

# TEMA 1. DISEÑO ALGORÍTMICO

Como cualquier tema que deseemos abordar, el diseño algorítmico requiere la comprensión de ciertos conceptos básicos que constituirán el andamiaje sobre los que el aprendiz o practicante sustentará sus nuevos conocimientos, habilidades y destrezas. Por tanto para iniciar nuestra aventura en los fundamentos de programación, y con la conciencia de que mucho bla, bla, bla sería sumamente aburrido, buscaré ser breve y concisa en la descripción de dichos conceptos.

## CONCEPTOS BÁSICOS

### *Sistema computacional*

Para empezar es necesario recordar que un *sistema* es un conjunto de partes relacionadas entre sí que tienen un propósito común. Por ejemplo el sistema solar integrado por el sol, los planetas, asteroides, etc., o nuestro sistema respiratorio por las vías aéreas, pulmones y músculos respiratorios.

Luego entonces un *sistema computacional* está constituido por (Blanco Vega, 1998):

- *Hardware (hw)*. Es la parte tangible, los fierros, lo que puedo tocar: el monitor, el teclado, la Unidad central de procesamiento, el disco duro, la impresora, etc.
- *Software (sw)*. Es intangible, lo que no puedo tocar: los programas, los datos, el sistema operativo, etc.
- *Peopleware (pw)*. Se refiere al factor humano, lo esencial de un sistema. Las personas que dan vida al hw y al sw: analistas, diseñadores, programadores, usuarios finales, administradores, etc. Es importante mantener en mente que sin ellos el sistema pierde su valor, ya que una computadora no es mejor que una persona, solo es un equipo electrónico con la capacidad de procesar información a una gran velocidad.

El propósito del *sistema computacional* es el manejo de datos, de tal forma que éstos sean correctos, adaptables, flexibles y se encuentren disponibles para apoyar el proceso de toma de decisiones de una organización (Loomis, 1991). Lo que se traduce en que la información debe encontrarse a la mano del usuario, de la forma y en el momento que éste la precise, no antes, ni después. Hago énfasis en este punto, porque será determinante en tu formación como programador, el que comprendas que aun cuando podemos sugerir, hacer propuestas e incluso guiar al cliente para que puntualice lo que le hace falta. Nuestro trabajo será entonces, resolver su problema facilitándole lo que necesita y no lo que nosotros creamos que necesita.

### *Clasificación del software de: sistemas y aplicación*

Tal como una persona se le deben proveer instrucciones precisas para realizar una actividad determinada, para que tu equipo de cómputo o dispositivo móvil funcione requiere de cierto software (programas), es decir, de instrucciones electrónicas que le muestren el camino a seguir para realizar tareas específicas como crear documentos, diseñar presentaciones, realizar cálculos, es decir, hacernos la vida más fácil. En general dicho software puede ser dividido en dos grandes categorías: *software del sistema* y *software de aplicación*.

#### **Software de sistema**

El *software de sistema* o *software de base*, es quien le indica a la computadora como interactuar con el usuario y cómo usar los dispositivos de entrada/salida y de almacenamiento. Una vez que la computadora ha sido encendida, realiza las siguientes actividades:

1. Una autoprueba en la que detecta los dispositivos que tiene conectados, la cantidad de memoria de que dispone y verifica que su propio funcionamiento sea correcto.
2. Busca el Sistema Operativo, lo carga en la memoria y lo mantiene en ejecución hasta que el equipo sea apagado.
3. Una vez realizado lo anterior, el equipo estará listo para recibir (a través del sistema operativo), los comando necesarios para manipular los dispositivos o incluso para ejecutar otros programas de uso específico.

Investigación individual del estudiante: ¿Cuáles son los tipos de software de sistema? Describa brevemente para que sirven.

## Software de aplicación

Una vez que la computadora está funcionando, la utilidad que esta tendrá para el usuario estará determinada por los programas con los que cuente, que por supuesto deberán ser los que le faciliten las tareas específicas que desea realizar. Dichos programas son llamados *software de aplicación*, cotidianamente llamadas *aplicaciones*, y *apps*. Aun cuando hay muchos tipos de software de aplicación, se pueden considerar las siguientes categorías como las más importantes (Norton, 1995):

- *Aplicaciones de negocios:* procesador de palabras, hoja de cálculo, diseño de presentaciones, bases de datos, software de CAD, etc.
- *Aplicaciones de utilerías:* programas para facilitar tareas de administración de dispositivos y recursos difíciles o no incluidas en el sistema operativo.
- *Aplicaciones personales:* Agenda personal, rediseño de casa, calendario de citas, correo electrónico, pagos en línea, etc.
- *Aplicaciones de entretenimiento:* videojuegos, simuladores de vuelo, rompecabezas, etc.

## Algoritmo

Un *algoritmo* es una serie de pasos organizados que describe el proceso que se debe seguir, para dar solución a un problema específico. Al diseñar una solución es importante considerar que un algoritmo debe ser (Joyanes Aguilar, Fundamentos de programación. Algoritmos, estructura de datos y objetos, 2008):

- *Preciso.* Una computadora no tiene criterio, ni sentimientos, por lo tanto los pasos a seguir para realizar una tarea, deberán estar claramente establecidos en un orden que no deje lugar a dudas.
- *Definido.* Al probar un algoritmo con los mismos datos de entrada, deberá dar siempre los mismos resultados de salida.
- *Finito.* En un algoritmo no caben los etcéteras, ni los puntos suspensivos, lo que significa que los pasos que lo integren siempre podrán ser contados con precisión.

- *Independiente del lenguaje de programación en el que será codificado.* Cuando el analista trabaje en la solución del problema, lo hará con independencia del lenguaje que utilizará para escribir el programa, es decir que solo centrará su atención y habilidades en el diseño de una solución que podrá representar a través de un lenguaje algorítmico.

Quedemos claros entonces que todos nosotros hemos diseñado un algoritmo, cuando hemos planeado unas vacaciones, cuando establecimos una ruta para acudir a un compromiso, cuando definimos una estrategia para que nuestros padres o el jefe nos dieran algún permiso. En fin, cada vez que hemos resuelto un problema o simplemente determinamos para una situación dada, que íbamos a decir o hacer, el orden en que lo haríamos, que necesitábamos para hacerlo, con quien debíamos hablar, como íbamos a ir vestidos, etc., lo que estábamos haciendo era diseñar una solución, paso por paso, de manera precisa, definida y finita.

### *Lenguaje de programación*

Tal como las personas utilizamos un lenguaje verbal, corporal, o de señas para comunicarnos, se necesita un *lenguaje de programación* para escribir los programas para la computadora.

*Investigación individual del estudiante:* *¿cuál ha sido la evolución de los lenguajes de programación? ¿Cuáles son las diferencias entre dichos lenguajes?*

### *Programa*

Para que el algoritmo pueda ser comprendido y ejecutado por la computadora, es necesario transcribirlo utilizando un lenguaje de programación, logrando con ello la construcción de un *programa*. Por lo tanto un programa es una secuencia de instrucciones que indican a la computadora lo que debe hacer, es el *software*.

### *Programación*

El término *programación*, es utilizado en distintos ámbitos, y tiene que ver con el proceso de pensar cómo se llevará a cabo cierto evento, estableciendo acciones, tiempos, involucrados, responsables, requerimientos, etc. Específicamente cuando hablamos de

programación de computadoras nos referimos al proceso que nos lleva desde la definición de un problema hasta la prueba, depuración y validación del sistema.

### *Paradigmas de programación*

Un *paradigma* es la forma en que un individuo percibe e interpreta el mundo que lo rodea, filtrado a través de sus creencias, juicios mentales, predisposiciones, miedos, en fin toda su filosofía de vida.

Un *paradigma de programación* ha recuperado las mejores prácticas a lo largo de la evolución de los lenguajes. Es una manera aceptada y estandarizada, como un punto de vista o filosofía para abordar y resolver problemas computacionalmente solubles.

Los paradigmas de programación van evolucionando conforme los programadores encuentran formas más sencillas y apegadas a la concepción humana, para resolver problemas a través del diseño de software.

*Investigación individual del estudiante:* *¿cuál ha sido la evolución de los paradigmas de programación? ¿Cuáles son las diferencias entre dichos paradigmas?*

### *Editores de texto*

Un *editor de texto* es una aplicación cuya finalidad es facilitar la edición de documentos. Cuando hablamos de programación, la transcripción de los algoritmos en programas, se puede hacer mediante la utilización de cualquier editor de texto que permita la captura de las instrucciones sin formato. Afortunadamente en la actualidad los lenguajes de programación cuentan con Entornos Integrados de Desarrollo que además de facilitar la captura, prueba y depuración del código, brindan herramientas para el diseño de interfaces.

### *Entorno Integrado de Desarrollo*

*Entorno Integrado de Desarrollo (EID)*, también denominado Ambiente de Desarrollo Integrado, AID (Integrated Development Environment, IDE) es una aplicación que provee al programador de un entorno de desarrollo que le permite editar, compilar, depurar y en su caso, construir la interfaz gráfica de sus aplicaciones de forma amigable, sencilla y ágil.

*Investigación individual del estudiante:* *¿Cuáles EID son más utilizados en la actualidad? y ¿Cuál es su software de base?*

## Compiladores e intérpretes

Una vez capturado el código, existen dos maneras en las que la computadora podrá “comprender” y ejecutar las instrucciones, mediante un compilador o de un intérprete.

Los *compiladores* revisarán línea por línea las instrucciones del programa, asegurándose de que todo está bien escrito y que puede ser ejecutado. Y solo una vez que todo está en orden, procederá a la ejecución de la aplicación. Es decir, que si encuentra algún error o incongruencia, por mínima que sea, se lo hará saber al programador para su correspondiente corrección.

Los *intérpretes*, por otro lado, revisan y ejecutan línea por línea. Es decir, que instrucción que ha sido “aprobada” es ejecutada.

Los *compiladores* son traductores, como es el caso en la vida cotidiana, de las personas que se dedican a traducir libros o artículos, estos serán publicados hasta que todo el documento ha sido correctamente traducido al nuevo idioma.

Los *intérpretes* trabajan tal como lo hacen las personas que durante una transmisión de un evento por televisión, van realizando la traducción en tiempo real, frase por frase, comprendiendo lo que dice el audio original e inmediatamente comunicándolo al público.

Investigación individual del estudiante: *¿qué reciben como entrada y que producen como salida tanto los compiladores como los intérpretes?*

## Ejecutables

Un *programa ejecutable* es aquel que puede ser puesto en marcha sin necesidad de contar con el EID o compilador del lenguaje de programación con el que fue escrito. Es decir, se trata de todas las aplicación que utilizamos con solo hacer clic sobre su ícono o al invocarlas desde la Línea de comando del sistema.

## Consola de línea de comandos

Es aquella ventana o área a través de la que se pueden invocar *comandos del sistema operativo y aplicaciones* diseñadas para ser ejecutadas de esa forma.

Investigación individual del estudiante: *¿Cómo se accede a la Consola de línea de comandos? Menciona 5 comandos que pueden ser ejecutados desde la Línea de comandos y explique brevemente para que sirven.*

*¡Recuerda que todo tiene que ver!*

Mientras avanzas en los temas, percibe como todo tiene que ver con tu vida, con la forma en que las personas procesamos nuestra propia existencia. Cuanto más encontramos la relación de lo que debemos aprender con lo que sabemos, más fácil será integrarlo.

## ***REPRESENTACIÓN DE ALGORITMOS***

Como ya se ha mencionado un algoritmo es una serie de pasos organizados que describe el proceso que se debe seguir, para dar solución a un problema específico. A la serie de símbolos y reglas utilizados para describir de manera explícita un algoritmo se les llama *lenguaje algorítmico*.

Existen dos tipos de lenguajes algorítmicos:

- **Gráfico:** Es la representación gráfica de las operaciones que realiza un algoritmo (Diagrama de flujo, Diagramas N-S, etc.).
- **Narrativo (No Gráfico):** Representa en forma descriptiva las operaciones que debe realizar un algoritmo (Pseudocódigo).

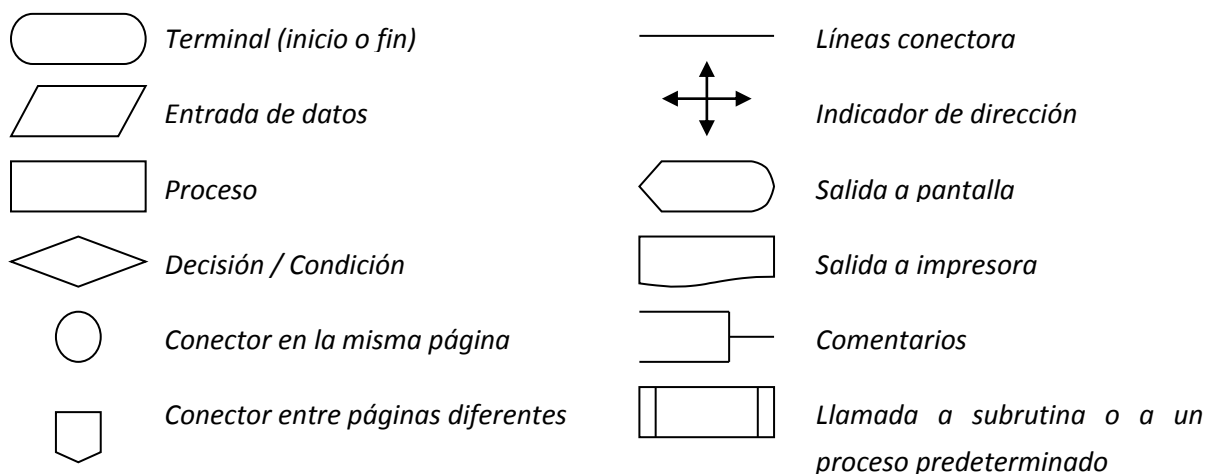
### ***Representación gráfica (Diagramas de flujo)***

Un *diagrama de flujo* es la representación gráfica de un procedimiento y de la secuencia u orden en que deben ejecutarse los pasos que lo componen. En otras palabras, es la representación gráfica de la solución del problema.

Los diagramas que se realizan durante el desarrollo de una aplicación informática deben ser claros, esquemáticos, y especialmente, independientes del lenguaje de programación que se vaya a utilizar. Así mismo, deben ser comprensibles para cualquier analista o programador que los examine, procurando no presenten excesiva complejidad.

El Instituto de Normalización Americano (ANSI- American National Standards Institute) ha diseñado un conjunto de símbolos y signos estándar que prácticamente han sido adoptados

internacionalmente. Algunos de ellos, útiles para representar las principales operaciones se muestran a continuación en la *Figura 2*:



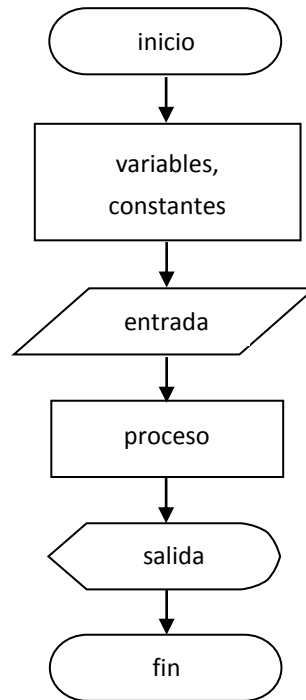
*Figura 2. Símbolos principales de los Diagramas de flujo.*

Reglas prácticas para la construcción de diagramas de flujo:

- Todo diagrama debe tener un inicio y un fin.
- Las líneas de flujo deben ser siempre rectas y de ser posible verticales u horizontales.
- Las líneas deben estar todas conectadas.
- Se deben dibujar los símbolos de modo que el diagrama pueda ser seguido de arriba abajo y de izquierda a derecha.
- Procurar que el centro del diagrama coincida con el centro de la hoja.
- Evitar la utilización de terminología propia de cualquier lenguaje de programación.
- Dejar uno o dos bloques de proceso libres al inicio del diagrama para declarar variables.
- Colocar los comentarios al margen del diagrama.
- Recurrir preferentemente a la lógica positiva y no a la negativa.
- A cada símbolo se accede por arriba o por la izquierda y se sale por abajo o por la derecha.
- Procurar en lo posible que el diagrama no sobrepase una página de tamaño, y si es así utilizar correctamente los conectores correspondientes que indiquen sin dudas la dirección del flujo del diagrama. (Joyanes, Metodología de la programación. Diagramas de flujo y programación estructurada, 1987)



En la *Figura 3* se puede apreciar la estructura básica de un algoritmo representado con un diagrama de flujo:



*Figura 3. Estructura básica de un algoritmo respresentado con diagrama de flujo.*

### **Representación narrativa (Pseudocódigo)**

Otra herramienta muy útil en el diseño de algoritmos es el *pseudocódigo*. *Pseudo* o *seudo*, significa falso o imitación y código se refiere a las instrucciones escritas en un lenguaje de programación; pseudocódigo no es realmente un código sino una imitación y versión abreviada de instrucciones reales para las computadoras.

El pseudocódigo es pues, una técnica para expresar en lenguaje natural la solución de un problema, es decir la lógica de un programa, de modo que sea sencillo para el programador traducirlo a cualquier lenguaje de alto nivel.

Para representar las diferentes instrucciones válidas, el pseudocódigo utiliza ciertos identificadores predefinidos llamados palabras clave o reservadas (*ver Tabla 1*), mencionadas a continuación:

<u>algoritmo</u>	<u>entero</u>	<u>fin procedimiento</u>	<u>leer</u>	<u>repetir</u>
<u>arreglo</u>	<u>entonces</u>	<u>fin registro</u>	<u>lógico</u>	<u>regresar</u>
<u>cadena</u>	<u>escribir</u>	<u>fin si</u>	<u>mientras</u>	<u>si</u>
<u>carácter</u>	<u>fin</u>	<u>funcion</u>	<u>no</u>	<u>sino</u>
<u>const</u>	<u>fin caso</u>	<u>hacer</u>	<u>o</u>	<u>var</u>
<u>de</u>	<u>fin desde</u>	<u>hasta</u>	<u>procedimiento</u>	<u>y</u>
<u>desde</u>	<u>fin funcion</u>	<u>hasta que</u>	<u>real</u>	<u>//comentarios</u>
<u>en caso de</u>	<u>fin mientras</u>	<u>inicio</u>	<u>registro</u>	

Tabla 1. Palabras clave o reservadas en pseudocódigo.

Si bien no existe un estándar único para la estructura básica de un algoritmo representado con pseudocódigo, a continuación se muestra el que utilizaremos en este curso (ver Figura 4):

```

algoritmo IdentificadorAlgoritmo
const
    //sección de declaración de constantes
var
    //sección de declaración de variables
inicio
    //cuerpo del algoritmo o sección de código
fin

```

Figura 4. Estructura básica de un algoritmo representado con pseudocódigo.

## DISEÑO DE ALGORITMOS

Cuando en la vida cotidiana se nos presenta una situación que requerimos resolver, comúnmente buscamos hacerlo utilizando la menor cantidad de tiempo, dinero y esfuerzo, es decir, de la mejor manera posible. Si bien, si se nos pierden las llaves no vamos a investigar una metodología formal para buscarlas, lo que si hacemos es tratar de ordenar nuestros pensamientos e iniciamos un proceso informal pero ordenado para lograr nuestro cometido. Si llegamos a pedir ayuda a alguna personal, esta generalmente nos hará preguntas y quizá hasta nos dirá algo como “haber vamos por partes...”. El asunto aquí, es que si deseamos resolver un problema, el desorden no será ayuda.

De igual manera, en el ámbito de los sistemas computacionales, cuando se desea dar solución a un problema con cierto nivel de complejidad y que el resultado final cumpla en tiempo y forma con las expectativas del cliente, será necesario que el desarrollo se realice con base en una metodología.

En este punto me parece relevante que comprendas, que si bien el desarrollo de software es un proceso sumamente creativo en donde el conocimiento, la experiencia y tus habilidades son determinantes, el uso de una metodología puede hacer la diferencia entre el éxito o el fracaso del proyecto. Así pues, aunque en materias posteriores tendrás la oportunidad de explorar a profundidad las distintas metodologías existentes, es oportuno que como parte de los fundamentos de programación empieces a familiarizarte con su utilización, lo que haremos a través de la *Metodología para la solución de problemas por medio de una computadora* (Blanco, 1998).

## *Metodología para la solución de problemas por medio de una computadora*

### **1. Definición del problema**

Podríamos decir que la *definición del problema*, no solo es crucial sino que es quizá la más importante en el desarrollo del software, pues si no se entiende claramente lo que el cliente está solicitando puede provocar que el producto final no sea lo que este necesita. Es entonces sumamente necesario que quien tiene este primer contacto comprenda claramente lo que el cliente está pidiendo: ¿qué quiere resolver?, ¿cuál es el problema que tiene?, ¿qué necesita exactamente?

La *definición del problema* es pues el primer acercamiento con el solicitante y esto puede darse a través de una llamada telefónica, un encuentro breve, o incluso con un mensaje. Lo relevante aquí es que la solicitud se haga de manera formal dejando claro el problema que se va a resolver, quedando finalmente descrito de forma breve y puntual.

Una buena *definición del problema* disminuirá en gran medida las probabilidades de error y permitirá que las siguientes etapas se lleven a cabo con mayor fluidez y un mejor aprovechamiento de los recursos.

### **2. Análisis del problema**

En la etapa de *análisis del problema* el programador toma conocimiento del problema antes de proceder a desarrollar una solución, sí, antes de empezar modelar cualquier solución. Un *análisis* inadecuado puede conducir a una mala interpretación del enunciado del problema. Los errores en esta etapa son, con frecuencia, difíciles de detectar y consumen mucho tiempo al arrastrarse hacia fases posteriores.

En esta etapa se responde a la pregunta qué: ¿qué desea el cliente?, ¿qué se está pidiendo?, ¿qué requerimientos se deben cumplir?, ¿Con qué información se cuenta?, ¿qué datos, fórmulas o formatos se requieren?, de la información que tengo ¿cuál necesito y cual es irrelevante?, etc. Necesito saber de manera precisa, en donde estoy y a dónde voy, porque si no lo sé... tampoco sé si ya llegué.

Los criterios o estrategias generales que se deben tener en cuenta al resolver un problema son:

- Usar toda la información útil (no superflua) disponible en el enunciado del problema.
- Hacer explícitas las reglas y datos que se encuentran implícitos (por ejemplo, en muchos problemas numéricos se pueden usar reglas convencionales de la aritmética o el álgebra).
- Profundizar en el problema considerado. Por ejemplo, empleando algún tipo de notación, utilizando determinados símbolos o dibujando algún diagrama que nos permitan captar y hacer visibles ciertos detalles del problema antes de resolverlo.
- Dividir un problema complejo en subproblemas más simples, que se puedan resolver independientemente y después combinar sus soluciones.
- Otra forma de abordar un problema consiste en trabajar “hacia atrás”; es decir, partir de la solución e intentar llegar al estado inicial.
- Los problemas no están aislados, sino que existen interrelaciones o afinidades, por tanto, se puede recurrir a la solución de problemas isomorfos o similares.

