USERS

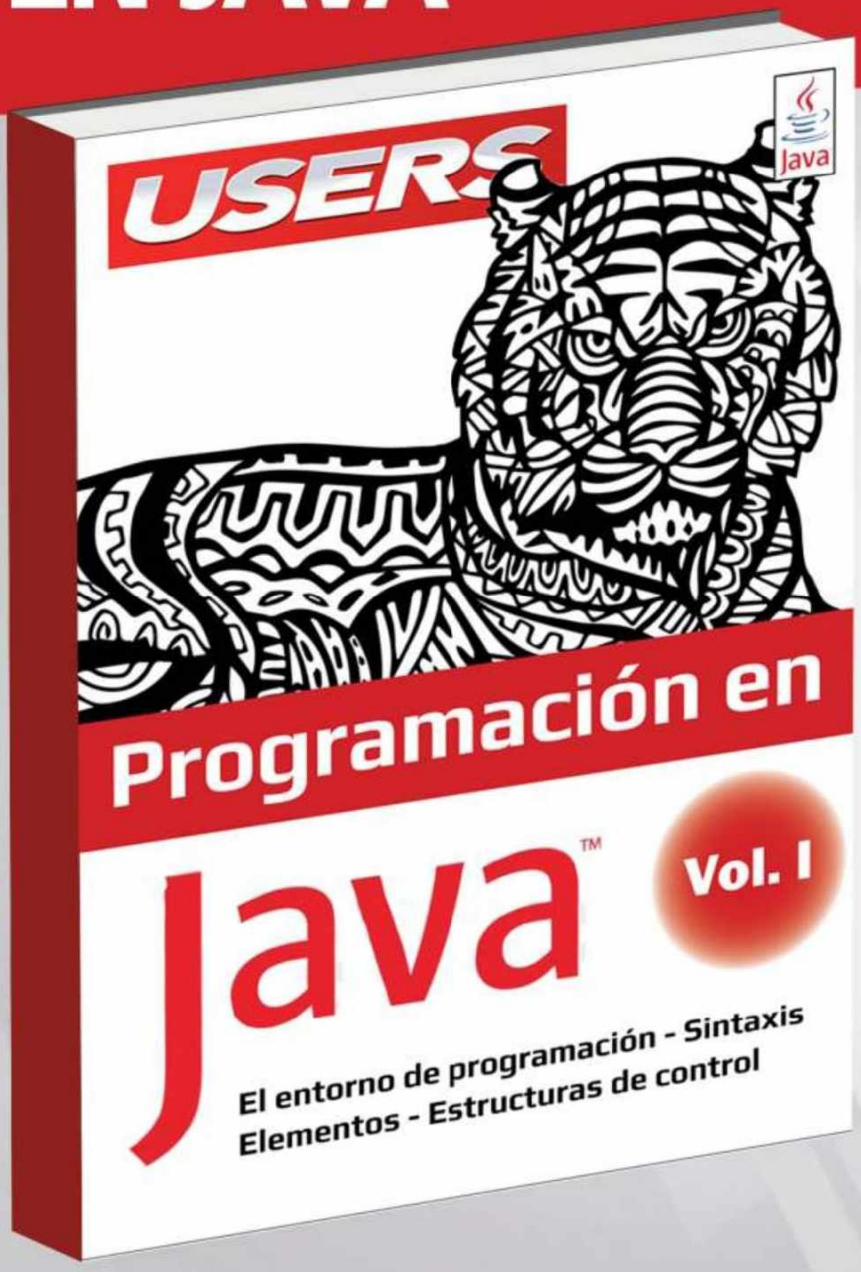
Programación en Java™

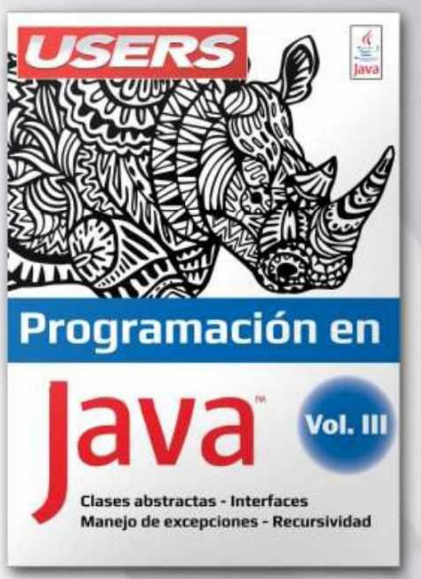
Vol. I

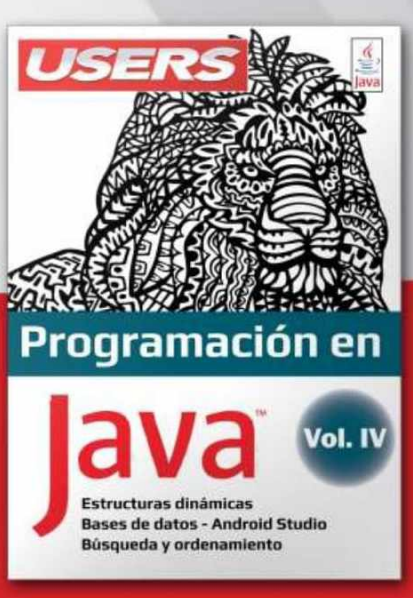
El entorno de programación - Sintaxis Elementos - Estructuras de control

CURSO DE

PROGRAMACION EN JAVA







Aprende a programar aplicaciones robustas y confiables. Escribe tus códigos una vez y ejecútalos en cualquier dispositivo.

Programación en Java™ Vol. I

Título: Programación en Java / **Autor**: Carlos Arroyo Díaz
Coordinador editorial: Miguel Lederkremer / **Edición**: Claudio Peña
Maquetado: Marina Mozzetti / **Colección**: USERS ebooks - LPCU287

ACERCA DE ESTE CURSO

Java es un lenguaje de programación que sigue afianzándose como un estándar de la web y, por eso, año tras año, aparece en el tope de las búsquedas laborales de programadores.

Es por esto que hemos creado este curso de **Programación en Java**, donde encontrarán todo lo necesario para iniciarse o profundizar sus conocimientos en este lenguaje de programación.

El curso está organizado en cuatro volúmenes, orientados tanto a quien recién se inicia en este lenguaje, como a quien ya está involucrado y enamorado de Java.

En el primer volumen se realiza una revisión de las características de este lenguaje, también se entregan las indicaciones para instalar el entorno de desarrollo y, posteriormente, se analizan los elementos básicos de la sintaxis y el uso básico de las estructuras de control.

En el segundo volumen se presentan las clases en Java, se realiza una introducción a los conceptos asociados a la Programación Orientada a Objetos y también se profundiza en el uso de la herencia, colaboración entre clases y polimorfismo.

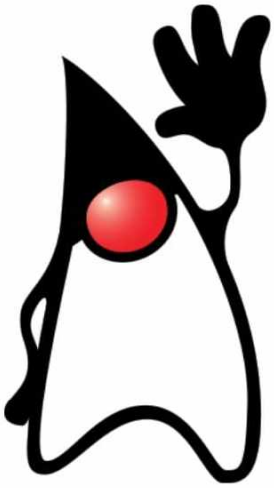
El tercer volumen contiene información sobre el uso de las clases abstractas e interfaces, el manejo de excepciones y la recursividad.

Finalmente, en el cuarto volumen se enseña el uso de las estructuras de datos dinámicas, el acceso a bases de datos y la programación Java para Android.

Sabemos que aprender todo lo necesario para programar en Java en tan solo cuatro volúmenes es un tremendo desafío, pero conforme vamos avanzando, el camino se va allanando y las ideas se tornan más claras.

¡Suerte en el aprendizaje!

SUMARIO DEL **VOLUMEN I**



PRÓLOGO

Java es un lenguaje maduro y robusto que, desde su nacimiento en el año 1995, ha demostrado que vino para quedarse y ha logrado evolucionar hasta convertirse en el lenguaje más utilizado en el mundo tecnológico.

Durante nuestro aprendizaje de cualquier lenguaje de programación podemos encontrarnos con variados libros, pero solo a través de la experimentación y escudriñando las entrañas de la Web (foros, blogs, under sites), podremos obtener los conocimientos necesarios para enfrentar el aprendizaje de mejor forma.

Mientras recorremos este camino, contar con un curso como este es fundamental pues se presenta como un compendio de todo lo que necesitamos para enfrentar un lenguaje de programación, permitiéndonos lograr, a través de ejemplos concretos, un mejor aprendizaje de los distintos temas necesarios para programar en Java.

Desde el primer momento somos optimistas en que este curso será para ustedes un desafío muy significativo y, por otro lado, logrará convertirse en una excelente guía del lenguaje Java.

Carlos Arroyo Díaz

Acerca del autor

Carlos Arroyo Díaz es programador de profesión, escritor especializado en tecnologías y docente por vocación. Se desempeña desde hace más de 20 años como docente en Informática General y en los últimos 10 años enseña programación en Java y Desarrollo Web. También se ha desempeñado como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires, enseñando a programar a grupos de jóvenes.

Introducción

¿Por qué un lenguaje de programación tiene tanta importancia en la informática? Todo lo concerniente a las tecnologías es un ámbito amplio donde están involucradas distintas áreas: redes, internet, hardware, comunicaciones, física, lógica; sin embargo, todas confluyen de alguna manera en los códigos binarios, que hacen que las computadoras realicen acciones que usamos a diario. Por esta razón, los lenguajes son tan relevantes.

01

UN POCO DE HISTORIA

James Gosling, un programador de **Sun Microsystems**, crea en 1995 el lenguaje Java, que, en sus orígenes, no se llamó de esa forma y, tras ciertos cambios, evolucionó hasta llegar a lo que hoy conocemos.

Al ser un lenguaje de programación multipropósito con el que podemos realizar cualquier tipo de programa, su uso se ha extendido enormemente. Java se ha hecho muy famoso por una de sus principales características: es un lenguaje independiente de la plataforma. Eso quiere decir que, si hacemos un programa en Java, podrá funcionar en cualquier computadora del mercado. Esto es una ventaja significativa para los desarrolladores de software, pues antes tenían que crear un programa para cada sistema operativo, como Windows, Linux, IOS, etcétera.

Java es independiente de la plataforma porque tiene una **máquina virtual** para cada sistema; esta máquina funciona como **puente** entre el sistema operativo y el programa en Java, y posibilita que este último se lea y se ejecute a la perfección.

Figura 1. En esta imagen vemos a James Gosling, el creador del lenguaje Java.

Origen del nombre

Existen varias historias sobre el origen del nombre de este lenguaje. Uno de los relatos en torno al nombre es que, en un principio, se llamó **Oak**, pero lo cambiaron pues ya existían derechos sobre ese nombre. También se especula que es el nombre de un café al que los fundadores concurrían o que se trata de las iniciales de los desarrolladores. Además, hay quienes conjeturan que es un nombre aleatorio devenido de un listado del pizarrón en que los creadores iban escribiendo los nombres propuestos.

Pasado, presente y futuro

En sus comienzos, Java fue concebido para ser utilizado en todo tipo de electrodomésticos; sin embargo, esta idea inicial no llegó a buen puerto. En ese momento, uno de los programadores, en vez de abandonar el proyecto, definió una nueva dirección, pero esta vez con un objetivo más ambicioso: orientarlo a Internet, lo que para esa época sonaba como una locura, pues se trataba de una tecnología que daba sus primeros pasos.

A partir de entonces, logró ser integrado al navegador **Netscape** (el más importante del momento), donde se encargaba de ejecutar programas o **applets** dentro de una página web.

Con el paso de los años, Java se convirtió en un lenguaje potente, seguro y universal, gracias a la posibilidad de usarlo en cualquier plataforma y por su carácter gratuito.

Actualmente, Java se utiliza para un amplio abanico de posibilidades, y casi todo lo que se puede hacer en cualquier otro lenguaje es posible lograrlo también en Java, muchas veces, con más ventajas. Para lo que aquí nos interesa, con Java podemos programar páginas web dinámicas con accesos a bases de datos, utilizar XML y aprovechar diversos tipos de conexión de red entre cualquier sistema, entre otras funciones.

En general, Java será útil para cualquier aplicación que deseemos desarrollar, por ejemplo, programas con acceso mediante la Web, y apps para sistemas móviles, como Android, lo que en la actualidad se encuentra en pleno auge.

Podemos concluir que Java es un lenguaje que llegó para quedarse, de eso no nos cabe ninguna duda; siempre ha sabido responder a los incesantes cambios tecnológicos y ha vuelto sobre sus pasos cuando fue necesario. Ahora apunta al mundo de los sistemas móviles; esto es importante pues las estadísticas indican que el uso de Android llega a casi un 70% de los teléfonos de este tipo en el mundo.

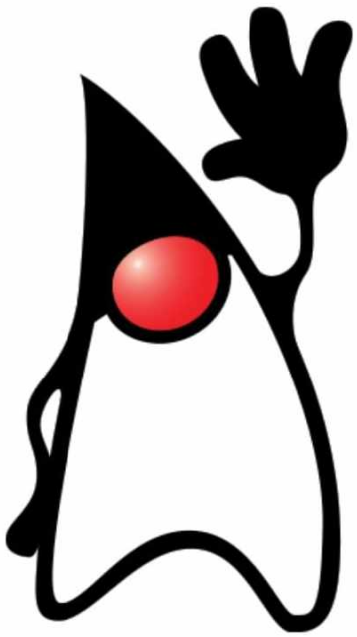


Figura 2. Duke, la curiosa mascota que por muchos años acompañó a las publicaciones de Java.

LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación se utiliza para escribir programas; estos constan de secuencias de instrucciones que se codifican y, a su vez, son traducidos para que las computadoras los entiendan.

A las secuencias traducidas, se las denomina **lenguaje de máquina** o **lenguaje de bajo nivel**, que es el lenguaje nativo de la computadora. Tales instrucciones, difíciles para los humanos, no son más que secuencias de ceros y unos, o patrones de bits. En consecuencia, para desarrollar programas, se necesitan lenguajes de programación amigables que nos permitan interactuar directamente con una computadora.

Cada lenguaje de programación incluye un conjunto de instrucciones que la computadora puede entender directamente en su código de máquina. Las instrucciones básicas y comunes en casi todos los lenguajes de programación son las siguientes:

1. **INSTRUCCIONES DE ENTRADA Y SALIDA**
   Instrucciones de transferencia de información entre dispositivos periféricos y la memoria principal.

2. **INSTRUCCIONES DE CÁLCULO**
   Se trata de instrucciones para que la computadora pueda realizar operaciones aritméticas.

3. **INSTRUCCIONES DE CONTROL**
   Son aquellas que modifican la secuencia de la ejecución del programa.

Por una parte, los lenguajes de bajo nivel se califican como lenguajes de máquina o lenguajes ensambladores. Por otra parte, los **lenguajes de alto nivel** permiten expresar los algoritmos de una forma adecuada para que sea entendida por los humanos. En este punto es necesario tener en cuenta que las computadoras solo ejecutan programas escritos en lenguajes de bajo nivel; los programas de alto nivel tienen que traducirse antes de ejecutarse, y esta traducción lleva tiempo, lo que constituye una pequeña desventaja para los lenguajes de alto nivel.

Aun así, las ventajas de los lenguajes de alto nivel son enormes. En primer lugar, la programación en lenguajes de este tipo es mucho más fácil; escribir programas con ellos toma menos tiempo, porque son más cortos y más fáciles de leer; y es más probable que estos programas sean correctos.

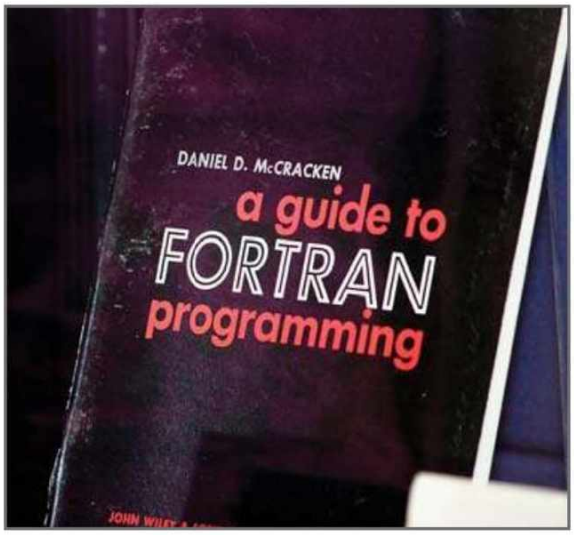


Figura 3. Fortran: antiguo lenguaje de programación ya casi en desuso.

En segundo lugar, los lenguajes de alto nivel son portables, lo que significa que pueden ejecutarse en tipos diferentes de computadoras sin modificación alguna o con pocas modificaciones. En cambio, los programas escritos en lenguajes de bajo nivel solo pueden ser ejecutados en un tipo de computadora y deben rescribirse para ejecutarlos en otra máquina. Debido a estas ventajas, casi todos los programas se escriben en un lenguaje de alto nivel, y los de bajo nivel solo se usan para unas pocas aplicaciones especiales.

Existen dos tipos de programas que traducen lenguajes de alto nivel a lenguajes de bajo nivel: **intérpretes** y **compiladores**.

Un intérprete lee un programa de alto nivel y lo ejecuta; esto significa que lleva a cabo lo que indica el programa; de esta forma, traduce el programa poco a poco, leyendo y ejecutando cada comando. Un **compilador** lee el programa y lo traduce todo al mismo tiempo antes de ejecutar cualquiera de las instrucciones. En este caso, el programa de alto nivel se llama **código fuente**, y el programa traducido se denomina **código objeto** o **código ejecutable**. Una vez compilado el programa, puede ejecutarse repetidamente sin volver a traducirlo.

Figura 4. En este esquema vemos la función del intérprete.

Evolución

La historia de los lenguajes de programación es relativamente reciente, (unos 60 años). Ada Byron, condesa de Lovelace (1815-1852), es considerada la primera programadora de la historia, pues intentó interpretar la máquina analítica de Charles Babbage y, para esto, desarrolló, en 1843, un algoritmo que podía ser utilizado en ella.

Desde 1954 hasta la actualidad, se han documentado más de 2.500 lenguajes de programación. Entre 1952 y 1972 (la primera época de los lenguajes de programación), se desarrollaron alrededor de 200 lenguajes. De ellos, solo una decena fueron realmente significativos y tuvieron influencia en el desarrollo de lenguajes posteriores, por ejemplo:

- **Fortran** (*Formula Translation*)
- **LISP** (*List Processor*)
- **COBOL** (*Common Business-Oriented Language*)

Los principales usos de estos tres lenguajes fueron aplicaciones para supercomputadoras, desarrollo de la inteligencia artificial y software empresarial.

A continuación, realizaremos un breve recorrido para ver cómo fueron apareciendo los lenguajes de programación y cuánta influencia tuvieron en los paradigmas modernos de desarrollo de software.

1960 Un hito en esta evolución es la llegada, en los años 60, de **Algol**, un lenguaje imperativo que no tuvo un gran éxito, pero sirvió de influencia para importantes lenguajes que vinieron después, como C.

1970 Ya por los 70, aparece el lenguaje **Pascal**, llamado así en honor al matemático y físico Blas Pascal; tiene un objetivo más bien educativo pues se utiliza para el aprendizaje de las estructuras de datos. También en los 70, se crea un lenguaje que será fundamental para el propósito de este curso: **Smalltalk** y que introduce el paradigma de orientación a objetos. En 1972 llega un lenguaje padre: **C**. Es desarrollado por el programador Dennis Ritchie, de los Laboratorios Bell. Se trata de un lenguaje de bajo nivel, creado para propósitos generales y con el que es reescrito el sistema Unix.

1983

Luego de C, vamos a saltarnos hasta 1983 en el que llega **C++**, que desciende de C, pero ya orientado a objetos; como sabemos, se trata de un paradigma que ejerce una influencia profunda en los lenguajes modernos. Ese mismo año aparece **Objetive-C**, que se utilizaba para programar todo lo relacionado con Apple.

1987

En el año 1987 se crea **Perl**, un poderoso lenguaje de alto nivel, conocido por su versatilidad, que ofrece un abanico interesante de usos, como aplicaciones de base de datos, administración de sistemas, programación web y programación de gráficos.

1991

En 1991 llega **Python**, un lenguaje con muchos propósitos, muy utilizado en la actualidad. Empresas como Google y Spotify, entre otras, lo incluyen dentro de sus líneas de código.

1993

Ya en 1993 nace un lenguaje que es considerado un bálsamo para los desarrolladores web: **Ruby**. Se trata de un lenguaje bastante versátil y con un framework propio (*Ruby on rails*). Twitter, Groupon y Hulu lo utilizan.

1995

Posteriormente, en 1995 nace **Java**. Este mismo año hace su aparición **PHP**, creado por el groenlandés Rasmus Lerdorf, muy utilizado para la creación de páginas web dinámicas. Encontramos ejemplos de su uso en Facebook, Wikipedia, Digg, WordPress y Joomla.
Cabe resaltar que, en estos años, también aparece **JavaScript** (JS), creado por un exdesarrollador de Netscape. JS ha logrado un importante posicionamiento entre los desarrolladores por las incontables bondades en el desarrollo de páginas web, por ejemplo, páginas dinámicas para el envío y validación de formularios, interactividad, animación, seguimiento de actividades de usuario, etcétera; es utilizado por Gmail, Photoshop y Mozilla Firefox.

Además de los lenguajes que hemos citado en orden cronológico, debemos tener en cuenta que aparecieron muchos otros, pero en esta ocasión hemos mencionado solo los más relevantes.

Clasificación

Entre los lenguajes de programación, encontramos dos grandes grupos: los lenguajes **imperativos** y los **declarativos**.

Dentro de los lenguajes imperativos mencionaremos los siguientes:

- **Secuenciales**: por ejemplo, Ensamblador.
- **Estructurados**: por ejemplo, BASIC.
- **Procedurales**: como C o Pascal.
- **Orientados a objetos**: por ejemplo, Java, C++, Ruby.

