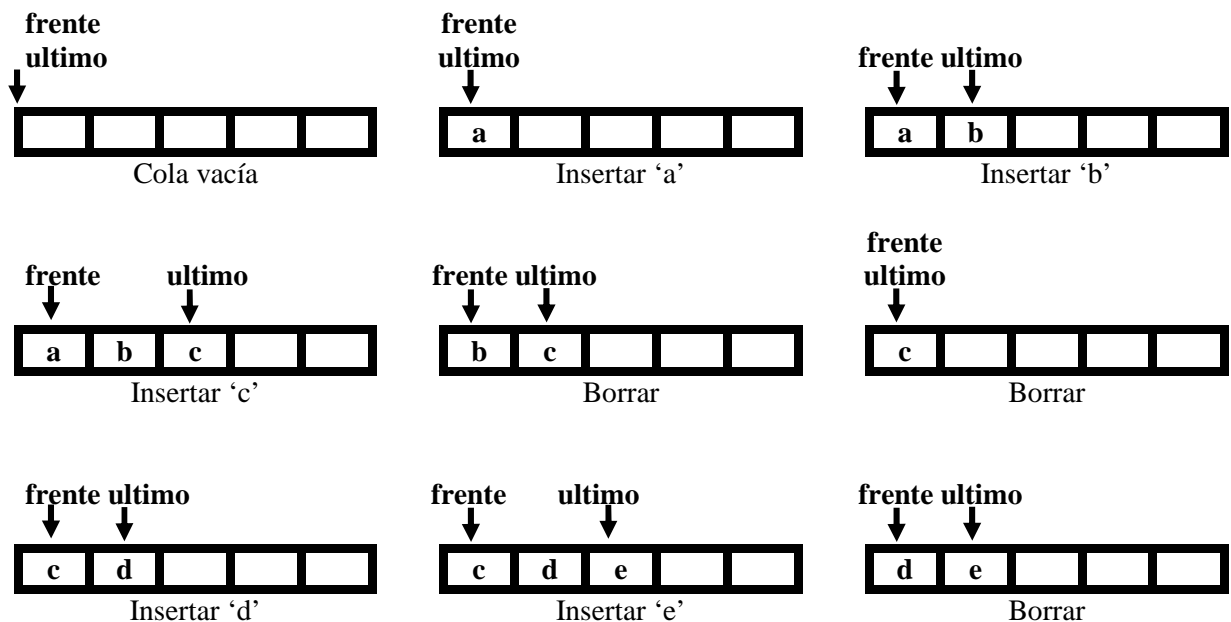
COLAS (FIFO)

El diccionario de la Real Academia Española define una acepción de cola como “hilera de personas que esperan turno para alguna cosa”: una hilera de vehículos esperando pasar una frontera, una hilera de personas para entrar en un cine, una hilera de personas en la caja de un supermercado, la cola de los bancos en las que los clientes esperan para ser atendidos o una cola de trabajos en un sistema de computadoras que espera la disponibilidad de algún dispositivo de salida tal como una impresora.

En cada uno de estos ejemplos, los elementos se atienden en el orden que llegaron; es decir, el primer elemento en entrar (primero en la cola) es el primero en ser atendido (salir). Por consiguiente, una cola es una estructura FIFO (First-In First-Out: primero en entrar primero en salir), también llamada a veces estructura FCFS (First-Come First-Served: primero que llega primero que se atiende).

Una cola (Queue) es una estructura de datos (objeto) sobre la cual se aplican las operaciones básicas de añadir y eliminar elementos por un extremo de la cola. Los elementos se borran por un extremo llamado frente y se insertan por un extremo llamado último.

Veamos la evolución de la siguiente cola:



Las operaciones básicas con las colas son:

- crear/inicializar: inicializar una cola vacía.
- cola vacía: determinar si una cola esta vacía.
- cola llena: determinar si una cola esta llena.
- limpiar cola: quita todos los elementos y deja la cola vacía.
- insertar: añade un elemento a la parte última de la cola.
- borrar: recupera y elimina el elemento al frente de la cola.