Para efectos de estandarizar, en nuestro curso el abordaje y *análisis de problema* se hará a través de la identificación de tres partes fundamentales:

- **Entrada:** Establecer los datos que el nuevo sistema requerirá para ser procesados y obtener el resultado deseado.
- **Proceso:** A través del uso de lenguaje cotidiano, describir la secuencia de pasos requeridos para llegar desde los datos de entrada hasta el resultado esperado.
- **Salida:** Definir la información que la aplicación desarrollada ofrecerá al usuario como producto.

### 3. Diseño de la solución

Una vez definido el problema y habiendo realizado el análisis que nos da cierta idea de cómo resolverlo, se puede utilizar alguna de las técnicas conocidas de diseño de algoritmos. Por ejemplo, si la complejidad interna del problema a resolver así lo requiere, se puede iniciar describiendo la solución como una secuencia de pasos bastante generales en lenguaje natural, para luego ir detallándolos o refinándolos más hasta obtener una solución en pseudocódigo o diagrama de flujo. En esta etapa también se empiezan a tomar decisiones sobre las estructuras de datos que se utilizarán para representar y almacenar los datos del problema (el término estructura de datos lo explicaremos con mayor profundidad en el Tema 4 de este curso).

Programar implica un proceso mental complejo, dividido en varias etapas. La finalidad de la programación, así entendida, es comprender con claridad el problema que va a resolverse o simularse por medio de la computadora, para entender también con detalle cual será el procedimiento mediante el cual la máquina llegará a la solución deseada.

En esta etapa se responde a preguntas como: ¿cómo se cumplirá con lo solicitado?, ¿cómo se almacenarán los datos?, ¿cómo es la solución?, ¿qué datos deben ser proporcionados por el usuario?, ¿Qué información espera el usuario que le sea proporcionada?, etc.

Al final de esta etapa se habrán diseñado los *algoritmos*, las *estructuras de datos* y las *interfaces* que servirán de comunicación entre el usuario y la aplicación.

Realizar *pruebas de escritorio (corridas de escritorio)* a los algoritmos en esta etapa, resulta ser una práctica efectiva para garantizar una menor aparición de errores lógicos durante la etapa de *codificación*.

En nuestro curso el *diseño de la solución* incluirá la especificación de:

- *Representación narrativa del algoritmo*: Algoritmo expresado utilizando pseudocódigo.
- *Representación gráfica del algoritmo*: Algoritmo expresado utilizando diagramas de flujo.
- *Interfaz*: Descripción gráfica del área de contacto, componentes y elementos de comunicación entre la aplicación y el usuario.

### 4. Codificación

Considerando que la solución algorítmica ha sido bien definida y probada, este proceso resulta casi completamente mecánico. El algoritmo se transcribe utilizando las reglas sintácticas y semánticas de un lenguaje de programación, así como teniendo en cuenta ciertos criterios de estilo y estructura.

### 5. Prueba, depuración y validación

Con la *prueba* se trata de comprobar que el programa (algoritmo codificado en la etapa anterior) es correcto; es decir, produce resultados correctos para todos los conjuntos posibles de datos válidos. Para algoritmos sencillos, donde el conjunto de datos sea pequeño, existe la posibilidad de realizar pruebas (donde se comprueba el comportamiento del algoritmo para ciertos conjuntos significativos de los datos: extremos, intermedios y excluidos). De la misma manera la *validación* del programa consiste en asegurarse en lo posible que las entradas que brinde el usuario no quebrarán el programa de manera inesperada.

### 6. Documentación

Una vez que la aplicación ha sido probada, es importante dejar registro y en su caso explicación del diseño e implementación de las partes más importantes del sistema. Dicha *documentación* va dirigida tanto a los usuarios finales como a los programadores que proporcionarán el mantenimiento. Para tal efecto comúnmente se genera:

- *Documentación interna* incluida dentro del código fuente, integrada por los comentarios e incluso por la estructura dada a las instrucciones a través de las indentaciones o sangrías.
- *Documentación externa* que regularmente consiste en un documento técnico, un manual de usuario, ayudas y/o tutoriales.

### 7. Implantación

La *implantación* es la fase en la que un programa es puesto en marcha para ser utilizado por los usuarios. En ocasiones y a manera de dar seguridad al cliente la implantación es realizada de poco en poco (por módulos, áreas o departamentos) o en paralelo con el sistema anterior por un determinado tiempo.

## 8. Mantenimiento

Si se ha tomado el trabajo de planear cuidadosamente un sistema y de transformarlo en un conjunto bien estructurado de programas y módulos, seguramente tendrá una vida útil prolongada y no se utilizará sólo una o dos veces. Este simple hecho obliga a considerar un esquema de mantenimiento que asegure que el modelo ya sistematizado evolucione a un ritmo parecido al que lo haga la realidad que está siendo simulada.

Tal vez llegue el momento en que ese proceso o aspecto de la realidad para el que se construyó el sistema haya cambiado cualitativamente, lo significará el término de la vida útil del sistema. Mientras tanto, sin embargo, hay que ser capaces de hacer alteraciones no estructurales al sistema con costo mínimo en recursos de análisis y programación, lo cual; de alguna manera está asegurando si el sistema se ha construido de manera modular y estructurada, y si se dispone de la documentación adecuada que lo describa tanto en su diseño como en su uso.

Suele decirse que si un sistema sólo es comprensible por su creador, es un mal sistema. Dicha falta de flexibilidad, resulta imposible de tolerar para el caso de sistemas realmente grandes, que son creados incluso por cientos de ingenieros en sistemas y programadores.

Aunque nuestros programas no sean siquiera medianamente grandes, tenemos el compromiso de hacerlos claros y flexibles para que admitan mejoras o sugerencias posteriores.

Se consideran tres tipos de *mantenimiento*: el preventivo, el correctivo y las nuevas versiones.

### *Diseño de algoritmos aplicados a problemas*

En este apartado distinguiremos dos tipos de algoritmos que son del interés de nuestra asignatura:

- *Algoritmos cotidianos*. Son los algoritmos que muestran el conjunto de pasos a seguir para dar solución a una actividad o problema cotidiano. Por ejemplo el plan para ir de vacaciones, la receta para hacer la gelatina, etc. En realidad cualquier receta o instructivo preciso, definido y finito es un algoritmo cotidiano.
- *Algoritmos computacionales*. Son los algoritmos que describen paso a paso la solución para un problema computablemente soluble. Por ejemplo sumar dos

números enteros, multiplicar los valores de dos matrices, convertir grados Celsius a grados Fahrenheit, etc.

## Ejemplo de resolución de un problema cotidiano

A continuación se presenta un ejemplo de resolución de un algoritmo cotidiano representado de manera narrativa y gráfica. Para ello utilizaremos las primeras 3 etapas de la Metodología para la resolución de problemas por medio de una computadora.

### 1. Definición del problema

Escriba el instructivo que le permita hacer un litro de limonada azucarada y servir en un vaso.

### 2. Análisis del problema

Entrada	Proceso	Salida
<ul style="list-style-type: none"><li>- 4 limones</li><li>- 1/2 taza de azúcar</li><li>- 1 jarra con un litro de agua</li><li>- 1 cuchillo</li><li>- 1 cuchara o pala de mango largo</li><li>- vaso</li></ul>	<ul style="list-style-type: none"><li>- Agregar el azúcar a la jarra con agua.</li><li>- Mezclar el azúcar con una cuchara o pala, hasta que se haya disuelto completamente.</li><li>- Partir cada limón por la mitad.</li><li>- Exprimir una por una las 8 mitades de limón en la jarra con agua.</li><li>- Mezclar el agua utilizando la cuchara o pala hasta disolver.</li></ul>	<p>Servir limonada en un vaso.</p>



### 3. Diseño de la solución

Vemos como queda representado nuestro algoritmo utilizando como lenguaje algorítmico al pseudocódigo (ver Figura 5):

```

algoritmo Limonada
var
    4 limones
    1/2 taza de azúcar
    1 jarra con un litro de agua
    1 cuchillo
    1 cuchara o pala de mango largo
inicio
    agregar el azúcar a la jarra con agua.
    repetir
        mezclar el azúcar utilizando la cuchara o pala
    hasta que se haya disuelto completamente
    partir cada limón por la mitad
    repetir
        exprimir mitad de limón en la jarra con agua
    hasta completar las 8.
    repetir
        mezclar el agua utilizando la cuchara o pala
    hasta disolver.
    servir limonada en un vaso
fin
  
```

Figura 5. Algoritmo Limonada representado con pseudocódigo.

## Ejemplo de resolución de un problema computacional

A continuación se presenta un ejemplo de resolución de un problema computacional representado de manera narrativa y gráfica. Para ello utilizaremos las primeras 3 etapas de la Metodología para la resolución de problemas por medio de una computadora.

### 1. Definición del problema

Diseñe el algoritmo para elevar un número al cuadrado.

### 2. Análisis del problema

Entrada	Proceso	Salida
Numero	resultado = numero x numero	Cuadrado del número

### 3. Diseño de la solución

```

algoritmo Cuadrado
var
    numero, resultado: entero
inicio
    escribir "¿cuál es el número al que deseas calcular el cuadrado?"
    leer numero
    resultado ← numero * numero
    escribir "El cuadrado de " + numero + " es " + resultado
fin

```

Figura 6. Algoritmo Cuadrado representado con pseudocódigo.

Veamos que al transcribir nuestra solución obtenemos un algoritmo con elementos no visibles en el análisis (ver Figura 6):

- Los nombres del dato de entrada y de salida son anotados, separados por coma, seguidos de dos puntos y la palabra *entero*. Esto significa que ambos podrán almacenar datos numéricos sin decimales.
- Antes de la instrucción para leer el número, anotamos la orden escribir seguida de un mensaje que le clarifica al usuario lo nuestro algoritmo está esperando de él. Así mismo al mostrar el resultado, el mensaje de salida incluye también el dato de entrada. A partir de ahora, considera que este tipo de detalles son deseables en el desarrollo de una aplicación *amigable*.

No te preocupe, pues en los temas posteriores profundizaremos en todo lo antes descrito.

## ***DISEÑO DE FUNCIONES***

Las *funciones* son segmentos de código que tienen un propósito muy específico dentro de un programa. Esto es equivalentes a las *funciones* en una calculadora, que cuando presionas una tecla realiza la tarea para la que fue programada, utilizando como entrada el valor que has introducido a través del teclado numérico de la misma.

Cuando hablamos de desarrollo de software el diseño algorítmico de funciones implica que desde el análisis, el problema sea subdividido en pequeños *subproblemas* más simples de resolver. A ello se le llama *Programación Modular* y su lema es "*divide y vencerás*". Sí, tal como lo has escuchado alguna vez, la estrategia consiste en dividir el problema de tal forma

que al analizar cada segmento se diseñarán pequeñas soluciones que juntas contribuirán a la solución total del problema.

Un ejemplo cotidiano sería la organización de una fiesta. Cuando el organizador dice “Vamos a hacer una fiesta”, de manera natural el grupo empieza a segmentar el problema diciendo “¿Qué vamos a comer?, ¿Quién lleva la bebida?, ¿Quién pone la música?, etc. En ese momento inicia el análisis que con cada pregunta busca saber qué quiere hacer el grupo, con que se cuenta, quienes pueden apoyar, cuando será, etc. Una vez que los asistentes, se ponen de acuerdo en las responsabilidades, lugares y horarios, logran diseñar la solución del problema, teniendo cada pequeño grupo una responsabilidad que cumplir, es decir, una *función* por realizar para contribuir a la resolución del problema total.

Para la representación de una función en pseudocódigo frecuentemente se usa la palabra *funcion*, mientras que en los diagramas de flujo se utiliza el símbolo de llamada a subrutina incluido en la Figura 2.

Profundizaremos en este tema más adelante, una vez que contemos con más elementos para el diseño de algoritmos con funciones.

***Se vale tener miedo, se vale no entender, pero sobre todo ¡Se vale ver por ti!***

Y así como todo lo que estas aprendiendo toma significado cuando lo relacionas con tus experiencias. También lo que piensas, sientes y expresas tiene que ver e impacta fuertemente en el logro de tus objetivos. Procura tu bienestar, has lo que necesites para cuidar tu mente, cuerpo y espíritu.

# TEMA 2. INTRODUCCIÓN A LA PROGRAMACIÓN

## CONCEPTOS BÁSICOS

### *Programación Visual*

El hecho de que la mayoría de las personas pensemos en imágenes y que para muchos los lenguajes de programación sean difíciles de aprender, ha contribuido a que los métodos de desarrollo visual sean provechosos para el desarrollo de aplicaciones científicas y de simulación. De ahí la motivación para el estudio de Lenguajes de *Programación Visual (VPLs)*.

Para lograr una mejor comprensión de la programación en un lenguaje visual es necesario conocer alguna terminología mínima. Por lo cual a continuación se definen los conceptos que se manejarán durante la resolución de las prácticas:

- Un *proyecto* es el conjunto de archivos que integran a una aplicación: formularios, módulos, clases y recursos.
- Una *forma (formulario)* es una ventana sobre la que el programador dibuja los elementos que el usuario utilizará para comunicarse con la aplicación. A todo el conjunto se le llama *interfaz*.
- Los *controles (componentes)* son los elementos que dibujamos en el formulario, es decir, objetos gráficos a través de los que se dará la entrada o salida de datos, por ejemplo: cajas de texto, botones o etiquetas.
- Las *propiedades* son las variables que permiten dar valores a los atributos (características) de un control.

## Controles (componentes) básicos

Los controles más utilizados o básicos en la programación visual son:

- El *cuadro de texto (campo de texto-TextField)* es un área dentro del formulario en la que el usuario puede escribir o visualizar texto.
- El *botón de comando (Button)* tiene asociada una orden con él, la cual se ejecutará cuando se haga clic sobre el mismo.
- La *etiqueta (Label)* es utilizada para mostrar texto en la forma.

## Pasos para generar una aplicación en un lenguaje visual

En general para construir una aplicación en un lenguaje visual, se siguen los pasos indicados a continuación: (Ceballos, 1999)

1. Ejecutar el Ambiente Integrado de Desarrollo (AID), también llamado Entorno Integrado de Desarrollo (EID), en el que se desea construir la aplicación.
2. Crear el proyecto.
3. Agregar la forma (formulario).
4. Ajustar el tamaño de la forma (si es necesario).
5. Agregar los controles.
6. Definir las propiedades de la forma y de los controles.
7. Escribir el código para cada uno de los controles (con buen estilo de programación).
8. Guardar la aplicación (recuerde realizar esta operación frecuentemente).
9. Ejecutar, probar y depurar la aplicación.

## Estilo de programación

Un buen estilo de programación es una de las características más notables que debe tener un programador experto. Aunque la experiencia proporciona el estilo, existen una serie de reglas que se recomienda seguir desde el principio del aprendizaje en programación:

- ✓ Sangrado (indentación) correcto de instrucciones.
- ✓ Inclusión de comentarios (documentación).

- ✓ Utilización de líneas en blanco.
- ✓ Elección de identificadores significativos.
- ✓ Buen uso de mayúsculas y minúsculas.
- ✓ Escritura de cada instrucción en una línea distinta.
- ✓ Diseño Modular y Descendente.
- ✓ Generación de aplicaciones funcionales y sin dificultades para el usuario, en una palabra, amigables.

### *Sintaxis*

La *sintaxis* es la forma correcta de escribir una instrucción para que el compilador la entienda y la pueda ejecutar. Son las reglas de escritura de un lenguaje.

### *Declaración*

La *declaración* de un elemento, consiste en especificar su nombre, tipo, y en algunos casos estructura, con la finalidad de que el compilador sepa como almacenar e interpretar la información que contendrá. Es algo así como la presentación en sociedad del elemento en cuestión.

## ***CARACTERÍSTICAS DEL LENGUAJE DE PROGRAMACIÓN***

### *Java*

Java es un lenguaje de programación orientado a objetos apropiado para diseñar aplicaciones para Internet. Además se dice que es un lenguaje de plataformas cruzadas, ya que puede ser diseñado para correr de igual forma en Windows de Microsoft, que en Apple de Macintosh y la mayoría de las versiones UNIX.

Java es descendiente del lenguaje C++ que a su vez es descendiente de C. Del lenguaje C, Java heredó su sintaxis y de C++, las características fundamentales de Programación Orientada a Objetos.

## NetBeans

*NetBeans* es un Entorno Integrado de Desarrollo, EID (Ambiente de Desarrollo Integrado, AID - Integrated Development Environment, IDE), para diseñar, codificar, construir y depurar aplicaciones utilizando el lenguaje Java (ver Figura 7).

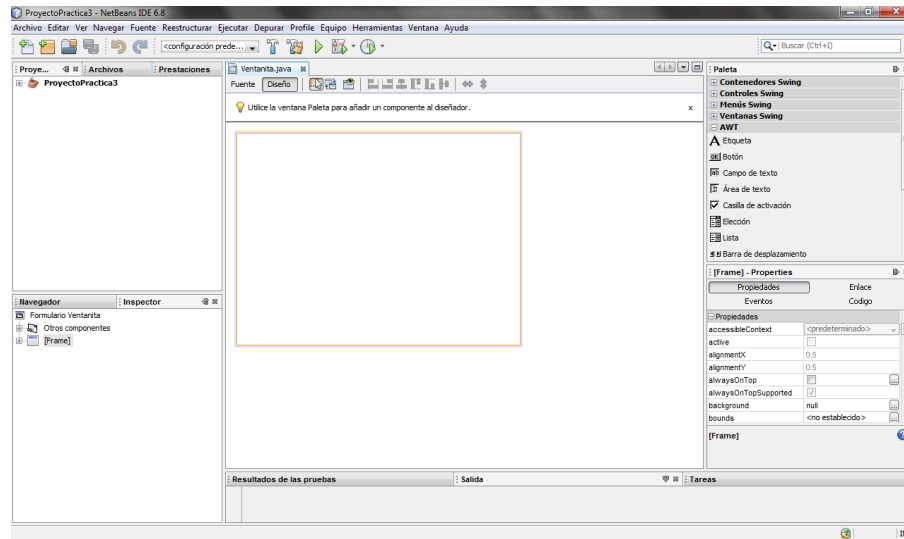


Figura 7. EID de NetBeans.

## ESTRUCTURA BÁSICA DE UN PROGRAMA

En Java un programa puede construirse creando una o más clases, sin embargo la *estructura básica* de un programa en Java está constituido del método principal *main()* en el que se incluyen las instrucciones del programa principal, separadas entre sí por el carácter “;” (ver Figura 8).

```
package claseprincipal;

import java.io.*;           // Inclusión de paquetes (librerías)

public class ClasePrincipal // La clase debe llamarse igual que el archivo que la contiene
{                           // Llave abre clase
    public static void main(String[] args) // Método principal, solo debe haber uno en la aplicación
    {                           // Llave abre método main
        instrucción1;
        instrucción2;
        // ...
        instrucciónN;
    }                           // Llave cierra método main
}                               //Llave cierra clase
```

Figura 8. Estructura básica de un programa en Java.

## ***ELEMENTOS DEL LENGUAJE***

Para la representación de algoritmos y su correspondiente traducción al lenguaje de programación (codificación), es necesario conocer los siguientes elementos:

- Tipos de datos.
- Variables.
- Constantes: literales y declaradas.
- Identificadores.
- Expresiones.
- Operadores.
- Estructuras de control: secuenciales, condicionales (decisión o selectivas) y cíclicas (iterativas o repetitivas).

### ***Tipos de datos***

Se le llama *dato* a la característica propia de una entidad (persona, cosa o animal). Por ejemplo la edad y el domicilio son dos datos que forman parte de la información de una persona.

Un software es útil para el usuario si tiene la capacidad de recibir *datos*, *procesarlos* y *obtener información*. Para almacenar los datos y la información, los programas utilizan *variables* y *constantes* a las que les es asociado un *tipo de dato* que define las características de los valores que se podrán guardar y el espacio que se requiere utilizar en la memoria de la computadora. Es algo equivalente a lo que sucede cuando vas a comprar una casa para un perrito, dependiente de la raza (tipo de perro), será el tamaño de la casa.

En los lenguajes algorítmicos los *datos* se clasifican en *cuatro tipos primitivos* (elementales, simples o básicos): *numérico*, *caracter*, *lógico* y *cadena*.



## 1. Numéricos: entero y real

En esta clasificación se incluyen dos tipos de datos:

- *entero*: Sirve para el almacenamiento de números positivos o negativos sin decimales, por ejemplo: 10, 258, 12548. Para que la computadora entienda el número es necesario no incluir comas para la separación de los miles.
- *real*: Permite almacenar números positivos o negativos con decimales en notación de coma flotante o científica, por ejemplo: 10.5, 0.48,  $1.185 \times 10^3$ ,  $9.1234 \times 10^{-5}$ . Para que la computadora entienda el número en coma flotante es necesario que *siempre tenga parte entera aunque esta sea un 0*, por ejemplo 0.57.

## 2. Caracter

El tipo *caracter* puede guardar un símbolo o *caracter* tomado del conjunto disponible en la computadora, es decir, todos los contenidos en el *UNICODE* (antes *Código ASCII*). Ejemplo: 'p', '@', 'ä', 'ú', '?', '4'.

El *caracter* debe ser colocado *entre apostrofes*. Es importante considerar que un número colocado entre comillas o apóstrofes es considerado un caracter, por lo tanto no es posible realizar operaciones matemáticas con él.

Los caracteres se clasifican en:

- Alfabéticos mayúsculas: 'A', 'B', 'C', ..., 'Z' (no incluye 'Ñ' ni vocales acentuadas)
- Alfabéticos minúsculas: 'a', 'b', 'c', ... 'z' (no incluye 'ñ' ni vocales acentuadas)
- Dígitos: '0', '1', '2', ..., '9'
- No imprimibles: #32 (espacio), #7 (beep), etc.
- Especiales: 'ñ', 'Ñ', '?', '\*', '+', '@', '\', '^', '/', 'á', etc.

## 3. Lógico

El tipo de dato *lógico* puede tomar solo dos valores: *falso (F)* y *verdadero (V)*.

## 4. Cadena

El tipo *cadena* sirve para guardar más de un carácter *colocados entre comillas*, por ejemplo: "María Luisa", "Calle 11 #451, Col. Centro", "www.itchihuahuauii.edu.mx".

Sin bien los lenguajes de programación *no* consideran a la cadena como un tipo elemental (pues en realidad se trata de un conjunto de caracteres), los lenguajes algorítmicos si lo hacen debido a su gran utilización.

### *Tipos de datos primitivos en Java*

Para cada una de las categorías antes mencionadas, los lenguajes de programación cuentan con una variedad de tipos de datos predefinidos que se distinguen fundamentalmente unos de otros por la cantidad de memoria requerida para su almacenamiento. Dichos tipos de datos son llamados también *primitivos*, *básicos* o *elementales*.