Dentro de los lenguajes declarativos encontramos la siguiente clasificación:

- **Lógicos**: como Prolog.
- **Funcionales**: por ejemplo, Haskell, Erlang, LISP.

También es posible clasificar los lenguajes de programación por el **tipado**, es decir, si un lenguaje es fuerte o débilmente tipado, tema que profundizaremos en el capítulo.

Figura 5. En este diagrama podemos ver la función de un compilador desde el código fuente hasta la salida.

Código fuente

Se trata de un texto escrito en un lenguaje de programación específico y que puede ser leído por un programador. Debe traducirse a lenguaje máquina para que pueda ser ejecutado por la computadora o a **bytecode** para que pueda ser ejecutado por un intérprete. Este proceso se denomina **compilación**. Acceder al código fuente de un programa significa acceder a los algoritmos desarrollados por sus creadores, se trata de la única manera de modificar eficazmente un programa.

ENFOQUE ALGORÍTMICO

Sin importar el lenguaje de programación que decidamos utilizar, existen diferentes fases en el proceso de programación, esto se conoce como **ciclo de vida** de un software. Para enfrentar este ciclo de vida, debemos seguir una serie de fases que son obligatorias; por esta razón, es necesario tenerlas en cuenta y acostumbrarnos a ellas, veámoslas:

ANÁLISIS DEL PROBLEMA
El problema se analiza teniendo en cuenta los requerimientos del cliente, previendo alcances y dimensionándolo con la anticipación necesaria.

DISEÑO DEL ALGORITMO
Es el momento de recurrir al lápiz y al papel, o tal vez a algún software adecuado para realizar los algoritmos necesarios que solucionen el problema.

CODIFICACIÓN
Llamada **implementación de los algoritmos**; para esto se utiliza la sintaxis de un lenguaje de programación y, como resultado, obtendremos el código fuente.

COMPILACIÓN Y EJECUCIÓN
Mediante diversos mecanismos, en esta etapa se procede a compilar y ejecutar el programa.

VERIFICACIÓN
En esta etapa se ejecutan las pruebas de rigor para verificar el programa; si aparecen errores o bugs, pasamos a la fase siguiente.

DEPURACIÓN
En caso de encontrar errores, que pueden ser de distinta índole, en esta etapa se procede a corregirlos (debugging).

MANTENIMIENTO
Una vez lanzado el programa, debemos tener en cuenta que es probable que en algún momento requiera ser actualizado o modificado en alguna de sus instancias.

DOCUMENTACIÓN
En todas las fases de la programación se deben escribir los manuales del usuario (además de los comentarios dentro del código) y también las normas adecuadas para su correcto mantenimiento.

En esencia, el enfoque algorítmico enfatiza la importancia de seguir una secuencia ordenada de pasos o actividades, para, en este caso, enfrentar el ciclo de vida de un software.

Algoritmos

La palabra **algoritmo** es una traducción del árabe Alkhô-warîzmi, en honor a un astrónomo y matemático árabe que trabajó en la manipulación de números y ecuaciones en el siglo IX.

Un algoritmo es la manera en que resolvemos un problema mediante una serie de pasos precisos, definidos y finitos. Las formas en que utilizamos los algoritmos se denominan métodos **algorítmicos**, a diferencia de los métodos **heurísticos**, que implican algún juicio o interpretación. Veamos un ejemplo sencillo en el que, sin darnos cuenta, aplicamos algoritmos.

Supongamos que queremos preparar una cena especial. Nos esmeramos para tener a la mano todos los ingredientes, para lo cual preparamos un listado y, si es necesario, vamos al supermercado para comprar los elementos. Una vez que tenemos todo, empezamos a seguir la receta (si no la tenemos en la cabeza); para ello, antes hay que disponer de los utensilios que utilizaremos (sartenes, bol, cuchillo, etcétera). Procedemos a cortar los distintos ingredientes, verificamos los condimentos y, para terminar, cocinamos.

Al final, una vez comprobada la correcta cocción y seguros de que los sabores sean los esperados (corregimos con sal, por ejemplo), será el momento en que habremos terminado el proceso.

En este caso, podemos ver que hemos seguido un método algorítmico para dar solución a un problema inicial.

Pero, para enfrentar la programación de software, necesitamos métodos un poco más sofisticados que en el ejemplo anterior. Por esta razón existen varios métodos, como **diagramas de flujo** y **pseudocódigo**, que conoceremos en las secciones posteriores de este capítulo.

De cualquier forma, el primer paso será identificar y definir el problema, y para esto, debemos plantearnos algunas preguntas:

¿Qué entradas se requieren?
Tipo de datos con los que se trabaja y su cantidad.

¿Cuál es la salida deseada?
Tipo de datos de los resultados y la cantidad.

¿Qué método produce la salida deseada?
Requerimientos adicionales y las restricciones.

Veamos un ejemplo un poco más específico. Necesitamos saber cuánto gana un obrero que trabaja por horas; para ello será útil hacernos las preguntas que ya hemos citado:

¿Qué entradas se requieren? ENTRADA.
Nombre del empleado, cantidad de horas, costo por hora.

¿Cuál es la salida deseada? SALIDA.
Nombre del empleado, total por cobrar.

¿Qué método produce la salida deseada? PROCESO.
La multiplicación de la cantidad de horas por el costo hora nos dará el total que cobrará el empleado.

Pseudocódigo

Para resolver este problema, en principio utilizaremos el método de los pseudocódigos. Se trata de la forma de escribir las instrucciones tal como lo hacemos en nuestro idioma, claro que siguiendo algunas reglas inherentes a este método. Para entenderlo mejor, tomaremos el ejemplo que planteamos en la sección anterior:

```
1. Inicio_Proceso
2. Leer: nombreEmpleado, Horas, costoHora,
3. Escribir: nombreEmpleado, Horas, costoHora
4. Calcular: totalCobrar = Horas * costoHora
5. Escribir: nombreEmpleado, totalCobrar
6. Fin_Proceso
```

El pseudocódigo puede utilizar palabras reservadas en un lenguaje como: **start**, **end**, **stop**, **if**, **then**, **else**, **while**, **repeat**, **until**, etcétera. También se puede utilizar **indentación** o **tabulación** desde el margen izquierdo, para que el pseudocódigo quede más ordenado.

Por cierto, la computadora no entiende este pseudolenguaje; sin embargo, nos será de mucha ayuda, sobre todo cuando comenzamos a programar. A la hora de implementar algoritmos, no importa el método que finalmente utilicemos, pero sí son necesarios.

Diagramas de flujo

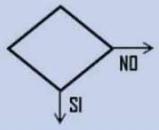
El método de **diagramas de flujo** o **flowchart** es una técnica muy antigua y, tal vez, la más utilizada en comparación con el pseudocódigo. Para trabajar con ella, se requiere conocer una serie de símbolos o cajas estándar, que contendrán cada uno de los pasos de la solución del algoritmo y estarán unidas por medio de flechas, que indican una secuencia.

Los símbolos más utilizados van a representar:

- proceso
- decisión
- conectores
- entrada/salida
- dirección del flujo
- inicio/fin

En la siguiente **Tabla** conoceremos los símbolos que podemos usar al crear un diagrama de flujo.

SÍMBOLOS PARA DIAGRAMAS DE FLUJO

SÍMBOLOS PRINCIPALES	NOMBRE	FUNCIÓN
	**Terminal**	Representa el inicio o el final del diagrama.
	**Entrada/salida**	Representa los datos de entrada o de salida.
	**Proceso**	Indica todas las acciones o los cálculos que se ejecutarán con los datos de entrada u otros obtenidos.
	**Decisión**	Representa las comparaciones de dos o más valores, tiene dos salidas: Verdadero y Falso.

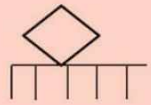
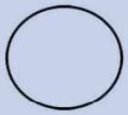
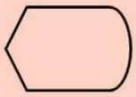
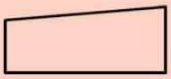
SÍMBOLOS PRINCIPALES	NOMBRE	FUNCIÓN
	**Decisión múltiple**	En función del resultado, se seguirá uno de los diferentes caminos de acuerdo con dicho resultado.
	**Conector**	Permite identificar la continuación de la información, si el diagrama es muy extenso
	Indicador de dirección	Indica el sentido de ejecución de las operaciones.
	Línea conectora	Une los símbolos.
	**Conector**	Conecta dos puntos del diagrama, pero en diferentes hojas.
	**Llamada a subrutina**	Llamada a subrutina o proceso predeterminado.
	**Pantalla**	Ocasionalmente para representar E/S.
	**Impresora**	Ocasionalmente para representar E/S.
	**Teclado**	Ocasionalmente para representar E/S.
	**Comentarios**	Para añadir comentario a algún símbolo.

Tabla 1. En esta tabla vemos los símbolos que podemos utilizar en un diagrama de flujo.

Para comprender de mejor forma el uso del pseudocódigo y los diagramas de flujo, veamos algunos ejemplos resueltos:

1. Desarrollemos un algoritmo que permita leer dos valores distintos y, luego, determinar cuál de los dos es el mayor.

Primero realicemos el pseudocódigo adecuado para este problema:

```
1. Inicio
2. Inicializar variables: A=0, B=0
3. Solicitar dos valores distintos
4. Leer los dos valores
5. Asignarlos a las variables A y B
6. Si A=B Entonces vuelve a 3
7. Si A>B Entonces
    Escribir A, "Es el mayor"
8. De lo contrario: Escribir B, "Es el mayor"
9. Fin_Si
10. Fin
```

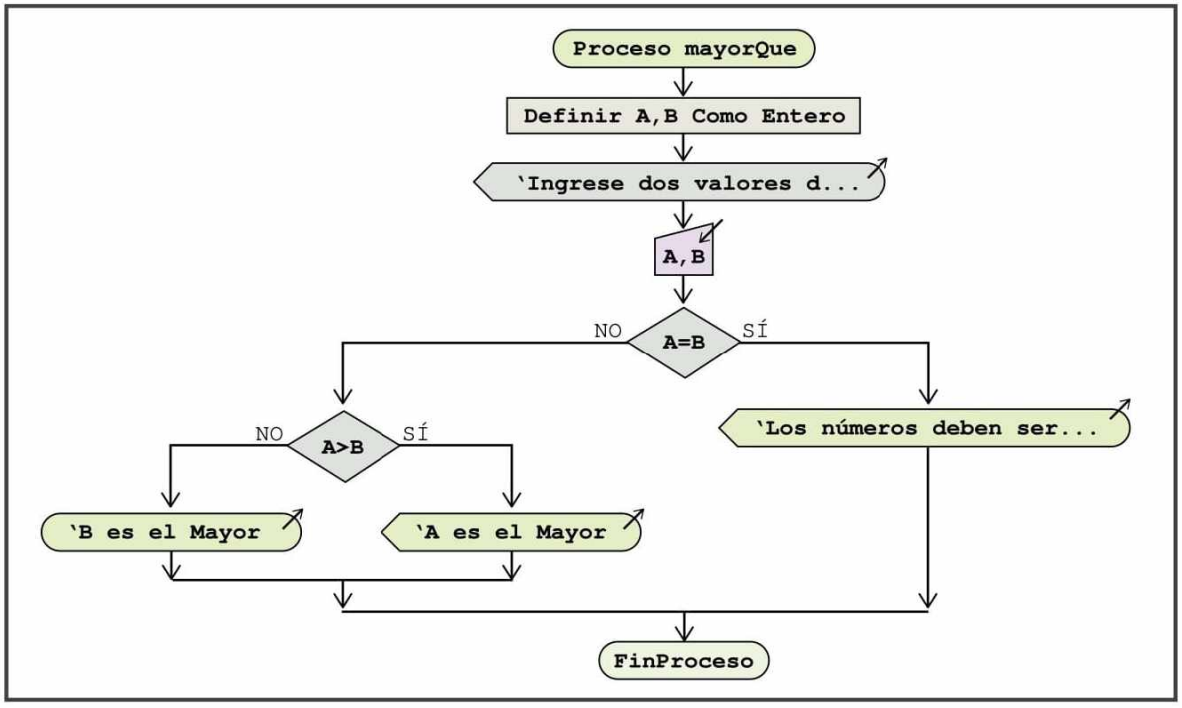


Figura 6. Este diagrama de flujo corresponde al problema que nos pide leer dos valores distintos y, luego, determinar cuál es mayor.

2. Dadas tres notas, determinar si un alumno aprueba o no una materia; considerar que, para aprobar, el promedio debe ser igual o mayor que 7.

Primero creamos el pseudocódigo adecuado para resolver este problema:

```
1. Inicio
2. Definir las variables n1,n2,n3, prom
3. Leer los 3 valores
4. Almacenar en las variables n1,n2,n3
5. prom = (n1+n2+n3)/3
6.  Si prom>=7 Entonces
7. Escribir "El estudiante aprobó con: ", prom
8. Sino
9.  Escribir "El estudiante reprobó con: ", prom
10. Fin_Si
11. Fin
```

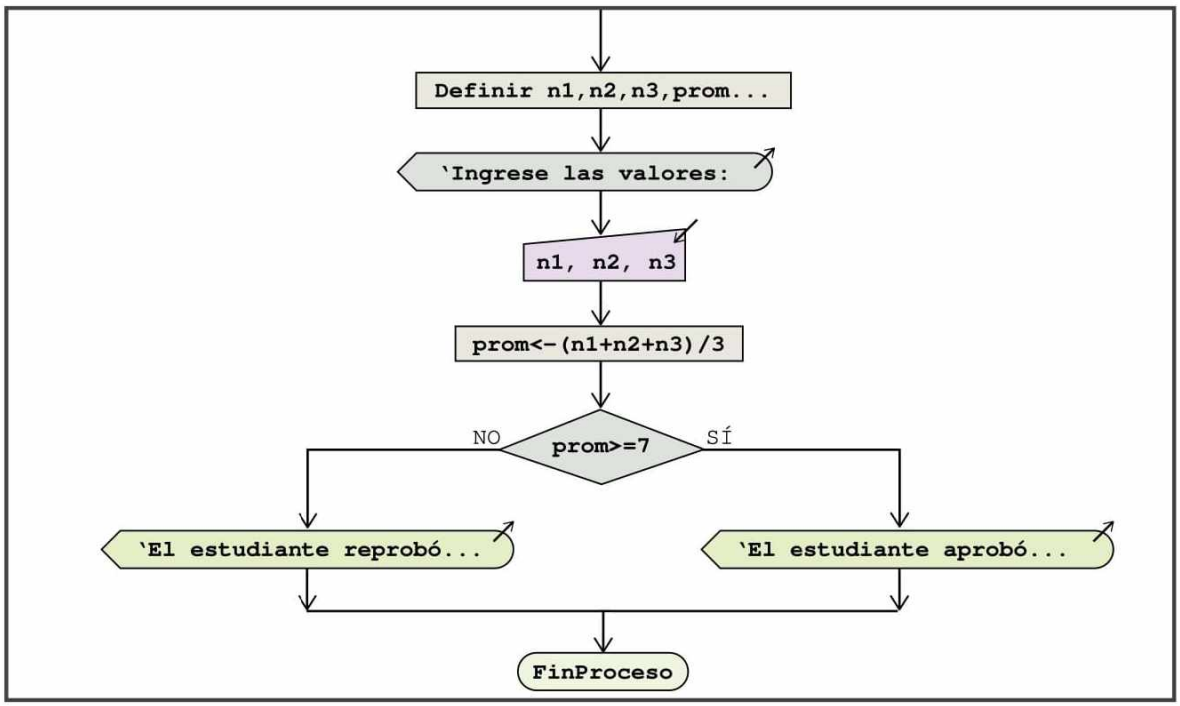


Figura 7. Este diagrama de flujo corresponde al problema en el que determinamos la situación del alumno sobre la base de su promedio.

JAVA

Java es un lenguaje de programación que combina dos aspectos dignos de mencionar: su diseño y su popularidad.

Este lenguaje ofrece una implementación muy limpia de los conceptos de programación, y su popularidad ha alcanzado niveles insospechados, lo que nos asegura que la cantidad de recursos disponibles son diversos e importantes, todo lo que necesitamos para comenzar en el mundo del desarrollo.

Como todos los lenguajes de programación, Java posee sus propias características para describir algoritmos, es decir, sus propios fundamentos. En este sentido, debemos tener en cuenta que este curso no pretende hacer una descripción exhaustiva de la totalidad de modos en que podemos expresarnos en este lenguaje, sino que intenta entregar la base que necesitamos para crear nuestros primeros programas.

En torno a Java existen muchas confusiones, que es necesario aclarar antes de continuar.

Ya conocemos algo de la filosofía que se encuentra en la base de Java: *Write once, run anywhere*, es decir, no importa dónde escribamos el código, ya que funcionará en cualquier sistema operativo y hardware en los que decidamos hacerlo correr, y esto es posible porque posee una máquina virtual.

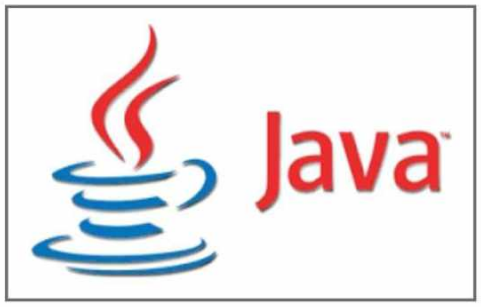


Figura 8. En esta imagen vemos el logo que acompaña a Java.

PSeInt

Podemos descargar y utilizar un programa libre, altamente recomendado para trabajar con diagramas. Se trata de **PSeInt**, que encontramos en la dirección **http://pseint.sourceforge.net**. Este sencillo programa nos ofrece las herramientas que necesitamos para crear y editar diagramas, así como también útiles ejemplos de algoritmos para nuestras prácticas iniciales.

Java Virtual Machine o **JVM** es una máquina de proceso nativo, es decir, ejecuta en una plataforma específica, y es capaz de interpretar y ejecutar las instrucciones expresadas en bytecode de Java (un código binario especial), generado por el compilador del lenguaje. JVM es la piedra fundamental de este lenguaje y hace que este funcione como lo conocemos, es decir, respetando la filosofía que ya citamos.

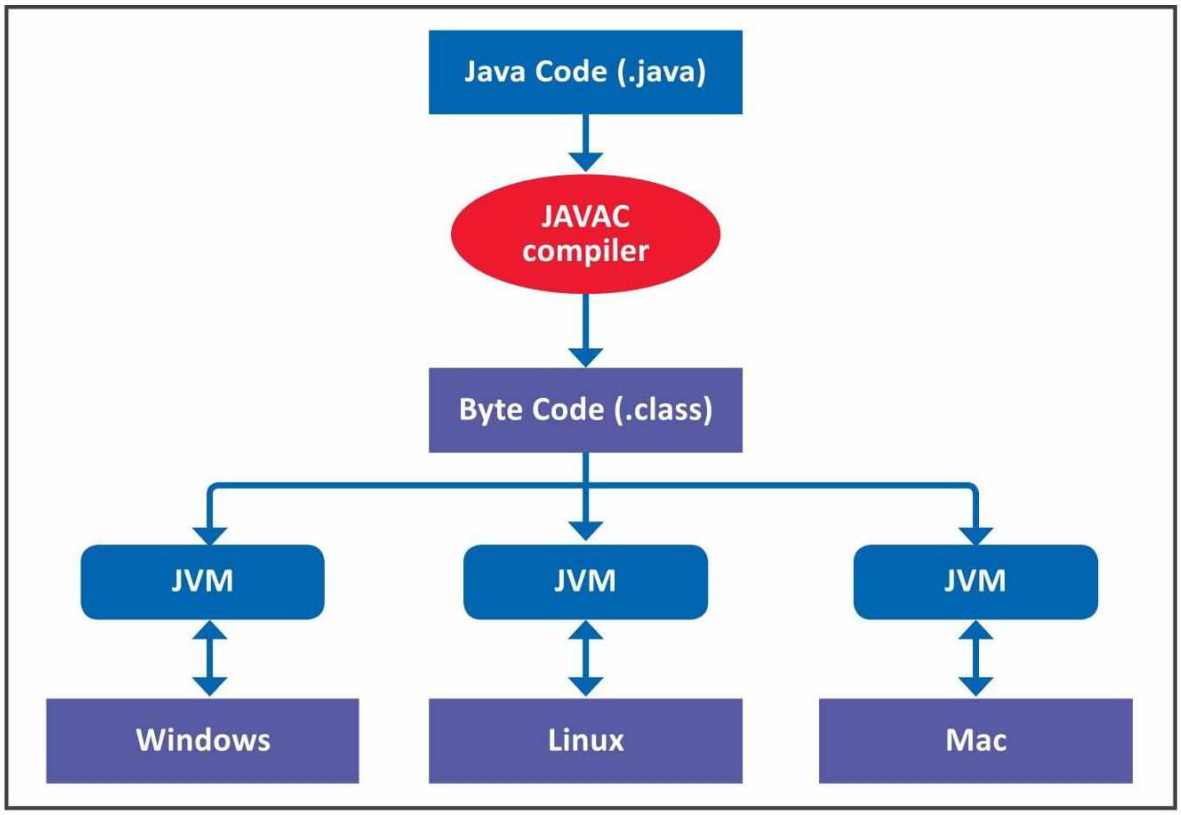


Figura 9. Esquema de la arquitectura general de un programa en ejecución en una Máquina Virtual de Java.

Curiosidades de Java

El 'Hola Mundo' es un clásico en todos los lenguajes de programación y es lo que difiere en cada uno de ellos. El primer 'Hola Mundo' en Java salió en mayo de 1995. Se trata de un lenguaje que no fue planeado por su creador, ya que se dice que trabajaba depurando algunos códigos con C++. Actualmente, Java ha entrado en una importante etapa madurativa y existen numerosos frameworks disponibles.

Existen diferentes ediciones de Java, y esto podría confundirnos, sobre todo, cuando estamos comenzando a trabajar con este lenguaje. Para aclarar esto, presentamos sus ediciones más comunes:

JAVA EE
Se trata de la edición *Enterprise* de Java, utilizada para proyectos web.

JAVA SE
Es la edición *Standard* de Java, la versión que utilizaremos para la gran mayoría de los capítulos que componen este curso.