Estas operaciones se pueden ver como los siguientes métodos para la clase *Cola*:

```
Cola()           //crea-inicializa-limpia una cola
estaVacía()     //determina si la cola está o no vacía
```

Apunte de Cátedra

```
estaLlena() //determina si la cola está o no llena  
limpiar() //quita todos los elementos de la cola  
insertar(elem) //inserta el elemento elem en la cola  
borrar() //borra y devuelve un elemento de la cola
```

Posibles situaciones de error que se pueden presentar con las colas:

- Intentar insertar un elemento en una cola llena. Forma parte de una precondición. Entonces, antes de insertar un elemento en la cola:

```
if(!estaLlena())  
    //insertar el elemento  
else  
    //mensaje
```

- Intentar borrar un elemento en una cola vacía. Forma parte de una precondición. Entonces antes de borrar un elemento de la cola:

```
if(!estaVacía())  
    //borrar el elemento  
else  
    //mensaje
```

Implementación de una Cola

Al igual que las pilas, las colas se pueden implementar utilizando arreglos o listas enlazadas. Veremos también el primer caso según los siguientes niveles:

1. **Frente fijo y último movable**: esta solución es similar a la pila. En los siguientes esquemas se representa las distintas situaciones. El frente (subíndice del arreglo) debe estar siempre fijo (en 0), se insertan elementos por el último (subíndice) y cuando se borran elementos de la cola, para poder mantener la premisa del frente fijo, será necesario desplazar los elementos restantes de la cola. La ventaja de este diseño es su simplicidad, dado que se utilizan prácticamente los mismos algoritmos que para una pila. La desventaja es tener que mover todos los elementos de la cola cada vez que se borra un elemento de la cola.
2. **Frente y ultimo movibles o flotantes (colas circulares)**: La desventaja de la implementación anterior se puede resolver llevando un índice para el frente al igual que para el ultimo y mantener ambos extremos flotantes. La operación de insertar sigue siendo la misma, incrementando el último. La operación de suprimir, en este caso incrementaría el frente. Pero se plantea un problema: cuando el índice último llega al último elemento del arreglo, podríamos pensar que la cola esta llena, sin embargo es posible que no sea así. Para ello, trataremos al arreglo como una estructura circular (Figura 1.a). Es decir, el elemento que le sigue al último es el primero. Para obtener la siguiente posición del último, utilizamos el siguiente método:

```
private int siguiente(int subind)  
{  
    if(subind == maxcola -1)  
        return 0;  
    else  
        return subind++;  
}
```

Apunte de Cátedra

Pero ahora se nos plantea un nuevo problema: ¿Cómo sabemos si la cola esta llena o vacía?. Para ello, el atributo frente indica la posición del arreglo que “precede” al primer elemento de la cola, en lugar del índice al propio elemento frente. Y el atributo último es la posición en donde se hizo la última inserción, último elemento de la cola. Se avanza en el sentido de las agujas del reloj.

La cola estará vacía cuando ambos índices sean iguales (Figura 1.b). La cola estará llena cuando el siguiente al último sea igual al frente. Para insertar un elemento a la cola se debe: incrementar el último y agregar el elemento (Figura 1.c). Para borrar un elemento de la cola se debe: incrementar el frente.