La siguiente tabla muestra los 4 tipos de datos descritos en la sección anterior y su correspondencia con los ocho *tipos de datos básicos* ofrecido por Java. Los cuales como se puede observar deben ser escritos completamente en minúsculas (ver *Tabla 2*):

Lenguaje algorítmico (pseudocódigo y diagrama de flujo)	Java	# bytes que ocupa y Rango de valores	Valor inicial
<u>lógico</u>	<b>boolean</b>	<b>true y false</b>	<b>false</b>
<u>caracter</u>	<b>char</b>	Caracter Unicode de 2 bytes \u0000 a \uFFFF	\u0000
<u>entero</u>	<b>byte</b>	Entero de 1 byte -128 a 127	0
	<b>short</b>	Entero de 2 bytes -32 768 a 32 767	0
	<b>int</b>	Entero de 4 bytes -2 147 483 648 a 2 147 483 647	0
	<b>long</b>	Entero de 8 bytes -9 223 372 036 854 775 808 a 9 223 372 036 854 775 807	0
<u>real</u>	<b>float</b>	Número de coma flotante de 4 bytes	+0.0
	<b>double</b>	Números de coma flotante de 8 bytes	+0.0

Tabla 2. Tipos de datos primitivos en Java.

## Tipos de clase

Además de los ocho tipos de datos primitivos, una variable puede tener una clase como su tipo. En tal caso lo que se está declarando es un objeto o instancia. Un ejemplo de tipo de clase es *String* que corresponde al tipo *cadena*.

## Variables

Una *variable* representa al espacio en memoria en donde se puede almacenar la información mientras se ejecuta una aplicación. Su nombre se debe a que el valor que contiene *puede ser modificado en cualquier punto del programa durante su ejecución*.

Para poder utilizar una variable, se le debe dar un nombre e indicar qué tipo de información podrá almacenar. Adicionalmente durante la declaración se le puede asignar un valor inicial. Por tanto en la declaración de una variable se debe incluir lo siguiente (ver *Tabla 3*):

- Identificador de la variable: nombre que utilizaremos para referirnos a la variable dentro del código de la aplicación.
- Tipo de dato: el tipo de dato determina la clase de valores que podrá almacenar la variable.

	Lenguaje algorítmico	Java
Sintaxis	identificadorVariable: <u>tipoDato</u>	tipoDato identificadorVariable;
Ejemplos	edad: <u>entero</u> nombre: <u>cadena</u> caracter1, sexo: <u>carácter</u>	<b>int</b> edad; <b>String</b> nombre; <b>char</b> caracter1, sexo;
	(No está definido como hacerlo)	<b>int</b> suma=0;

Tabla 3. Sintaxis de declaración de variables.

Así pues, declarar un variable significa determinar el nombre de la misma y el tipo de dato que podrá almacenar. Cada lenguaje de programación tiene definida una sección específica para la realización de esta actividad. Una vez que se declara una variable en un programa, el compilador busca espacio suficiente en memoria para guardar el valor, el cual es definido por el tipo de dato que le fue asociado.

## Constantes

Las *constantes* se usan para almacenar datos que *mantendrán un mismo valor durante toda la ejecución del programa*. Hay dos tipos de constantes: *literales* y *definidas por el programador*.

### Constante literal

Las *constantes literales*, son los símbolos que representan al valor en sí mismas, es decir los números y las cadenas: 9, 153, 10.34, “Calle Zapotecas # 4578, Colonia Antepasados”. Cuando se requiere usar una constante literal en un cálculo, esta puede ser usada directamente, por ejemplo  $\text{suma} \leftarrow x + 10$ ,  $\text{costo} \leftarrow 50 * 200$ .

### Constante definidas por el programador

Las *constantes definidas por el programador* son declaradas cuando se requiere usar un valor constante varias veces dentro del código o el valor es difícil de recordar.

El símbolo utilizado para guardar un valor en una constante es el =. En Java la palabra reservada para declarar una constante es *final* (ver Tabla 4).

	Lenguaje algorítmico	Java
<b>Sintaxis</b>	<u>const</u> identificador = valor	<b>final</b> tipoDato identificador= valor;
<b>Ejemplo</b>	<u>const</u> edadMinima = 18 pi=3.1415926 carrera = “ISC”	<b>final byte</b> edadMinima=18; <b>final float</b> pi=3.1415926; <b>final String</b> carrera=”ISC”;

Tabla 4. Sintaxis de declaración de constantes.

## Identificadores

Un *identificador* es el nombre que se le da a cualquier elemento de un programa. Como todos los lenguajes de programación, Java cuenta con dos tipos de identificadores: las palabras clave y los definidos por el programador.

### Palabras clave

Las *palabras clave* o *reservadas (keywords)*, son identificadores definidos previamente para uso exclusivo del lenguaje, es decir, que *tienen un significado* para el compilador de Java y

*no pueden ser utilizadas para otros fines.* A continuación se listan las palabras reservadas de Java:

abstract	class	false	implements	new	return	throws
boolean	const	final	import	null	short	transient
break	continue	finally	inner	operator	static	true
byte	default	float	instanceof	outer	super	try
byvalue	delegate	for	int	package	switch	var
case	do	future	interface	private	synchronized	void
cast	double	generic	long	protected	this	volatile
catch	else	goto	multicast	public	threadsafe	while
char	extends	if	native	rest	throw	

### Identificadores definidos por el programador

Los *identificadores definidos por el programador* son los nombres que este da a los elementos declarados por él para cierta aplicación. Y para que estos sean reconocidos por Java, es necesario que cumplan con las siguientes reglas de construcción:

- No debe ser una palabra reservada.
- No debe incluir espacios en blanco.
- Puede tener cualquier longitud (uno o más caracteres).
- El primer caracter solo puede ser una letra, el caracter \$ o el guión bajo.
- Después del primer carácter puede incluir cualquier combinación de letras, dígitos, \$ y \_.
- Las letras pueden ser mayúsculas y minúsculas incluyendo, en ambos casos, las acentuadas y la ñ.
- Debe ser referenciado (utilizado) *siempre de la misma manera*, sin cambiar mayúsculas por minúsculas ni viceversa, ya que sería interpretado como un identificador diferente.
- En Java los identificadores pueden emplear caracteres del alfabeto inglés y de otros idiomas internacionales tomados de Unicode.

Cuando se da nombre a un elemento de una aplicación, *es importante recordar que Java es sensible a mayúsculas/minúsculas*. Por tal razón los identificadores *Edad*, *EDAD* y *edad* son diferentes.

En la mayoría de los programas escritos en Java a los elementos, entre ellos las variables, se les dan nombres con significado útil que incluyan varias palabras unidas. Para facilitar la identificación de las palabras y la categorización de los elementos de una aplicación, se aplica la siguiente *Guía estándar* (Ingalls & Jinguji, 1999):

- a. En el nombre de una clase, cada palabra inicia con mayúscula. Por ejemplo: Color, JFrame, TextField.
- b. En el nombre de una variable miembro o local, la primera palabra es toda en minúsculas y de la segunda en adelante inician con mayúscula. Por ejemplo: minimumSize, textField1, background.
- c. En los nombres de variables se utilizan sustantivos o adjetivos. Por ejemplo: edad, font, carrera, text.
- d. En los nombres de métodos se utiliza un verbo seguido del nombre de variable o propiedad (atributo) a la que afecta. Por ejemplo: getText, setText, calcularArea.
- e. En general, evite el uso de guión bajo, porque el lenguaje les tiene ya un uso específico.

Para la separación visual de palabras en los identificadores (en lugar de espacios o guiones), se utilizan las iniciales mayúsculas, según se ha descrito en los puntos anteriores.

## Comentarios

Los comentarios son la documentación interna de una aplicación. Es *información ignorada por el compilador*, incluida entre el código para beneficiar a quienes requieran saber lo que sucede dentro del mismo. Una manera de mejorar la legibilidad de las instrucciones de un programa es el uso de comentarios.

Java provee tres formas diferentes para escribir comentarios dentro del código:

- *Comentario de una línea*. Se antepone doble diagonal (//) al texto, de tal forma que todo lo que se encuentre después y hasta el final de la línea se considerará comentario. Por ejemplo: `char sexo; // Utilizar 'F' para femenino, 'M' para masculino`

- *Comentarios de más de una línea.* Inician con `/*` y finalizan con `*/`. Todo lo que se encuentre entre ambos delimitadores se considerará comentario (ver Figura 9).

```
/* Módulo que determina si un carácter es dígito. Recuerde que al tratarse de un método
de tipo boolean, este regresará como resultado un valor lógico, es decir, true o false. */

boolean esDigito(String n){
    if ( n.equals("0") || n.equals("1") || n.equals("2") || n.equals("3") || n.equals("4") ||
        n.equals("5") || n.equals("6") || n.equals("7") || n.equals("8") || n.equals("9") )
        return (true);
    else
        return(false);
}
```

Figura 9. Ejemplo de uso de comentario de más de una línea.

- Los comentarios oficiales del lenguaje inician con `/**` y finalizan con `*/`. Todo lo que se encuentre entre ambos delimitadores se considera un comentario que puede ser entendido también por la computadora. Este tipo de comentarios proveen información acerca del funcionamiento de las clases y métodos públicos. Además puede ser leído por utilerías como la herramienta javadoc incluida en JDK, las cuales toman este tipo de comentarios para crear un conjunto de páginas Web que documentan el programa, sus clases y sus métodos.

## TRADUCCIÓN DE UN PROGRAMA

### Compilación

La *compilación* de un programa consiste en convertir el código fuente del mismo a un lenguaje máquina que pueda ser interpretado y ejecutado por la computadora.


El código fuente es el que ha sido escrito en algún lenguaje de programación de Alto nivel (por ejemplo Java), es decir con instrucciones que fácilmente pueden ser entendidas por los humanos. En cambio el código objeto está escrito en lenguaje máquina (bajo nivel), sumamente difícil de comprender por el programador, pues su representación dista mucho de la forma en que nos expresamos.

Al programa escrito en código fuente se le llama archivo fuente y al que se encuentra en código máquina se le denomina archivo objeto o ejecutable

### Enlace

El *enlace* de un programa consiste en que un programa montador o enlazador (linker) se encarga de insertar al programa objeto el código máquina de las funciones de las librerías (archivos de biblioteca) usadas en el programa y de realizar el proceso de montaje, que producirá un programa ejecutable .exe. Las librerías son una colección de código (funciones) ya programado y traducido a código máquina, listo para utilizar en un programa y que facilita la labor del programador (Rodríguez, 2012).

### Ejecución

Una vez que la construcción del proyecto ha sido finalizada (interfaz y codificación), es momento de ejecutar su nueva aplicación haciendo clic en el icono , de la barra de herramientas o eligiendo la opción Ejecutar *Main Project (Run Main Project)* del menú *Ejecutar (Run)*, o pulsando *F6*.

NOTA: Si al ejecutar el proyecto, aparece una caja de dialogo para elegir el *Proyecto Principal (Class Main)*, debes elegir el proyecto y Aceptar.

Para detener la ejecución hay tres formas distintas:

- Elegir la opción *Detener generación/ejecución (Stop Build/Run)*, del menú *Ejecutar (Run)*, o
- *Alt + F4*.
- *Hacer clic en el botón de cerrar de la ventana (icono con cruz de la esquina superior derecha)*.

### Errores

La prueba de un programa es el proceso de ejecución con una amplia variedad de datos de entrada, llamados datos de prueba, que tiene como finalidad determinar si la aplicación “corre” sin errores. Para realizar una buena prueba se deben considerar valores normales y extremos que comprueben los límites del sistema. La depuración, entonces, consiste en encontrar los errores del programa y corregirlos.



Al ejecutar un programa se pueden producir tres tipos de error:

- *De compilación (syntax)*. Ocurren cuando una o varias instrucciones infringen reglas de sintaxis del lenguaje. Ejemplos de ellos sería omitir una letra en una palabra reservada, declarar una variable con un nombre y utilizarla con otro, omitir un “;”, etc.
- *De ejecución*. Ocurren cuando durante su ejecución el flujo del programa conduce a una operación que no puede ser manejada por el sistema, produciendo que este se quiebre de forma abrupta. Un ejemplo de ello es la división entre cero.
- *De lógica (diseño)*. Este tipo de error no impide que el programa sea ejecutado, pues las instrucciones han sido escritas respetando las reglas sintácticas del lenguaje. Sin embargo, el sistema no ofrece al usuario los resultados esperados, pues existen incongruencias derivadas del diseño de la solución. Por ejemplo, la incorrecta definición de una expresión.

### *Creencias*

Correcto o incorrecto, bueno o malo, lo que puedes y no puedes, lo que deberías o no deberías ser, pensar, decir o hacer. ¿Hasta donde tus creencias te ayudan o limitan?, ¿De verdad son tuyas? ¿Tienen que ver con tu experiencia? ¿Sabes de dónde vienen?

¿Crees que valdrá la pena revisar una por una? Yo creo que sí.

Analiza, reflexiona y suelta lo que ya no te haga click, lo que no tome sentido en tu vida. La pregunta es, ¿te aporta o protege de algo? ¿O solo se trata de juicios hacia ti y/o hacia las demás personas?

## TEMA 3. CONTROL DE FLUJO

Se llaman *estructuras de control* al grupo de instrucciones que guían la ejecución del programa, es decir controlan el flujo de la aplicación. Los lenguajes cuentan con tres tipos de estructuras de control:

- Instrucciones secuenciales.
  - a) Instrucción de asignación ( $\leftarrow$ ).
  - b) Instrucción de entrada (lectura).
  - c) Instrucción de salida (escritura).
- Instrucciones condicionales o de decisión.
  - a) Condición simple (si).
  - b) Condición doble (si-sino).
  - c) Condición múltiple (en caso).
- Instrucciones cíclicas, repetitivas o iterativas.
  - a) Predefinido (desde).
  - b) Condicionales (mientras, hacer-mientras).

Para comprender su funcionamiento te invito a que encuentres su correspondencia en tu vida cotidiana, recuerda que el ser humano crea a partir de lo que conoce. Ten la seguridad que de una u otra forma ya las has usado. Si no me crees, vamos a ver si te suenan estas expresiones:

- ✓ Si comes te puedes levantar de la mesa.
- ✓ Si terminas la tarea sales a jugar, sino aquí te quedas.
- ✓ Mientras tengo hambre como.
- ✓ La tienda está a 7 cuadras, avanza de una en una hasta que la veas del lado izquierdo.
- ✓ Guarda tus cosas en el segundo cajón.

Se ve fácil, pues lo es. Entonces para empezar, elije derrumbar tus barreras, es más dilo en voz alta, ¡es fácil! ¡Es muy fácil para mí comprenderlo! Pan comido, listos para continuar.

## ***ESTRUCTURAS SECUENCIALES***

Las estructuras secuenciales son instrucciones que serán ejecutadas por el procesador una detrás de otra. Es decir, que este tipo de instrucción no estará pendiente de que pasó antes o después, simplemente realizan su trabajo y continúan con el flujo del programa. Los lenguajes de programación incluyen *3 tipos de instrucciones secuenciales: instrucción de asignación, instrucción de lectura (entrada) e instrucción de escritura (salida)*.

Por un lado las instrucciones de *asignación* y *lectura* constituyen las dos formas en que las aplicaciones almacenan datos e información en la memoria de la computadora. Mientras que la *instrucción de escritura* constituye el medio disponible para comunicarse con el usuario.

En términos cotidianos podemos hacer las siguientes analogías:

- La *instrucción de asignación* es equivalente a guardar algo en un lugar determinado. Por ejemplo: guardar unos calcetines en el cajón de la cómoda, colocar una lata en la alacena, poner el comprobante de pago del seguro del vehículo en la guantera, poner agua en la cartera de hielos, etc.
- La *instrucción de lectura* es correspondiente a recibir información del exterior. Por ejemplo: escuchar, tocar con nuestras manos, sentir a través de la piel, percibir, etc.
- La *instrucción de escritura* se relaciona con enviar información hacia el exterior. Hablar, emitir sonidos, hacer señas con las manos, nuestra postura corporal, los gestos de la cara, etc.

### ***Instrucción de asignación***

La *instrucción de asignación* permite al programador indicar al procesador que se desea guardar un valor en la memoria.

	Lenguaje algorítmico	Java
Sintaxis	variable ← expresión	variable = expresión;
Ejemplo	edad ← 19 nombre ← "Andrés" caracter1 ← '2' sexo ← 'M'	edad= 19; nombre= "Andrés"; caracter1= '2'; sexo= 'M';

Tabla 5. Sintaxis de declaración de una asignación.

Una expresión puede ser una variable, constante declarada, constante literal, llamada a un método o la combinación de ellos a través de una fórmula (ver Tabla 5).

### Reglas de construcción de una instrucción de asignación

- Una variable en el lado derecho de una asignación debe tener un valor antes de que la instrucción se ejecute.
- Del lado izquierdo de una asignación solo puede haber un identificador válido de una variable.
- El tipo de la expresión debe ser del mismo tipo que la variable.
- La operación de asignación es una operación destructiva debido a que el valor almacenado en una variable se pierde y se sustituye por el nuevo valor en la asignación.

### Expresiones

Una *expresión* es un conjunto de *operadores* y *operandos* o inclusive un solo operando. Es decir que una expresión puede ser una constante declarada, una constante literal, una variable o una combinación de constantes, variables y funciones con operadores.

Los *operadores* son los símbolos que representan a las operaciones permitidas en un lenguaje de programación. Los operadores de Java los podemos clasificar de la siguiente manera (ver Tabla 6, Tabla 7, Tabla 8, Tabla 9, Tabla 10 y Tabla 11).

Operadores aritméticos	
+	Operador unario signo positivo. Suma
-	Operador unario negativo. Resta
*	Multiplicación
/	División entera si los operandos son enteros
/	División real si los operandos son reales
%	Módulo. Resto de la división entera. No es necesario que los operandos sean enteros.

Tabla 6. Operadores aritméticos.

Operadores relacionales	
==	Igual
!=	Diferente
>	Mayor que
<	Menor que
>=	Mayor o igual
<=	Menor o igual

Tabla 7. Operadores relacionales.

Operadores lógicos	
&&	AND en corto circuito
	OR en corto circuito
!	NOT unario lógico
&	AND lógico
	OR lógico
^	XOR

Tabla 8. Operadores lógicos.

Operador alfanumérico	
+	Concatenación o unión de cadenas.

Tabla 9. Operador alfanumérico.

Tabla de verdad de los operadores lógicos					
Operando1	Operando2	&&		! Operador1	^
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	true
false	false	false	false	true	false

Tabla 10. Tablas de verdad.

Operadores de asignación	
=	Asignación simple
++	Incremento y asignación
--	Decremento y asignación
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
+=	Suma y asignación
- =	Resta y asignación

Tabla 11. Operadores de asignación.

El diseño de un programa implica el uso de valores y fórmulas de todo tipo. Por ello, la *evaluación y conversión de expresiones* es sumamente relevante, pues una vez que comprendes claramente que significan y como se llevan a cabo, podrás construir correctamente cualquier tipo de expresión para las aplicaciones que desarrolles.

## Evaluación de expresiones

*Evaluar una expresión* significa realizar todas las operaciones indicadas por los operadores que contiene, hasta obtener un último y único resultado. Para asegurar que la *expresión* arroje el resultado correcto, es necesario que los operadores sean evaluados en un orden ya determinado por el lenguaje de programación, a dichos lineamientos se les denomina *reglas de prioridad o precedencia*. Cabe señalar que entre lenguajes de programación puede haber variaciones, por lo que se recomienda que siempre te asegures de conocer las del lenguaje que utilizarás para la codificación de tu aplicación.

### Reglas de prioridad o precedencia de Java

1. Los operadores se evalúan de acuerdo a la tabla de prioridad (*ver Tabla 12*). El orden se lee en *forma vertical*, siendo los paréntesis lo más relevante, y la asignación lo de menor prioridad. En la tabla, el acomodo horizontal no tiene relevancia.

Tabla de Prioridad de operadores	
mayor	()
	- ! ++ --
	new (tipo)
	* / %
	+ -
	> >= < <=
	== !=
	&
	^
	&&
menor	= *= /= %= += -=

Tabla 12. Tabla de prioridad o precedencia de operadores en Java.

2. Si dos operadores con la misma prioridad se encuentran contiguos, su evaluación se hará de izquierda a derecha, tal como se muestra a continuación. Observa como para una misma expresión los resultados son totalmente distintos cuando la resolución es correcta e incorrecta (ver Figura 10).

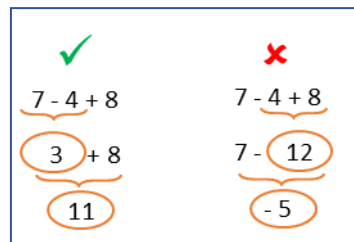


Figura 10. Ejemplo de evaluación de expresión con operadores de misma precedencia.

3. Para *cambiar el orden de precedencia* marcado por el lenguaje se utilizan *paréntesis*. Por ejemplo si en la expresión mostrada en la Figura 10, quisiéramos que la evaluación se hiciera de la segunda manera, requerimos agregar paréntesis (ver Figura 11 ).

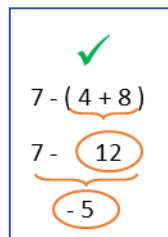


Figura 11. Ejemplo de uso de paréntesis en una expresión.

A continuación se muestran algunos ejemplos de expresiones evaluadas (ver Figura 12):

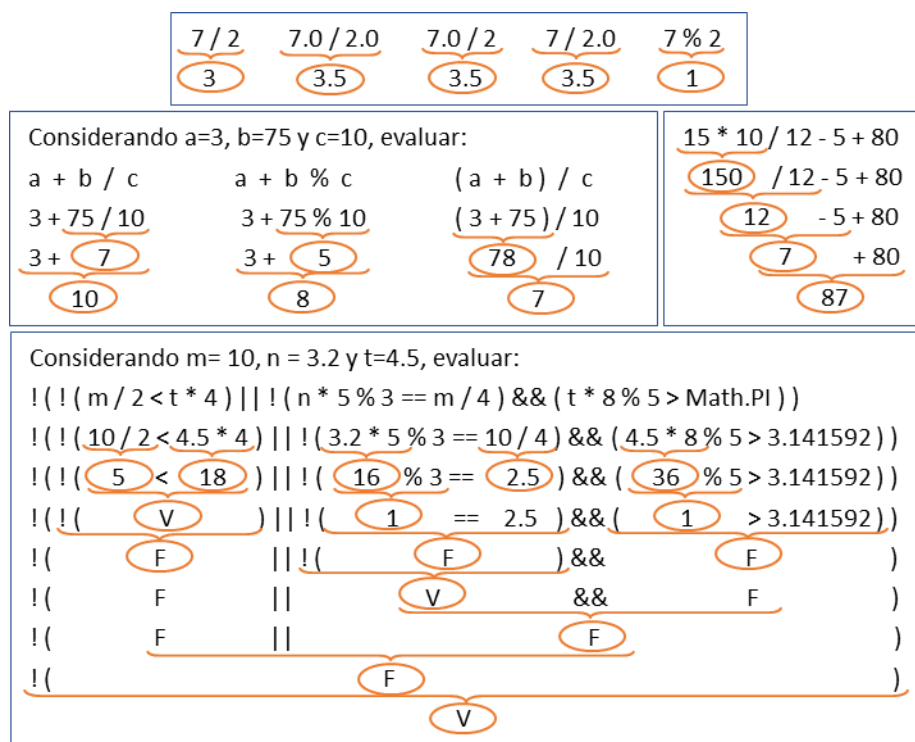


Figura 12. Ejemplo de expresiones evaluadas.