JAVA ME
Es la *Micro* edición de Java: es utilizada para el desarrollo de aplicaciones de bajo nivel, como los electrodomésticos.

JRE
Java Runtime Environment es un conjunto de utilidades que permite la ejecución de programas Java. El JRE actúa como un intermediario entre el sistema operativo y Java.

JDK
Más conocido como *Java Developer Kit*, es el kit de desarrollo en Java, que contiene todas las clases y librerías para poder trabajar y desarrollar nuestros programas. Este software es necesario para luego poder instalar nuestra IDE de desarrollo (Eclipse o Netbeans). Actualmente se encuentra en su versión 8.

JDBC
Java Database Connectivity, es la API que permite la ejecución de operaciones sobre bases de datos.

Para saber qué versión de Java se encuentra instalada en nuestra computadora, necesitamos entrar a la consola de Windows (CMD) y escribir el comando **java–version**, luego presionamos **ENTER**. De inmediato veremos la información que necesitamos.

A la fecha, podemos probar la versión 9 de Java y hallaremos variada documentación en el sitio web de Oracle.

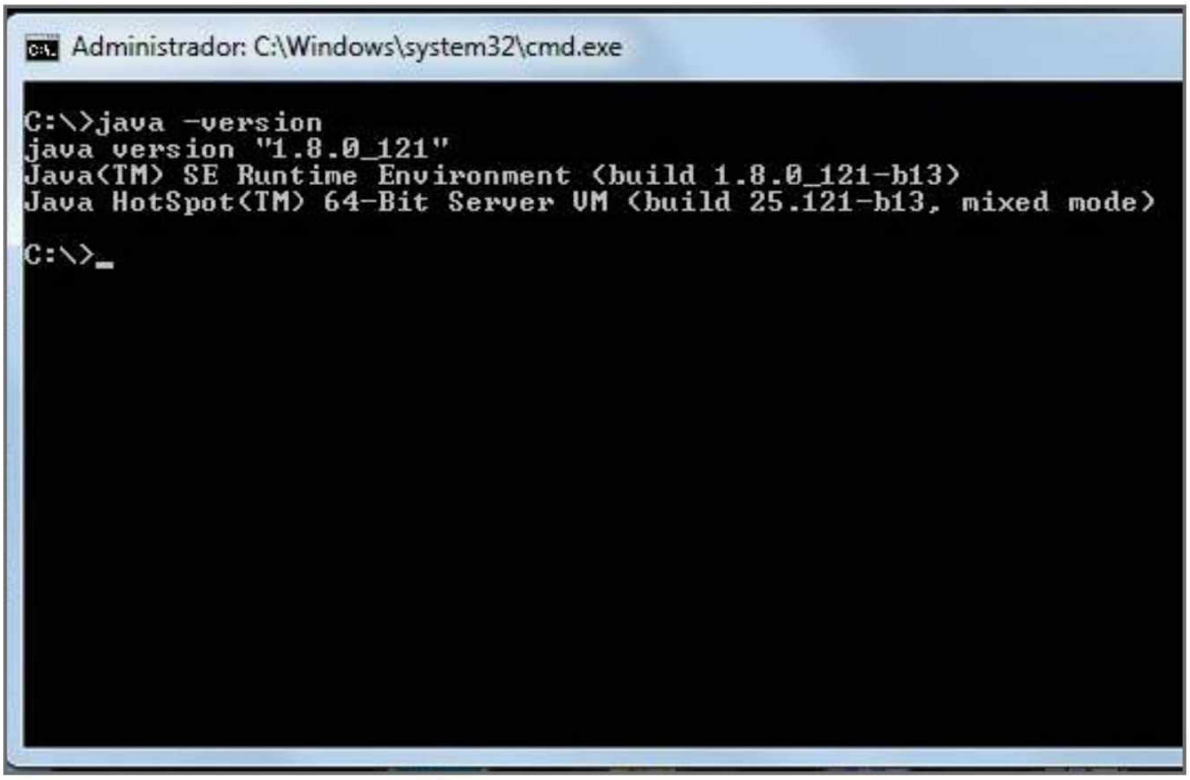


Figura 10. Esta imagen muestra el resultado de la ejecución del comando **java-version** en una consola de comandos de Microsoft Windows.

RESUMEN CAPÍTULO 01

Como punto de partida, en el capítulo inicial de esta obra hicimos una pequeña introducción a los lenguajes de programación, revisamos su evolución y, también, el momento en que apareció Java. Vimos algo de la historia de los lenguajes de programación más importantes, cómo nacieron, cuáles son las empresas que los usan y su clasificación. Además aprendimos a planificar nuestros programas a través de la algoritmia, conocimos qué es el ciclo de vida de un software y qué etapas lo componen. Para terminar, realizamos un acercamiento al lenguaje objeto de este curso: Java. Vimos algunas de sus características y analizamos sus diferentes ediciones.

Actividades 01

Test de Autoevaluación

1. ¿Quién creó el lenguaje Java?
2. ¿Qué es un applet?
3. ¿Qué es un lenguaje de programación?
4. Clasifique los lenguajes de programación.
5. ¿Qué es un lenguaje de bajo nivel? Cite dos ejemplos.
6. ¿Qué es un algoritmo?
7. ¿Qué es un compilador?
8. ¿Qué es lo que hace que Java sea un lenguaje tan popular?
9. ¿Cuál es la filosofía de Java?
10. Mencione las ediciones de Java.

Ejercicios prácticos

1. Resuelva un algoritmo para cambiar la rueda de un auto. Use pseudocódigo.
2. Descargue el programa PSeInt para realizar las actividades de diagramas de flujo y pseudocódigos.
3. Dados tres números, halle la suma del primero con el segundo, y el producto del segundo con el tercero. Resuélvalo con ambos métodos.
4. En una playa de estacionamiento cobran $150 la primera hora y $100 a partir de la segunda hora. Diseñe un algoritmo que determine cuánto debe pagar un cliente, conociendo el tiempo de estacionamiento en horas.
5. Obtenga el IVA de una venta, si esta es superior a $ 10.000 aplique un descuento del 25%, en caso contrario no se le aplica el impuesto.

Primeros Pasos

Ya sabemos qué son los lenguajes de programación y para qué sirven, vimos qué tipo de lenguaje es Java y analizamos sus principales características. En este capítulo, pondremos manos a la obra, conoceremos la sintaxis de este lenguaje y realizaremos nuestras primeras incursiones con Java.

02

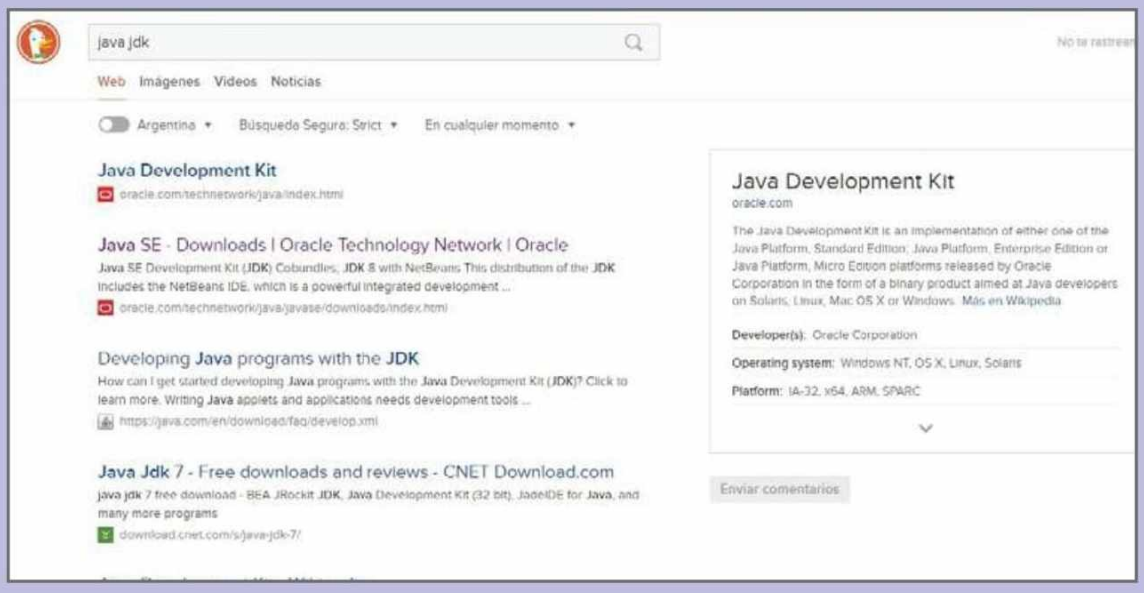
INSTALACIÓN DEL JDK

El **JDK** es un conjunto de aplicaciones útiles para programar en Java, que, además de la máquina virtual, contiene el compilador y el intérprete.

A continuación revisaremos la forma adecuada de preparar el entorno de trabajo necesario para realizar la codificación de nuestros programas.

PASO A PASO: INSTALAR EL JDK DE JAVA

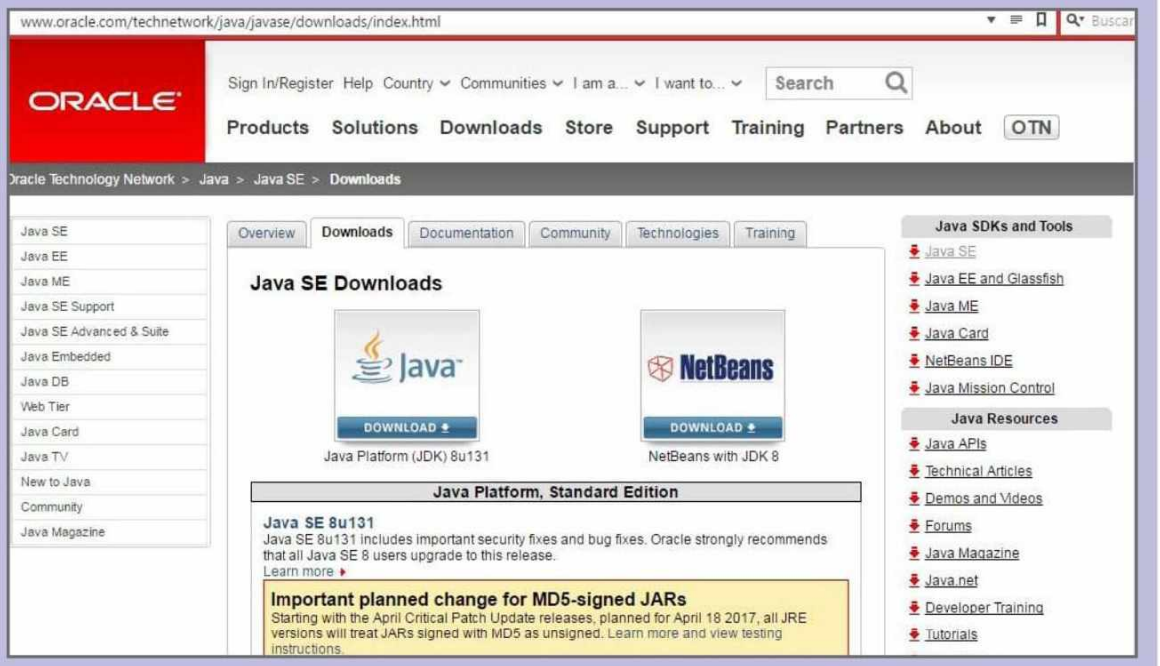
01 Para comenzar, ejecute el navegador web de su preferencia y acceda a la página oficial de Oracle en la dirección **www.oracle.com/technetwork/java/javase/downloads/index.html**.



Máquina Virtual

Una de las formas más comunes de emular un sistema es usar una máquina virtual, por ejemplo, de sistemas operativos. Entonces se emula una computadora como si fuera una real y se lo hace dentro de nuestro propio sistema, con los recursos que necesita, como memoria, espacio y memoria gráfica.

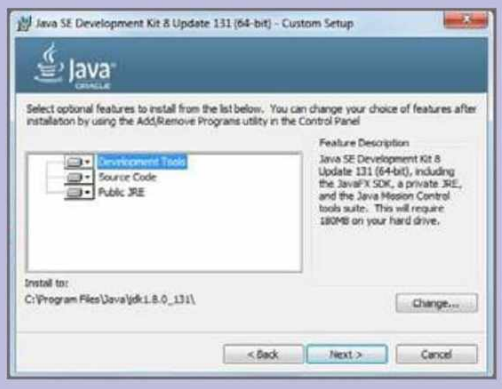
02 En el lado izquierdo del panel de navegación de la página, se muestran las diferentes versiones de Java que conocimos y explicamos en el Capítulo 1.



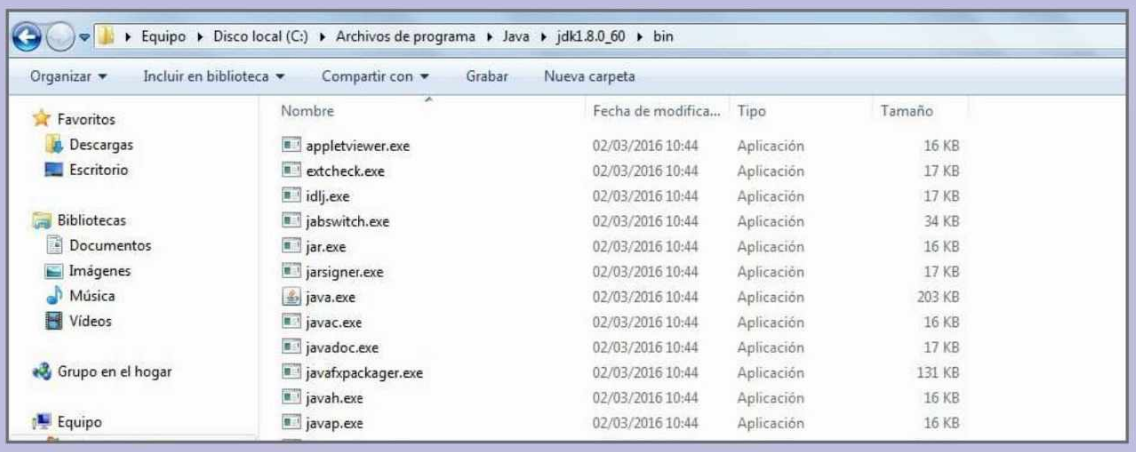
03 Antes de descargar la versión de Java que necesita, verifique en su computadora qué tipo de sistema operativo se encuentra instalado, es decir, si se trata de un SO de 32 o de 64 bits. Haga clic en el enlace que corresponda al tipo de SO que incluya su PC.

04 Espere mientras el proceso de descarga se completa, esto puede tardar algunos minutos dependiendo de la velocidad de su conexión.

05 Ubique el archivo en la carpeta de descargas; para instalarlo deberá hacer doble clic en el icono adecuado. La instalación del JDK puede tardar algunos minutos, y se trata de un proceso que no presenta complejidad alguna, solo hay que seguir las indicaciones del asistente de instalación.



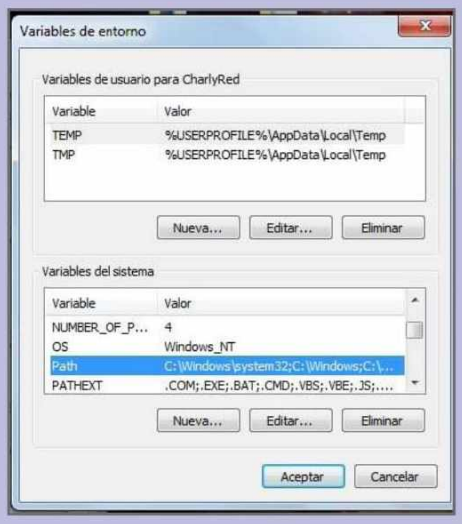
06 En este momento es necesario que verifique el contenido de la carpeta de instalación. Navegue hasta **Archivos de Programa/Java/jdk1.8.0_60/bin** donde se encuentran los ejecutables. El corazón del kit lo conforman los archivos **javac.exe** (compilador) y **java.exe** (la máquina virtual Java).



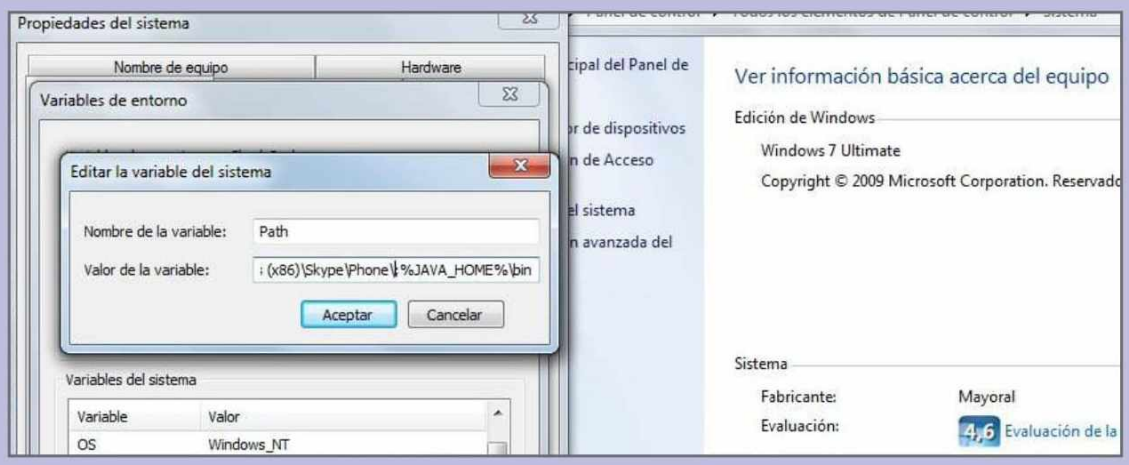
En Java, como en otros lenguajes de programación, por lo general nos encontraremos con dos formas de programar: empleando un IDE (como Eclipse o NetBeans) o utilizando un editor de texto plano cuyos resultados se verán en la línea de comandos del sistema. Para utilizar esta segunda opción, es necesario configurar el path de Java. Esto es importante pues, para invocar el compilador desde cualquier lugar en la línea de comandos, precisamos que la carpeta bin figure en el path del sistema. Para lograrlo debemos copiar la ruta que corresponde a la carpeta bin (por ejemplo **C:/Archivos de Programa/Java/jdk1.8.0_60/bin**) y seguir las instrucciones planteadas en el siguiente **Paso a paso**:

PASO A PASO: CONFIGURAR EL PATH DE JAVA

01 En primer lugar, acceda al cuadro de diálogo **Propiedades del sistema**, en **Panel de control/ Sistema y seguridad/Sistema** y seleccionando la opción adecuada en el panel lateral de opciones. Dentro de la pestaña **Opciones avanzadas**, haga clic sobre **Variables de entorno**, seleccione **Path** y pulse sobre **Editar**.



02 Ubique el cursor al final de la línea **Valor de la variable**, escriba un punto y coma (**;**) y luego pegue la ruta de la carpeta **bin**. Haga clic en **Aceptar**.



03 Para probar que se configuró el path correctamente, ejecute una ventana de comandos presionando las teclas **WINDOWS + R**, escriba **cmd** y luego presione **ENTER**. En cuanto aparezca la ventana de comandos, escriba **javac –version** y presione **ENTER**. Luego escriba **java –versión** y presione **ENTER**.

04 Si todo está correcto, tal como muestra la imagen, ya se encuentra en condiciones de escribir su primer programa en Java.

Comunidades de Desarrollo

En la Comunidad Java Codegeeks podemos encontrar aspectos avanzados del lenguaje, actualizaciones y hasta manuales interesantes, aunque están en inglés.

ENTORNOS DE DESARROLLO

Aunque es posible realizar nuestros programas con herramientas sencillas, como el Bloc de notas o Notepad++, es necesario considerar que el uso de un entorno de desarrollo o IDE (Integrated Development Environment) nos permitirá trabajar en aplicaciones más completas, gracias a la posibilidad de acceder a servicios y ventajas que los métodos más simples no proporcionan. Entre las ventajas de un IDE encontramos las siguientes:

- Un área de trabajo para escribir los códigos y acceso a la ayuda que permite generar código de manera automática; asimismo a un corrector de sintaxis y tabuladores automáticos, de tal manera que los códigos sean más ordenados.
- Herramientas para compilar y ejecutar el código escrito.
- Opciones para organizar los proyectos de programación.
- Herramientas auxiliares para programadores, adecuadas para efectuar la detección de errores o el análisis de programas (**debuggers**).
- Opciones adicionales, tales como utilidades para realización de pruebas o carga de librerías, entre otras.
- Importación de proyectos locales a una red e, incluso, desde otras IDE.

Como vemos, la importancia y las ventajas de los entornos de desarrollo no es menor, por lo tanto, es necesario conocer algunos de los IDE que podemos utilizar.

IDE, tomar la decisión adecuada

El IDE permite que tengamos todo lo necesario para programar. Hay que tomar una buena decisión a la hora de elegirlo, ya que pasaremos horas trabajando en él.

ECLIPSE

Se trata de un software libre que se descarga desde la dirección **https://www.eclipse.org**. Es uno de los entornos Java más utilizados por los profesionales; su paquete básico se puede expandir mediante la instalación de plugins que agregan funciones a medida que se van necesitando.

Figura 1. En esta imagen vemos la página oficial de **Eclipse**, uno de los IDE más populares.

NETBEANS

Es una aplicación libre que se puede descargar desde la dirección **www.netbeans.org**. Es otro de los entornos Java comúnmente utilizados; al igual que en Eclipse, pueden agregarse funciones mediante el uso de plugins.

INTELLIJ IDEA

Como ocurre con NetBeans y Eclipse, también puede incorporar soporte para otros lenguajes de programación. Pero, a diferencia de los IDE anteriores, se trata de un producto comercial.

No obstante podemos acceder a una edición reducida, denominada **Community**, que se obtiene en forma gratuita. Para descargarlo, debemos visitar la dirección **www.jetbrains.com/idea/download**.