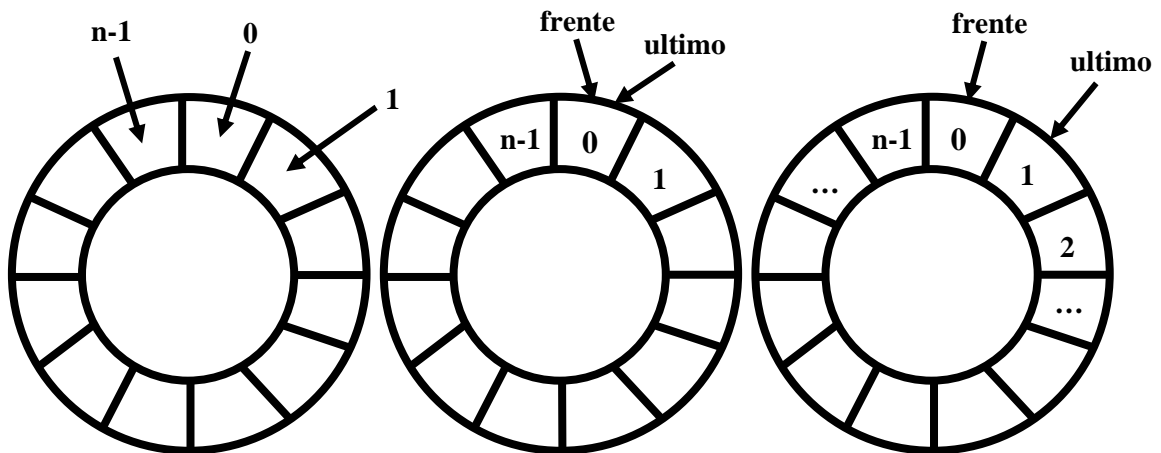


Figura 1: a) cola circular; b) cola circular vacía; c) cola circular que contiene un elemento.

```
class Cola
{
private final int maxcola=50;           //cantidad de elementos de la cola
private int[] elementos;                //arreglo que representa a la cola
private int frente, ultimo;             //índices para insertar y borrar

// Constructor de la cola
public Cola()
{
    elementos=new int[maxcola];
    frente=0;
    ultimo=0;
}

//verifica si la cola esta o no vacía
public boolean estaVacía()
{
    return (ultimo == frente);
}

//verifica si la cola esta o no llena
public boolean estaLlena()
{
    int sigultimo=siguiente(ultimo);
    return (sigultimo == frente);
}
```

Apunte de Cátedra

```
//establece cuál es el siguiente elemento
private int siguiente(int subind)
{
    if(subind == maxcola - 1)
        return 0;
    else
        return ++subind;
}

//inserta un elemento en la cola
//precondición: cola no llena y elemento entero
//postcondición: elemento entero insertado en cola si no está llena
public void insertar(int elem)
{
    if(!estaLlena())
    {
        ultimo=siguiente(ultimo);
        elementos[ultimo]=elem;
    }
    return;
}

//borra un elemento de la cola
//precondición: cola no vacía
//postcondición: elemento entero borrado de la cola si no está vacía
public int borrar()
{
    if(!estaVacía())
    {
        frente=siguiente(frente);
        return elementos[frente];
    }
    else
        return Integer.MIN_VALUE; //-2147483648
}
}
```

Aplicaciones de las colas

Las colas se utilizan con frecuencia en informática, siempre que más de un proceso requiera un recurso específico, tal como una impresora, una unidad de disco o la unidad central de proceso. Tal proceso, al solicitar un recurso específico, se sitúa en una cola a la espera de atender a ese proceso. Por ejemplo: diferentes computadoras pueden compartir la misma impresora, y una cola spool se utiliza para planificar las diferentes salidas. Si una petición de impresión llega y la impresora esta libre inmediatamente pasa a impresión, mientras se este imprimiendo las peticiones que lleguen serán almacenadas en la cola spool y a medida que la impresora termine serán realizadas, liberando la cola spool.

Otro uso importante de las colas, es para las operaciones de entrada/salida de buffers. Los buffers son memorias intermedias que se comportan como colas entre los dispositivos de E/S y memoria principal. Otro uso, es la CPU de un sistema operativo multiusuario, trabaja como una cola, cuando distintos usuarios (workstation) solicitan procesos, van entrando a una cola de procesos y saliendo a medida que el procesador va ejecutándolos (según las prioridades). Esto se conoce como colas de prioridad.

Apunte de Cátedra

RECURSIVIDAD

La recursividad es una técnica de programación muy potente que puede ser utilizada en lugar de la iteración (bucles). Ello implica una forma diferente de ver las acciones repetitivas, permitiendo que un subprograma se llame así mismo para resolver una versión “más pequeña” de su problema original.

Algunas definiciones de recursividad:

- Recursión: ver recursividad.
- Un objeto es recursivo si figura en su propia definición.
- Una definición recursiva es aquella en la que el objeto que se define forma parte de la definición.