## Conversión de expresiones algebraicas en expresiones algorítmicas

Es común que algunas de las fórmulas que se usan en la solución de problemas, ya existan en un *formato algebraico*, en cuyo caso es necesario convertirlas a una representación que pueda ser reconocida y evaluada correctamente por el compilador, denominada *expresión algorítmica*.

Para llevar a cabo este tipo de conversiones se recomienda analizar el orden en que la fórmula original se resuelve manualmente (de manera cotidiana), para luego iniciar la transcripción de adentro hacia afuera, es decir, empezando por los operadores que se evalúan primero hasta llegar a la última operación. Conforme se avanza, cada vez que una operación es transcrita, se verifica que el orden de evaluación sea el correcto o si es necesario incluir paréntesis (ver Figura 13).



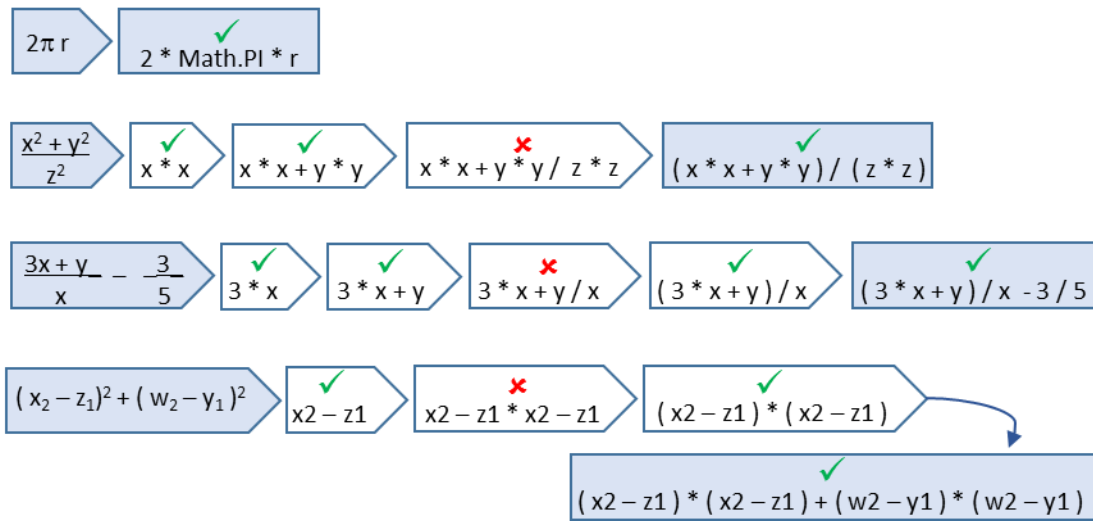


Figura 13. Ejemplo de conversión de expresiones.

### Y a ti, ¿Qué te motiva?

La valentía no es la ausencia de miedo, pues si no hay miedo en donde esta la conquista. El valiente es aquel que vence lo que le asusta. De la misma forma, la fortaleza la cultiva quien se cae, se permite vivirlo, se desahoga, lo integra y encuentra la energía, su motivación para continuar.

### Instrucción de lectura (entrada)

La *instrucción de entrada* es la que toma valores provenientes de un dispositivo periférico y los dirige a algún lugar de la memoria. Los datos que se pueden leer son enteros, reales, caracteres o cadenas (ver Tabla 13). Los dispositivos de entrada del ser humano serían los oídos, los ojos o la piel, que nos permiten recibir información del exterior.


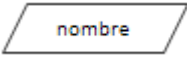
	Lenguaje algorítmico		Java
	Pseudocódigo	Diagrama de flujo	
<b>Sintaxis</b>	<u>leer</u> variable		variable= identif_TextField.getText();
			variable= JOptionPane.showInputDialog(null, mensaje, título, tipo_caja);
			<b>Scanner</b> entrada= new Scanner(System.in); variable= entrada._____. En la línea se usa el método que corresponda al tipo de información que se desea recibir. Por ejemplo: nextBoolean, nextInt, nextByte o nextFloat.
<b>Ejemplo</b>	<u>leer</u> nombre		nombre = txtNombre.getText();
			nombre=       JOptionPane.showInputDialog(null, "Escribe tu nombre", "Captura de datos, 1);  <b>NOTA:</b> tipo_caja puede tomar valores del 0 al 3.
			<b>Scanner</b> entrada = new Scanner(System.in); nombre=entrada.nextLine();

Tabla 13. Sintaxis de declaración de la instrucción de lectura.

### Instrucción de escritura (salida)

La *instrucción de salida* es la operación que proporciona información al usuario (ver Tabla 14). Dicha instrucción toma información de la memoria y la envía a la pantalla, a un dispositivo de almacenamiento o a un puerto de E/S. De la misma forma las personas usamos nuestros dispositivos de salida cuando hablamos o hacemos señas, gestos o ruidos; incluso nuestra postura corporal transmite información.

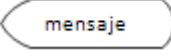
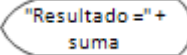
	Lenguaje algorítmico		Java
	Pseudocódigo	Diagrama de flujo	
<b>Sintaxis</b>	<u>escribir</u> mensaje		identif_Label.setText(mensaje);
			JOptionPane.showMessageDialog(null, mensaje, título, tipo_caja); System.out.printf(mensaje);
<b>Ejemplo</b>	<u>escribir</u> "Resultado = " + suma		lblMensaje.setText("Resultado = " + suma);
			JOptionPane.showMessageDialog(null, "Resultado=" + suma, "Facturación", 1); <b>NOTA:</b> tipo_caja puede tomar valores del 0 al 3. System.out.printf("Resultado= %d \n", suma);

Tabla 14. Sintaxis de declaración de la instrucción de escritura.

## Ejemplo de resolución de un problema utilizando instrucciones secuenciales

A continuación se plantea un *problema* al que le diseñaremos una solución mediante la *metodología* abordada en el Tema 1 de este curso.

### 1. Definición del problema

Diseñe un programa que solicite el nombre y apellido al usuario, para a continuación mostrarle el mensaje *Hola mundo*, seguido de una bienvenida personalizada. Y si te estas preguntando ¿Por qué Hola mundo?... bueno es que programador que se respete siempre inicia con el programa Hola mundo jajaj. Se trata de una regla ancestral para el éxito 😊.

### 2. Análisis del problema

Entrada	Proceso	Salida
<ul style="list-style-type: none"> <li>Nombre</li> <li>Apellido</li> </ul>	<ul style="list-style-type: none"> <li>Construir un mensaje con el nombre, el apellido y un saludo de bienvenida.</li> </ul>	<ul style="list-style-type: none"> <li>Mensaje <i>Hola mundo</i></li> <li>Bienvenida personalizada.</li> </ul>

Una vez que se han identificado los aspectos generales anteriores, es necesario preguntarse si ello es suficiente para ser transcrito a un algoritmo. Si la respuesta es sí estamos listos para pasar a la etapa de Diseño de la solución..., pero si la respuesta es no o no sé, habría que continuar con un análisis más detallado, el cual puede plasmarse en el formato anterior o según le sea más claro al analista:

- **Entrada y salida.** Es importante tener en cuenta que cada vez que una aplicación solicita datos o provee de información al usuario, es recomendable que lo haga de manera *amigable*, utilizando mensajes que le brinden la información necesaria para comprender lo que el sistema está haciendo. Por tal razón, lo consideraremos en la etapa de diseño.
- **Proceso.** El algoritmo debe construir un mensaje de bienvenida utilizando el nombre y apellido que el usuario le provea. Para ello deberá utilizar el operador de concatenación o unión de cadenas.

Recuerda que el producto que obtengas en esta etapa deberá ser un documento claro, ordenado y listo para servir de sustento en el diseño de la solución.

### 3. Diseño de la solución

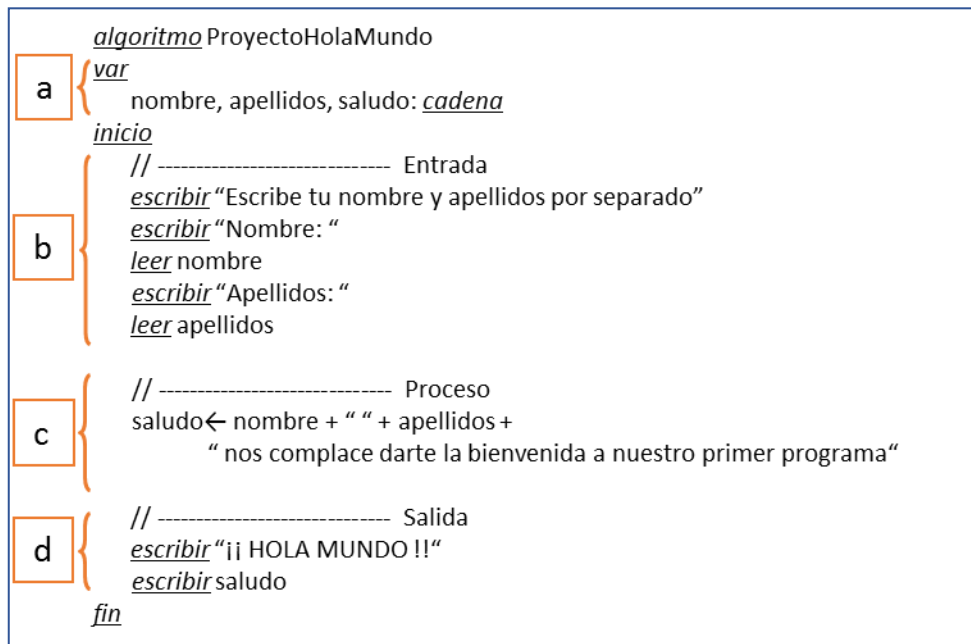


Figura 14. Algoritmo ProyectoHolaMundo representado con pseudocódigo.

Antes de continuar es conveniente que precisemos algunos detalles del pseudocódigo presentado en la *Figura 14*:

- En la sección *var* se han declarado 3 variables de tipo cadena: dos para leer el nombre y apellido; y otra para guardar en ella el saludo personalizado.
- En la *Entrada*. El primer mensaje instruye al usuario sobre cómo debe introducir su nombre. Luego antes de cada lectura se le indica lo que se espera que escriba. Observa que el mensaje incluye un espacio después de los dos puntos, esto se debe a que la *instrucción de salida*, mostrará en la pantalla *exactamente lo que se encierre entre comillas*.
- En el *Proceso*: La *instrucción de asignación* debe evaluar la *expresión alfanumérica* para asignar el resultado a la variable *saludo*. Se trata de una *concatenación*, en donde el operador *+* une el contenido de las variables, a las cadenas escritas entre comillas. Una vez más, es importante poner atención en los espacios que se deben incluir.
- La *Salida* es muy sencilla. Se llama dos veces a la instrucción *escribir*, para indicar que el mensaje *Hola mundo* y la bienvenida personalizada, se mostrarán en renglones diferentes.

## 4. Codificación

```
package proyectoholamundo;
import java.util.Scanner;
/**
 * @author Guadalupe
 * Marzo 2020
 */
public class ProyectoHolaMundo {

    public static void main(String[] args) {
        // --- Declaración de variables
        String nombre, apellidos, saludo;

        // ----- Entrada
        System.out.print("Escribe tu nombre y apellidos por separado");

        Scanner entrada = new Scanner(System.in);
        System.out.print("\nNombre: ");
        nombre=entrada.nextLine();
        System.out.print("Apellidos: ");
        apellidos=entrada.nextLine();

        // ----- Proceso
        saludo = "¡¡ HOLA MUNDO !!\n" + nombre + " " + apellidos +
            " nos complace darte la bienvenida a nuestro primer programa\n";

        // ----- Salida
        System.out.printf(saludo);
    }
}
```

Figura 15. Código en Java del ProyectoHolaMundo.

Veamos algunos aspectos relevantes del código anterior (ver Figura 15):

- En la parte superior se incluyen dos líneas de código:
  - ✓ **package** proyectoholamundo;
 

La palabra reservada *package* indica que este programa se encuentra almacenado en una carpeta llamada proyectoholamundo. Es la forma en que Java facilita la organización los archivos de un proyecto.
  - ✓ **import** java.util.Scanner;
 

Al escribir un programa generalmente se requerirá utilizar código que se encuentra almacenados en otros archivos y para ello Java utiliza la palabra reservada *import*. La instrucción está importando la clase *Scanner*, porque más adelante se invocarán métodos de entrada incluidos en ella.

- Tal como en el algoritmo solo fue necesario declarar 3 variables de tipo *String*.
- La instrucción *Scanner entrada = new Scanner(System.in);* se encarga de declarar y crear una entrada de datos. A la variable del ejemplo se le llama entrada, pero en realidad puedes identificarla de la manera que desees, siempre y cuando el nombre sea válido para el lenguaje. Esta instrucción solo es necesario escribirla una sola vez antes de la primera lectura, la cual es realizada por la instrucción *nombre=entrada.nextLine();*
- En la instrucción de asignación, la concatenación utiliza la secuencia *\n*, que sirve para incluir *enter* en una posición dada de una cadena. Por ello, se utiliza una sola instrucción de salida, en lugar de las 2 del algoritmo.

## ESTRUCTURAS SELECTIVAS

Las *estructuras selectivas (condicionales o de decisión)*, son instrucciones que brindan al programador la oportunidad de establecer diversas alternativas de solución que podrán ser realizadas o no de acuerdo al comportamiento de la aplicación durante el tiempo de ejecución. Los lenguajes de programación incluyen 3 tipos de instrucciones selectivas: simple, doble y múltiple.

### Instrucción condicional simple

La *instrucción condicional simple* permite que la aplicación decida en tiempo de ejecución si se utilizará o no cierto bloque de código (ver Tabla 15).

	Lenguaje algorítmico		Java
	Pseudocódigo	Diagrama de flujo	
<b>Sintaxis</b>	<u>si</u> expresión_lógica <u>entonces</u> instrucción(es) <u>fin si</u>		<b>if</b> (expresión_lógica) instrucción(es);

Tabla 15. Sintaxis de declaración de la instrucción condicional simple.

Instrucción condicional doble

La *instrucción condicional doble* permite al programador plantear dos alternativas de solución, de las que el procesador solo ejecutará una de acuerdo al comportamiento de la aplicación (ver Tabla 17 y Tabla 17).

	Lenguaje algorítmico	
	Pseudocódigo	Diagrama de flujo
Sintaxis	<div>si expresión_lógica entonces instrucción(es) sino instrucción(es) fin si</div>	<pre>graph TD; A{expresión_lógica} -- V --&gt; B[instrucción(es)]; A -- F --&gt; C[instrucción(es)]; B --&gt; D[instrucción(es) flujo común]; C --&gt; D; D --&gt; E[ ];</pre>

Tabla 16. Sintaxis de declaración de la instrucción condicional doble en lenguaje algorítmico.

	Java
Sintaxis	<div>if (expresión_lógica) instrucción; else instrucción;  ó  if (expresión_lógica){ instrucción(es); }else{ instrucción(es); }  }</div>

Tabla 17. Sintaxis de declaración de la instrucción condicional doble en Java.

Instrucción de condicional múltiple

La *instrucción condicional múltiple* será utilizada por el programador cuando la solución del problema requiera establecer varias alternativas, de las cuales serán ejecutadas una o varias de acuerdo al estado de la aplicación en determinado momento (ver Tabla 19 y Tabla 18 ).

	Java
Sintaxis	<div>switch (var){ case valor1: instrucción(es); break; case valor2: instrucción(es) ; break; ... case valorN: instrucción(es) ; break; default: instrucción(es); }  }</div>

Tabla 18. Sintaxis de declaración de la instrucción condicional múltiple en Java.

Lenguaje algorítmico		
	Pseudocódigo	Diagrama de flujo
Sintaxis	<u>en caso</u> var opción 1: instrucción(es) opción 2: instrucción(es) ... opción N: instrucción(es) <u>sino</u> instrucción(es) <u>fin caso</u>	

Tabla 19. Sintaxis de declaración de la instrucción condicional múltiple en lenguaje algorítmico.

## Ejemplo de resolución de un problema utilizando instrucciones selectivas

A continuación se plantea un *problema* al que le diseñaremos una solución mediante la *metodología* abordada en el Tema 1 de este curso.

### 1. Definición del problema

Diseñe un programa que juegue a adivina un número. Para ello el usuario deberá pensar un número de un dígito e indicarle al programa si es par o impar y cuál es el residuo al dividirlo entre 5.

### 2. Análisis del problema

#### Entrada

- Paridad
- Residuo

#### Proceso

La restricción de que el usuario solo puede pensar un valor de un dígito, disminuye las posibilidades (0 al 9). No es magia, se requiere encontrar la manera de identificar de manera única cada dígito, y para ello lo primero que se tiene a la mano son los datos que el usuario está proporcionando.

(Considerando al 0 como par)

dígito	paridad	residuo5		dígito	paridad	residuo5
0	par	0	Ordenados por paridad y residuo	0	par	0
1	impar	1		6	par	1
2	par	2		2	par	2
3	impar	3		8	par	3
4	par	4		4	par	4
5	impar	0		5	impar	0
6	par	1		1	impar	1
7	impar	2		7	impar	2
8	par	3		3	impar	3
9	impar	4		9	impar	4

#### Salida

- Número adivinado

Una vez que hemos identificado los aspectos generales anteriores, es necesario preguntarse si ello es suficiente para ser transcrito a un algoritmo. Si la respuesta es sí estamos listos



para pasar a la etapa de Diseño de la solución, pero si la respuesta es no o no sé, habría que continuar con un análisis más detallado, el cual puede plasmarse en el formato anterior o según le sea más claro al analista:

- *Entrada y salida.* No requiere mayor aclaración.
- *Proceso.* Se puede observar que efectivamente la paridad y el residuo diferencian a los dígitos de forma única, es decir que las combinaciones entre los valores de paridad y residuo no se repiten. De tal forma que solo se requiere comparar su contenido para decidir de qué dígito se trata y adivinarlo. Considerando que para la paridad solo hay dos opciones, con ella se utilizará una *condición doble*. Mientras que para el residuo se invocará una *condición múltiple*.

### 3. Diseño de la solución

Precisemos algunos detalles del pseudocódigo presentado en la *Figura 16*:

- En la sección *var* se han declarado 4 variables:
  - *paridad*, en esta variable se lee el primer dato solicitado al usuario, el cual consiste en una sola letra: 'I', 'i', 'P' o 'p'; por ello es de tipo *caracter*.
  - *resp*, es de tipo *caracter* porque guarda la respuesta del usuario, como 's', 'S', 'n' o 'N'.
  - *digito* y *residuo*, en ellas se almacena el dígito que está siendo “adivinado” y el segundo dato solicitado al usuario. Ambos son números pequeños y sin decimales, por lo que su tipo es *entero*.
- En la *Entrada*. Se le pregunta al usuario si desea jugar y si la respuesta es afirmativa, se le solicitan los dos datos de entrada establecidos en el análisis: paridad y residuo.
- En el *Proceso*. En este ejemplo se puede observar el uso de los 3 tipos de instrucciones selectivas:
  - La *instrucción condicional simple*, si *resp='s'* o *resp='S'* entonces, está *decidiendo* si se ejecutará el código del juego. De lo contrario dirigirá flujo del algoritmo al cierre de la instrucción, al fin si, para finalizar con escribir “¡Buen día!”

```

algoritmo ProyectoAdivinaNumero
  a { var
    paridad, resp: char
    digito, residuo: entero
    inicio
    // ----- Entrada
    escribir "¿Deseas jugar? S/N?... "
    leer resp
    b { si resp='s' o resp='S' entonces
      escribir "Nombre: "Piensa un número de un dígito... "
      escribir "Si es par escribe una p o una i para impar... "
      leer paridad
      escribir "Ahora, divídelo entre 5 y dime cual es el residuo... "
      leer residuo
    }
    // ----- Proceso
    si paridad='p' o paridad='P' entonces
      en_caso residuo
        0: digito ← 0
        1: digito ← 6
        2: digito ← 2
        3: digito ← 8
        4: digito ← 4
        sino digito ← -1
      fin_caso
    sino
      c { si paridad='i' o paridad='I' entonces
        en_caso residuo
          0: digito ← 5
          1: digito ← 1
          2: digito ← 7
          3: digito ← 3
          4: digito ← 9
          sino digito ← -1
        fin_caso
      }
    sino
      digito ← -1;
    fin_si
  }
  // ----- Salida
  d { si digito=-1 entonces
    escribir "Uno de los datos de entrada fue inválido"
  }
  sino
    escribir "El número que pensaste fue " + digito
  fin_si
  escribir "¡Buen día!"
fin

```

Figura 16. Algoritmo ProyectoAdivinaNumero representado con pseudocódigo.

- Por su parte, la *instrucción condicional doble* compara el contenido de paridad, para determinar que segmento de código se debe ejecutar, el de par o el de impar. Fíjate que en realidad en el ejemplo se están usando dos condiciones dobles, una dentro de la otra, a esto se le llama *condiciones anidadas*, y sirve para cuando, como en este caso, se requiere decidir entre más de 2 opciones, pero no es necesario o adecuado utilizar la condición múltiple.

En el ejemplo, si la *evaluación de la expresión* de la *primera condición* resulta *verdadera* el flujo del algoritmo se irá directamente a ejecutar el segmento de código inmediato. Pero si es *falsa*, se irá al *sino* que contiene una *segunda condición*. Si esta expresión resulta *verdadera*, se ejecuta el bloque de código inmediato, de lo contrario se va al *sino* y guarda un -1 en le digito.

- La condición múltiple es la que finalmente selecciona y asigna a la variable digito el valor pensado por el usuario. Nota que si el residuo no coincide con los valores del 0 al 4, a digito se le asigna un -1.
- d. La *Salida*. Como se puede observar el segmento de código se encuentra dentro de la primera condición, es decir que solo se ejecuta si está fue verdadera. Además utiliza una condición doble para verificar el contenido de la variable dígito, de tal forma que si el usuario escribe un valor incorrecto en paridad o residuo, se le muestra un mensaje de valor inválido, de lo contrario se adivina el número que pensó.