JCREATOR

Este IDE se distribuye en forma comercial, aunque se pueden obtener versiones de prueba o versiones simplificadas gratuitas en la web **www.jcreator.com**. Está escrito en C++ y omite herramientas para desarrollos gráficos, por lo que es más rápido y eficiente que otros IDE.

BLUEJ

Software libre que se puede descargar desde **http://bluej.org**, se trata de un entorno de desarrollo dirigido al aprendizaje de Java (entorno académico), por lo que no se considera su uso en el ámbito profesional. Se destaca por ser sencillo e incluir algunas características dirigidas a que las personas que estén aprendiendo comprendan con mayor facilidad los aspectos clave de la programación orientada a objetos.

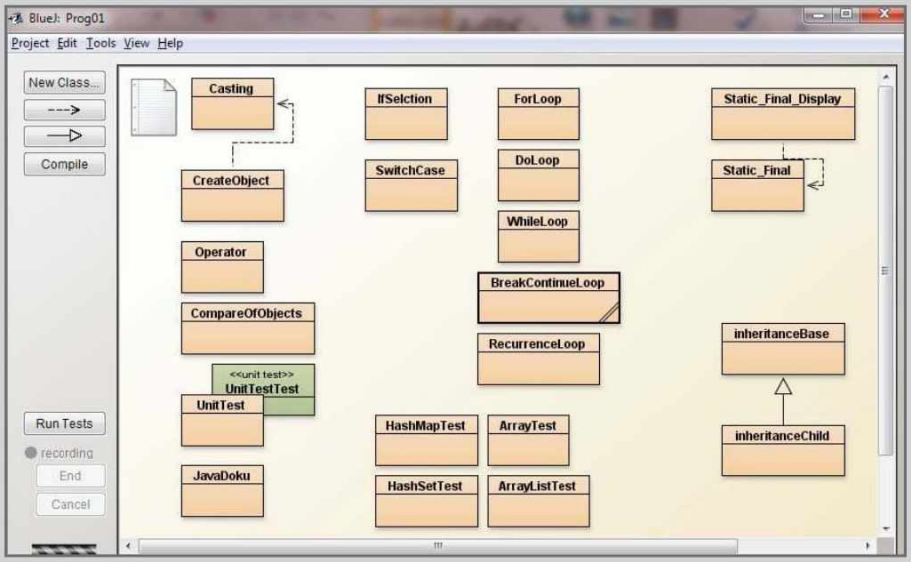


Figura 2. **BlueJ** es un IDE de software libre muy popular en la comunidad académica.

JGRASP

Es un IDE superligero creado por la **Universidad de Auburn** (Alabama). Su punto fuerte se encuentra en la capacidad de generar visualizaciones del código que escribimos en forma de diagramas de estructura de control (Control Structure Diagrams). Para descargarlo debemos visitar la dirección **www.jgrasp.org**.

Librerías en Java

Una librería en Java es un conjunto de clases que poseen una serie de métodos y atributos. Lo realmente interesante de estas librerías para Java es que facilitan muchas operaciones. De una forma más completa, las librerías en Java nos permiten reutilizar código, es decir, podemos hacer uso de los métodos, clases y atributos que componen la librería y así evitaremos implementar nosotros mismos esas funcionalidades.

Instalación del IDE

Decidir por uno de los entornos de desarrollo integrados que hemos presentado parece una tarea algo complicada; para esta obra utilizaremos NetBeans, pues se trata de la opción ofrecida directamente en la página oficial de Java.

Figura 3. NetBeans es el IDE con el que realizaremos los códigos en este curso.

Para descargarlo visitaremos el sitio **http://netbeans.org/downloads**, aquí encontraremos diferentes versiones para descargar, en este caso necesitamos **Java SE**, por lo tanto, hacemos clic sobre **Download** en la columna que corresponde.

Figura 4. Sitio oficial de descarga del IDE NetBeans. Debemos tener en cuenta la versión del SO y el idioma que deseamos descargar.

Una vez descargado nuestro entorno de desarrollo, es necesario ejecutarlo y seguir las indicaciones del asistente de instalación. Cuando se completa el proceso, ya estamos preparados para comenzar con nuestros primeros programas. Recordemos que es necesario contar con espacio suficiente en el disco duro en el que se instalará el IDE, pues en esa ubicación también se grabarán los archivos que generemos.

Figura 5. Una vez instalado el IDE lo ejecutamos. Mientras su proceso de inicio se completa, veremos información sobre los desarrolladores, la versión del IDE y otros datos técnicos importantes.

Luego de instalar el IDE podemos ejecutarlo. Una vez que nos encontremos en su interfaz principal, será un buen momento para que comencemos a explorar el árbol de proyectos que contendrá todo lo relacionado con nuestros programas.

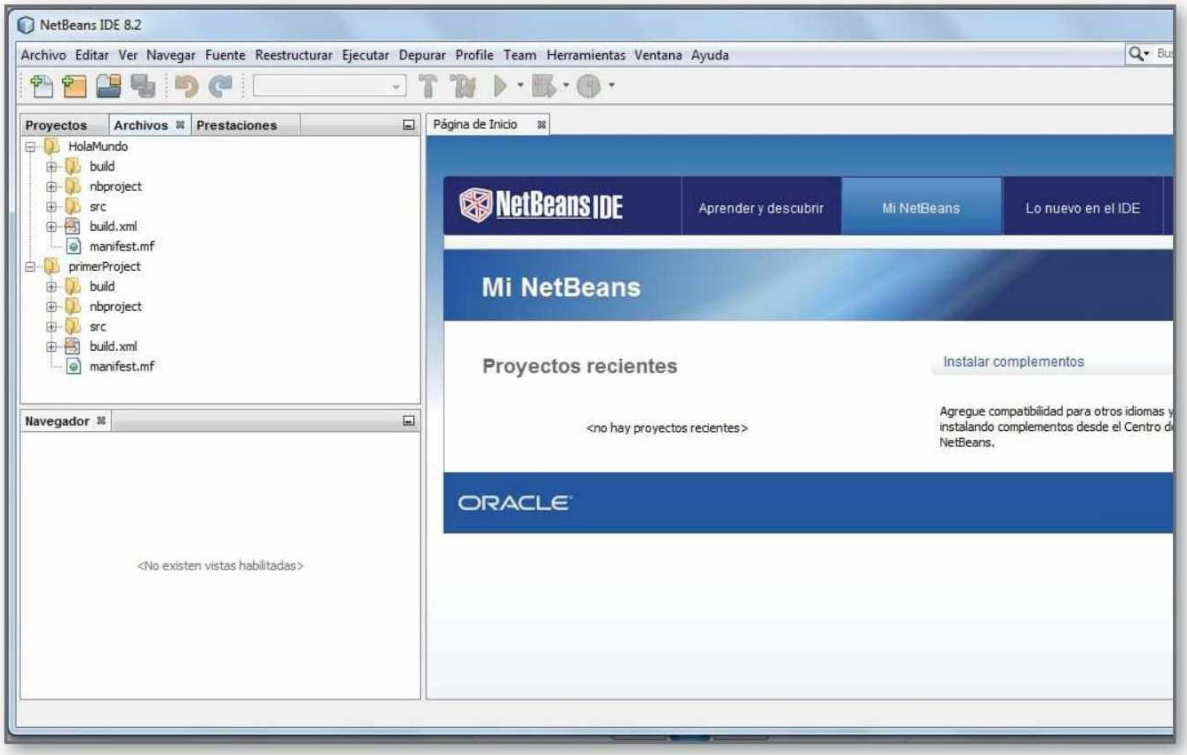


Figura 6. Vista principal del entorno NetBeans una vez iniciado. Observamos el panel de **Archivos**, el panel **Navegador** y, en su sección principal, la **Página de inicio**.

SINTAXIS Y REGLAS

La **sintaxis** es un conjunto de reglas que necesitamos conocer para programar de manera correcta, utilizando un lenguaje de programación específico. A continuación, revisaremos algunas consideraciones iniciales para tener en cuenta cuando escribamos código en Java.

Sintaxis básica

En esta sección conoceremos las reglas más recurrentes a la hora de iniciar la codificación, así estaremos preparados para enfrentar los primeros errores que, seguramente, aparecerán conforme avancemos en nuestro aprendizaje:

Java es un lenguaje del tipo **case sentive**, es decir, sensible a las mayúsculas y las minúsculas. Por esta razón, no es lo mismo escribir **numero** que **Numero**.

Para nombrar una **clase** se debe comenzar con mayúscula. Si la clase tuviera dos o más palabras, se recomienda que las iniciales empiecen con mayúsculas, a esto se denomina **Camel Case**, por ejemplo:

```
class MiPrimeraClase
```

Para nombrar un **método**, se recomienda comenzar con minúscula; si usamos varias palabras, las interiores deberían empezar en mayúsculas. Por ejemplo:

```
public void miPrimerMetodo()
```

Al nombrar un **archivo**, debe coincidir con el nombre de la clase, de otro modo este no compilará. Por ejemplo, si el nombre de la clase es **MiPrimerPrograma**, entonces el archivo debe ser guardado como **MiPrimerPrograma.java**.

La ejecución de un programa en Java se inicia desde el método **main()**, por lo tanto, se trata de una parte obligatoria en el código.

Los bloques de código deben estar dentro de llaves **{}**, por ejemplo:

```
public class NombreDeClase {
//este es el bloque 1
public static void main(String args[]){
//este es el bloque 2
}//fin del bloque 1
}//fin del bloque 2
```

Las instrucciones deben terminar en punto y coma (**;**), por ejemplo:

```
System.out.println("Hola mundo");
```

Identificadores

Los **identificadores** permiten asignar nombres a los distintos componentes de un programa Java (clases, métodos, variables) y deben ser utilizados sin excepción alguna. Existen algunas reglas para nombrarlos adecuadamente:

- Todos los identificadores deben comenzar con una letra mayúscula o minúscula (**A-Z** o **a-z**), con el carácter de moneda (**$**) o con un guión bajo (_).

- Una **palabra clave** no se puede utilizar como un identificador. No es posible usarlas para nombrar variables, ya que poseen un propósito definido dentro de un programa y, por lo tanto, se trata de palabras reservadas. Los IDE nos indicarán cuándo una palabra es clave y la resaltarán con un color distinto al resto del código, generalmente se utiliza el color azul.

- Los identificadores distinguen entre mayúsculas y minúsculas.

Teniendo en cuenta estas reglas, algunos ejemplos de identificadores son: **edad**, **$salario**, **_valor**, **_1_valor**; y **123abc**, **-salario**, ejemplos de identificadores inválidos.

Secuencias de escape

Una **secuencia de escape** corresponde al uso de signos que no veremos en la consola o salida del programa, pero que pueden resolver los problemas que surgen cuando deseamos imprimir ciertos signos que el compilador identifica como signos internos.

Una secuencia de escape siempre representa un solo carácter, aunque para escribirla utilicemos dos o más caracteres. Por ejemplo, se utilizan para realizar acciones, como un salto de línea, o para presentar caracteres no imprimibles.

En la **Tabla 1** presentamos algunas de las secuencias de escape que se encuentran definidas en Java.

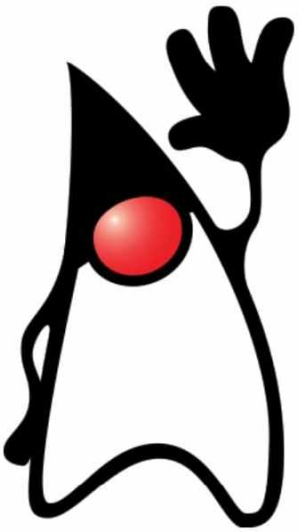
Secuencias de Escape

SECUENCIA DE ESCAPE	DESCRIPCIÓN
\n	Ir al principio de la siguiente línea
\b	Retroceso
\t	Tabulador horizontal
\r	Ir al principio de la línea
\"	Comillas
\'	Comilla simple
****	Barra inversa
\udddd	Carácter Unicode

Tabla 1. Secuencias de escape definidas en Java junto a sus respectivas descripciones.

Caracteres especiales y espacios en blanco

Cuando trabajemos con codificación Java, debemos tener en cuenta la existencia de caracteres especiales y de signos de puntuación; se trata de los siguientes: **+ - * / = % & # ! ? ^ " ' ~ \ | < > ( ) [ ] : ; . , { }**. Por otra parte, también existen los espacios en blanco; estos corresponden a **tabulador**, **espacio** y **salto de línea**, y su labor es realizar una separación entre los diferentes elementos de un programa.



Comentarios

Un **comentario** es una anotación que permite explicar algo referente al código que se está produciendo, por lo que no será tomado en cuenta por el compilador.

En los comentarios, se acostumbra incluir información que pueda ayudar a comprender mejor el código elaborado.

Existen tres tipos de comentarios, los que conoceremos a continuación:

1

DE UNA LÍNEA

Lo usamos para comentar alguna secuencia corta o instrucción simple; para crearlo comenzamos con dos barras (**//**). Por ejemplo:

```
//Esto es un comentario de una línea
```

2

DE VARIAS LÍNEAS

Si lo que necesitamos explicar ocupará mucho espacio, será necesario usar un comentario de varias líneas; para crearlo ponemos el comentario entre los símbolos **/*** y ***/**. Por ejemplo:

```
/*Este se considera un comentario de varias líneas.
No importa que se haya cambiado de línea, continúa
el comentario*/
```

3

PARA EL DOCUMENTADOR DE JAVA:

Este tipo de comentario es tenido en cuenta por el documentador de Java, es muy recomendable al comienzo de todo programa; para crearlo utilizamos los símbolos **/**** y ***/**. Por ejemplo:

```
  /**Este se considera un comentario de varias líneas.
El cual será de vital importancia cuando se pida a la
herramienta java.doc que genere la documentación
correspondiente a la clase que tenga este comentario.*/
```

NUESTRO PRIMER PROGRAMA

Antes de enfrentarnos al desarrollo de nuestro primer programa en Java, debemos familiarizarnos con el concepto de **proyecto**.

Un proyecto es un conjunto ordenado de carpetas y archivos que nos permiten mantener el código organizado, suele constar de archivos **.java**, **.class** y también documentación.

El código fuente se encuentra en los archivos **.java** dentro de la carpeta **src** (**source**); sin embargo, los archivos **.class**, que contienen el **Bytecode**, se ubican en la carpeta **bin**.

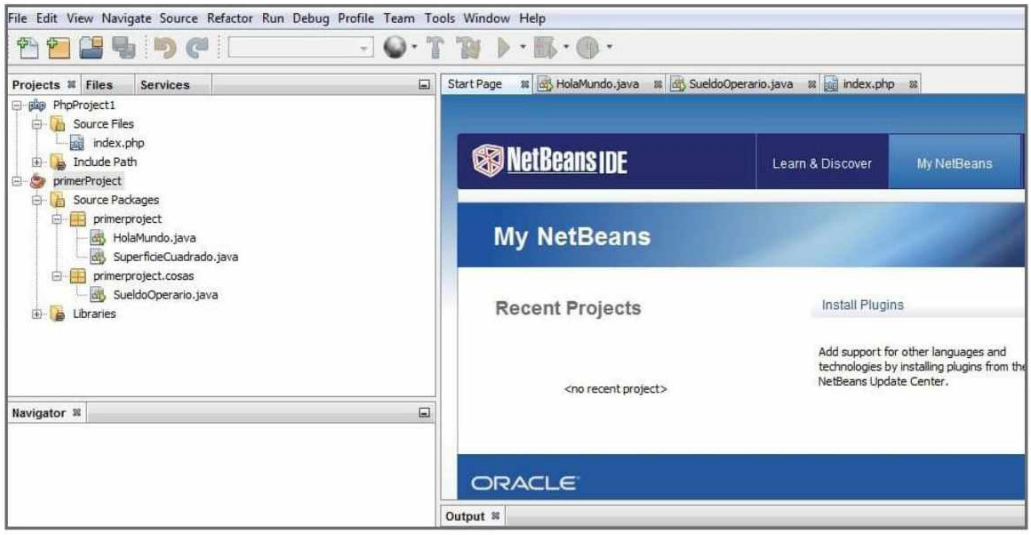


Figura 7. Vista de la organización de las carpetas de nuestro proyecto.

La organización de los archivos en carpetas y la presencia de elementos adicionales dependen del entorno de desarrollo que utilicemos. Además, Java introduce un esquema organizativo a través de paquetes (**packages**).

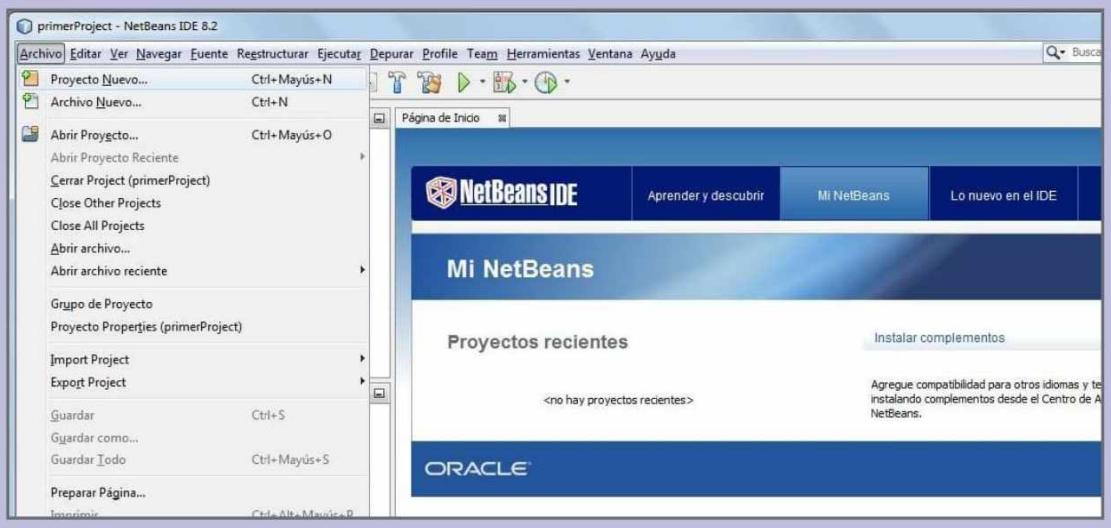
Comentarios iniciales

Si bien los comentarios no son de carácter obligatorio, es recomendable realizarlos; la comunidad de programadores lo recuerda cada vez que subimos un programa a la Web con el propósito de que sea corregido. Algunas organizaciones requieren que todos los programas comiencen con un comentario que explique el objetivo del programa, el autor, la fecha y la hora de la última modificación.

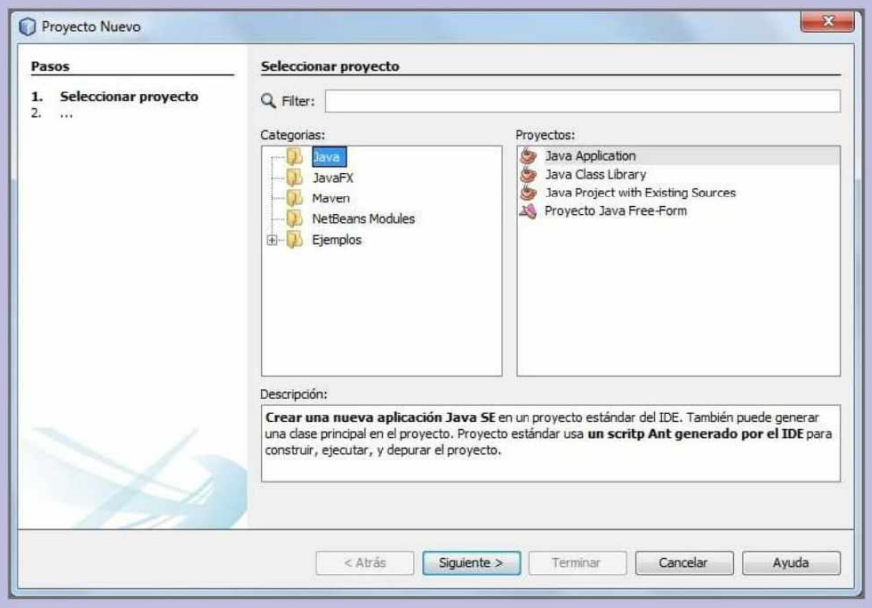
Para crear un proyecto en nuestro IDE, debemos seguir las indicaciones que presentamos en el siguiente **Paso a paso**:

PASO A PASO: CREACION DE UN PROYECTO EN NETBEANS

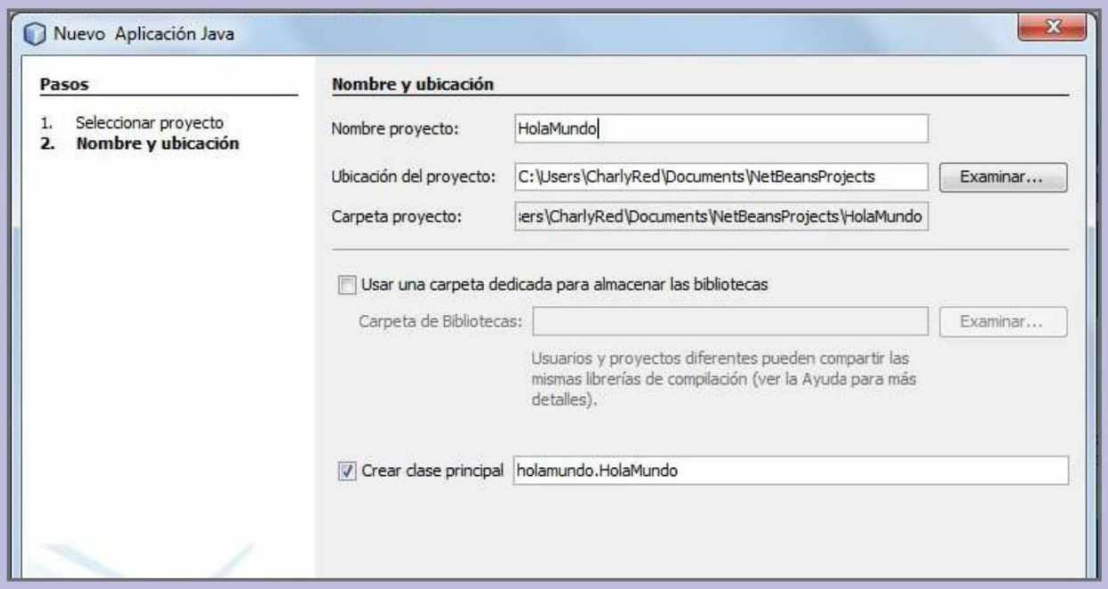
01 Para comenzar debe ejecutar el IDE. Una vez iniciado, haga clic en el menú **Archivo** y elija **Proyecto nuevo**.