Veamos un ejemplo:

Analicemos la definición matemática del factorial de N (N!).

$$\begin{aligned}N &= 5 \\ N! &= N * (N-1) * (N-2) * (N-3) * \dots * (N-(N-1)) \\ 5! &= 5 * (5-1) * (5-2) * (5-3) * (5-4) \\ 5! &= 5 * 4 * 3 * 2 * 1 \\ 5! &= 120\end{aligned}$$

El método para calcular el factorial es el siguiente:

```
public int factorial(int N)
{
    //declaración de variables locales
    int cont, fact;

    //cálculo
    fact=1;
    for(cont=2; cont<=N; cont++)
        fact=fact * cont;
    return fact;
}
```

Puede verse también como que 5! es igual a 5 * 4!

$$\text{Generalizando N!}: \begin{cases} 1 & \text{si } N = 0 \\ N * (N - 1)! & \text{si } N > 0 \end{cases}$$

Esta generalización es una definición recursiva donde:

1. La condición de salida es: $N = 0$ (denominado **CASO BASE**)
2. La parte recursiva es: $N! = N * (N - 1)!$ (denominado **CASO GENERAL**)

Entonces, podemos afirmar que esta función es recursiva, dado que puede ser expresada en función de sí misma, es decir: $\text{factorial}(N) = N * \text{factorial}(N - 1)$

Por lo tanto, la solución recursiva sería:

```
public int factorial(int valor)
{
    if(valor == 0)
        return 1;
    else
        return valor * factorial(valor - 1);
}
```

Apunte de Cátedra

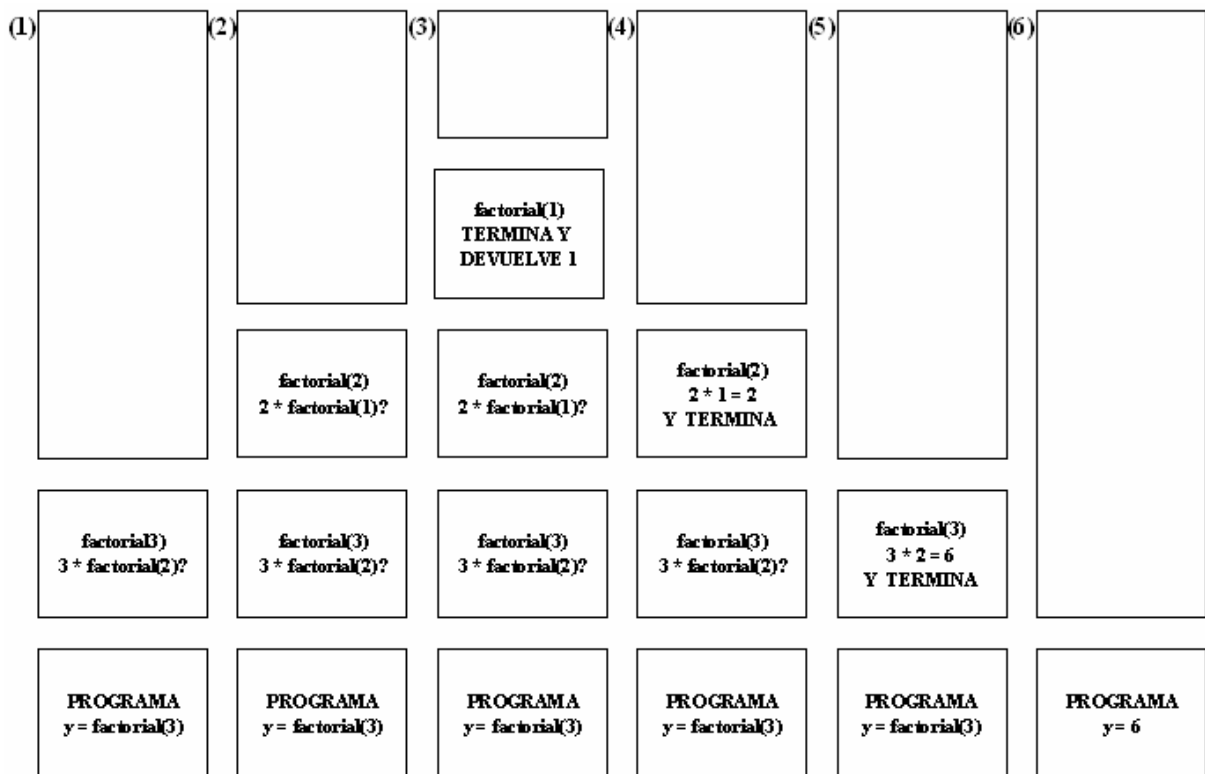
Si la llamada es factorial(3) esto se ejecuta de la siguiente manera:

- 1er. llamada: $N \neq 0$
factorial = $3 * \text{factorial}(2)$
- 2da. llamada: $N \neq 0$
factorial = $2 * \text{factorial}(1)$
- 3er. llamada: $N \neq 0$
factorial = $1 * \text{factorial}(0)$
- 4ta. llamada: $N = 0$
factorial = 1

*** Vuelve hacia atrás ***

- 3er. llamada: factorial = $1 * 1 = 1$
- 2da. llamada: factorial = $2 * 1 = 2$
- 1er. llamada: factorial = $3 * 2 = 6$

Por lo tanto, factorial(3) = 6



Suponiendo que la llamada se genere de la siguiente manera:

$$y = \text{factorial}(3);$$

Lo que se hace, es ir guardando todas las llamadas sin resolver en una pila, cuando llega a una que tiene solución comienza a extraer de la pila las llamadas y a resolverlas.

Este tipo de recorrido para comprobar la validez de un algoritmo recursivo consume tiempo. Existe una técnica que sirve para verificar o comprobar si una solución o algoritmo se puede implementar en

Apunte de Cátedra

forma recursiva, puesto que la traza consume tiempo, es confusa y tediosa. Esta técnica nos ayuda inductivamente a determinar si un algoritmo recursivo funciona o no, y se denomina método de las tres preguntas.

El Método de las Tres Preguntas

Para comprobar que una solución recursiva funciona debe ser capaz de contestar SI a estas tres preguntas:

- La pregunta CASO BASE: ¿Hay una salida no recursiva de la función?. ¿La rutina funciona correctamente para este caso “base”?.

CASO BASE: Instancia que se puede resolver sin hacer llamadas recursivas. Toda llamada recursiva debe tender hacia un CASO BASE.

- La pregunta LLAMADOR MAS PEQUEÑO: ¿Cada llamada recursiva a la función se refiere a un caso más pequeño del problema original?.

La respuesta a esta pregunta debe buscarse en los parámetros pasados en la llamada recursiva.

- La pregunta CASO GENERAL: Suponiendo que la(s) llamada(s) recursivas funcionan correctamente, ¿funciona correctamente todo la función?.

Apliquemos estas preguntas a la función factorial:

- La pregunta CASO BASE: la respuesta es SI, cuando $N=0$ se produce el CASO BASE.
- La pregunta LLAMADOR MAS PEQUEÑO: la respuesta es SI, esta dado por el parámetro de la llamada ($N-1$).
- La pregunta CASO GENERAL: la respuesta es SI, dado que hemos implementado la función de acuerdo a su definición matemática o fórmula $N * (N-1)!$

Escritura de Algoritmos (Métodos) Recursivos

- 1) Obtener una definición exacta del problema a resolver.
- 2) Determinar el tamaño del problema completo que hay que resolver. Este tamaño determinará los valores de los parámetros en la llamada inicial a la función.
- 3) Resolver el CASO BASE donde el problema puede expresarse no recursivamente. Asegurando la respuesta SI a la pregunta CASO BASE.
- 4) Resolver el CASO GENERAL correctamente en términos de un caso más pequeño del mismo problema (una llamada recursiva). Esto asegurará una respuesta SI a las preguntas LLAMADOR MAS PEQUEÑO y CASO GENERAL.

La Eficiencia (Iteración vs. Recursividad)

La recursividad parece un método simple para resolver problemas mediante relaciones recursivas; sin embargo, en ocasiones se puede volver un método ineficiente ya que se realizan más cálculos que su equivalente solución repetitiva.