#### 4. Codificación

Veamos algunos aspectos relevantes del código (ver Figura 17):

- Tal como en el algoritmo solo fue necesario declarar 4 variables: 3 de tipo *char* y 2 *byte* (pues los valores que guardan son pequeños y sin decimales).
- En la instrucción *resp=entrada.nextLine().charAt(0);*, el método *nextLine* lee lo que el usuario escribe hasta encontrar un *enter*, interpretándolo como una *cadena*. Aunque en este caso solo se trata de un *caracter*, no es válido asignarlo directamente, Por ello es necesario usar el método *charAt(0)*, que toma el carácter indicado de la cadena. Cabe señalar que en una cadena los caracteres que contiene se numeran del 0 en adelante.
- La entrada puede hacerse para distintos tipos de datos, simplemente sustituyendo el método *nextLine* por el que corresponda. Por ejemplo el *nextByte* que se utiliza en el programa para leer la variable *residuo* que ha sido declarada de tipo *byte*.

```

package proyectoadivinanumero;
import java.util.Scanner;
/**
 * @author Guadalupe
 * Marzo 2020
 */
public class ProyectoAdivinaNumero {

    public static void main(String[] args) {
        // --- Declaración de variables
        char paridad, resp;
        byte digito, residuo;

        // ----- Entrada
        Scanner entrada = new Scanner(System.in);
        System.out.print("¿Deseas jugar? S/N?... ");
        resp=entrada.nextLine().charAt(0);
        if (resp=='s' || resp=='S')
        {
            System.out.print("Piensa un número de un dígito...\nSi es par escribe una p o una i para impar... ");
            paridad=entrada.nextLine().charAt(0);
            System.out.print("Ahora, divídelo entre 5 y dime cual es el residuo... ");
            residuo=entrada.nextByte();

            // ----- Proceso
            if (paridad=='p' || paridad=='P')
                switch(residuo){
                    case 0: digito=0; break;
                    case 1: digito=6; break;
                    case 2: digito=2; break;
                    case 3: digito=8; break;
                    case 4: digito=4; break;
                    default: digito=-1;
                }
            else
                if (paridad=='i' || paridad=='I')
                    switch(residuo){
                        case 0: digito=5; break;
                        case 1: digito=1; break;
                        case 2: digito=7; break;
                        case 3: digito=3; break;
                        case 4: digito=9; break;
                        default: digito=-1;
                    }
                else
                    digito=-1;

            // ----- Salida
            if (digito==-1)
                System.out.printf("Uno de los datos de entrada fue inválido");
            else
                System.out.printf("El número que pensaste fue " + digito);
        }
        System.out.printf("\n¡Buen día!\n");
    }
}

```

Figura 17. Código en Java del ProyectoAdivinaNumero.

## ESTRUCTURAS ITERATIVAS

Las *estructuras iterativas* son instrucciones que permitirán al programador diseñar soluciones que contemplen la repetición de bloques de código, ya sea un número de veces predeterminado o condicionado al comportamiento de la aplicación. Los lenguajes de programación incluyen 3 tipos de instrucciones iterativas: desde, mientras, hacer-mientras.

Es importante considerar que para el buen funcionamiento de cualquier estructura de control resulta contundente la utilización de una *variable de control*, para la cual se deben establecer de manera congruente: el *valor inicial*, la *condición de permanencia o salida* y la *modificación del valor de la variable*.

### Instrucción desde (para)

En lenguaje algorítmico el *ciclo desde*, es también conocido como ciclo predeterminado o automático, ya que inicialmente los lenguajes de programación incluían para esta instrucción, una sintaxis muy simple que solo requería establecer una variable de control inicializada y un valor final, lo que automáticamente determinaba el número de veces que cierto bloque de código se ejecutaría.

Aunque actualmente la sintaxis establecida por algunos lenguajes para dicha instrucción es más sofisticada, la estructura mantiene un formato sencillo, muy conveniente para soluciones en las que de manera clara se visualiza la cantidad de iteraciones que se desean generar (ver Tabla 21 y Tabla 20).

	Java
Sintaxis	for (var=val_ini ; exp_lógica ; modif_var) instrucción;
	ó  for (var=val_ini ; exp_lógica ; modif_var) { instrucción(es); }

Tabla 20. Sintaxis de declaración de la instrucción desde en Java.

Lenguaje algorítmico	
Pseudocódigo	Diagrama de flujo
<b>Sintaxis</b> <u>desde</u> var←val_ini <u>hasta</u> N <u>hacer</u> instrucción(es) <u>fin desde</u>	<pre> graph TD     Start(( )) --&gt; Init[varCtrl ← valorInicial]     Init --&gt; Cond{expresión_lógica}     Cond -- F --&gt; Instr1[instrucción(es)]     Instr1 --&gt; End1(( ))     Cond -- V --&gt; Instr2[instrucción(es) modificación varCtrl]     Instr2 --&gt; Cond   </pre> <pre> graph TD     Start(( )) --&gt; Init[varCtrl ← valorIni]     Init --&gt; Cond{expr_lógica}     Cond -- F --&gt; Instr1[instrucción(es)]     Instr1 --&gt; End1(( ))     Cond -- V --&gt; Instr2[instrucción(es) modif varCtrl]     Instr2 --&gt; Cond   </pre> <pre> graph TD     Start(( )) --&gt; Init[varCtrl ← valorIni hasta valorFin]     Init --&gt; Cond{varCtrl &lt;= valorFin}     Cond -- V --&gt; Instr1[instrucción(es)]     Instr1 --&gt; End1(( ))     Cond -- F --&gt; Instr2[instrucción(es)]     Instr2 --&gt; Cond   </pre>

Tabla 21. Sintaxis de declaración de la instrucción desde en lenguaje algorítmico.

## Instrucción mientras

La *instrucción mientras* es un ciclo condicionado que controla su ejecución a través del establecimiento de una *condición (expresión lógica) de permanencia* ubicada *al inicio del bucle*. Mientras la evaluación de dicha expresión sea verdadera, el bucle continuará ejecutándose, y una vez que resulte falsa, este se detendrá y pasará a la siguiente instrucción (ver Tabla 22).

## Instrucción hacer-mientras

La *instrucción hacer-mientras*, es un ciclo condicionado que controla su ejecución a través del establecimiento de una *condición (expresión lógica) de permanencia* ubicada *al final del bucle*. Mientras la evaluación de dicha expresión sea verdadera, el bucle continuará ejecutándose (ver Tabla 23).

	Lenguaje algorítmico		Java
	Pseudocódigo	Diagrama de flujo	
<b>Sintaxis</b>	<pre> var←val_ini <b><u>mientras</u></b> exp_lógica <b><u>hacer</u></b>   instrucción(es)   modif_var <b><u>fin_mientras</u></b>           </pre>	<pre> graph TD     Start([Start]) --&gt; Init[varCtrl←valorInicial]     Init --&gt; Cond{expresión_lógica}     Cond -- F --&gt; Body[instrucción(es)]     Body --&gt; Cond     Cond -- V --&gt; BodyMod[instrucción(es) modificación varCtrl]     BodyMod --&gt; Cond     </pre>	<pre> var=val_ini; <b>while</b> (exp_lógica ) {   instrucción(es);   modif_var; }           </pre>

Tabla 22. Sintaxis de declaración de la instrucción mientras.

	Lenguaje algorítmico		Java
	Pseudocódigo	Diagrama de flujo	
<b>Sintaxis</b>	<pre> var←val_ini <b><u>hacer</u></b>   instrucción(es)   modif_var <b><u>mientras</u></b> exp_lógica           </pre>	<pre> graph TD     Start([Start]) --&gt; Init[varCtrl←valorInicial]     Init --&gt; Body[instrucción(es) modificación varCtrl]     Body --&gt; Cond{expresión_lógica}     Cond -- V --&gt; Body     Cond -- F --&gt; EndBox[instrucción(es)]     EndBox --&gt; End([End])     </pre>	<pre> var=val_ini; <b>do</b>{   instrucción(es);   modif_var; } <b>while</b> (exp_lógica );           </pre>

Tabla 23. Sintaxis de declaración de la instrucción hacer-mientras.

## Ejemplo de resolución de un problema utilizando instrucciones iterativas

A continuación se plantea un *problema* al que le diseñaremos una solución mediante la *metodología* abordada en el Tema 1 de este curso.

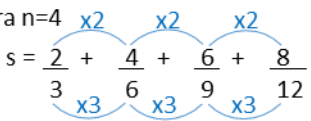
### 1. Definición del problema

Diseña un programa que calcule el resultado de la siguiente serie:

$$s = 1 + \frac{2}{3} + \frac{4}{6} + \frac{6}{9} + \dots + \frac{(n*2)}{(n*3)} \quad \text{en donde } n \geq 1$$

Resuelva para los tres tipos de ciclos.

## 2. Análisis del problema

Entrada	Proceso	Salida
n	<p>Para resolver una serie se precisa identificar el número de términos que deben calcularse y la relación existente entre ellos. Esto se puede lograr analizando el comportamiento de la serie para uno o varios valores de n según se requiera:</p> $s = \frac{2}{3} + \frac{4}{6} + \frac{6}{9} + \dots + \frac{(n*2)}{(n*3)} \quad n \geq 1$ <p>Para n=1</p> $s = \frac{2}{3}$ <p>Para n=4</p>  <p>El comportamiento regular de la serie, nos deja ver que su cálculo puede ser automatizado mediante el uso de una instrucción cíclica que cuente de 1 hasta n, agregando un término en cada iteración.</p>	s

Una vez que hemos identificado los aspectos generales anteriores, es necesario preguntarse si ello es suficiente para ser transcrito a un algoritmo. Si la respuesta es sí estamos listos para pasar a la etapa de Diseño de la solución, pero si la respuesta es no o no sé, habría que continuar con un análisis más detallado, el cual puede plasmarse en el formato anterior o según le sea más claro al analista:

- **Entrada y salida.** No requiere mayor aclaración.
- **Proceso.** Será implementado con los tipos de 3 ciclos.

Recuerda que el producto que obtengas en esta etapa deberá ser un documento claro, ordenado y listo para servir de sustento en el diseño de la solución.

## 3. Diseño de la solución

Antes de continuar me parece conveniente que precisemos algunos detalles del pseudocódigo presentado en la *Figura 18*:

- En la sección *var* se han declarado 3 variables:
  - *n*, en esta variable se lee el número de términos que tendrá la serie. Es un valor pequeño y sin decimales, por ello es *entera*.



- $c$ , es de tipo *entera*, pues es la variable que utilizan los ciclos para contar desde 1 hasta  $n$ .
- $s$ , es la variable encargada de acumular el resultado de la serie. Es de tipo *real* debido a que la expresión para calcularla contiene una división.

```

    algoritmo ProyectoSerie
    a { var
        n, c: entero
        s: real
    b { inicio
        // ----- Entrada
        escribir "Escribe el valor de n para calcular la serie"
        escribir "N: "
        leer n

        // ----- Proceso
        // Cálculo de la serie con ciclo desde
        s ← 0
        desde c ← 1 hasta n hacer
            s ← s + (c*2)/(c*3)
        fin desde
        // ----- Salida
        escribir "Con el ciclo desde, S = " + s

        // Cálculo de la serie con ciclo MIENTRAS
        s ← 0
        c ← 1
        mientras ( c<=n ) hacer
            s ← s + (c*2)/(c*3)
            c ← c+1
        fin mientras
        // ----- Salida
        escribir "Con el ciclo mientras, S = " + s

        // Cálculo de la serie con ciclo HACER-MIENTRAS
        s ← 0
        c ← 1
        hacer
            s ← s + (c*2)/(c*3)
            c ← c + 1
        mientras ( c<=n )
        // ----- Salida
        escribir "Con el ciclo hacer-mientras, S = " + s
    c {
    fin
  
```

Figura 18. Algoritmo ProyectoSerie representado con pseudocódigo.

- b. En la *Entrada*. Simplemente consiste en solicitar y recibir el valor de  $n$ .

c. En el *Proceso*. En este ejemplo se puede observar el uso de los 3 tipos de instrucciones repetitivas:

- Fuera de cada ciclo.
  - ✓ Antes, se *inicializa la variable acumulador* *s* en 0, esto debido a que lo que se acumulará es una sumatoria, por lo tanto el 0 limpia la variable sin afectar el resultado final.
  - ✓ Después del ciclo, se muestra el valor de *s* al usuario.
- *Desde*. Su sintaxis incluye las tres partes fundamentales de un ciclo. La inicialización de la variable de control,  $c \leftarrow 1$ ; la condición de permanencia, implícita en el hasta *n*; y la modificación de la variable de control, que hace automáticamente.

El ciclo cuenta automáticamente desde 1 hasta *n* y en cada iteración va acumulando el resultado de la expresión dada para el cálculo de los términos.

- *Mientras*. Su sintaxis solo incluye la condición de permanencia, por lo que la inicialización de la variable de control se hace antes de iniciar el ciclo y la modificación es realizada dentro del ciclo.

El ciclo se ejecuta mientras la condición de permanencia es verdadera y finaliza cuando se hace falsa. En cada iteración acumula el resultado de la expresión dada para el cálculo de los términos e incrementa la variable de control.

- *Hacer-mientras*, Al igual que la instrucción *mientras*, su sintaxis solo incluye la condición de permanencia, por lo que la inicialización de la variable de control se hace antes de iniciar el ciclo y la modificación es realizada dentro del ciclo.

Este ciclo funciona de manera similar al *mientras*, pero su condición de permanencia la tiene al final, por lo que aunque en este ejemplo no hace diferencia, habrá otros usos en lo que si lo haga.

d. La *Salida*. Como se mencionó antes la salida está incluida en el proceso, justo después de cada ciclo.

## 4. Codificación

```

package proyectoserie;
import java.util.Scanner;
/**
 * @author Guadalupe
 * Marzo 2020
 */
public class ProyectoSerie {

    public static void main(String[] args) {
        // --- Declaración de variables
        byte n, c;
        float s;

        // ----- Entrada
        System.out.print("Escribe el valor de n para calcular la serie");
        Scanner entrada = new Scanner(System.in);
        System.out.print("\nN: ");
        n=entrada.nextByte();

        // ----- Proceso
        // Cálculo de la serie con ciclo FOR
        s= 0;
        for (c=1; c<=n; c++)
            s = s + (float)(c*2)/(c*3);
        // ----- Salida
        System.out.printf("Con el ciclo for, S = %2.3f\n", s);

        // Cálculo de la serie con ciclo WHILE
        s= 0;
        c=1;
        while ( c<=n )
        {
            s = s + (float)(c*2)/(c*3);
            c++;
        }
        // ----- Salida
        System.out.printf("Con el ciclo while, S = %2.3f\n", s);

        // Cálculo de la serie con ciclo DO-WHILE
        s= 0;
        c=1;
        do
        {
            s = s + (float)(c*2)/(c*3);
            c++;
        }while ( c<=n );
        // ----- Salida
        System.out.printf("Con el ciclo do-while, S = %2.3f\n", s);
    }
}

```

Figura 19. Código en Java del ProyectoSerie.

Veamos algunos aspectos relevantes del código anterior (*ver Figura 19*):

- Las variables fueron declaradas tal como se planteó en el algoritmo: dos de tipo byte (entero pequeño) y una float (real).
- Una diferencia significativa la encontramos en la sintaxis del ciclo for, el cual aunque en algunos lenguajes se escribe de forma muy similar al pseudocódigo, en Java las 3 partes del ciclo se escriben de manera explícita: `for (c=1; c<=n; c++)` .
- En la salida se utiliza la instrucción `printf` que permite dar formato a los valores que se van a mostrar. El `%` indica la posición que ocupará el valor en el mensaje, luego se expresa el número de dígitos antes y después del punto decimal. Finalmente la `f` significa que se trata de un valor flotante.

*Para el cerebro lo mismo es pensar que hacer*

No me creas, haz la prueba. Cierra tus ojos e imagina que tienes un limón frente a ti, de esos verdes y jugosos. Luego miras como lo partes suavemente, mientras empiezan a escaparse pequeñas gotas de zumo... Ahora dime ¿cómo te fue?

## TEMA 4. ORGANIZACIÓN DE DATOS

Cuando se trata de dar solución a cualquier problema, tan importante como definir los pasos a seguir, será identificar los datos que vamos a necesitar y la información que deberá obtenerse al procesarlos. Para que esto tome mayor sentido, que te parece si empezamos por imaginar que me quiero ir de vacaciones. Lo primero que necesito saber es a dónde quiero ir. Recorro a internet, pregunto a mis amigos y empiezo a tomar nota de todos los lugares que me parecen atractivos. Pero entonces me doy cuenta de que para decidir entre todos ellos, necesito saber más. Procedo entonces a identificar lo que tengo que hacer para obtener más información, los datos me permitirán planear mi viaje: destino, transporte, hospedaje, presupuesto, costos, duración, etc. Fíjate como al resolver de manera efectiva cualquier *problema* cotidiano, requerimos recopilar cierta información para establecer con precisión el punto de partida (*origen*), el resultado que deseamos obtener (*destino*) y los pasos a seguir para llegar de uno a otro (*proceso*). Piénsalo así, si una persona te pide que le expliques como llegar a un lugar, pero no sabe decirte donde está y/o a dónde quiere llegar, será imposible trazar una ruta, aun cuando conozcas perfectamente la ciudad.

Por esa razón, como se mencionó y has empezado a practicar desde el *Tema 1*, cuando se diseña la solución de un problema, además del algoritmo, y la interfaz, es preciso identificar los datos que el nuevo sistema recibirá y procesará. Ya hemos explicado que todos los datos que el programa utiliza los guarda en la memoria ya sea mediante la ejecución de una instrucción de *asignación* o de una *lectura* (entrada), por ello en este tema profundizaremos un poco más en la complejidad que conlleva la manipulación de la información.

No se trata de utilizar simples valores aislados, sino de captar, clasificar y *organizar datos* para que su almacenamiento y procesamiento garantice su *seguridad*, *integridad*, *flexibilidad* y, por supuesto, su *disponibilidad*.

## Conceptos básicos

### Datos e Información

Podemos aseverar que aunque en lo cotidiano, los términos datos e información son utilizados esencialmente como sinónimos. Esto no sucede en el área de informática, pues se considera que los *datos* son la representación de algún hecho, concepto o entidad real, que pueden tomar diferentes formas como texto, números, imágenes o sonido, y que esencialmente no conlleva ningún significado. Mientras que la *información* implica datos debidamente organizados y procesados de manera que puedan ser comprendidos e interpretados por el usuario permitiéndole tomar decisiones y acciones (Joyanes Aguilar, Fundamentos de programación. Algoritmos y estructura de datos, 1996).

### Atributo

Un *atributo* es un rasgo de una entidad, una característica que constituye un solo aspecto (dato) de un todo, una partecita que por sí misma no tiene significado alguno.

### Abstracción

Para el programador la *abstracción* constituye una de sus habilidades fundamentales, ya que nos permite representar segmentos de la realidad a través de las aplicaciones: juegos, controladores de vuelo, procesadores de texto, calendarios, etc.

La *abstracción* consiste en el descubrimiento de un aspecto o característica esencial de una entidad. Es decir, identificar un atributo que caracterice de manera única al objeto que deseamos definir o describir.

En mi experiencia como docente, este es un concepto que comúnmente causa ruido en los estudiantes, pues cuando pregunto qué significa, inmediatamente les viene a la mente algo extraño, que no tiene forma, y generalmente me dicen “esas pinturas que no les entiendes o las esculturas raras”. Es aquí donde necesitamos hacer un alto y preguntarnos si de verdad el arte abstracto no tiene forma o en realidad quien lo entiende o no, es quien lo mira.

Vamos pues a un ejemplo cotidiano, e imagina que tu hermanito(a) o sobrino(a), viene a ti diciendo “mira te dibuje”. Y cuando miras te encuentras con algo que parecen unos rayones de colores. Si te ha sucedido, o lograste imaginarlo, dime... ¿Qué cara pondrías?, Le dirías “Hay mijo esto no es nada, son puras rayas” o “Que bonito, haber

explícame...”. Lo más probable es que si le pides que te explique él o ella te dirá exactamente donde está tu cabeza, los brazos, las manos, las piernas y los pies. Y si tienes suerte tendrás ojos nariz, boca y orejas. Jajajaja, ya sé, es tierno pero extraño, porque aunque con toda seguridad te explique, quizá tú no alcances a ver la mayor parte de lo que te está describiendo. Pues bien, eso es abstracción, el niño te miro y según su apreciación, experiencia y habilidades hizo una representación tuya integrando en ella las características que considera esenciales, las que unidas te representan. Porque si se dibuja a sí mismo justo a tu lado, no lo hará de la misma manera, pues su representación será otra, parecida o diametralmente distinta, pero otra.

Como programadores, nuestra labor consiste en analizar la realidad para a través de la abstracción encontrar la forma de representarla mediante una aplicación que de manera amigable se comunice con el usuario, procese sus datos y le ofrezca la información que le ayude a tomar decisiones. Si aún no te queda claro, piensa en un juego y en como cuando estás interactuando con él, sientes que en realidad vas manejando aquél auto de carreras, o como rompes una pared. Pero no es así, se trata solamente de una muy buena representación de la realidad, un muy buen trabajo de *abstracción*.

Cuando desarrollamos la solución de un problema, realizamos dos tipos de *abstracción*:

- *Abstracción de datos*: Esto se da cuando identificas las características de aquello que deseas representar y le pones un nombre significativo a cada una. Luego determinas las relaciones que se dan entre ellas, es decir te das cuenta que organizados de determinada forma representan a ese algo. Y finalmente estableces las operaciones que se podrán realizar con esos datos o lo que es lo mismo, para que sirven, que se podrá hacer con ellos. Estarás de acuerdo en que si los datos no se relacionan de ninguna forma, no se pueden organizar y no sirven para nada, pues son inútiles y no tendrá ningún sentido almacenarlos en la aplicación que estas desarrollando.
- *Abstracción de procedimientos*: Cuando diseñas un algoritmos, es decir, los pasos que darán solución al problema, estás haciendo *abstracción de procedimientos*, porque al analizar la realidad identificas como es que las cosas suceden en ella y buscas la forma de que la computadora pueda simular que hace lo mismo. A ese código que generes también le darás un nombre a través del cual podrás utilizarlo cuando lo necesites. Por ejemplo cuando presionas en tu calculadora la tecla que calcula la raíz cuadrada de un número, en realidad estas mandando ejecutar un

procedimiento o módulo, o sea un segmento de código que procesa el dato de entrada y muestra en la pantalla el resultado, esto sucede porque en algún momento un programador investigó como se calcula la raíz cuadrada de un número y diseñó los pasos para que la calculadora pudiera hacerlo.

### Estructura de datos

"Una *estructura de datos* es una clase de datos que se puede caracterizar por su organización, y las operaciones definidas sobre ella." (Loomis, 1991)

Dicho de otra forma, una *estructura de datos* es una colección de datos, relacionados entre sí y las operaciones que se pueden realizar con ellos. Describe el formato en que los valores serán almacenados, así como la manera en que podrán ser accedidos para su consulta y modificación (Sena, 2019).

Hay varias clasificaciones para las *estructuras de datos*, sin embargo por la amplitud del tema este será tratado a profundidad en asignaturas posteriores. Para efectos de este curso bastará saber que se dividen en *estáticas* y *dinámicas*. Y que abordaremos a las estáticas, a través del estudio de las que se consideran básicas, *arreglos* y *registros*.

## ARREGLOS

### Definición

Un *arreglo* o *array* es una estructura de datos homogénea, de acceso directo y estática. Se trata de un conjunto de datos relacionados entre sí, considerado:

- **Homogéneo:** Todos elementos que lo conforman son del mismo tipo, es decir que no se puede guardar un dato de tipo distinto a los demás. Por ejemplo, si el arreglo sirve para guardar perros, no puedo meter un elefante.
- **Acceso directo:** Se puede visitar directamente a cualquiera de sus elementos (datos) individuales en el momento deseado. Un ejemplo sería cuando quieres guardar o tomar algo de la alacena, puedes abrir directamente el cajón deseado sin necesidad de ir cajón por cajón hasta llegar al que necesitas. Jajaja bueno, siempre y cuando sepas lo que estás buscando y el contenido de la alacena se encuentre en orden.