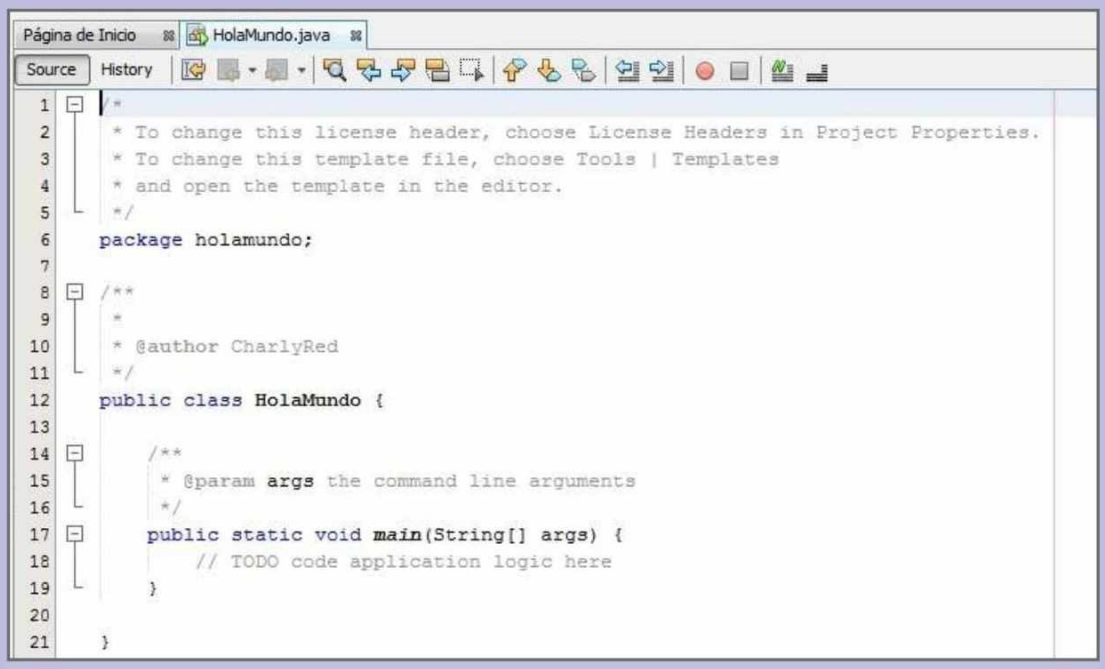
02 En la ventana **Proyecto nuevo** se presentan varias carpetas. Elija **Java;** luego, haga clic en la opción **Java application** y presione el botón **Siguiente**.



03 En la casilla adecuada escriba el nombre del proyecto, por ejemplo **HolaMundo**. También puede utilizar el botón **Examinar** e indicar una ruta diferente para almacenarlo.



04 Haga clic sobre el botón **Terminar**, con esto habrá creado su proyecto y se presentará el código que corresponde a **HolaMundo.java** en la sección principal del IDE.



Con el proyecto ya creado escribiremos el siguiente código; tengamos en cuenta que el IDE ya agregó algunos bloques de código en forma predeterminada, por lo tanto, no es necesario escribirlos.

```
//Bienvenidos a mi primer programa
// Programa para imprimir texto

public class HolaMundo{
//el método main empieza la ejecución del programa
   public static void main(String[] args){
          System.out.println("Hola Mundo Java");

          }//fin del bloque main
}//fin de la clase HolaMundo
```

Para ejecutar el programa que acabamos de codificar, debemos presionar el pequeño signo verde que representa **play** o **ejecutar**, situado en la barra de herramientas del IDE; también podemos hacerlo presionando la tecla **F6**. Luego de ejecutarlo deberíamos ver el mensaje **Hola Mundo Java** en la consola inferior.

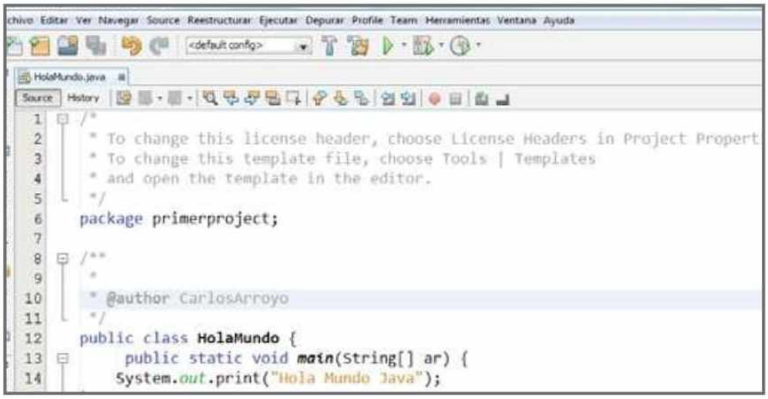


Figura 8. En la imagen se muestra el botón de ejecución del programa.

Sangrías y tabulador

Muchos IDE insertan las sangrías en los lugares adecuados, pero también podemos usar la tecla **TAB** para ampliarlas. La mayoría de los IDE dan la posibilidad de realizar la configuración de los tabuladores de tal forma que se inserte un número específico cada vez que oprimamos dicha tecla, ordenando nuestro código. Además de las sangrías, una buena práctica es integrar líneas en blanco, pues facilitan la lectura de nuestros programas.

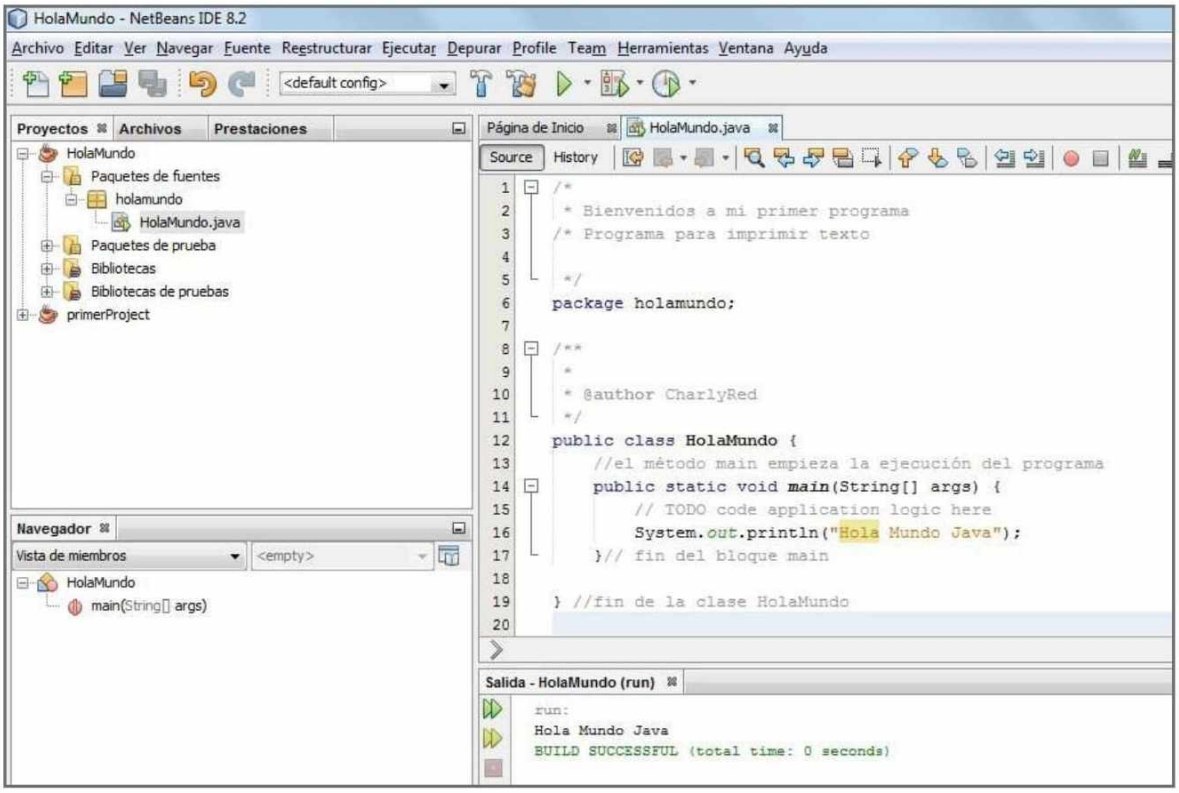


Figura 9. En esta imagen observamos la ejecución del programa **Hola Mundo** en la consola que se ubica en la parte inferior del IDE.

Siempre es necesario considerar que, si el programa que escribimos presenta errores de algún tipo, el IDE lo marcará con un signo que indique la línea que contiene el error. Para saber a qué tipo de error nos enfrentamos, bastará con pasar el cursor por sobre este signo. Es importante corregir los errores que aparezcan, de lo contrario el programa no se compilará y, por lo tanto, no podrá ejecutarse.

Aplicación Java

Una **aplicación Java** es un programa de computadora que se ejecuta cuando utilizamos el comando **java** para iniciar la máquina virtual de Java (JVM), que se encargará de compilarlo y ejecutarlo debidamente. Debemos saber que, en el funcionamiento de estas aplicaciones, intervienen, en forma directa o indirecta, muchos archivos, además de las bibliotecas que pueden ser usadas u omitidas, dependiendo de la complejidad del programa.

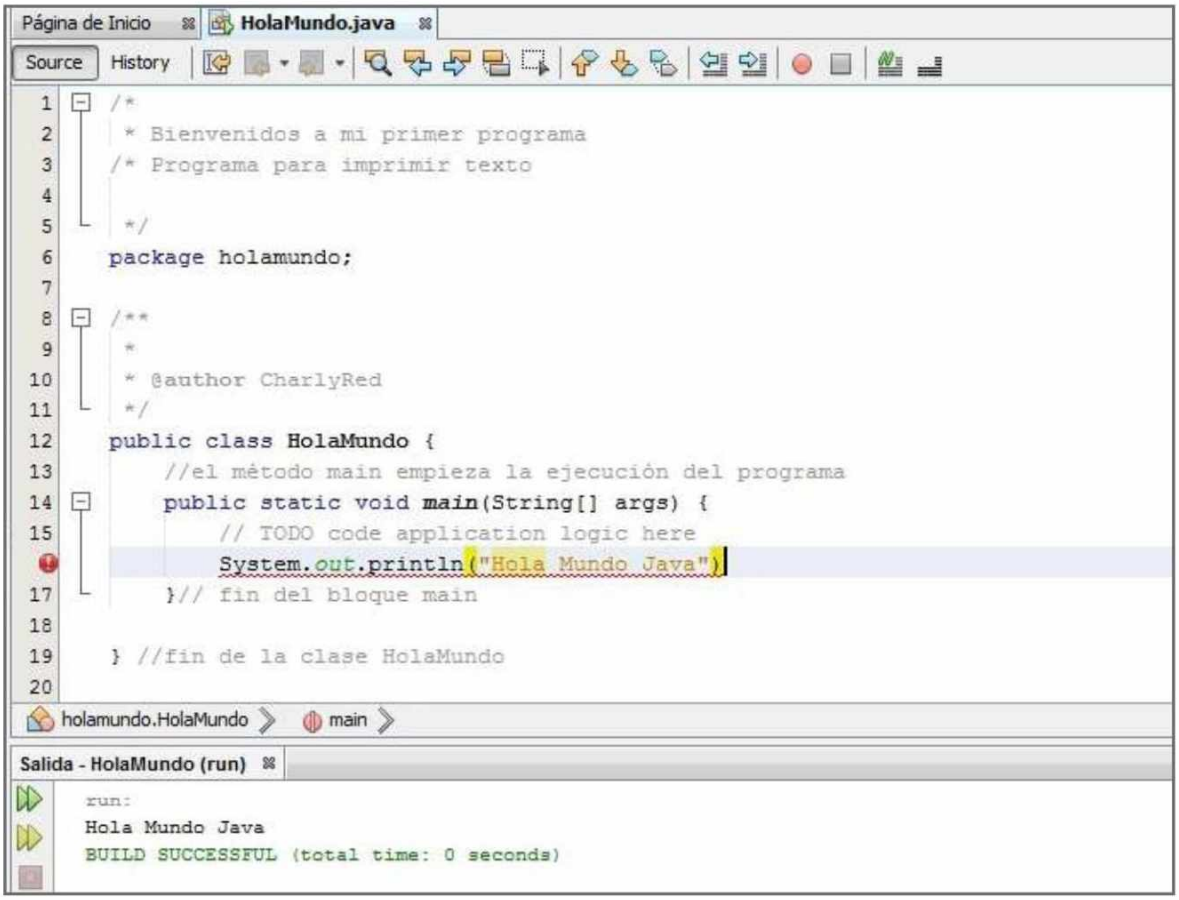


Figura 10. Como vemos en esta imagen, el IDE ha detectado un error en la fila 16. En este caso se trata de la falta del punto y coma (**;**) al final de la línea.

RESUMEN CAPÍTULO 02

En este capítulo aprendimos cómo instalar el JDK, revisamos algunos de los entornos de desarrollo disponibles para trabajar con Java y vimos la forma en que podemos instalar NetBeans, el IDE que usaremos en este curso. Nos adentramos en el conocimiento de las reglas básicas que debemos tener en cuenta antes de comenzar a codificar, conocimos la forma adecuada de agregar comentarios y también creamos nuestro primer proyecto. Para terminar, vimos de qué manera el IDE nos indica la presencia de un error en el programa creado.

Actividades 02

Test de Autoevaluación

1. Descargue el JDK e instálelo.
2. Descargue el IDE NetBeans desde su sitio oficial e instálelo.
3. ¿Qué ventajas proporciona utilizar un entorno de desarrollo integrado?
4. ¿Qué es un programa de Java?
5. ¿Para qué sirven las palabras clave?
6. ¿De qué forma se crea un proyecto Java?
7. ¿Por qué es tan importante la sintaxis en un lenguaje de programación?
8. ¿Cómo reconocemos un bloque?
9. ¿Qué signo debemos colocar al final de una instrucción?
10. ¿Qué importancia tiene main() en un programa de Java?

Ejercicios prácticos

1. Ejecute NetBeans y explore el contenido de la ventana de proyectos.

2. Codifique una pequeña aplicación que muestre su nombre, su edad y el país en el que reside. Agregue diversos comentarios que expliquen el funcionamiento de la aplicación y los datos que muestra. Ejecútela.

3. Provoque algunos errores de sintaxis en la aplicación recién creada y verifique qué recomienda el IDE para corregirlos.

Elementos del Lenguaje JAVA

En este capítulo conoceremos más sobre el lenguaje Java, analizaremos sus elementos fundamentales y cómo funcionan dentro del código así como también su interacción e importancia. Veremos las variables y las constantes, el uso de las expresiones regulares, los operadores y también los métodos o funciones, todo esto nos permitirá crear programas más avanzados.

03

TIPOS DE DATOS

Los tipos de datos están asociados a las variables, es decir, determinan lo que la variable puede contener y las operaciones que se pueden realizar con ella.

Java es un lenguaje orientado a objetos que proviene de C, por ello no se ha desprendido de elementos como los **tipos primitivos**, que precisamente no son objetos. Por esta razón tenemos dos categorías de datos en Java, los **primitivos** y los **referenciados**.

Los datos primitivos contienen un solo valor e incluyen enteros, punto flotante y caracteres. En la **Tabla 1** vemos los tipos primitivos soportados hasta la versión 8 de Java.

TIPOS DE DATOS PRIMITIVOS

DATOS PRIMITIVOS	DESCRIPCIÓN	RANGO DE VALORES
long	Representa un número entero real. Usa 8 Bytes o 64 bits.	±9.223.372.036.854.775.808
int	Representa un número entero real. Usa 4 Bytes o 32 bits.	± 2.147.486.647
short	Representa un número entero real. Usa 2 Bytes o 16 bits.	-32768 a 32768
byte	Representa un número entero real. Usa 1 Byte u 8 bits.	-128 a 127
double	Representa un número de punto flotante; es compatible con una variable con un entero real, pero no lo es con **double**. Usa 8 Bytes o 64 bits.	1.2345e300d, -1.2345e-300f, 1e1d
float	Similar al **double**. Usa 4 Bytes o 32 bits. Debemos usar una f después del número.	1.23e100f, -1.23e-100f, .3ef, 3.14f
boolean	Solo puede contener **true** o **false**. Se usa para condiciones lógicas. Usa 2 Bytes.	True o false
char	Representa un número, letra o símbolo según la tabla ASCII. Es compatible con un número entero real. Si es un carácter, va entre comillas simples. Usa 2 Bytes o 16 bits.	caracteres
String	No es un tipo primitivo, pero Java da soporte para manejar las cadenas de caracteres.	cadenas

Tabla 1. Tipos de datos primitivos usados en Java.

Los tipos de datos referenciados no están integrados en el sistema operativo; en realidad se trata de objetos de Java. Entre ellos encontramos las cadenas de caracteres o datos de tipo String, Array y Object, etcétera.

Podemos identificarlos porque, en el momento de declararlos, se utiliza una palabra clave para hacerlo; esta debe comenzar con mayúscula. Veamos un ejemplo para los tres tipos de datos que mencionamos:

```
String unaCadena = "Hola Mundo Java";
Array unArreglo;
Object C;
```

ENTRADA Y SALIDA DE DATOS

En Java, la entrada y salida (I/O) se basa en el concepto de **flujo de datos**, que es una secuencia ordenada de datos con una fuente o flujos de entrada y un destino o flujos de salida.

Java nos proporciona clases de E/S que se encargan de aislar los detalles específicos de su funcionamiento y hacen posible acceder a recursos del sistema por medio de archivos. Para la lectura de estos datos usamos las clases Scanner y BufferedReader.

BigInteger en Java

La librería **BigInteger** de Java nos permite procesar números grandes con sus propios métodos; para usarla debemos importar la librería **math.BigInteger**, luego creamos un objeto BigInteger mediante el código **BigInteger A = new BigInteger("1");**. De esta forma es posible ingresar, como argumentos, números grandes como cadenas, datos de tipo **long**, o simplemente leerlos desde el teclado.

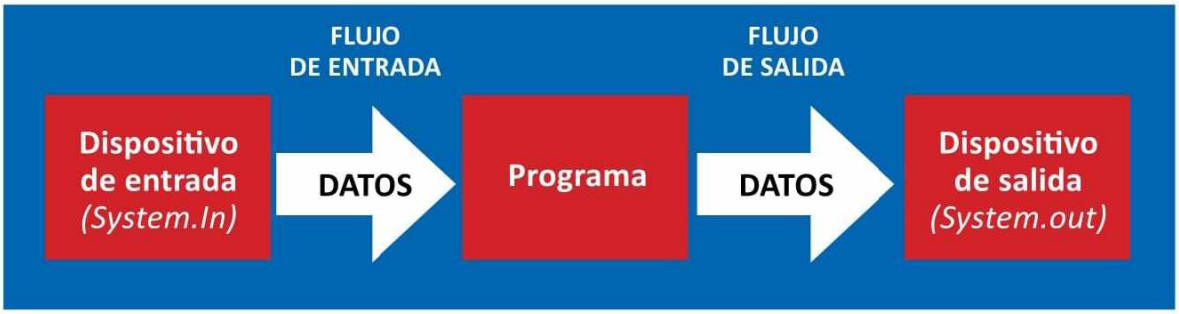


Figura 1. Esta imagen nos presenta un diagrama del flujo de datos en sistemas de información.

Cuando programamos en Java, es posible mostrar información a través de la **consola** o mediante un **cuadro de diálogo**.

En el primer caso es necesario ejecutar el IDE, pues la información que deseamos mostrar se presentará en una ventana de comandos o consola. Por ejemplo, para mostrar un mensaje en consola, escribimos el siguiente código:

```
System.out.println("esto es una cadena de texto");
```

Además de cadenas, podemos mostrar el valor de una variable, para lo cual escribimos el nombre de la variable sin comillas dobles, no importa el tipo de variable de la que se trate. Veamos un ejemplo, en este caso mostraremos el valor almacenado en la variable **num**:

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

También se puede mostrar información que incluya una cadena de texto y el valor de una variable; para hacerlo escribimos la cadena con comillas dobles, un signo **+** y el nombre de la variable. Por ejemplo:

```
System.out.println("El precio final es "+ precio +
" Pesos");
```

Si necesitamos escribir una fórmula, es necesario ponerla entre paréntesis, de la siguiente manera:

```
  System.out.println("El precio del producto, incluyendo
el IVA, es "+(precioProducto+(precioProducto*IVA)));
```

Por otra parte, si necesitamos que el resultado no dé un salto de línea, escribimos lo siguiente:

```
  System.out.print("texto sin salto de línea");
```

A continuación presentamos un ejemplo integrador que nos permitirá observar la manera de presentar información en diferentes casos:

```
  public class SalidaDatos {

      public static void main(String[] args) {

          int precio=100;
          final double IVA=0.21;
          System.out.println("Información del producto");
          System.out.println("El precio del producto es
"+precio);
          System.out.println("El precio del producto, in-
cluyendo el IVA, es "+(precio+(precio*IVA)));
      }
  }
```

Uso de sangrías en los programas

La indentación hace referencia a los espacios que vamos a dejar en el código respecto a la posición izquierda, es decir, la parte que se encuentra en el inicio de las líneas. Uno de los conceptos importantes para los iniciados en Java es saber colocar la indentación (sangrías) y saber respetar los niveles con los que se van colocando los bloques de código.