Por ejemplo, supongamos que utilizamos un algoritmo recursivo para hallar un valor en un arreglo de 100 elementos. Cada vez que se invoca al método, se hacen copias de todos los parámetros, anidando más y más niveles de llamadas recursivas. Si el valor buscado no se halla en el vector, en un momento tendremos 100 copias del arreglo de 100 elementos en memoria. Con el tiempo, podemos agotar la memoria.

La eficacia de una solución recursiva viene medida por una serie de factores, entre otros se destacan:

- *Tiempo de ejecución*: Se consideran más eficientes aquellos algoritmos que cumplen con la especificación del problema en el menor tiempo posible.
- *Espacio en memoria ocupado por el algoritmo*: Serán eficientes los algoritmos que utilicen las estructuras de datos adecuadas, a fin de minimizar la memoria ocupada.

Apunte de Cátedra

- **Legibilidad y facilidad de comprensión:** Mayor sencillez al momento de interpretar el algoritmo.

Desde el punto de vista de la ejecución, los algoritmos recursivos son más lentos y ocupan mayor espacio en memoria, con el inconveniente que puede suponer en el caso de poca memoria. Sin embargo, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa; en estos casos la ganancia en claridad puede compensar con creces el costo de tiempo y espacio en memoria de la ejecución de programas recursivos.

Las definiciones recursivas se caracterizan por su elegancia y simplicidad en contraste con la dificultad y falta de claridad de las definiciones repetitivas.

La solución recursiva es estéticamente más agradable, y una vez acostumbrado, la forma recursiva es normalmente más fácil de leer y comprender que la forma iterativa.

Como guía general, si la solución no recursiva es más corta - o no mucho más larga - que la versión recursiva, no utilizar recursividad. Por ejemplo, el caso del factorial.

Veamos un cuadro comparativo entre las dos soluciones:

Solución Iterativa	Solución Recursiva
Utiliza una construcción cíclica (for, while, do-while)	Utiliza una estructura de ramificación (if-else)
Necesita variables locales (contadores)	Utiliza los parámetros de la función para dar toda su información
Los datos utilizados para el ciclo son inicializados antes del mismo (dentro de la rutina)	Los datos utilizados son, usualmente, inicializados en la elección del valor del parámetro en la llamada inicial
Son más largos (más líneas de código)	Son más cortos (menos líneas de código)

Formas y Tipos de Recursividad

Existen dos formas de recursividad:

- Directa: la función se llama directamente a sí mismo.
- Indirecta: un método llama otro método, y éste, en algún momento, llama nuevamente al primero.

Existen tres tipos de recursividad

- 1) Recursividad simple. Es todo algoritmo en el cual solo aparece una llamada recursiva. Este caso fácilmente se transforma en iteración

Algoritmo Recursivo (parámetros)

```
{
  if(caso base)
    [fin]
  else //caso general
    AlgoritmoRecursivo(llamador_más_pequeño)
}
```

Apunte de Cátedra

2) Recursividad múltiple. Se presenta cuando existe más de una llamada a si mismo dentro del cuerpo del algoritmo. Este caso es más complicado para transformar en una iteración.

AlgoritmoRecursivo(parámetros)

```
{
  if(caso base)
    [fin]
  else //caso general
    AlgoritmoRecursivo(llamador_más_pequeño),
    AlgoritmoRecursivo(llamador_más_pequeño)
}
```

3) Recursividad anidada. Es el caso donde existe más de un caso general.

AlgoritmoRecursivo(parámetros)

```
{
  if(caso base)
    [fin]
  else
    if(caso general)
      AlgoritmoRecursivo(llamador_más_pequeño)
    else
      AlgoritmoRecursivo(llamador_más_pequeño)
}
```

❖ Ejemplos.