- *Estático*: La cantidad máxima de elementos una vez definida queda fija (no puede aumentarse ni disminuirse). Esto funciona como un camión de pasajeros que cuenta con un número específico de asientos. Pueden subir tantas personas como asientos hay, no se puede agregar ningún asiento más. Y si el número de personas que ingresa es menor al cupo, tampoco se desaparecen asientos, simplemente quedan vacíos. Esto se da por que la capacidad del camión es *estática*.

## Clasificación

La *clasificación* de *arreglos* que utilizaremos en este curso será:

- *Unidimensionales*. El conjunto de datos se organiza como una lista, en la que los elementos se almacenan uno detrás de otro.
- *Multidimensionales*. El conjunto de datos se organiza en n dimensiones, como una tabla (bidimensional), como un cubo (tridimensional), etc.

Aun cuando la concepción humana solo nos permite imaginarnos arreglos unidimensionales, bidimensionales y tridimensionales, una vez que se comprende la relación que existe entre los elementos, es fácil definir arreglos de n dimensiones.

# ARREGLOS UNIDIMENSIONALES

En un arreglo *unidimensional*, también conocido como *lista* o *vector*, sus elementos se almacenan uno detrás de otro (*organización física lineal*). Este tipo de estructura nos permite guardar en la memoria datos cuya *organización lógica es lineal*. Por ejemplo: la lista del mandado, las edades de los estudiantes del grupo, las calificaciones de los aspirantes en el examen de admisión, los nombres de los invitados a un evento, etc.

## Conceptos básicos

### Elemento o componente

Un dato o valor guardado en un arreglo, es llamado *elemento* o *componente*.

## Casilla o celda

A cada espacio del arreglo en el que se almacena un dato o valor se le denomina *casilla* o *celda*.

## Índice

Las celdas o casillas se distinguen unas de otras por un número al que se le llama *índice*.

## Tamaño

El *tamaño* se refiere al número máximo de elementos que caben en un arreglo.

## Organización física

*Organización física* se refiere a la relación entre los elementos en la memoria, es decir a como están almacenados físicamente.

## Organización lógica

*Organización lógica* se refiere a la relación que de manera virtual tiene los elementos, es decir la forma en que los gestionamos, que puede ser distinta a la organización que guardan de físicamente.

## Representación gráfica

Los *arreglos unidimensionales* pueden ser visualizados como una *secuencia de elementos adyacentes*. Pudiendo ser representados indistintamente de manera vertical u horizontal, según considere conveniente el programador (ver Figura 20 y Figura 21).

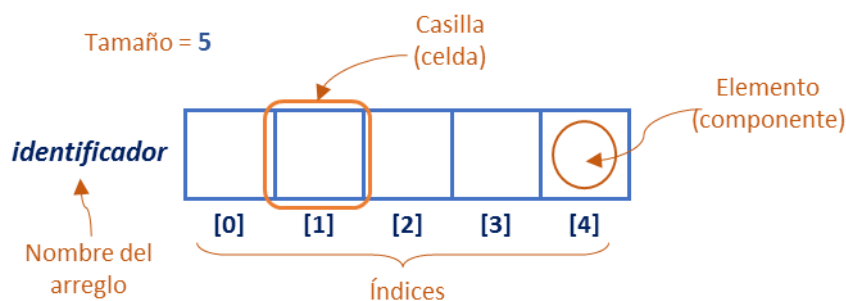


Figura 20. Representación gráfica horizontal de un arreglo unidimensional.

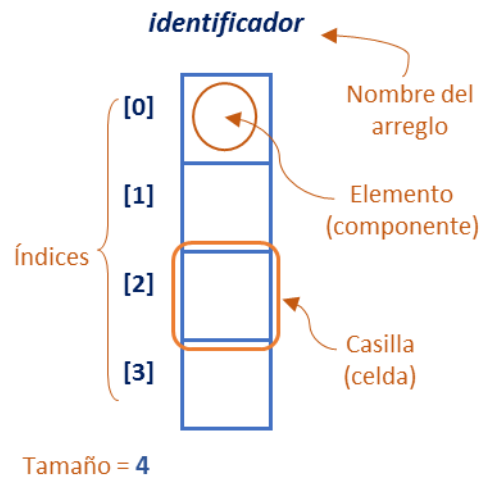


Figura 21. Representación gráfica vertical de un arreglo unidimensional.

En un *arreglo* los elementos *comparten* un mismo *identificador*, por tal razón si se requiere acceder a uno de ellos, es necesario especificar el índice de la celda en la que se encuentra. Para que quede más claro, imagina que estás chateando con un(a) amigo(a), te invita a su casa y te pasa su ubicación para que sepas llegar. Muy probablemente para empezar solamente pondrás atención al rumbo de la ciudad en el que se encuentra tu destino, pero una vez que vas llegando, lo más importante será saber el nombre de la calle y el número de la casa, necesitas las dos cosas porque si solo te dan una de ellas no podrás llegar. Haciendo analogía encontramos que:

- La calle es el identificador del arreglo.
- El número de la casa es el índice.
- La casa de tu amigo(a) es la casilla o celda.
- Y tu amigo(a) es el elemento o componente.

Existen dos formas de declarar un *arreglo unidimensional*: vacío o con datos. En el primer caso, en *lenguaje algorítmico* solo se debe hacer su declaración, pero en *Java* es necesario declararlo y además crearlo mediante la invocación de la palabra reservada **new**. Por otro lado, si se desea que el arreglo contenga datos desde un inicio, en ambos lenguajes solo se requiere su declaración (ver Tabla 25 y Tabla 24).

Algo muy importante a tener en cuenta es que en los *lenguajes algorítmicos* generalmente los índices de las celdas del arreglo, van del 1 al tamaño, mientras que los lenguajes de

programación los manejan de forma distinta, algunos inclusive permiten al programador utilizar índices negativos o letras. Sin embargo, es importante que recuerdes que en *Java* *siempre* se numeran de *0 a tamaño-1*, de tal forma que si tamaño es igual a 7, el índice de la primera casilla será 0 y el de la última 6.

	Java
<b>Sintaxis</b>	<pre>tipoDato identificadorArreglo[]; identificadorArreglo=new tipoDato[tamaño]; ó tipoDato identificadorArreglo[]=new tipoDato[tamaño];</pre>
<b>Ejemplos</b>	<pre>double promedios[]; promedios=new double[30]; ó double promedios=new double[30]; String nombres []={"Carola", "Lidia", "Pedro"};</pre>

Tabla 24. Sintaxis de declaración de un arreglo unidimensional en Java.

	Lenguaje algorítmico
<b>Sintaxis</b>	identificadorArreglo: <u>arreglo</u> [1..tamaño] <u>de</u> tipoDato
<b>Ejemplos</b>	<pre>promedios: <u>arreglo</u>[1..30] <u>de</u> <u>real</u> nombres: <u>arreglo</u>[] <u>de</u> <u>cadena</u> ={"Carola", "Lidia", "Pedro"}</pre>

Tabla 25. Sintaxis de declaración de un arreglo unidimensional en lenguaje algorítmico.

## Operaciones

En un *arreglo unidimensional* se pueden realizar, las operaciones mencionadas a continuación (Joyanes Aguilar, Fundamentos de programación. Algoritmos, estructura de datos y objetos, 2008):

- Declaración y/o creación de la estructura.
- Asignación, lectura y escritura de elementos, recorrido (acceso secuencial).
- Actualización: adición, inserción y eliminación de elementos.
- Ordenación y búsqueda

## Aplicaciones

Los *arreglos unidimensionales* pueden ser utilizados cuando en un programa se requiere guardar un conjunto de  $n$  elementos relacionados y del mismo tipo, en cuyo caso sería poco práctico declarar  $n$  variables independientes para almacenarlos en la memoria. En lugar de eso se declara una sola *estructura (arreglo)* que permita guardar y mantener disponibles los datos de manera ágil y segura. Por ejemplo para calcular el promedio de las calificaciones finales de un grupo, para registrar el promedio de ventas diarias de un establecimiento en el último mes, para guardar la lista de los nombres de los invitados a un evento, etc.

### Ejemplo de resolución de un problema utilizando un arreglo unidimensional

A continuación se plantea un *problema* al que le diseñaremos una solución mediante la *metodología* abordada en el Tema 1 de este curso.

#### 1. Definición del problema

Diseñe un programa que procese los promedios finales del grupo de Programación Avanzada. A partir de los promedios proporcionados por el profesor, se deberá mostrar el promedio general, la calificación más alta.

#### 2. Análisis del problema

Entrada	Proceso	Salida
<ul style="list-style-type: none"> <li>Número de estudiantes en el grupo.</li> <li>Promedios de los estudiantes del grupo.</li> </ul>	<ul style="list-style-type: none"> <li>Calcular el promedio general.</li> <li>Determinar la calificación más alta.</li> </ul>	<ul style="list-style-type: none"> <li>promedio general.</li> <li>calificación más alta.</li> </ul>

Una vez que hemos identificado los aspectos generales anteriores, es necesario preguntarse si ello es suficiente para ser transcrito a un algoritmo. Si la respuesta es sí estamos listos para pasar a la etapa de Diseño de la solución, pero si la respuesta es no o no sé, habría que continuar con un análisis más detallado, el cual puede plasmarse en el formato anterior o según le sea más claro al analista:

- Entrada y salida:* Como se ha venido comentando, tanto para solicitar datos como para mostrar los resultados, el sistema deberá hacerlo de forma *amigable*.

Específicamente para este ejemplo, vislumbrar que los promedios son un conjunto homogéneo de valores que deberán ser recibidos y almacenados uno por uno, de manera que luego sean fáciles de procesar.

- **Proceso.** Por lo general es aquí en donde se requiere hacer un análisis más detallado, pues los pasos que identifiquemos serán la parte fundamental de la solución. Por ejemplo, es necesario saber que tanto para calcular el promedio como para determinar la calificación más alta, se requiere *recorrer el arreglo*, es decir, visitar todas las casillas, desde la primera hasta la última. Con el tiempo, esto último no será un problema, incluso de memoria te sabrás el segmento de código para hacerlo, pues constituye una de las acciones más comunes cuando de arreglos se trata.

Recuerda que el producto que obtengas en esta etapa deberá ser un documento claro, ordenado y listo para servir de sustento en el diseño de la solución.

### 3. Diseño de la solución

Precisemos algunos detalles del pseudocódigo presentado en la *Figura 22*:

- a. En la sección *var* hemos declarado 6 variables, las 4 que habíamos identificado desde el análisis, y 2 variables que se requieren para realizar cálculos intermedios.
- b. En la *Entrada*. Como podrás observar, el *primer ciclo desde* utiliza como *variable contador* a *cont*, que va de 1 hasta *numEst*. Esto se debe a que aunque se ha definido el tamaño del vector en 40, también se le ha solicitado al usuario el número de estudiantes del grupo, permitiéndole establecer la cantidad de calificaciones que desea almacenar. La variable contador *cont* está siendo usada como índice del arreglo, de tal forma que cada vez que se incrementa, la instrucción *calif[cont]* accede a una casilla diferente y va guardando el dato de entrada. Es decir, cuando *cont* vale 1, el valor leído es guardado en *calif[1]* (celda 1 del arreglo); cuando vale 2, en *calif[2]* (celda 2 del arreglo); y así sucesivamente.
- c. En el *Proceso* el *segundo ciclo desde*:
  - Nota que tanto las instrucciones de inicialización de *suma* y de *mayor*, como la del cálculo del *promedio* son realizadas fuera del ciclo, cada una en el lugar correspondiente de acuerdo al momento en que deseamos que sean ejecutadas. Estos detalles son sumamente importantes pues hacen la diferencia entre obtener los resultados deseados o no.

- Con la *instrucción*  $\text{suma} = \text{suma} + \text{calif}[\text{cont}]$ ; se está calculando la sumatoria de las calificaciones de los estudiantes. Esto sucede porque una vez más la variable contador del ciclo está siendo utilizada como índice del arreglo, de tal forma que cuando incrementa su valor el contenido de  $\text{calif}[\text{cont}]$  es acumulado en la variable suma. Es decir, cuando  $\text{cont}$  vale 1,  $\text{suma} = \text{suma} + \text{calif}[1]$ ; cuando vale 2,  $\text{suma} = \text{suma} + \text{calif}[2]$ ; y así sucesivamente.
  - Se incluye una *instrucción condicional simple* que va comparando el valor de  $\text{calif}[\text{cont}]$  contra el que contiene la variable *mayor* y solo cuando la condición sea verdadera, se ejecutará la asignación ahí indicada. Con ello nos aseguramos de que la variable *mayor* mantendrá el *valor más grande*.
- d. La *Salida* como podrás observar es muy sencilla. Consta de dos instrucciones de escritura encargadas de mostrar al usuario el promedio general y la calificación más alta.

```

algoritmo ProyectoCalificaciones
  var
  a { numEst, cont: entero
      calif: arreglo[1..40] de real
      suma, prom, mayor : real
  inicio
    // ----- Entrada
    escribir "¿Cuántos estudiantes hay en tu grupo? (Máximo 40)"
    leer numEst
    b { desde cont ← 1 hasta numEst hacer
        escribir "Escribe la calificación del alumno " + (cont + 1) + " : "
        leer calif[cont]
      fin_desde

    // ----- Proceso
    suma ← 0.0
    mayor ← calif[1]
    c { desde cont ← 1 hasta numEst hacer
        suma = suma + calif[cont]
        si calif[cont] > mayor entonces
          mayor ← calif[cont]
        fin_si
      fin_desde
    prom ← suma/numEst

    d { // ----- Salida
        escribir "El promedio general del grupo es " + prom
        escribir "La calificación más alta es " + mayor
      fin

```

Figura 22. Algoritmo ProyectoCalificaciones representado con pseudocódigo.

## 4. Codificación

Veamos algunos aspectos relevantes del código (ver Figura 23):

```
package proyectocalificaciones;
import java.util.Scanner;
/**
 * @author Guadalupe
 * Abril 2020
 */
public class ProyectoCalificaciones {

    public static void main(String[] args) {
        // --- Declaración de variables
        int numEst, cont;
        double calif[], suma, prom, mayor;

        // ----- Entrada
        System.out.print("¿Cuántos estudiantes hay en tu grupo? (Máximo 40)");
        Scanner entrada = new Scanner(System.in);
        numEst=entrada.nextByte();
        calif = new double[numEst];

        for (cont=0;cont<numEst;cont++)
        {
            System.out.printf("Escribe la calificación del alumno " + cont + ": ");
            calif[cont]=entrada.nextDouble();
        }

        // ----- Proceso
        suma=0.0;
        mayor = calif[0];
        for (cont=0;cont<numEst;cont++)
        {
            suma = suma + calif[cont];
            if (calif[cont] > mayor)
                mayor = calif[cont];
        }
        prom=suma/numEst;

        // ----- Salida
        System.out.printf("El promedio general del grupo es %5.2f \n", prom);
        System.out.printf("La calificación más alta es %5.2f \n", mayor);
    }
}
```

Figura 23. Código en Java del ProyectoCalificaciones.

- Como ya hemos mencionado antes la declaración y creación del vector también pudo haberse realizado en una sola instrucción.

**double calif[]= new double[numEst];**



- Observa que los índices del arreglo en el *lenguaje algorítmico* corren desde *1* y los ciclos están contando hasta *numEst*, pero en *Java* lo hacen hasta *numEst-1* pues inician en *0*. Este tipo de detalles pueden provocar errores de lógica o de compilación.
- Por último, me parece importante mencionar que la instrucción `System.out.print("¿Cuántos estudiantes hay en tu grupo? (Máximo 40)");`, se ha dejado tal como en el algoritmo, con el único propósito de mantener la congruencia. Pues en Java es innecesario limitar al usuario en cuanto al número de estudiantes, ya que en realidad el arreglo es creado a partir del dato que él mismo nos está proporcionando.

## ARREGLOS MULTIDIMENSIONALES

Un arreglo *multidimensional*, como su nombre lo indica, es aquel que ha sido declarado de *n* dimensiones, en donde  $n > 1$ . Si bien esto es cierto, en la práctica es común que a los arreglos de dos dimensiones se les denomine bidimensionales; a los de tres dimensiones, tridimensionales; y solo se llame multidimensional al que excede la 3ª dimensión.

En este curso abordaremos al arreglo de dos dimensiones. Mejor conocido como *arreglo bidimensional (matriz o tabla)*, este tipo de estructura aunque cuenta con una *organización física lineal*, tiene una *organización lógica no lineal*, pues la relación entre sus elementos se visualiza en *renglones y columnas*. Un ejemplo de ello serían los datos en una hoja de Excel o los valores de una matriz de matemáticas.

### Conceptos básicos

#### Renglón y columna

Como ya hemos mencionado, aun cuando en la memoria los datos están almacenados de manera secuencial, su *organización lógica* se da en dos dimensiones, como una *tabla* o *matriz*. A la dimensión horizontal se le denomina *renglón* o *fila* y a la vertical se le llama *columna*.

## Índice de renglón e índice de columna

Las celdas o casillas se distinguen unas de otras por dos índices, uno para indicar el *renglón* o *fila* en el que se encuentra y otro para la *columna*.

## Tamaño

En el caso de un arreglo bidimensional su *tamaño* implica definir la cantidad de renglones y la de columnas que tiene.

## Representación gráfica

Los *arreglos bidimensionales* pueden ser visualizados como una *tabla* o *matriz* (ver Figura 24).

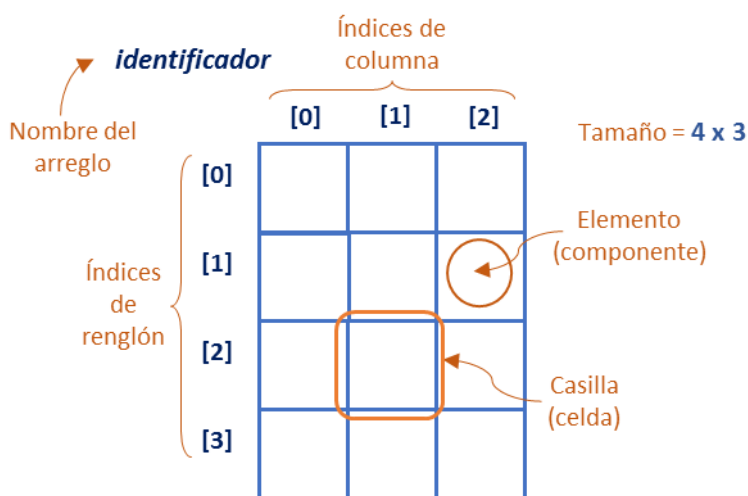


Figura 24. Representación gráfica de un arreglo bidimensional.

En un *arreglo bidimensional* los elementos *comparten* un mismo *identificador*, por tal razón si se requiere acceder a uno de ellos, es necesario especificar los índices de la celda en la que se encuentra, es decir, [*renglón*, *columna*]. Para que quede más claro, imagina que estás hablando por teléfono con un(a) compañero(a) de trabajo que te está dictando algunos valores que necesitas capturar en una hoja de Excel. Lo más lógico es que para cada dato te dé como referencia el renglón y la columna en la que deberás colocarlo.

Existen dos formas de declarar un *arreglo bidimensional*: vacío o con datos. En el primer caso, en *lenguaje algorítmico* solo se debe hacer su declaración, pero en *Java* es necesario declararlo y además crearlo mediante la invocación de la palabra reservada **new**. Por otro

lado, si se desea que el arreglo contenga datos desde un inicio, en ambos lenguajes solo se requiere su declaración (ver Tabla 26 y Tabla 27).

Es importante recordar que en los *lenguajes algorítmicos* generalmente los índices de las celdas del arreglo inician en 1, mientras que en *Java* siempre lo hacen en 0.

	Lenguaje algorítmico
<b>Sintaxis</b>	identificadorArreglo: <u>arreglo</u> [1..numRen, 1..numCol] <u>de</u> tipoDato
<b>Ejemplos</b>	matriz: <u>arreglo</u> [1..4, 1..3] <u>de entero</u>
	nombres: <u>arreglo</u> [][] <u>de cadena</u> = { {"Batman", "Superman"}, {"Robin", "Luisa Lane"}, {"El Guasón", "Lex Luthor"} }

Tabla 26. Sintaxis de declaración de un arreglo bidimensional en lenguaje algorítmico.

	Java
<b>Sintaxis</b>	tipoDato identificadorArreglo[][]; identificadorArreglo= <b>new</b> tipoDato[numRen] [numCol]; ó tipoDato identificadorArreglo[][]= <b>new</b> tipoDato[numRen] [numCol];
<b>Ejemplos</b>	<b>int</b> matriz[][]; matriz= <b>new int</b> [4] [3]; ó <b>int</b> matriz= <b>new int</b> [4] [3];
	<b>String</b> nombres [][]={ {"Batman", "Superman"}, {"Robin", "Luisa Lane"}, {"El Guasón", "Lex Luthor"} };

Tabla 27. Sintaxis de declaración de un arreglo bidimensional en Java.

## Operaciones

En un *arreglo bidimensional* se pueden realizar, las operaciones mencionadas a continuación (Joyanes Aguilar, Fundamentos de programación. Algoritmos, estructura de datos y objetos, 2008):

- Declaración y/o creación de la estructura.
- Asignación, lectura y escritura de elementos, recorrido (orden principal de renglón, orden principal de columna).
- Actualización: adición, inserción y eliminación de elementos.
- Ordenación y búsqueda

## Aplicaciones

Los *arreglos bidimensionales* pueden ser utilizados cuando en un programa se requiere guardar un conjunto de  $n$  elementos del mismo tipo y organizados como una tabla, en cuyo caso sería poco práctico declarar  $n$  variables independientes para almacenarlos en la memoria. En lugar de eso se declara una sola *estructura (arreglo)* que permita guardar y mantener disponibles los datos de manera ágil y segura. Por ejemplo para calcular el promedio de las calificaciones finales de mis 4 grupos, para registrar el promedio de ventas diarias de un establecimiento en los últimos 6 meses, para guardar la lista de los nombres de los invitados a 6 eventos, etc.

### Ejemplo de resolución de un problema utilizando un arreglo bidimensional

A continuación se plantea un *problema* al que le diseñaremos una solución mediante la *metodología* abordada en el Tema 1 de este curso.

#### 1. Definición del problema

Diseñe un programa que procese el total de ventas por día en las últimas “ $n$ ” semanas. A partir de los datos proporcionados por el usuario, se deberá mostrar el promedio por día de las “ $n$ ” semana (promedio vendido los lunes, los martes, etc.) y el promedio por semana.