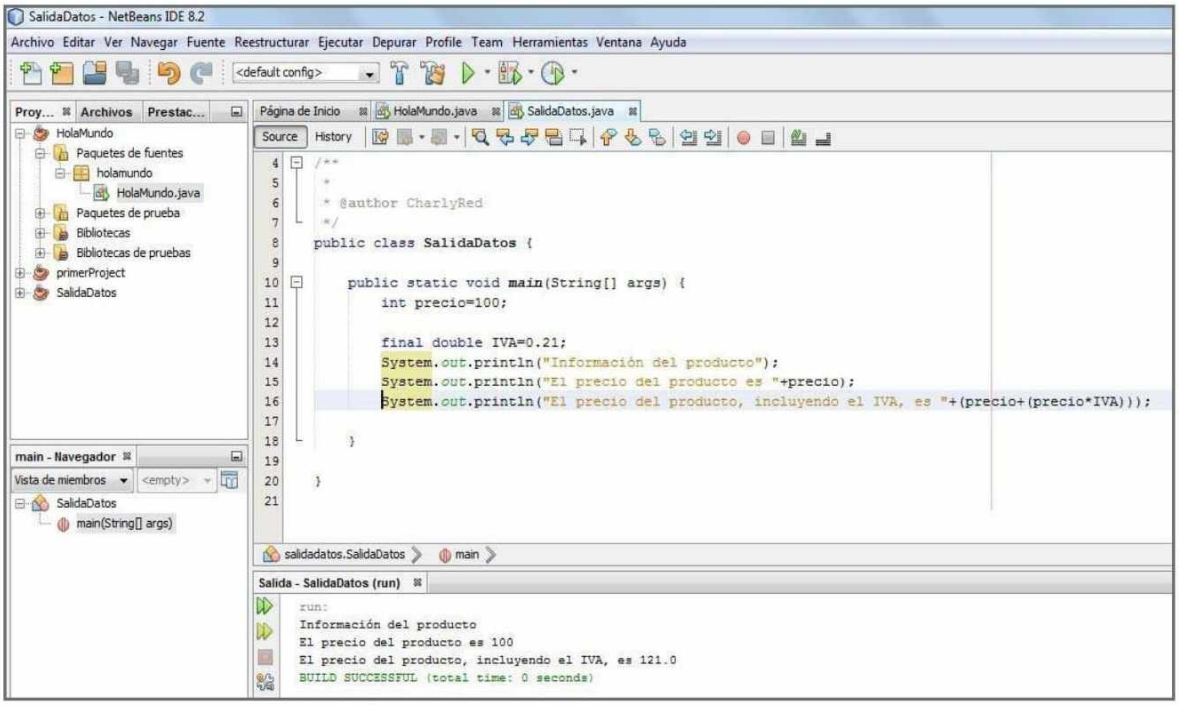


Figura 2. Aquí vemos el resultado de la ejecución del código de ejemplo. Este resultado se presenta en la consola de NetBeans.

La segunda forma de mostrar información es utilizando cuadros de diálogo. Analizaremos esta práctica en detalle más adelante, **cunado** expliquemos el uso de las interfaces gráficas mediante **Java Swing**.

VARIABLES

Una **variable** es una estructura de datos que puede variar a lo largo de la ejecución de un programa. Ocupa un espacio en la memoria y siempre referenciará a un tipo de dato primitivo o a cualquier objeto que haya sido creado en nuestro programa.

Antes de usar una variable, es necesario declararla; esto significa que debemos asignarle el tipo de dato que usará.

Además de declararla, es opcional inicializarla, es decir, asignarle un valor, que debe ser compatible con el tipo de dato que hemos declarado. Por ejemplo, si declaramos una variable de tipo numérico, esta no puede contener valores que correspondan a una cadena de caracteres.

La sintaxis adecuada para declarar una variable es **tipo_variable nombre_variable;**. Veamos un ejemplo:

```
int num;
```

Es recomendable inicializar estas variables con un número genérico, por ejemplo, uno o cero:

```
int num= 0;
```

Antes de crear nuestras variables, debemos tener en cuenta algunas reglas importantes:

- El nombre de una variable no puede comenzar con un número, pero sí puede formar parte del nombre.
- Los espacios no pueden formar parte del nombre.
- No se deben utilizar operadores como +, -, !, etcétera.
- No es posible crear dos variables con el mismo nombre en el mismo ámbito, aunque sean de distintos tipos.
- Para nombrar una variable, está prohibido el uso de palabras reservadas; en el IDE estas palabras se muestran en color azul.
- Debemos tener cuidado con las mayúsculas y las minúsculas, no es lo mismo mivariable que miVariable.

Nombres de variables

Debemos escribir las variables con nombres significativos, así se evitan largas documentaciones. En caso de que los nombres se tornen complejos, no olvidemos dejar un espacio entre cada declaración para agregar un comentario.

Ámbito

El **ámbito** se establece en el momento de crear una variable y se relaciona con la posibilidad de acceder a la variable desde cualquier parte de un programa.

Por norma, la variable tiene validez dentro del bloque en el que está encerrada entre llaves ({}) y depende del lugar donde fue declarada, por ejemplo, dentro de una función o dentro de una estructura de control (**if... else**).

Para aclarar este punto, es necesario que revisemos el siguiente código:

```
  public class AmbitoDeUnaVariable {
  public static void main(String[] args) {
   if (true) {
   int y=5;
  }
    System.out.println(y);
  }
}
```

Al ejecutar el código presentado, el compilador nos dirá que hay un error pues se intenta acceder a la variable **y**, pero no está definida puesto que fue declarada en un bloque distinto, fuera de donde intentamos utilizarla.

Para acceder a las variables, debemos considerar si en la clase está declarada como **public** o como **private**. Las variables declaradas como **public** se acceden directamente a través de **NombreClase.nombreVariable**;

Nombrar variables

Por convención, se suele escribir el nombre de las variables en minúscula, si esta tuviera más de una palabra, la siguiente debería empezar en mayúscula; también se puede usar guión bajo (_) para separarlas. Algunos ejemplos de variables correctamente nombradas son los siguientes: **variablePrueba** o **variable_prueba**. Estas reglas nos permitirán tener un código más ordenado y fácil de leer.

en cambio, una variable **private** solo puede ser utilizada mediante los métodos de esa clase.

Desde la declaración de cualquier función propia de una clase, podemos acceder a las variables internas de esa clase en forma directa. Es posible realizar esta declaración en el bloque de código de una clase, en el bloque de código de una función o también en un bloque de código que se encuentra en el interior de una función. Esta ubicación es importante pues solo podrá utilizarla el código que corresponde al bloque donde declaramos la variable. Si el mismo bloque de código se ejecuta varias veces durante la ejecución de la función, por ejemplo en el caso de un bucle **while**, la variable se creará con cada iteración del bucle. En este caso, la inicialización de la variable es obligatoria.

No podemos tener dos variables con idéntico nombre en el mismo ámbito, pero existe la posibilidad de declarar una variable interna a una función o un parámetro de una función con el mismo nombre que una variable declarada al nivel de la clase. En este caso, la variable declarada al nivel de la clase queda oculta por la variable interna de la función.

Nivel de acceso

El ámbito de las variables se combina con su nivel de acceso; esto determinará qué fragmento de código va a ser leído y escrito en la variable.

Un conjunto de palabras clave nos permiten controlar el nivel de acceso de una variable; estas se utilizan en la declaración y deben ser ubicadas delante del tipo de datos que corresponde a la variable. Estos niveles de acceso solo se usan para efectuar la declaración de una variable en el interior de una clase, por lo tanto, no se pueden usar en el interior de un método. A continuación, listaremos los niveles de acceso de las variables:

private	Donde la clase está definida es donde se utilizará la variable. `private String nombre;`
protected	La variable se utiliza en la clase donde está definida, en las subclases de esta clase y en las clases que forman parte del mismo paquete. `protected String codigo;`
public	La variable es accesible desde cualquier clase sin importar el paquete. `public String nombre;`
ningún modificador	La variable es accesible desde todas las clases que forman parte del mismo paquete. `int num1, num2;`
static	Se usa para transformar una declaración de variable de instancia en declaración de variable de clase (utilizable sin que exista una instancia de la clase). `static float impuesto;`

Ciclo de vida

El **ciclo de vida** de una variable nos dirá el tiempo que ella estará disponible en un programa. Para una variable que ha sido declarada en una función, el ciclo de vida de la variable terminará cuando la función deje de ejecutarse. En cuanto termine la ejecución del procedimiento o de la función, la variable se eliminará de la memoria y volverá a ser creada con la próxima llamada a la función.

Una variable declarada en el interior de una clase puede ser utilizada mientras esté disponible una instancia de la clase, pero las variables declaradas con la palabra clave **static** serán accesibles todo el tiempo.

CONSTANTES

A diferencia de las variables, las **constantes** son valores que no se pueden modificar durante la ejecución del programa.

Su mayor utilidad se relaciona con la posibilidad de definir valores que siempre serán iguales durante la ejecución del programa; para cambiar su valor deberá hacerse directamente en el código. Ejemplos de constantes son el IVA o el valor de PI.

En Java, cualquier tipo de dato puede ser una constante. Para definirla debemos escribir la palabra **static final** antes del tipo de dato, y luego, el nombre de la constante en mayúsculas.

Para entender cómo se implementan las constantes, veamos un ejemplo sencillo:

```
  public class UsoDeConstantes {

      public static void main(String[] args) {
            static final double IVA = 0.21;
            int producto=400;
            double resultado= producto*IVA;

          System.out.println("El IVA del producto es "+re-
sultado);
      }
  }
```

Clases en Java

Una **clase** es la unidad fundamental de programación en Java. Se trata de una especie de plantilla o molde en el que escribiremos los atributos y el comportamiento de los objetos. Cuando ejecutamos un programa, este crea y manipula los objetos en concreto, siguiendo el modelo de la **Programación Orientada a Objetos** o **POO**, que es uno de los principales paradigmas de desarrollo de software en la actualidad.

En el código anterior hemos declarado la constante **IVA**. Notemos que difiere de la declaración de variables en dos puntos: escribimos la palabra **static final**, y el nombre de la constante se recomienda en mayúsculas. Fuera de estas consideraciones iniciales, el uso de las constantes es idéntico al de las variables.

OPERADORES

Se trata de palabras clave del lenguaje, que nos permiten ejecutar operaciones con el contenido de ciertos elementos, por ejemplo, variables, constantes, valores literales o retornos de funciones.

Si combinamos uno o varios operadores y otros elementos, formaremos una **expresión**. Estas expresiones se evalúan en el momento en que se ejecutan, teniendo en cuenta los operadores y los valores asociados.

Java nos ofrece dos tipos de operadores: **unarios** (usan un solo operando) y **binarios** (utilizan dos operandos).

Otro punto por considerar es que el operado puede ir antes o después de la variable, es decir, en forma prefijada o posfijada. Veamos un ejemplo:

```
int i;
i=2;
System.out.println(i++);
```

Este código mostrará **2** como resultado, pues el incremento se realiza después de la utilización de la variable con la instrucción **println**. Sin embargo, en el siguiente ejemplo resultará algo distinto:

```
int i;
i=2;
System.out.println(++i);
```

Este código mostrará **3** como resultado, porque el incremento se ejecuta antes que la instrucción **println**.

En Java podemos dividir los operadores en siete categorías principales:

OPERADORES UNARIOS

En esta categoría encontramos los siguientes operadores:

-	valor negativo
~	complemento a uno
++	incremento
--	decremento
!	negación, solo se puede utilizar en las variables del tipo **boolean** o en comparaciones del tipo booleanas

OPERADORES DE ASIGNACIÓN

El operador **=** es el único en esta categoría. Se lo puede combinar con un operador aritmético lógico o binario, por ejemplo,

x+=2	que es equivalente a **x=x+2**.

OPERADORES ARITMÉTICOS

Nos permiten efectuar cálculos en el contenido de las variables, encontramos las siguientes opciones:

+	suma
-	resta
*	multiplicación
/	división
%	módulo o resto entero de la división

OPERADORES BIT A BIT

Operan solo con enteros y trabajan a nivel de bit.

&	y binario
|	o binario
^	o exclusivo
>>	desplazamiento a la izquierda (división por 2)
<<	desplazamiento a la derecha (multiplicación por 2)

OPERADORES DE COMPARACIÓN

Se trata de operadores utilizados en las estructuras de control. Devolverán un valor de tipo **boolean** en función del resultado de la comparación efectuada; luego la estructura de control utilizará este valor.

==	igualdad
!=	desigualdad
<	menor que (variante **<=**)
>	mayor que (variante **>=**)
instanceof	comparación del tipo de variable con el tipo indicado

OPERADORES DE CONCATENACIÓN

El operador **+** (más) utilizado en la suma se usa para unir cadena de caracteres o también para unir un String a una variable, por ejemplo:

```
System.out.println("El precio del producto es "+precio);
```

OPERADORES LÓGICOS

Se trata de operadores que permiten combinar las expresiones en estructuras condicionales o en estructuras de iteración:

&	y lógico
|	o lógico
^	o exclusivo
!	negación
&&	y lógico; a diferencia del **&** simple, este operador evaluará solo si la primera condición es falsa
||	o lógico; a diferencia del **|** simple, este operador evaluará solo si la primera condición es falsa.

División por cero

Un error que sabremos subsanar es la división por cero; se trata de un típico error en tiempo de ejecución. Cuando se produce una excepción, se muestra en la pantalla un mensaje de error y finaliza la ejecución del programa.

Orden de evaluación de los operadores

Es importante considerar el orden en que se ejecutarán los operadores cuando se encuentran combinados, pues se evalúan en un orden muy preciso.

Los incrementos y decrementos prefijados se ejecutan primero; luego, las operaciones aritméticas; le siguen las operaciones de comparación; después, los operadores lógicos y, finalmente, las asignaciones.

Los operadores aritméticos poseen un orden de evaluación específico dentro de una expresión, veámoslo a continuación:

- Negación (-)
- Multiplicación y división (*, /)
- División entera (\)
- Módulo (%)
- Suma y sustracción (+, -), concatenación de cadenas (+)

Si las operaciones presentan paréntesis, existe prioridad para las expresiones dentro de ellos.

En una expresión, podemos utilizar tantos niveles de paréntesis como deseemos; sin embargo, es importante que la expresión contenga tantos paréntesis de cierre como de apertura, porque de lo contrario, el compilador generará un error.

Los literales

En Java, un **literal** es un valor que suele aparecer dentro de un programa; puede ser de tipo entero, real, lógico, carácter, cadena de caracteres o un null. Por ejemplo, **12**, **13.6**, **3.14161722D**, **"Users"**, **null**, **'@'**, **"Presione \"Enter\" para continuar"**. Hay que aclarar que los tipos enteros pueden expresarse en decimal (base 10), octal (base 8) y hexadecimal (base 16), mientras que los de tipo real pueden terminar en **E** seguida de un exponente (positivo o negativo), por ejemplo, **14E-3**.

EXPRESIONES REGULARES

Una **expresión regular** es un **patrón** que describe a una cadena de caracteres. Se trata de elementos que se usan desde hace años en otros lenguajes de programación, como Perl. Desde la versión 1.4 del JDK, se incluye el paquete **java.util.regex**, que proporciona una serie de clases para poder hacer uso de este tipo de expresiones en Java. Veamos algunos casos en donde se pueden utilizar:

- Cuando deseamos que una fecha ingresada por el usuario cumpla con un patrón específico, por ejemplo, **dd/mm/aaaa**.
- Para comprobar que un DNI está formado por 8 cifras, un guion y una letra, o que una dirección de correo electrónico es válida.
- Para asegurar que una contraseña cumple con determinadas condiciones o que una URL sea válida.

También podemos utilizar las expresiones regulares para comprobar cuántas veces una secuencia de caracteres determinada se repite dentro de la cadena. En este caso, el patrón se busca en el String de izquierda a derecha; cuando se determina que un carácter cumple con el patrón, este carácter ya no vuelve a intervenir en la comprobación.

Símbolos comunes

Dentro de las expresiones regulares podemos utilizar muchos símbolos; gracias a ellos es posible representar elementos u operaciones dentro de una expresión. Los conoceremos en la **Tabla 2**.

Simbología en Java

Mucha de la simbología que utiliza el lenguaje Java proviene de C y posteriormente de C++, por lo que, a los conocedores de ambos lenguajes, se les hará más fácil enfrentar la programación.

SÍMBOLOS COMUNES

EXPRESIÓN	DESCRIPCIÓN
.	El punto indica cualquier carácter.
^expresión	Indica el inicio de un String, en este caso debe contener la expresión al principio.
expresión$	Indica el final del String, en este caso debe contener la expresión final.
[abc]	Representa una definición de conjunto, el String debe contener **a**, **b** o **c**.
[abc][23]	El String debe contener **a**, **b** o **c**, seguidas de **1** o **2**.
[^abc]	Indica negación, en este caso el String debe contener cualquier carácter excepto **a**, **b** o **c**.
[a-z1-9]	Se trata de un rango. Indica las letras minúsculas desde la **a** hasta la **z** (ambas incluidas) y los dígitos desde el **1** hasta el **9** (ambos incluidos)
A|B	Es un **OR**, **A** o **B**.
AB	Concatenación, **A** seguida de **B**.

Tabla 2. Símbolos utilizados en expresiones regulares.

Por ejemplo, para encontrar la frase **'encuentro frase'**, utilizamos el siguiente código:

```
(\W|^)encuentro\frase(\W|$)
```

En este caso, **\W** coincide con cualquier carácter que no sea una letra, un número o un guion bajo. Impide que la expresión **regex** tenga en cuenta los caracteres que aparezcan antes o después de la frase.

En cambio, si buscamos una coincidencia con cualquier palabra o frase de la lista siguiente: **papa** – **batata** – **remolacha** – **zanahoria**, haremos esto:

```
(?i)(\W|^)(papa|batata|remolacha|zanahoria|\sla|frutas)
(\W|$)
```

Metacaracteres

Los **metacaracteres** también pueden formar parte de las expresiones regulares. Se trata de un conjunto de caracteres con formatos especiales, por ejemplo, alfanuméricos, dígitos y signos de puntuación, entre otros.

METACARACTERES

METACARÁCTER	DESCRIPCIÓN
\d	Dígito, equivale a **[0-9]**.
\D	No dígito, equivale a **[^0-9]**.
\s	Espacio en blanco, equivale a **[\t\n\x0b\r\f]**.
\S	No es espacio en blanco, equivale a **[^\s]**.
\w	Una letra mayúscula o minúscula, un dígito o el carácter _, equivale a **[a-zA-Z0-9_]**.
\W	Equivale a **[^\w]**
\b	Límite de una palabra.

Tabla 3. Metacaracteres utilizados en las expresiones regulares.

En Java debemos usar una doble barra invertida ****, por lo tanto, para utilizar **\w,** es preciso escribir **\\w**. Si vamos a indicar que la barra invertida es un carácter de la expresión regular, será necesario escribir ****.

Veamos un ejemplo de cómo se usan los metacaracteres:

```
  String a = "Ejemplo, de. separar- String+ por* caracter";
      //Para usar los caracteres puede usar esta expresión
regular:
      String[] parts = a.split("[\\W]");

      for(String i:parts){
          System.out.println("===" +i);
      }
```

Cuantificadores

Los **cuantificadores** son elementos utilizados para controlar la cantidad de repeticiones o coincidencias de un patrón que hemos definido previamente. Se especifican luego de los corchetes (**[ ]**) en los que predefinimos los caracteres permitidos que podemos inicializar, luego de haber indicado entre llaves (**{ }**) la cantidad de veces que puede repetirse el patrón por buscar.

CUANTIFICADORES

EXPRESIÓN	DESCRIPCIÓN
{X}	Indica que lo que va justo antes de las llaves se repite **X** veces.
{X,Y}	Indica que lo que va justo antes de las llaves se repite mínimo **X** veces y máximo **Y** veces. También podemos poner **{X,}** indicando que se repite un mínimo de **X** veces sin límite máximo.
*	Indica **0** o más veces. Equivale a **{0,}.**
+	Indica **1** o más veces. Equivale a **{1,}.**
?	Indica **0** o **1** veces. Equivale a **{0,1}.**

Tabla 4. Símbolos usados en cuantificadores.

Veamos un ejemplo sobre el uso de los cuantificadores, para ello crearemos un patrón **regex**, que nos permita buscar números hexadecimales:

```
0[xX][0-9a-FA-F]
```

En este caso deseamos encontrar un conjunto de caracteres en el que el primer carácter sea el **0**, el segundo carácter sea una **x** o una **X**, el tercer carácter sea un dígito de **0** a **9**, una letra de **a** a **f** o una mayúscula de **A** a **F**.

La clase String

Muchas veces **String** es confundido con un tipo de dato, sin embargo, no lo es. Se trata de uno de los casos extraños de este lenguaje, ya que, al ser una clase, vamos a utilizarla para tipificar las variables que contendrán una cadena de caracteres, es decir, textos. Por esta razón, la primera letra debe escribirse en mayúsculas.

MÉTODOS

En Java, un **método** es un conjunto de instrucciones definidas dentro de una clase; realiza una determinada tarea y puede ser invocado mediante un nombre. En esta sección aprenderemos cómo debemos declarar y utilizar métodos; de esta forma, podremos facilitar el diseño, la implementación, la operación y el mantenimiento de programas extensos. Veamos algunos ejemplos de métodos:

- **Math.pow()**: adecuado para hallar la potencia de un número.
- **Math.sqrt()**: para hallar la raíz de un número.
- **Character.isDigit()**: permite determinar si un carácter es dígito.
- **System.out.println()**: adecuado para imprimir resultados.

Es importante saber que, cuando se llama a un método, la ejecución del programa pasará al método; pero, cuando este acaba, la ejecución continuará a partir del punto donde se produjo la llamada.

Construcción de métodos

Como sabemos, la mayoría de los problemas son muy extensos, por lo tanto, las soluciones que podemos implementar a nivel de software también serán extensas. Para enfrentar esto, es posible crear programas modulares, pues, si dividimos un programa en pequeñas piezas o módulos, lograremos que un programa extenso sea más entendible. Las ventajas relacionadas con la modularidad y la construcción de métodos tienen que ver con la posibilidad de reutilizar el código (refactorización); esto permite que, en lugar de escribir el mismo código varias veces, solo hagamos una llamada al método adecuado.