1. El siguiente método muestra los elementos de un vector en orden inverso.

```
//ultimo: posición del último elemento (dimension-1)
public void invertirVector(int[] vector, int ultimo)
{
  if(ultimo == 0)
    System.out.println(vector[0]);
  else
  {
    System.out.println(vector[ultimo]);
    invertirVector(vector, ultimo - 1);
  }
  return;
}
```

2. El siguiente método devuelve la suma de los elementos de un vector.

```
public int sumatoriaVector(int[] vector, int ultimo)
{
  if(ultimo == 0)
    return vector[0];
  else
    return (vector[ultimo] + sumatoriaVector(vector, ultimo - 1));
}
```

Apunte de Cátedra

La misma solución con distinto algoritmo:

```
//primero: posición del primer elemento (0)
public int sumatoriaVector(int[] vector, int primero)
{
    if(primero == dimension - 1)
        return vector[primero];
    else
        return (vector[primero] + sumatoriaVector(vector, primero+1));
}
```

ANEXO: Los Lenguajes de Programación

Para que un procesador realice un proceso, se le debe suministrar en primer lugar un algoritmo adecuado. El procesador debe ser capaz de **interpretar** el algoritmo, lo que significa:

- comprender las instrucciones de cada paso.
- realizar las operaciones correspondientes.

Cuando el procesador es la computadora, el algoritmo se ha de expresar en un formato que se denomina *programa*. Un programa se escribe en un *lenguaje de programación* y las operaciones que conducen a expresar un algoritmo en forma de programa se llaman *programación*.

Instrucciones a la computadora

Los diferentes pasos de un algoritmo se expresan en los programas como *instrucciones*. Por consiguiente, un programa consta de una secuencia de instrucciones, cada una de las cuales especifican ciertas operaciones que debe ejecutar la computadora.

Las instrucciones básicas y comunes a casi todos los lenguajes de programación se pueden agrupar en:

- **Instrucciones de entrada/salida:** Instrucciones de transferencia de información y datos, entre los dispositivos periféricos (teclado, impresora, unidad de disco, etc.) y la memoria central.
- **Instrucciones aritmético-lógicas:** Instrucciones que ejecutan operaciones aritméticas (suma, resta, multiplicación, división, potenciación), lógicas (operaciones &, |, etc.).
- **Instrucciones selectivas:** Instrucciones que permiten la selección de tareas alternativas en función de los resultados de diferentes expresiones condicionales.
- **Instrucciones repetitivas:** Instrucciones que permiten la repetición de secuencias de instrucciones un número determinado o indeterminado de veces.

Principales tipos de lenguajes utilizados en la actualidad

1. Lenguajes Máquina.

Son aquellos que están escritos en lenguajes directamente inteligibles por la máquina (computadora), ya que sus instrucciones son cadenas binarias (cadenas o series de caracteres –dígitos- 0 y 1) que especifican una operación, y las posiciones (dirección) de memoria implicadas en la operación se denominan instrucciones de máquina o código máquina. El código máquina es el conocido código binario.

Las instrucciones en lenguaje máquina dependen del hardware de la computadora y, por lo tanto, diferirán de una computadora a otra.

Ventajas:

Posibilidad de cargar (transferir un programa a memoria) sin necesidad de traducción posterior, lo que supone una velocidad de ejecución superior a cualquier otro lenguaje de programación.

Desventajas:

- Dificultad y lentitud en la codificación.
- Poca fiabilidad.
- Dificultad de verificar y poner a punto los programas.
- Los programas sólo son ejecutables en el mismo procesador (UCP).

2. Lenguajes de Bajo Nivel.

Son más fáciles de usar, pero al igual que los lenguajes máquina, dependen de la máquina en particular. El lenguaje de bajo nivel por excelencia es el *ensamblador (Assembly Language)*.

Apunte de Cátedra

Un programa escrito en lenguaje ensamblador no puede ser ejecutado directamente por la computadora - en esto se diferencia esencialmente del lenguaje máquina -, sino que requiere una fase de *traducción* al lenguaje máquina.

El programa original escrito en lenguaje ensamblador se denomina *programa fuente* y el programa traducido en lenguaje máquina se conoce como *programa objeto*, ya directamente inteligible por la computadora.

Ventaja:

Respecto a los lenguajes máquina:

- Mayor facilidad de codificación.
- Mayor velocidad de cálculo.

Desventajas:

- Dependencia total de la máquina, lo que impide la transportabilidad de los programas.
- La formación de los programadores es más compleja que la correspondiente a los programadores de alto nivel, ya que exige no sólo las técnicas de programación, sino también el conocimiento del interior de la máquina.