#### 2. Análisis del problema

Entrada	Proceso	Salida
<ul style="list-style-type: none"> <li>Número semanas</li> <li>Ventas por día en las “<math>n</math>” semanas.</li> </ul>	<ul style="list-style-type: none"> <li>Calcular el promedio por semana.</li> <li>Calcular el promedio por día de las “<math>n</math>” semanas.</li> </ul>	<ul style="list-style-type: none"> <li>Promedios por semana.</li> <li>Promedios por día de las “<math>n</math>” semanas.</li> </ul>

Una vez que hemos identificado los aspectos generales anteriores, es necesario preguntarse si ello es suficiente para ser transcrito a un algoritmo. Si la respuesta es sí estamos listos para pasar a la etapa de Diseño de la solución, pero si la respuesta es no o no sé, habría que continuar con un análisis más detallado, el cual puede plasmarse en el formato anterior o según le sea más claro al analista:

- *Entrada y salida:* Considerar que aunque las ventas son un conjunto homogéneo de valores, su relación lógica no es lineal, pues se clasifican en semanas, es decir que tendremos 7 datos para la semana 1, 7 para la semana 2, y así sucesivamente. Y que

deberán ser recibidos y almacenados uno por uno, de manera que luego sean fáciles de procesar.

- **Proceso.** Es necesario visualizar que para calcular el promedio por semana y el promedio por día de las “n” semanas, se requiere *recorrer el arreglo* de 2 maneras distintas. Por lo tanto lo primero que hay que hacer es definir como se almacenarán los datos, por ejemplo, podemos decidir que las columnas serán los días y los renglones las semanas. De tal forma que para el primer caso calcularemos los promedios por renglón y para el segundo lo haremos por columna. Al igual que con los arreglos unidimensionales, esto con el tiempo será pan comido para ti, incluso te aprenderás de memoria los segmentos de código para hacerlo.

Recuerda que el producto que obtengas en esta etapa deberá ser un documento claro, ordenado y listo para servir de sustento en el diseño de la solución.

### 3. Diseño de la solución

Precisemos algunos detalles del pseudocódigo presentado en la *Figura 25*:

- En la sección *var* hemos declarado 8 variables:
  - *promedio* y *suma*, para calcular y mostrar la información solicitada en el problema.
  - *semanas* y el arreglo *ventas*, en los que se reciben el número de semanas a procesar y los valores las ventas de los 7 días de las “n” semanas, proporcionados por el usuario.
  - *dia*, utilizada como el número de columnas del arreglo.
  - *ren* y *col*, variables contador que harán las veces de índices del arreglo.
  - *diaSemana*, arreglo inicializado con los nombres de los días de la semana.
- En la *Entrada*:
  - Se inicializa la variable *dia* con el valor 7.
  - Para recorrer la matriz, se utilizan dos *ciclos desde anidados*, en donde el contador del ciclo exterior cuenta los renglones (*ren*) y el del interior las columnas (*col*). A esto se le llama recorrer el arreglo en *orden principal de renglón*. Para comprender su funcionamiento imagina que son un par de

engranes, siendo el más grande el que está en el exterior, es decir el que está contando los renglones. Mientras el mayor da una vuelta, el más pequeño, que está contando las columnas, gira varias veces.

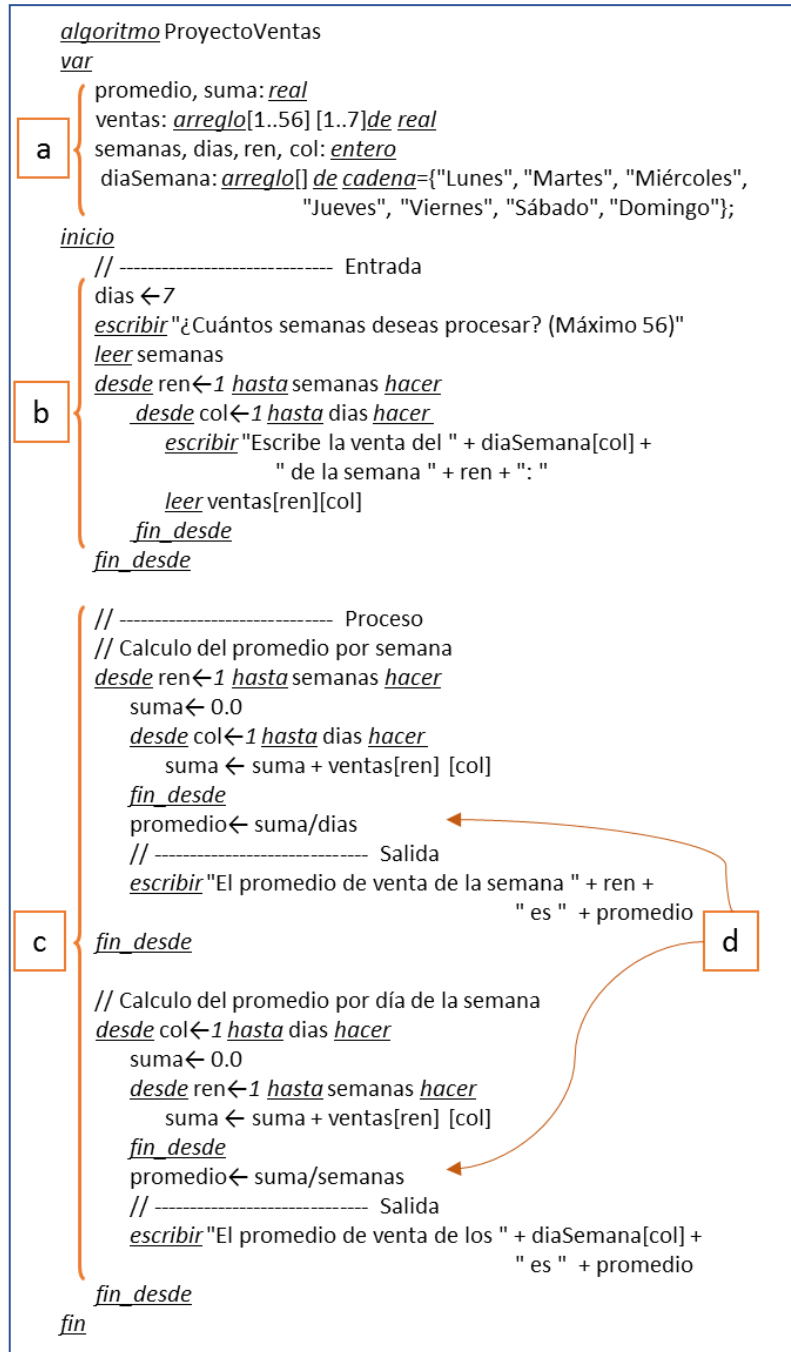


Figura 25. Algoritmo ProyectoVentas representado con pseudocódigo.

Regresando a nuestro ejemplo, cuando la ejecución del algoritmo llega al primer *desde*, el contador *ren* es inicializado en 1. Luego pasa el control a la siguiente

instrucción, segundo *desde*, lo que hace que *col* tome el valor de 1. Sin embargo como se trata de un ciclo este permanecerá ejecutándose e incrementando a *col* hasta que llegue al límite marcado por *dia*. Entonces, mientras *ren* vale 1, *col* tomará todos los valores *desde 1 hasta dia*, lo que significa que en cada vuelta del ciclo la instrucción *leer ventas[ren][col]*, estará guardando los valores en las celdas *ventas[1][1]*, luego en *ventas[1][2]*, en *ventas[1][3]*, en *ventas[1][4]*, en *ventas[1][5]*, en *ventas[1][6]*, hasta llegar a *ventas[1][7]*.

Una vez que el ciclo interior termina, regresa el control al ciclo exterior que incrementa la variable *ren* a 2, para luego dar paso a una nueva ejecución del ciclo interior, que esta vez estará almacenando valores en las celdas del segundo renglón: *ventas[2][1]*, *ventas[2][2]*, sucesivamente hasta *ventas[2][7]*. Todo lo anterior sucede hasta que el ciclo exterior haya terminado.

c. En el *Proceso*:

- Como podrás observar con los *primeros ciclos anidados* se está recorriendo el arreglo en *orden principal de renglón*, lo que permite ir fila por fila. Es importante visualizar que la sumatoria se lleva a cabo dentro del ciclo interior y que una vez que ha terminado, se calcula y muestra el promedio, para luego regresar el control al ciclo exterior.
- Los *segundos ciclos anidados*, recorren el arreglo en *orden principal de columna*, es decir, que el ciclo exterior controla las columnas y el interior las filas. Nota que solo los ciclos se invierten, pero los índices en el arreglo *ventas* se usan en el mismo orden *ventas[ren][col]*. Recuerda que siempre se le indica el renglón y después la columna.

Como te imaginarás, se usa el *orden principal de columna* pues lo que se requiere ahora, es calcular los promedios por día de la semana, es decir por columna. De tal forma que cuando *col* vale 1, se hace la sumatoria de *ventas[1][1]*, luego en *ventas[2][1]*, en *ventas[3][1]* hasta llegar a *ventas[semana][1]*, luego se calcula y muestra el resultado. Y así para todas las 7 columnas (los 7 días de la semana).

- d. La *Salida*. En este ejemplo, los resultados son mostrados al usuario durante el procesamiento de los datos, pues es cuando ha sido necesario hacerlo. Lo relevante de esto es que recuerdes que quien diseña decide en donde y como se usan las

instrucciones, según considere pertinente. La única regla es respetar la sintaxis del lenguaje y mantener un buen estilo de programación.

#### 4. Codificación

Veamos algunos aspectos relevantes del código anterior (ver Figura 26):

```
package proyectoventas;
import java.util.Scanner;
/**
 * @author Guadalupe
 * Abril 2020
 */
public class ProyectoVentas {

    public static void main(String[] args) {
        // --- Declaración de variables
        double promedio, suma;
        double ventas[][];
        int semanas, dias=7;
        String diaSemana[]={"Lunes", "Martes", "Miércoles", "Jueves",
                           "Viernes", "Sábado", "Domingo"};

        // ----- Entrada
        System.out.print("¿Cuántos semanas deseas procesar?");
        Scanner entrada = new Scanner(System.in);
        semanas=entrada.nextByte();
        ventas = new double[semanas][dias];

        for (int ren=0; ren<semanas; ren++)
            for (int col=0; col<dias; col++)
            {
                System.out.printf("Escribe la venta del " + diaSemana[col] +
                                   " de la semana " + (ren+1) + ": ");
                ventas[ren][col]=entrada.nextDouble();
            }

        // ----- Proceso
        // Calculo del promedio por semana
        for (int ren=0; ren<semanas; ren++)
        {
            suma=0.0;
            for (int col=0; col<dias; col++)
                suma = suma + ventas[ren][col];
            promedio= suma/dias;
            // ----- Salida
            System.out.printf("El promedio de venta de la semana %d es %5.2f \n",
                              (ren+1), promedio);
        }

        // Calculo del promedio por día de la semana
        for (int col=0; col<dias; col++)
        {
            suma=0.0;
            for (int ren=0; ren<semanas; ren++)
                suma = suma + ventas[ren][col];
            promedio= suma/semanas;
            // ----- Salida
            System.out.printf("El promedio de venta de los %s es %5.2f \n",
                              diaSemana[col], promedio);
        }
    }
}
```

Figura 26. Código en Java del ProyectoVentas.



- Como ya hemos mencionado antes la declaración y creación de la matriz también pudo haberse realizado en una sola instrucción.

**double** ventas[][]= **new double**[semanas][dias];

- Observa que los índices del arreglo en el *lenguaje algorítmico* corren desde **1** y los ciclos cuentan hasta *semanas* y *dias*, pero en *Java* lo hacen hasta *semanas-1* y *dias-1* pues inician en **0**. Este tipo de detalles pueden provocar errores de lógica o de compilación.
- Por último, observa que en el programa no le hacemos la indicación al usuario de “*máximo 56 semanas*”. Como ya habíamos explicado, en *Java* es innecesario pues en realidad el arreglo es creado a partir del dato que él mismo nos está proporcionando.

## ***ESTRUCTURAS O REGISTROS***

Un *registro* es una estructura de datos heterogénea, de acceso directo y estática. Denominados como *estructura* en algunos lenguajes de programación, se trata de un conjunto de datos relacionados entre sí, considerado:

- **Heterogéneo:** Los datos que lo conforman pueden ser de tipo distinto. Por ejemplo puedo guardar los datos de una persona: nombre, apellidos, edad, sexo y dirección.
- **Acceso directo:** Se puede visitar directamente a cualquiera de sus elementos individuales en el momento deseado. Imagina que en un solo cajón tienes guardados diferentes tipos de objetos, organizados en cajitas. Por ejemplo, calcetines, corbatas y cintos. En cualquier momento puedes tomar, agregar o cambiar un objeto sin necesidad de recorrer todo el cajón.
- **Estático:** Una vez declarados los elementos que conforman la estructura no se puede modificar el tamaño. Considera que en el ejemplo anterior las cajas que elegiste ocupan exactamente el espacio total del cajón, así que no puedes agregar ninguna más. Esto se debe a que el tamaño del cajón es *estático*.

El lenguaje Java no contempla el uso de registros, pues en su lugar utiliza la declaración de clases, tema que verás a mayor profundidad en la siguiente asignatura denominada *Programación Orientada a Objetos*.

## Conceptos básicos

### Campo

A cada espacio del registro en el que se almacena un dato o valor se le denomina *campo*.

### Representación gráfica

Los *registros* pueden ser visualizados de la siguiente manera (ver Figura 27).

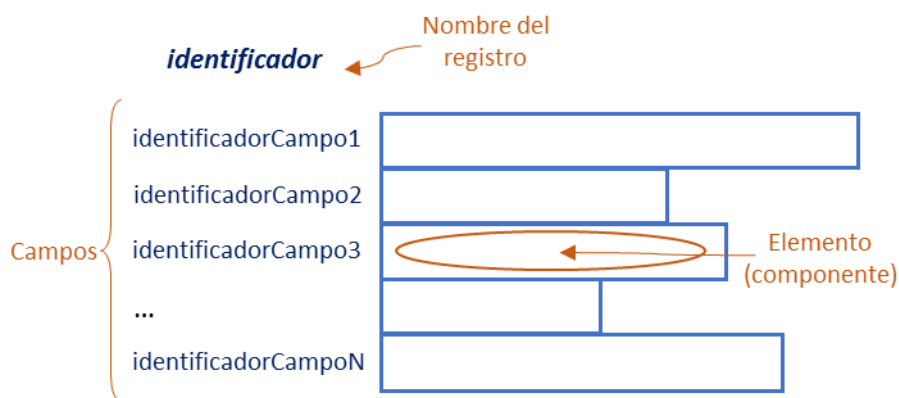


Figura 27. Representación gráfica de un registro.

En un *registro* todos los elementos *comparten* un mismo *identificador*, por tal razón si se requiere acceder a uno de ellos, es necesario especificar el nombre del *campo* correspondiente. Para que quede más claro, imagina que estás haciendo una tarea y requieres investigar la información de 5 libros. Lo más lógico es que primeramente identifiques cuales datos de un libro son los más relevantes, es decir, los que lo representan de manera única, para luego empezar a anotar dicha información para cada libro que elijas.

Cabe mencionar que los campos de un registro pueden ser de cualquier tipo, incluyendo de arreglo. A continuación te mostramos la sintaxis de declaración que usaremos para los *registros*, en *lenguaje algorítmico* y en *Java* (ver Tabla 28).

	Lenguaje algorítmico	Java
<b>Sintaxis</b>	identificadorRegistro: <u>registro</u> identificadorCampo1: tipoDato identificadorCampo2: tipoDato identificadorCampo3: tipoDato ... identificadorCampoN: tipoDato <u>fin registro</u>	modificador_de_acceso <b>class</b> identClase{ tipoDato identificadorAtributo1; tipoDato identificadorAtributo2; tipoDato identificadorAtributo3; ... tipoDato identificadorAtributoN; }
<b>Ejemplo</b>	Libro: <u>registro</u> titulo: <u>cadena</u> autor: <u>cadena</u> numPags: <u>entero</u> <u>fin registro</u>	modificador_de_acceso <b>class</b> Libro{ <b>String</b> titulo; <b>String</b> autor; <b>int</b> numPags; }

Tabla 28. Sintaxis de declaración de un registro.

## Operaciones

En un *registro* se pueden realizar, las operaciones mencionadas a continuación:

- Declaración y/o creación de la estructura.
- Asignación, lectura y escritura de elementos.

## Aplicaciones

Los *registros* pueden ser utilizados cuando en un programa se requiere guardar un conjunto de elementos de distinto tipo, relacionados entre sí porque describen a un objeto o entidad. En cuyo caso sería poco práctico declarar variables independientes para almacenarlos en la memoria. En lugar de eso se declara una sola *estructura (registro)* que permita guardar y mantener disponibles los datos de manera ágil y segura.

### Ejemplo de resolución de un problema utilizando un registro

A continuación se plantea un *problema* al que le diseñaremos una solución mediante la *metodología* abordada en el Tema 1 de este curso.

#### 1. Definición del problema

Diseñe un programa que lea los datos de los estudiantes del grupo de Kundalini Yoga y los ordene alfabéticamente.

## 2. Análisis del problema

Entrada	Proceso	Salida
<ul style="list-style-type: none"> <li>• Número estudiantes.</li> <li>• Datos de los estudiantes:               <ul style="list-style-type: none"> <li>- nombre</li> <li>- apellido</li> <li>- edad</li> <li>- sexo</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Ordenar los datos de los estudiantes.</li> </ul>	<ul style="list-style-type: none"> <li>• Datos de los estudiantes ordenados.</li> </ul>

Una vez que hemos identificado los aspectos generales anteriores, es necesario preguntarse si ello es suficiente para ser transcrito a un algoritmo. Si la respuesta es sí estamos listos para pasar a la etapa de Diseño de la solución, pero si la respuesta es no o no sé, habría que continuar con un análisis más detallado, el cual puede plasmarse en el formato anterior o según le sea más claro al analista:

- **Entrada y salida:** Podemos ir teniendo en mente los tipos de datos necesarios para guardar la información que hemos identificado. Además es muy importante considerar que aunque son heterogéneos, vamos a necesitar guardar esa información para N estudiantes, recibirlos y almacenarlos de manera que luego sean fáciles de procesar. En otras palabras, usaremos un arreglo unidimensional de registros.
- **Proceso.** Para ordenar los datos en un *arreglo* existen varios métodos, por lo que desde este punto se puede ir investigando y determinando cual se utilizará. En nuestro caso usaremos el más sencillo, denominado método Burbuja.

Recuerda que el producto que obtengas en esta etapa deberá ser un documento claro, ordenado y listo para servir de sustento en el diseño de la solución.

## 3. Diseño de la solución

Precisemos algunos detalles del pseudocódigo presentado en la *Figura 28*:

- En la sección *var* hemos declarado 5 variables y antes de ella un *registro*:
  - Antes de iniciar la declaración de variables se encuentra declarada la estructura del *registro Estudiante* requerido para guardar los datos de un estudiante.
  - *numEst*, para recibir la cantidad de estudiantes de los que se leerán y ordenarán los datos.

- $i$  y  $j$ , variables contador que ayudarán en el método Burbuja para recorrer el arreglo unidimensional y ordenar los datos.
  - *est*, arreglo unidimensional en el que se almacenarán los datos. Nota que está declarado de tipo *Estudiante*, es decir, que cada una de sus celdas guardará un registro de este tipo.
  - *aux*, utilizada como una variable auxiliar que permitirá hacer intercambios de datos entre las celdas del arreglo. Esto con el propósito de ordenarlo.
- b. En la *Entrada*:
- En la variable *numEst* se guarda el número de estudiantes indicados por el usuario.
  - Para leer los datos de los estudiantes se usa un *ciclo desde* cuya variable  $i$  cuenta de 1 hasta *numEst*. El propósito es recorrer el arreglo celda por celda e ir guardando los datos.
- c. En el *Proceso*.
- El método *Burbuja* consta de dos *ciclos desde anidados*: uno exterior que incrementa la variable  $i$  de 1 a *numEst*; y otro interior con  $j$  que va de 1 hasta *numEst-1*.
  - Dentro del *ciclo desde interior* la única instrucción existente es una *condición simple*. ¡Si es la única! Y si lo que estás pensando es que ahí hay más de una instrucción, la respuesta es NO. Es decir, que el *ciclo desde exterior* contiene un *ciclo desde interior*. Dentro del ciclo interior hay una instrucción *si*, y ella controla a tres *instrucciones de asignación*.
- Veamos que hace el código, de adentro hacia afuera:
- ✓ La condición solo se ejecuta *si* *est*[ $j$ ].apellido > *est*[ $j+1$ ].apellido, es decir que cuando  $j$  vale 1, compara a *est*[1].apellido > *est*[2].apellido, cuando  $j$  vale 2 compara a *est*[2].apellido > *est*[3].apellido, y así sucesivamente. Esa es la razón de que  $j$  cuente solo hasta *numEst*, pues siempre coteja a la celda  $j$  con la  $j+1$ . Si llegara hasta *numEst*, al final intentaría comparar al último elemento con un valor que no existe. Lo que obviamente causaría un error de ejecución.

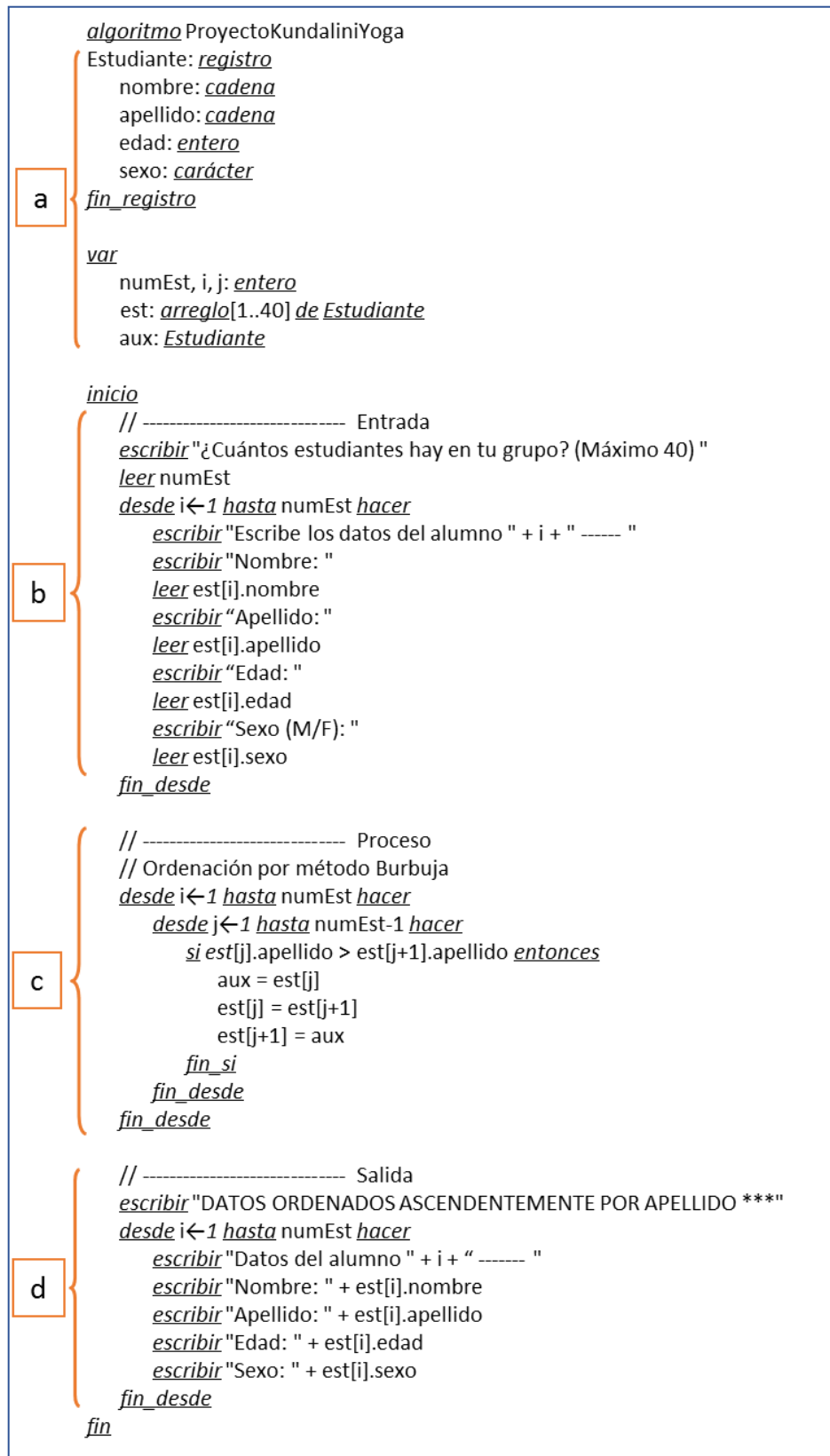


Figura 28. Algoritmo ProyectoKundaliniYoga representado con pseudocódigo.