En Java, un método siempre pertenece a una clase. Como sabemos, todo programa Java posee un método llamado **main**; se trata del punto de entrada al programa y, también, el punto de salida. Para construir métodos, hay que considerar que siempre debemos implementarlos fuera del **main**.

La estructura general de un método Java es la siguiente:

```
  [Modificador de acceso] [Otros modificadores] Valor_retorno
nombreMetodo (parámetros){
        Instrucciones
      [return valor;]
  }
```

Los elementos que aparecen entre corchetes son opcionales, veamos su sintaxis en detalle:

Modificador de acceso	Se trata de un elemento opcional que determina el tipo de acceso al método.
Valor de retorno	Este elemento se encarga de indicar el tipo del valor que devuelve el método; siempre hay que indicar este tipo de valor y, si el método no devuelve ningún valor, debemos indicarlo mediante **void**.
Nombre del método	Se trata del nombre que se le da al método. Para crearlo, hay que seguir las mismas normas que aprendimos para crear variables.
Parámetros	Se trata de un elemento opcional que se ubica después del nombre del método y siempre entre paréntesis. Estos parámetros o argumentos son los datos de entrada que recibe el método para operar. Un método puede recibir cero o más argumentos y, para cada argumento, debemos especificar su tipo. Los paréntesis son obligatorios, aunque estén vacíos.
Return	Se utiliza para devolver un valor. La palabra clave **return** va seguida de una expresión que será evaluada para saber el valor de retorno; esta expresión puede ser compleja o ser simplemente el nombre de un objeto, una variable de tipo primitivo o una constante. El tipo del valor de retorno debe coincidir con el tipo devuelto que se ha indicado en la declaración del método. Si el método no devuelve nada (**void**), la instrucción **return** es opcional.

Un método tiene un único punto de inicio, representado por la llave de inicio **{** . Su ejecución termina cuando se llega a la llave final **}** o cuando se ejecuta la instrucción **return**. Esta instrucción puede aparecer en cualquier lugar dentro del método, no necesariamente al final.

Los pasos que debemos seguir para implementar un método son los siguientes:

1. Describir lo que el método debe hacer.
2. Determinar las entradas correspondientes.
3. Definir los tipos de las entradas.
4. Determinar el tipo del valor de retorno.
5. Escribir las instrucciones que corresponden al cuerpo del método.
6. Probar el método mediante el diseño de distintos casos de prueba.

```
/*
 * Suma de dos números enteros
 */
package sumanumeros;
import java.util.*;
public class SumaNumeros {

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        int numero1, numero2, resultado;
        System.out.print("Ingrese el primer número: ");
        numero1 = entrada.nextInt();
        System.out.print("Ingrese el segundo número: ");
        numero2 = entrada.nextInt();
        resultado = sumar(numero1, numero2);
//sumar es el método
        System.out.println("Suma: " + resultado);
```

```
    }
     public static int sumar(int a, int b){
           int c;
           c = a + b;
           return c;
     }
}
```

Mientras escribimos el código, notaremos que **sumar** marcará un error; se trata de un método que aún no se ha implementado.

El método se llama **sumar** y recibe dos números enteros **a** y **b** como argumentos. En la llamada al método, los valores de las variables **numero1** y **numero2** se copian en las variables **a** y **b**; el método suma los dos números y guarda el resultado en la variable **c**. Finalmente, devuelve, mediante la instrucción **return**, la suma calculada.

Para indicar al método que no devuelva nada, debemos escribir **void** en lugar de **static**.

RESUMEN CAPÍTULO 03

En este capítulo hicimos una descripción completa de los elementos fundamentales que encontramos en el lenguaje Java, esto nos servirá para contar con la base de conocimientos que necesitamos para no tener falencias en nuestros códigos y también para entender para qué sirven los elementos en los bloques de código y que rol cumple cada uno de ellos. Además, vimos en detalle las expresiones regulares, su uso y los elementos que podemos utilizar para crearlas y, finalmente, realizamos una completa introducción a los métodos, que son muy importantes para el inicio de la construcción de las clases.

Actividades 03

Test de Autoevaluación

1. ¿Qué importancia tienen los tipos de datos en un programa?
2. ¿Cuándo es necesario el uso de las constantes?
3. ¿Qué son las expresiones regulares?
4. Mencione algunos ejemplos de cuantificadores.
5. ¿Qué son los literales en Java?
6. ¿Cómo hacemos para ingresar datos en Java?
7. ¿Qué importancia tienen las variables en la programación de Java?
8. ¿Qué es el ámbito de una variable?
9. ¿Para qué sirven los operadores de comparación?
10. ¿Qué ventajas presenta el uso de métodos?

Ejercicios prácticos

1. Cree un programa que solicite el ingreso de su nombre y su edad, y que imprima ambas entradas.
2. Instancie e inicialice algunas variables del tipo numéricas. Luego imprímalas junto con una cadena de caracteres.
3. Realice un programa que comience mostrando el número 100 y disminuya de dos en dos hasta mostrar 0.
4. Dados dos números enteros, haga que un programa realice las cuatro operaciones aritméticas.
5. Utilice métodos para crear un programa que permita ingresar tres valores por teclado, y luego muestre el mayor y el menor.

Estructuras de control y arreglos

En todos los lenguajes de programación encontraremos tres tipos de estructuras de control: secuenciales, selectivas o condicionales, y repetitivas. En este capítulo conoceremos estas estructuras y las aplicaremos al trabajo con Java. También, daremos los primeros pasos en el mundo de los arreglos o arrays, y de esta forma, dotaremos de mayor complejidad a nuestros códigos.

04

ESTRUCTURAS DE CONTROL

Las **estructuras de control** determinan la secuencia de ejecución de las sentencias de un programa, por lo tanto, las necesitamos para controlar el flujo de información dentro de los bloques de código.

Si en un programa tenemos una estructura que solo posee operaciones junto a entradas y salidas de datos, estamos frente a **estructuras secuenciales**, es decir, una instrucción del programa sigue a otra.

También existen **estructuras de selección** o decisión, en las que para ejecutarse alguna sentencia, es necesario que se cumplan una o varias condiciones. Por último, están las **estructuras de repetición** o iteración, que se dan cuando un proceso se repite (bucle) hasta que se cumpla cierta condición establecida para que dicho proceso finalice.

Estructuras secuenciales

Este tipo de estructuras constituyen las más sencillas y fáciles de entender, pues solo se componen de operaciones junto a elementos de entrada y de salida. Para comenzar a trabajar con estas estructuras, será necesario importar la clase **Scanner** pues esta nos permitirá obtener datos desde el teclado; esto lo haremos mediante la siguiente instrucción:

```
import java.util.Scanner;
```

Veamos un ejemplo completo de su uso:

```
// Importamos la clase Scanner
import java.util.Scanner;
 class EstructurasSimples1 {

 public static void main(String arg[]){

//declaramos las variables enteras
 int num1, num2;

// Se declara e inicializa una instancia de la clase
Scanner
```

```
    Scanner entrada=new Scanner(System.in);

     System.out.println("Ingrese el primer número: ");
     num1 = entrada.nextInt();
     System.out.println("Ingrese el segundo número: ");
     num2 = entrada.nextInt();
       int suma = num1 + num2;
       int producto = num1 * num2;

   System.out.println("La suma de los dos valores es: " +
suma);
   System.out.println(" Y el producto de los dos valores es:
" + producto);
    }

   }
```

En el código anterior se importa la clase **Scanner**; luego, se declara una instancia de ella, denominada **entrada**, y se inicializa pasando como parámetro un objeto **InputStream**, el que es devuelto por la clase **System** con ayuda de su método **in**.

Después se imprime el aviso correspondiente del tipo de datos que se van a pedir y se guarda todo lo que se escriba en la variable hasta que se pulse **ENTER**. Todo esto se logra mediante el objeto **Scanner**, llamado **entrada**, y su método **next()**.

En esta oportunidad ingresaremos los datos por consola, sin embargo, en la realidad se debe hacer a través de una caja de texto.

La clase Scanner

Para implementar la entrada de información por teclado, importaremos la clase **Scanner**, que proviene de la biblioteca de **java.util**. Luego debemos instanciarla (utilizando **new**), para crear un objeto que podamos utilizar en el programa. Recordemos que cada vez que se requiera el objeto creado, será necesario hacer el **casting** al tipo de dato correspondiente.

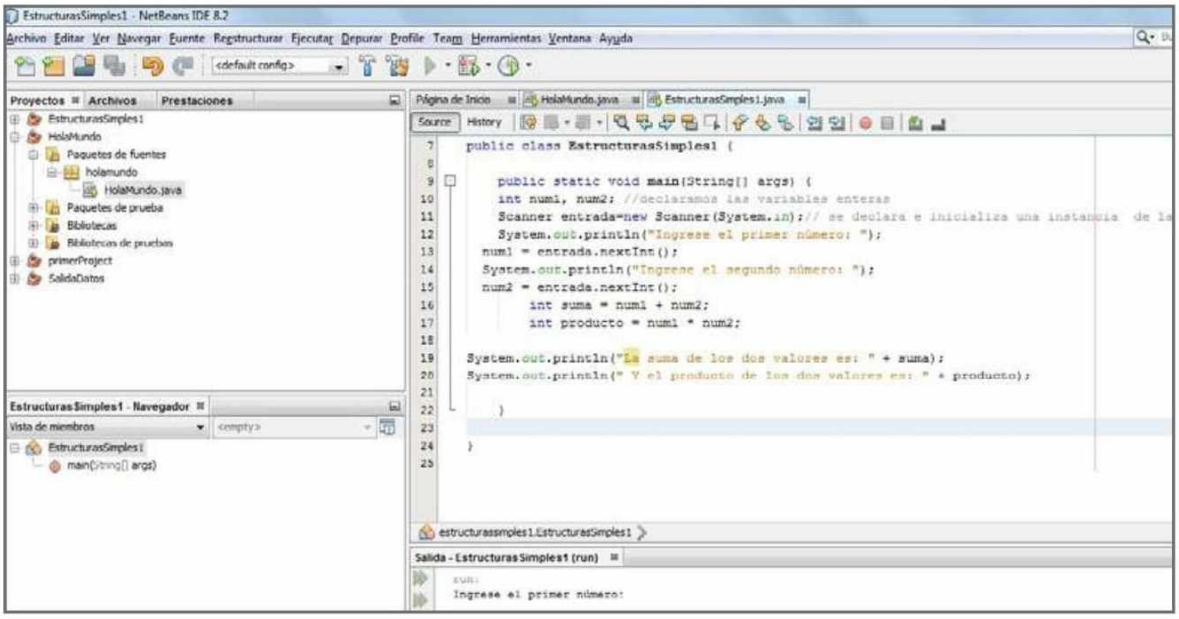


Figura 1. En esta imagen presentamos una vista del código de ejemplo en el IDE; observamos cómo nos expone la primera pregunta, esto se debe a la importación de la clase Scanner.

En la consola ingresamos los números que nos solicita el programa, por ejemplo, escribimos el número **5** y luego **6**; después podemos obtener el resultado, en este caso, directamente en la consola del IDE.

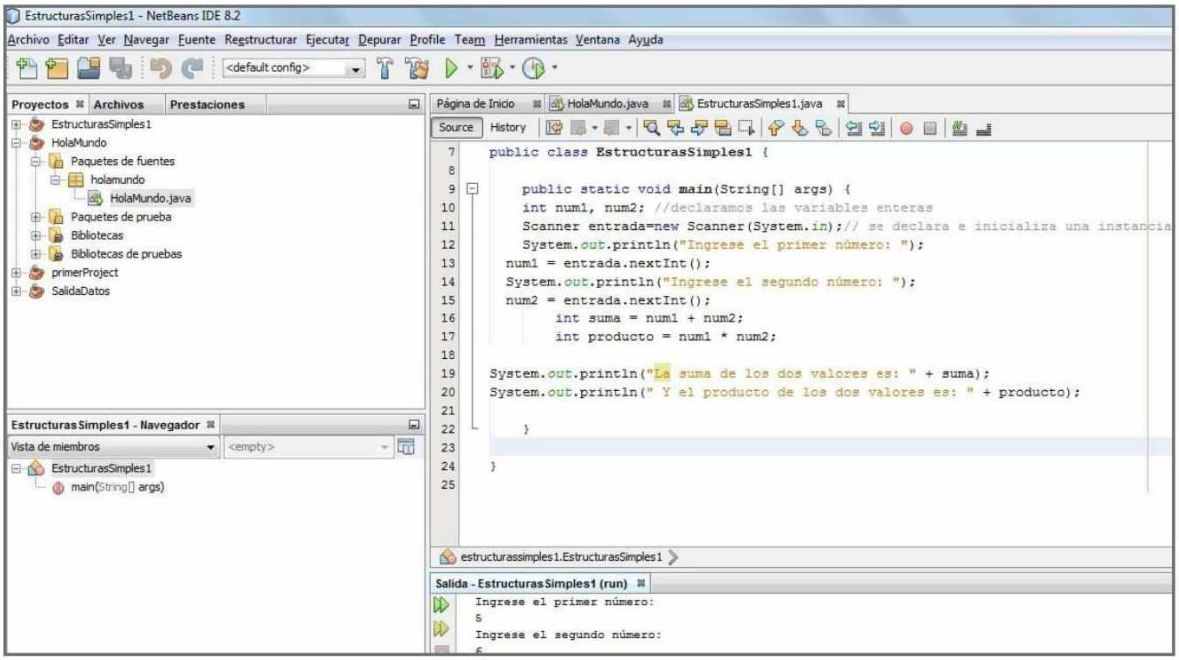


Figura 2. Una vez que ingresamos los números solicitados, en la consola aparecen los resultados con los strings prolijamente concatenados a las variables suma y producto guardadas.

Estructuras de selección

Las estructuras de selección o condicionales nos permiten ejecutar una serie de instrucciones si se cumple una determinada condición.

Es importante saber que la condición debe entregar un **resultado booleano**, por lo tanto, se usarán operadores relacionales y condicionales. Existen varios tipos de estructuras condicionales: **simples**, **compuestas** y **anidadas**.

Estructura condicional simple

Estas estructuras aparecen cuando se presenta una elección y debemos realizar una acción; en caso contrario, no se realizará ninguna acción.

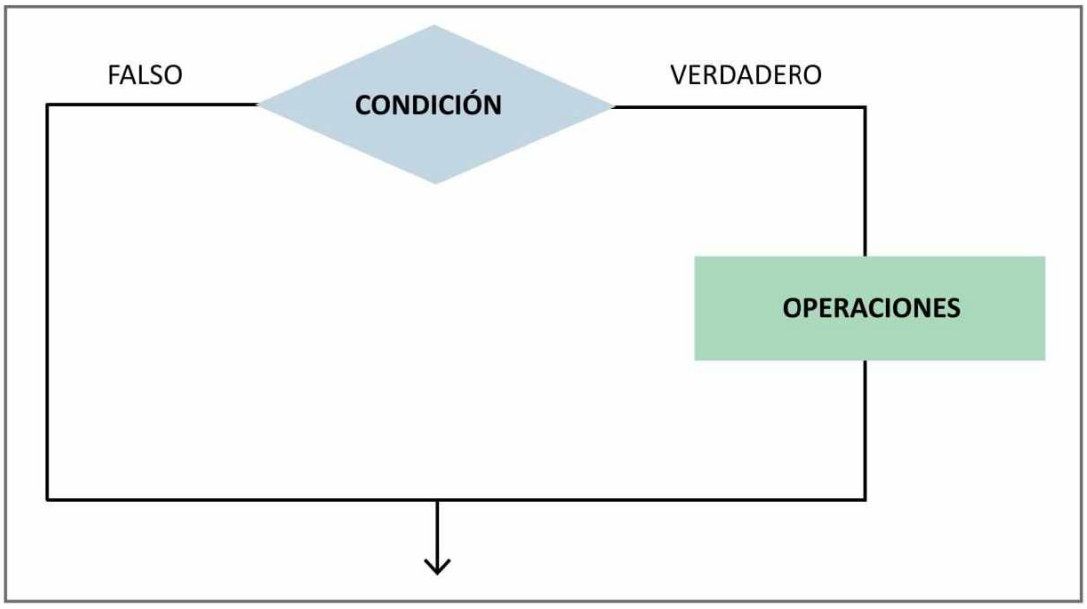


Figura 3. En esta imagen podemos apreciar una representación gráfica de las estructuras condicionales simples.

Flujo de ejecución

En los lenguajes de programación las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa. Con las estructuras de control se puede, de acuerdo con una condición, ejecutar un grupo u otro de sentencias (**If-Then-Else**). En Java el flujo de ejecución es lineal, es decir, se va ejecutando línea por línea y en el orden en que va encontrando las sentencias.

Estructura condicional compuesta

Aparece cuando se presenta una elección, pero esta vez podemos optar entre una acción u otra.

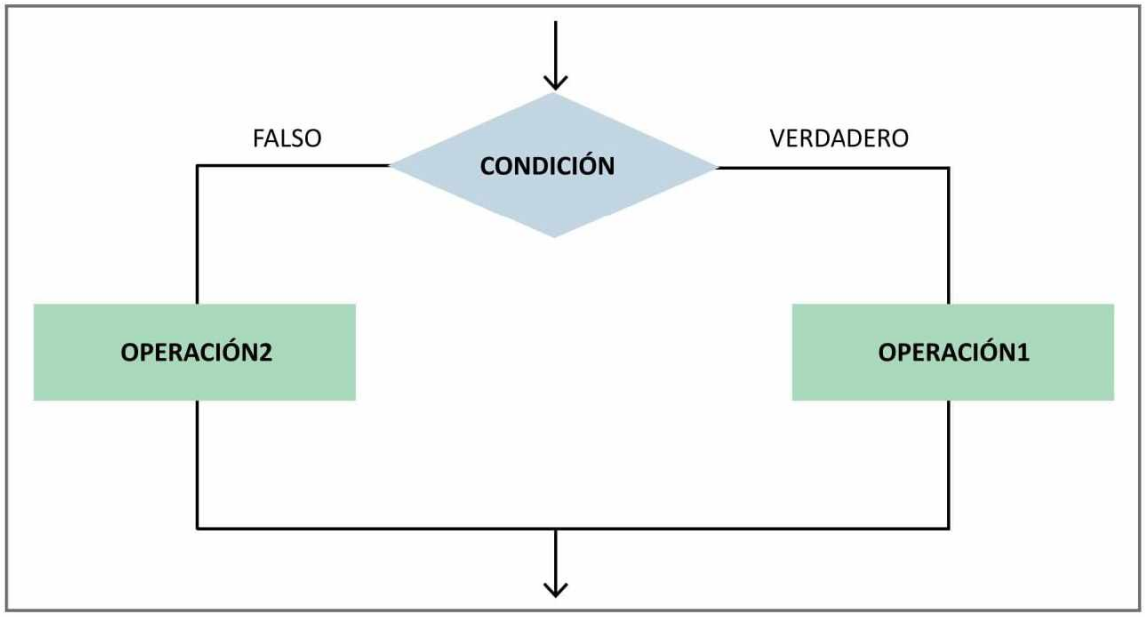


Figura 4. Representación gráfica de las estructuras condicionales compuestas.

Para continuar, veremos un código en el que el programa solicitará el ingreso de dos valores, luego deberá mostrar el mayor de ambos valores:

```
import java.util.Scanner;
class EstructurasCondicionalCompuesta1 {

 public static void main(String arg[]){

  int num1, num2;

  Scanner entrada=new Scanner(System.in);
  System.out.println("Ingrese el primer valor: ");
  num1 = entrada.nextInt();
  System.out.println("Ingrese el segundo valor: ");
  num2 = entrada.nextInt();
    if (num1>num2){
System.out.println(num1 +" es el mayor");
```

```
}else{
System.out.println(num2 +" es el mayor");
}
 }
}
```

Una vez que ejecutemos este código, dependiendo de los valores que ingresemos mediante el teclado, obtendremos el mayor de ellos.

Estructura condicional anidada

Una estructura es anidada cuando, en la estructura condicional del verdadero o falso, se agregan una o más estructuras condicionales.

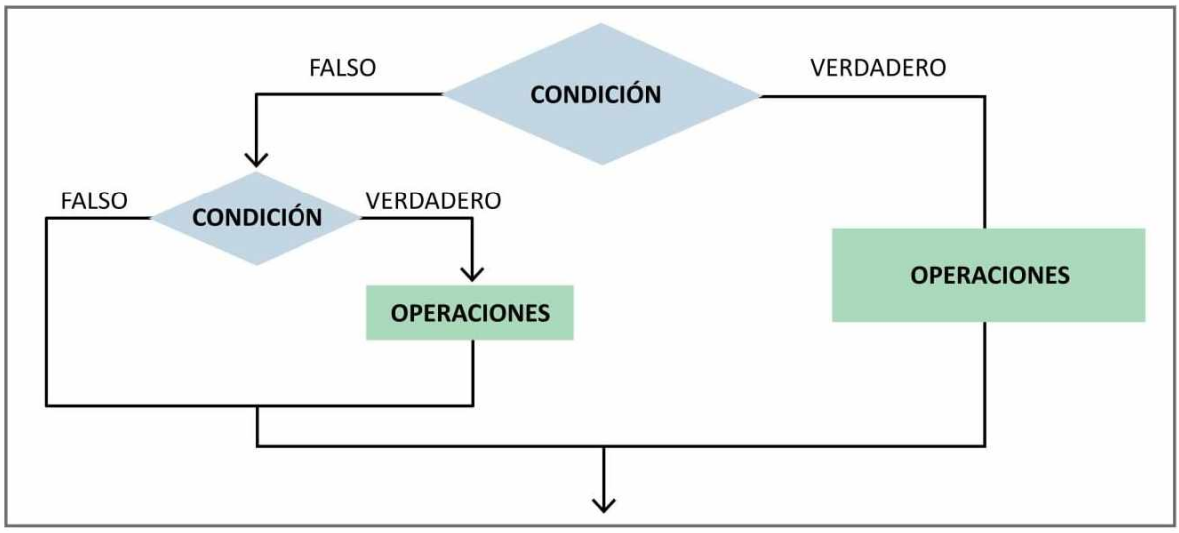


Figura 5. Representación gráfica de las estructuras condicionales anidadas.