3. Lenguajes de Alto Nivel.

Son los más utilizados por los programadores. Un programa escrito en un lenguaje de alto nivel es independiente de la máquina, esto es, las instrucciones del programa no dependen del diseño del hardware o de una computadora particular. En consecuencia, los programas escritos en lenguajes de alto nivel son *portables* o *transportables*, lo que significa la posibilidad de poder ser ejecutados con poca o ninguna modificación en diferentes tipos de computadoras.

Los programas fuentes tienen que ser traducidos por programas traductores llamados en este caso *compiladores* e *intérpretes*.

Los lenguajes de alto nivel existentes son muy numerosos, algunos de ellos son: Java, Pascal, C, C++, Cobol, Fortran, Basic, VisualBasic, Clipper, Prolog, Delphi, Lisp, etc.

Intérprete: es un traductor que toma un programa fuente, lo traduce y a continuación lo ejecuta.

Compilador: es un programa que traduce los programas fuente escritos en lenguajes de alto nivel a lenguaje máquina.

Los programas escritos en lenguaje de alto nivel se llaman *programas fuente* y el programa traducido *programa objeto*. El programa objeto obtenido de la compilación (proceso de traducción) ha sido traducido normalmente a código máquina. Para conseguir el programa máquina real se debe utilizar un programa llamado *montador* o *enlazador* (*linker*). El proceso de montaje conduce a un programa en lenguaje máquina directamente ejecutable.

Apunte de Cátedra

BIBLIOGRAFIA

1. “Algoritmos + Estructuras de Datos = Programas”. Nicklaus Wirth. Prentice Hall. 1989.
2. “Algoritmos, Datos y Programas: Conceptos Básicos”. Armando De Giusti, Cristina Madoz, Rodolfo Bertone, Marcelo Naiouf, Laura Lanzarini, Gladys Gorga y Claudia Russo. Editorial Exacta. Argentina. 1998.
3. “Cómo Programar en JAVA”. Harvey M. Deitel y Paul J. Deitel. Prentice Hall. México. 1997.
4. “Construcción de Software Orientado a Objetos”. Bertrand Meyer. Prentice Hall. Segunda edición. España. 1999.
5. “Core JAVA Volume I-Fundamentals”. Cay S. Horstmann y Gary Cornell. The Sun Microsystem Press JAVA Series. 1999.
6. “El Lenguaje de Programación JAVA”. Ken Arnold, James Gosling y David Holmes. Addison – Wesley. Tercera edición. 2001.
7. “Estructuras de Datos”. Osvaldo Cairó y Silvia Guardati. Mc. Graw Hill. Tercera edición. México. 2006.
8. “Estructuras de Datos: Algoritmos, Abstracción y Objetos”. Luis Joyanes Aguilar e Ignacio Zahonero Martínez. Mc. Graw Hill. España. 1998.
9. “Estructuras de Datos en Java”. Mark Allen Weiss. Addison – Wesley. España. 1998.
10. “Estructuras de datos y algoritmos”. Alfred Aho, John Hopcroft y Jeffrey Ullman. Addison-Wesley. Estados Unidos. 1988.
11. “Fundamentos De Programación”. Luis Joyanes Aguilar. Mc Graw Hill. España. 1996.
12. “Introducción a la Programación Orientada a Objetos con Java”. C. Thomas Wu. Mc. Graw Hill. España. 2001.
13. “Pascal”. Nell Dale y Chip Weems. Mc. Graw Hill. 2da. Edición. 1989.
14. “Pascal. Introducción al lenguaje y resolución de problemas con programación estructurada”. Elliot B. Koffman. Addison – Wesley. 1986.
15. “Pascal y Estructuras de Datos”. Neil Dale y Susan Lilly. Mc. Graw Hill. Segunda edición. México. 1992
16. “Programación con C++”. Al Stevens y Clayton Walnum. Anaya. 2000.
17. “Programación en Java 2. Algoritmos, Estructuras de Datos y Programación Orientada a Objetos”. Luis Joyanes Aguilar e Ignacio Zahonero Martínez. Mc. Graw Hill. España. 2002.