- ✓ Por tanto, la función que tiene el *ciclo interior* es ayudar a recorrer el arreglo comparando al elemento  $j$  con el elemento  $j+1$ , e intercambiar los contenidos de las celdas cuando encuentre que el primero es mayor que el segundo, porque desea que los valores pequeños *burbujeen* hacia el inicio del arreglo y los grandes se vayan al final. De ahí el nombre del *Método Burbuja*. Jajaj ya sé que suena raro, pero así le gustó a quien lo diseñó.
- ✓ El *ciclo exterior* lo que hace es contar desde 1 hasta *numElem*, pero si observas la variable  $i$  no se usa en el arreglo. Su función es solo contar para asegurarse que el método recorra o pase ese número de veces por el arreglo. En la primera pasada, moverá el valor más grande que encuentre hasta el final del arreglo, en la segunda moverá el segundo más grande y así hasta que el arreglo queda en orden.

d. La *Salida*.

- Se indica al usuario que a continuación se mostrarán los datos de los estudiantes ordenados ascendentemente.
- Con un *ciclo desde* se recorre el *vector* celda por celda, mostrando los datos de los estudiantes.

#### 4. Codificación

Veamos algunos aspectos relevantes del código (ver Figura 29):

- Como ya hemos mencionado, Java no utiliza registros, por tal razón en su lugar el código inicia con la declaración de una clase privada (solo se puede usar en la aplicación ProyectoKundaliniYoga).
- De nuevo se puede observar que el índice  $j$  en el *lenguaje algorítmico* cuenta desde 1 hasta *numEst*; y en *Java* lo hace hasta *numEst-1* pues inicia en 0.
- 

*¿Ser o parecer?...eh ahí la cuestión*

El reconocimiento no se grita, ni se publica y mucho menos se exige. Se siente y se recibe cuando escuchas una voz que con profunda admiración dice maravillas de ti.  
El secreto está en Ser, no en parecer.

```

package proyectokundaliniyoga;
import java.util.Scanner;
/**
 * @author Guadalupe
 * Abril 2020
 */
public class ProyectoKundaliniYoga {
    // Declaración de la clase que funcionará como un registro
    private static class Estudiante{
        String nombre;
        String apellido;
        byte edad;
        char sexo;
    }

    public static void main(String[] args) {
        // --- Declaración de variables
        int numEst, i, j;
        Estudiante est[], aux;

        // ----- Entrada
        System.out.print("¿Cuántos estudiantes hay en tu grupo?");
        Scanner entrada = new Scanner(System.in);
        numEst= Byte.parseByte(entrada.nextLine());
        est = new Estudiante[numEst];

        for (i=0; i<numEst; i++)
        {
            est[i]= new Estudiante();
            System.out.printf("\nEscribe los datos del alumno " + (i+1) + " ----- \n");
            System.out.printf("Nombre: ");
            est[i].nombre= entrada.nextLine();
            System.out.printf("Apellido: ");
            est[i].apellido= entrada.nextLine();
            System.out.printf("Edad: ");
            est[i].edad= Byte.parseByte(entrada.nextLine());
            System.out.printf("Sexo (M/F): ");
            est[i].sexo=entrada.nextLine().charAt(0);
        }

        // ----- Proceso
        // Ordenación por método Burbuja
        for (i=0; i<numEst; i++)
            for (j=0; j<numEst-1; j++)
                if ( est[j].apellido.compareTo(est[j+1].apellido)>0 )
                {
                    aux= est[j];
                    est[j]= est[j+1];
                    est[j+1]= aux;
                }

        // ----- Salida
        System.out.printf("\nDATOS ORDENADOS ASCENDENTEMENTE POR APELLIDO *** \n");
        for (i=0; i<numEst; i++)
        {
            System.out.printf("\nDatos del alumno " + (i+1) + " ----- \n");
            System.out.printf("Nombre: " + est[i].nombre + " \n");
            System.out.printf("Apellido: " + est[i].apellido + " \n");
            System.out.printf("Edad: " + est[i].edad + " \n");
            System.out.printf("Sexo: " + est[i].sexo + " \n");
        }
    }
}

```

Figura 29. Código en Java del ProyectoKundaliniYoga.



# TEMA 5. MODULARIDAD

Al iniciar el estudio del *desarrollo de software* comúnmente los ejemplos abordados son sumamente sencillos, resolviendo problemas de muy fácil análisis y cuyo diseño de la solución es bastante intuitivo. Incluso cuando vamos adquiriendo experiencia es común que las soluciones a ese tipo de problemas aparezcan en nuestra mente como por arte de magia. Sin embargo la idea es avanzar, adquiriendo conocimientos, desarrollando habilidades, aptitudes y actitudes; mientras nos entrenamos para resolver situaciones complejas a través del planteamiento de soluciones computacionales. Y es aquí donde entra el concepto de *modularidad*, que podemos definir como la propiedad que tiene el *software*, de ser divididos en subproblemas más fáciles de analizar, diseñar, codificar, documentar y mantener.

## Paradigmas de programación

Como todo lo que le es útil al ser humano, los paradigmas de programación han evolucionado con el tiempo. Veamos una pequeña remembranza que nos permita ubicar nuestro tema.

### Programación no estructurada o de espagueti

La programación no tenía el orden que ahora conocemos. Los códigos resultantes tendían a ser confusos e incluso en ocasiones un programador prefería escribir un nuevo código antes que intentar comprender lo que había hecho tiempo atrás:

- Los programas no contaban con mucha documentación y tampoco con una cabecera.
- Las instrucciones se escribían en líneas numeradas sin indentación y sin líneas en blanco.
- No existía sección de declaración de variables, pues las variables eran utilizadas en el lugar en el que se requirieran sin previa definición.

- Era común la utilización de las instrucciones *goto* y *exit*, que forzaban brincos en la secuencia de ejecución de instrucciones y/o el quiebre del programa en cualquier momento.

### Programación Estructurada y Modular

De la necesidad de facilitar el desarrollo de software, la reutilización de código y la mejora del estilo de programación, llega de la mano de la *programación estructurada* que pone orden al desorden. En pocas palabras, el *paradigma estructurado* expresa que a cualquier problema que se pueda resolver mediante el uso de una computadora, puede modelársele una solución utilizando solamente 3 tipos de instrucciones: *secuenciales*, *condicionales* y *cíclicas*. Prohíbe el uso de la instrucción *goto* y el *exit* para quebrar el programa, asevera que un buen diseño no lo necesita, ya que las instrucciones antes mencionadas llevarán al sistema desde el inicio hasta el final sin ningún problema. Algunas de las recomendaciones de la este *paradigma* han sido enlistadas en el apartado *Estilo de programación* del Tema 2.

Adicionalmente y como complemento del *paradigma estructurado*, surge la *Programación Modular* que invita al programador a subdividir el problema tantas veces como sea necesario para que el desarrollo del software sea más sencillo y efectivo, facilitando incluso que ello sea realizado por un equipo de programadores. Como se mencionó en el Tema 1, el lema de la *Programación Modular* es “*Divide y vencerás*”, y explica que cada *módulo* podrá ser analizado, diseñado, codificado de forma independiente al resto de los módulos del sistema. Facilitando también, la documentación y mantenimiento del mismo.

Un *módulo* deberá ser tan general en su funcionamiento y tan específico en su propósito que su reutilización se facilite en futuras aplicaciones. Así mismo, es recomendable que el número de líneas de código que incluya sea como máximo entre 25 a y 30, es decir, que de preferencia se vea sin necesidad de hacer scroll a la pantalla.

### Programación Orientada a objetos

El *paradigma orientado a objetos* toma como base a la *programación Estructurada y Modular* para dar el siguiente salto, el desarrollo de software con un enfoque más cercano a la concepción humana, que mira a una aplicación como un conjunto de objetos interactuando. Pero este será el tema central de tu siguiente asignatura, por ahora continuemos con modularidad.

## DECLARACIÓN Y USO DE MÓDULOS

Dependiendo del lenguaje de programación y del paradigma; los módulos que integran al sistema, han sido denominados de distintas formas: subprograma, subrutina, procedimiento, función, método, utilería, unidad y clase.

Para el propósito de nuestra asignatura, baste decir que un *método* permite implementar una funcionalidad específica de una aplicación.

## Declaración de un módulo

En este curso utilizaremos el término *módulo* para referirnos a lo que en Java se denomina como *función* o *método*, el cual para ser utilizado requiere:

1. **Declaración.** Para que un método pueda ser reconocido por un programa es necesario declararlo, indicando su identificador, *parámetros formales* y código (ver Tabla 29 y Tabla 30).

Lenguaje algorítmico		
	Pseudocódigo	Diagrama de flujo
<b>Sintaxis</b>	<p><u>funcion</u> identificadorMetodo( parametrosFormales ):</p> <p style="text-align: right;">tipoResultado</p> <p>    //sección de declaración de variables locales</p> <p><u>inicio</u></p> <p>    //cuerpo del método o sección del código</p> <p>    <u>regresar</u> resultado //solo si se definió tipo de resultado</p> <p><u>fin funcion</u></p>	<p>El diagrama de flujo para un módulo se dibuja como cualquier otro (ver Figura 3), solo se intercambia el símbolo de inicio por el de subprograma (subrutina):</p> <div style="border: 1px solid black; padding: 10px; margin: 10px auto; width: fit-content;"> <div style="border: 1px solid black; padding: 5px; text-align: center;"> IdentificadorMetodo (parametrosFormales) </div> </div>

Tabla 29. Sintaxis de declaración de una función en lenguaje algorítmico.

	Java
Sintaxis	<pre> <u>modificadorAcceso</u> <u>tipoValorRetorno</u> identificadorMetodo( parametrosFormales ){     //sección de declaración de variables locales      //cuerpo del método o sección del código      <u>return</u> resultado; //solo si se definió tipo de valor de retorno } </pre>

*Tabla 30. Sintaxis de declaración de una función(método) en Java.*

2. *Invocación*. La instrucción para mandar ejecutar a un método es denominada comúnmente como *invocación* o *llamada al método*, la cual consiste en anotar el identificador del método seguido por los *parámetros actuales* entre paréntesis.

## ***PASO DE PARÁMETROS O ARGUMENTOS***

Algunos programadores usan los términos *parámetro* y *argumento* como sinónimo, pero la realidad es que no lo son. Los *parámetros formales* son las variables de entrada indicadas en la declaración de un método; mientras que los *parámetros actuales* o *argumentos* son las expresiones (variables, constantes, fórmulas, etc.), pasadas al método en la llamada.

## ***IMPLEMENTACIÓN***

### **Ejemplo de resolución de un problema utilizando un método**

A continuación se plantea un *problema* al que le diseñaremos una solución mediante la *metodología* abordada en el Tema 1 de este curso.

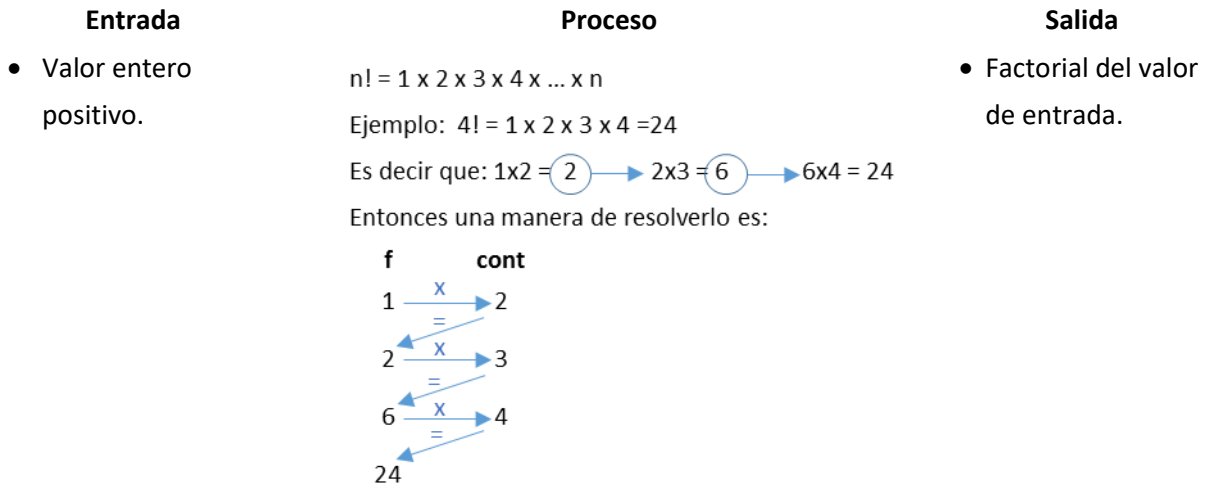
#### **1. Definición del problema**

Diseñe un programa que reciba un valor entero positivo y obtenga su factorial.

#### **2. Análisis del problema**

Entrada	Proceso	Salida
<ul style="list-style-type: none"><li>• Valor entero positivo.</li></ul>	<ul style="list-style-type: none"><li>• Calcular el factorial.</li></ul>	<ul style="list-style-type: none"><li>• Factorial del valor de entrada.</li></ul>

Una vez que quedan claros la entrada, proceso y salida del programa, la pregunta es si hay algo más que requiera ser analizado. En este caso, el cálculo del factorial es un procedimiento muy específico ya definido en matemáticas, pero que no es incluido en las funciones disponibles en el lenguaje. Por lo tanto, es necesario analizar como obtendremos dicho resultado:

*Función factorial*

- **Entrada y salida:** Serán fáciles de implementar pues se trata solo de recibir un valor entero y de mostrar como resultado la factorial.
- **Proceso.** Como hemos visualizado el cálculo del factorial requiere de dos variables, una que cuente de 2 hasta  $n$  y otra que acumule el factorial. Además es importante considerar que solo se puede calcular el factorial de un número igual o mayor que 0 y que el factorial de 0 por definición es 1.

Recuerda que el producto que obtengas en esta etapa deberá ser un documento claro, ordenado y listo para servir de sustento en el diseño de la solución.

### 3. Diseño de la solución

Precisemos algunos detalles del pseudocódigo presentado en la *Figura 30*:

- En la sección *var* hemos declarado solo 2 variables enteras, ya que solo necesitamos una para recibir el valor de entrada y otra para el resultado.
- En la *Entrada*.
  - Además del mensaje para solicitar el *valor de entrada*, el programa le indica al usuario que valor usar para terminar su ejecución.
  - El programa le está brindando la oportunidad al usuario de que calcule el factorial de más de un valor, por tal razón la lectura se hace en dos momentos, una antes del ciclo y antes de que termine.
- En el *Proceso*. Se ha simplificado con la implementación del módulo, de tal forma

que solo contiene un *ciclo* que controla la lectura de los valores, con una *condición de permanencia* que se hará falsa cuando reciba un dato menor que 0. El ciclo controla la llamada a la función factorial, la salida del resultado y la lectura del nuevo valor, quedando la *salida* implícita en este segmento de código. La invocación de la función lleva la variable valor, como *parámetro actual o argumento*.

d. La *función factorial*.

- Recibe como dato de entrada, *parámetro formal*, a la variable n de tipo entero y se ha declarado que regresará un resultado de ese mismo tipo.
- Se declaran dos variables enteras para el cálculo del factorial, el acumulador *f* y el contador *cont*.
- El ciclo *desde* cuenta desde 2 hasta n, para que en cada *iteración* la f vaya acumulando el producto de *f\*cont*.
- Finalmente *regresa* a *f* como resultado del módulo.

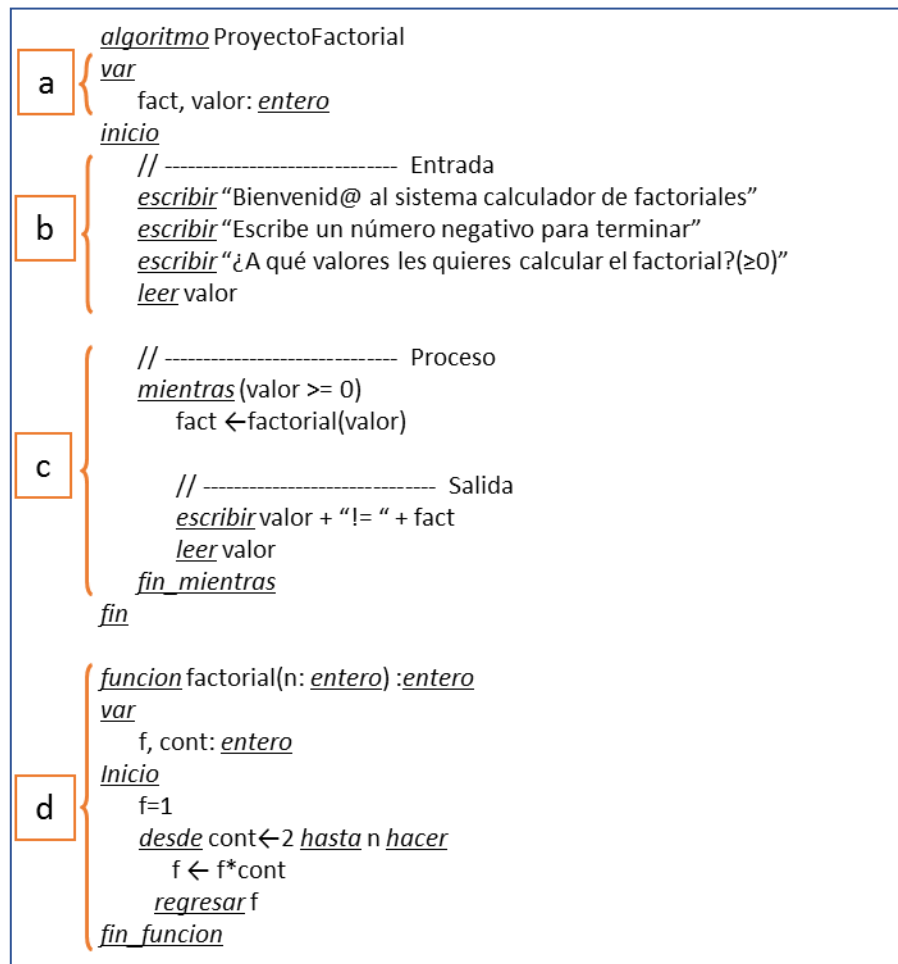


Figura 30. Algoritmo ProyectoFactorial representado con pseudocódigo.

## 4. Codificación

```

package proyectofactorial;
import java.util.Scanner;
/**
 * @author Guadalupe
 * Mayo 2020
 */
public class ProyectoFactorial {

    private static long factorial(byte n){
        long f=1;

        for(byte cont=2; cont<=n; cont++ )
            f = f*cont;
        return f;
    }

    public static void main(String[] args) {
        long fact;
        byte valor;

        // ----- Entrada
        System.out.print("Bienvenid@ al sistema calculador de factoriales");
        System.out.print("\nEscribe un número negativo para terminar");
        System.out.print("\n¿A qué valores les quieres calcular el factorial?(≥0)");
        Scanner entrada = new Scanner(System.in);
        valor=entrada.nextByte();

        // ----- Proceso
        while (valor >= 0)
        {
            fact=factorial(valor);

            // ----- Salida
            System.out.printf( "%d! = %d \n", valor, fact);
            valor=entrada.nextByte();
        }
    }
}

```

Figura 31. Código en Java del ProyectoFactorial.

Veamos algunos aspectos relevantes del código anterior (ver Figura 31):

- En Java el método factorial, ha debido declararse dentro de las llaves del proyecto, pues fuera de ellas no se puede escribir ningún tipo de código. Por lo demás la transcripción del algoritmo es casi directa.
- Nota que se utilizaron dos tipos de enteros distintos para el valor de entrada y la variable de salida, pues al tratarse de un cálculo que involucra la acumulación del producto, es necesario cuidar que el número recibido no sea tan grande y que la

variable fact sea de un tipo con un rango mayor. Por la misma razón, el contador y el acumulador de la función coinciden con dichos tipos de dato.

*Desetiquetate*

Las etiquetas son adjetivos que adoptaste porque alguien más te dijo que eso te identifica. Generalmente son simples juicios que sin importar si tuvieron una buena intención, pesan como piedras. Por ello, si de alguna forma crees que te motivan a mejorar, hazlo. Pero por favor despégate las etiquetas.

Hemos llegado al final de nuestro curso, pero tu camino apenas comienza en esta aventura profesional. Deseo que conviertas tu experiencia en algo maravilloso y divertido. Seguro no será perfecta, ¿Pero qué lo es? Si bien no todo lo que vivas te gustará tal como es, si no está en tus manos, mira lo que puedes cambiar y los demás déjalo fluir. Trabaja en ti, elije no aferrarte, suelta lo que te pesa, libérate de esas creencias y juicios que te impiden avanzar.

Respeta y da lo que te gustaría recibir. En tus manos está la elección de mirar lo que es y asentir ante la vida. Tal vez en algunos momentos no será fácil, pero posible si será. Piensa que todo lo que has vivido te trajo aquí, a lo que eres ahora, al momento en que puedes tomar tu vida en tus manos.

Mira menos hacia afuera, porque ver al otro te quita el tiempo y la atención para verte a ti, para conocerte. Eres lo más valioso que tienes, lo que nadie te podrá quitar. Tu mayor fortaleza será creer en ti, mirar lo que hay y amarte tal como eres, perfectible. Cambia lo que quieras y puedas, da tu mayor esfuerzo... y si un día cometes una equivocación o simplemente necesitas un respiro, detente, tienes derecho. Date el espacio, discúlpate, corrige, descansa, calla, has lo que sea necesario ¡se vale! pero luego ¡Avanza, siempre avanza! Enfoca bien tus sueños, ponlos en tu mira y ve por ellos.

Y como un gran maestro me dijo un día, el muy conocido y querido Ing. Blanco “estudie mijo(a), estudie, saber le dará seguridad.”