Operador ternario

En informática un operador ternario (a veces incorrectamente llamado operador terciario) es un operador que toma tres argumentos. Este operador puede pasar varias líneas de código a una sola línea en lenguajes que puedan usarlo. Puede sustituir al **if else** que va tener una sintaxis resultado**=condicion ? valor_si_se_cumple : valor_si_no_se_cumple**, lo cual reduce de tres líneas a una.

Para ejemplificar este tipo de estructuras, crearemos un programa que pedirá el ingreso de tres notas de un estudiante, luego calculará el promedio e imprimirá alguno de los siguientes mensajes:

- Si el promedio es **>=7** mostrar **Aprobado**.
- Si el promedio es **>=4** y **<7** mostrar **Diciembre**.
- Si el promedio es **<4** mostrar **Desaprobado**.

```
import java.util.Scanner;
class EstructurasCondicionalAnidada1 {

 public static void main(String arg[]){

  int nota1, nota2, nota3;

  Scanner entrada=new Scanner(System.in);
  System.out.println("Ingrese la primera nota: ");
  nota1 = entrada.nextInt();
  System.out.println("Ingrese la segunda nota: ");
  nota2 = entrada.nextInt();
  System.out.println("Ingrese la tercera nota: ");
  nota3 = entrada.nextInt();
   float promedio=1.0f*(nota1+nota2+nota3)/3;
   if (promedio>=7){
    System.out.println("Aprobado");
    }else{
          if(promedio>=4){
          System.out.println("Diciembre");
    }else{
    System.out.println("Reprobado");
    }
    }
 }
}
```

Estructura condicional compuesta (if – else if)

Esta estructura nos permite indicar otra condición cuando no se cumple la condición inicial; esto hace que el código sea más específico. Veamos un ejemplo en el que aplicamos esta estructura:

```
  import java.util.Scanner;
  class EstructurasCondicionalCompuesta2 {

   public static void main(String arg[]){

    int num;

    Scanner entrada=new Scanner(System.in);
    System.out.println("Ingrese un valor: ");
    num = entrada.nextInt();
     if (num>100){
      System.out.println("El número es mayor que 100 " +
num);
      }else if(num>=70){
      System.out.println("El número "+num+ " es mayor o es
igual que 70");
      }else{
      System.out.println("El número "+num+ " es menor que
70")
   }
  }
  }
```

Estructura condicional de selección múltiple (switch)

Esta estructura condicional nos permite asignar un valor (que puede ser una variable) y una lista de casos; si se cumple alguno de los casos, se ejecutan las instrucciones asociadas; en cambio, si no se cumple ningún caso, es posible indicar la ejecución de acciones predeterminadas. Normalmente, se utiliza para indicar la presencia de un error mediante la presentación de un mensaje. Veamos un código de ejemplo:

```
import java.util.Scanner;
class EstructurasSeleccionMultiple {

 public static void main(String arg[]){
    String dia="Lunes";
     System.out.println("Ingrese un día de la semana");
  String = entrada.next();

          switch (dia){
             case "Lunes":
                 System.out.println("Hoy es "+dia);
                 break;
             case "Martes":
                 System.out.println("Hoy es "+dia);
                 break;
             case "Miércoles":
                 System.out.println("Hoy es "+dia);
                 break;
             case "Jueves":
                 System.out.println("Hoy es "+dia);
                 break;
             case "Viernes":
                 System.out.println("Hoy es "+dia);
                 break;
             case "Sábado":
                 System.out.println("Hoy es "+dia);
                 break;
             case "Domingo":
                 System.out.println("Hoy es "+dia);
                 break;
             default:
                 System.out.println("No ha ingresado un
dato correcto");

  }
}
}
```

La sentencia **switch** recibe un valor para analizar, que comparará con un literal del mismo tipo en cada caso. Si son iguales, se ejecutará el código asociado, y la estructura se detendrá mediante la etiqueta **break**. Si no coincide en ninguno de los casos, se ejecutará la etiqueta **default**.

Estructuras iterativas

Las estructuras iterativas se relacionan con operaciones que se deben ejecutar un número determinado de veces. El conjunto de instrucciones que se ejecuta cierto número de veces se llama **ciclo**, **bucle** o **lazo**.

Cuando hablamos de **iteración**, nos referimos a cada una de las diferentes pasadas o ejecuciones de todas las instrucciones contenidas en el bucle; para salir de este bucle, se indica una condición por cumplir.

Estructura while

Esta estructura algorítmica se ejecuta mientras la condición evaluada resulte verdadera. Se evalúa la expresión booleana y, si es cierta, se ejecuta la instrucción que especifiquemos. Luego se vuelve a evaluar la expresión booleana; si todavía es cierta, se ejecuta de nuevo el cuerpo o conjunto de instrucciones. Este proceso de evaluación de la expresión booleana y la ejecución del cuerpo se repite mientras la expresión sea cierta.

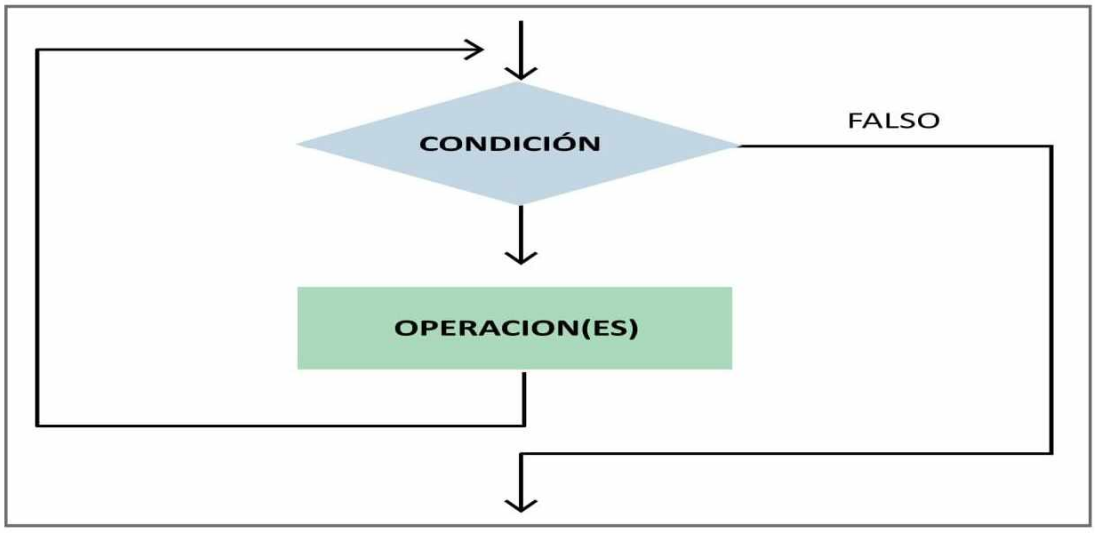


Figura 6. Diagrama de flujo de la estructura while.

La sintaxis de la estructura **while** es la siguiente:

```
while (condicion){
   instrucciones (cuerpo);
}
```

Veamos un ejemplo en el que usaremos la estructura **while** para imprimir en pantalla los 100 primeros números:

```
public class EstructuraWhile {

    public static void main(String[] args) {

        int x=1;
        while(x<=100){
          System.out.print(x);
          System.out.print(" - ");
          x = x++;
    }
    }
}
```

Como vemos, **x** vale **1** al principio, se comprueba la condición, como **1** es menor o igual que **100**, entra en el bucle para ejecutar las instrucciones, y así sucesivamente hasta que **x** vale **101**, ya que **101** no es menor o igual que **100**. Debemos considerar que, si **x** no fuera modificado, nunca se saldría del bucle.

Fases de un programa

Muchos programas pueden dividirse en tres fases: una fase de **inicialización** (en la que se inicializan las variables), una fase de **procesamiento** (en la que se ingresan los valores de los datos y se ajustan a las variables del programa) y una fase de **terminación** (que calcula y produce el efecto deseado del programa). También es necesario considerar que los programas en Java carecen de vida propia sin la implementación de una clase y la creación de los objetos adecuados.

Estructura for

El ciclo **for** es similar a **while**, pero, además de la condición, incluye la inicialización de una variable y un incremento o decremento de esa variable. En principio no es necesario que incluya estas tres partes, por lo que podemos inicializar o incrementar varias veces.

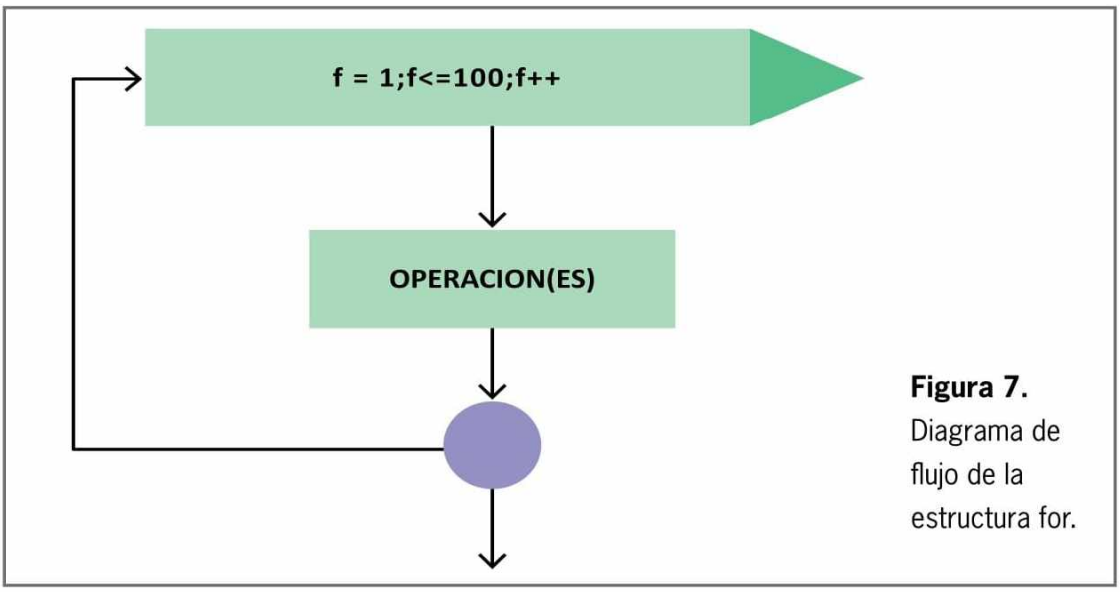


Figura 7. Diagrama de flujo de la estructura for.

La sintaxis de esta estructura es la siguiente:

```
for(inicializacion; condicion; incremento){
   instrucciones;
}
```

Veamos el mismo ejemplo que presentamos para **while**, de esta forma entenderemos cómo se construye una estructura **for**:

```
public class EstructuraFor {

    public static void main(String[] args) {

        int x=1;
        for(x=1; x<=100; x++){
          System.out.print(x);
```

```
            System.out.print(" - ");
            x = x++;
        }
        }
}
```

Estructura DO WHILE

Esta estructura ejecutará, al menos una vez, su bloque repetitivo, a diferencia de las estructuras anteriores, que podían no ejecutar el bloque.

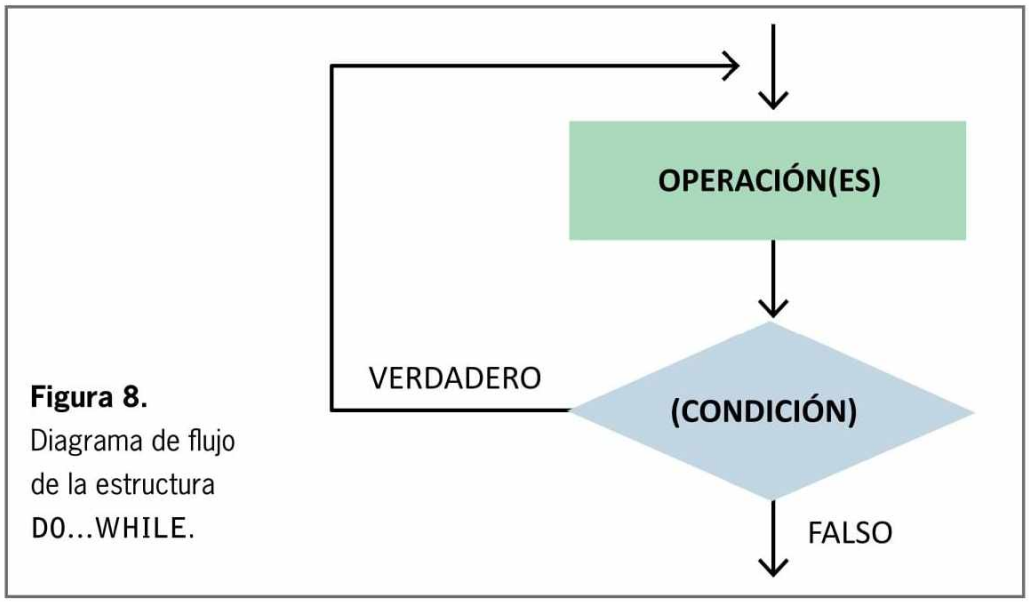


Figura 8.
Diagrama de flujo de la estructura DO...WHILE.

La sintaxis de esta estructura es la siguiente:

```
do{
   intrucciones
}while(condicion);
```

Veamos un ejemplo de esta estructura, para lo cual crearemos un código que nos pedirá introducir un número entre 1 y 10:

```
import java.util.Scanner;
public class EstructuraDoWhile {

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

```
        int num;
          Scanner entrada=new Scanner(System.in);
        do{
System.out.println("Ingrese un número entre 0 y 10: ");
     num = entrada.nextInt();
     }
     while(num>=10||num<0);
System.out.println("El número ingresado es: "+num);

  }
}
```

En este caso debemos ser cuidadosos con la condición de salida. Por ejemplo, si escribimos **5** (**5>=10** o **5<0**), devolverá **falso**; por otra parte, si escribimos **-1 (-1>=10** o **5<0**), devolverá **verdadero**, haciendo que vuelva a pedir un número.

Estructura for each

Con este ciclo es posible recorrer estructuras más complejas, como listas y determinadas colecciones (que veremos en el **Volumen 03** de este curso). A estas estructuras se las denominan **iterables**, ya que tienen un mensaje del tipo **iterator** que devolverá la instancia de **iterator**.

En Java no existe la palabra clave **foreach**, pero podemos lograr el resultado deseado usando **for**, por esta razón hablamos de **for each** o **for** extendido. La sintaxis de esta estructura es la siguiente:

For extendido

En las últimas versiones de Java se introdujo una nueva forma de uso del **for**, a la que se denomina "**for extendido**" o "**for each**". Esta forma de uso del for, que ya existía en otros lenguajes, facilita el recorrido de objetos existentes en una colección sin necesidad de definir el número de elementos a recorrer. Fue incorporado a partir de Java 5 y solventa de una manera sencilla el recorrido en estructuras como arrays o en colecciones.

```
for (TipoBase variable: ArrayDeTiposBase) {
    intrucciones
}
```

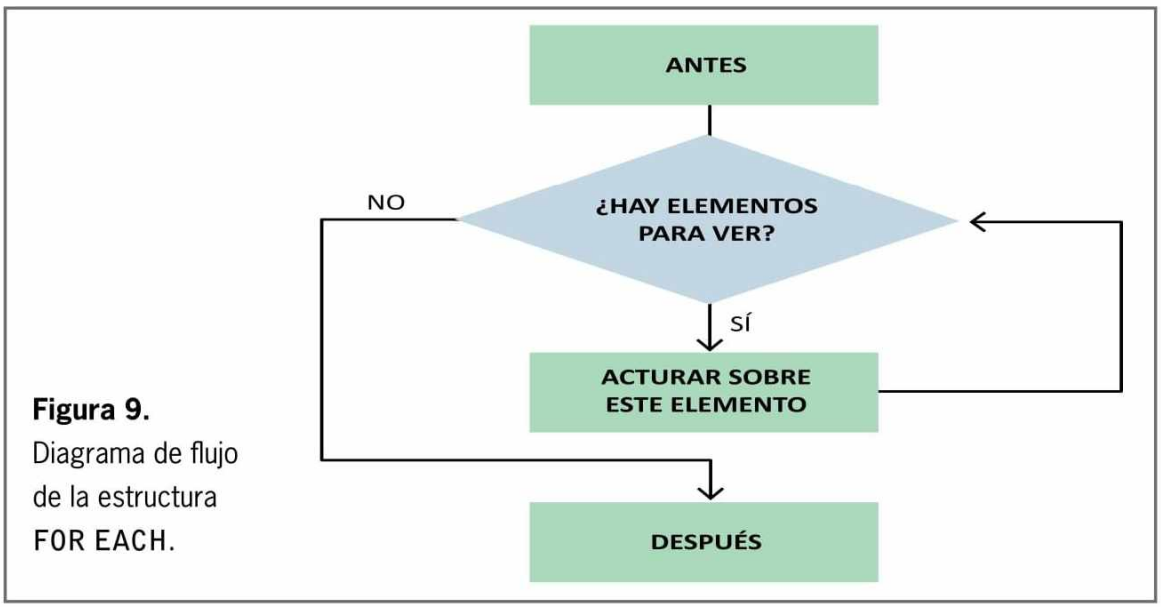


Figura 9. Diagrama de flujo de la estructura FOR EACH.

A continuación presentamos un ejemplo que nos permitirá ver la implementación de **for each**:

```
class EstructuraForeach  {
    public static void main(String args[]) {
        String [] arrStr = {"1", "2", "3", "4","5"};
        for(String elemento : arrStr) {
            System.out.println(elemento);
        }
    }
}
```

En el código observamos que se unifica la forma de listar los elementos y, de esta manera, se simplifica el código. Sin embargo, podemos lograrlo usando solo **for**:

```
for (int elemento=0; elemento<arrStr.length; elemento++)
 System.out.println(arrStr[elemento]);
```

VARIANTES DEL BUCLE FOR

El bucle **for** es una de las estructuras más versátiles en los lenguajes de programación; las variantes que podemos recrear son muchas. Veamos un ejemplo:

```
public class ForVariante1{
     public static void main(String args[]) {
     int i,j;
     // Dos variables de control del bucle
     for(i=0, j=10; i<j; i++, j--)
     System.out.println("i y j: "+ i +" --- " +j);
     }
}
```

Este singular programa hace que coexistan dos instrucciones de inicialización e iteración. Entonces, cada vez que **i** se incrementa, **j** se reduce.

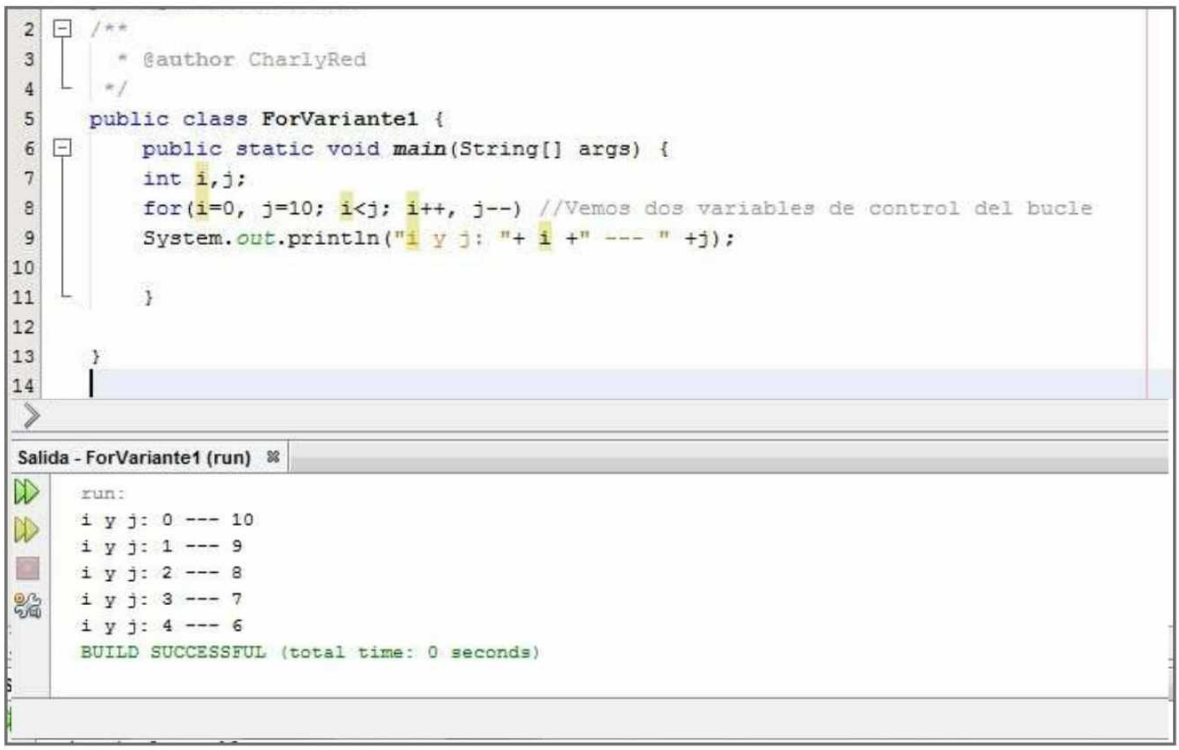


Figura 10. Observamos la salida que provoca el ejemplo de una de las variantes del ciclo FOR.

Piezas faltantes

Algunas de las variaciones de **for** se recrean al dejar vacías las piezas de la definición del bucle. En Java es posible que cualquiera o todas las inicializaciones, condiciones o partes de iteración del bucle **for** estén en blanco. Veamos un ejemplo:

```
public class ForVariante2{
public static void main(String args[]) {
     int i;
     for(i=0; i<10;){   //Falta la expresión de iteración
     System.out.println("Paso #: "+ i);
     i++; // Incrementa la variable de control de bucle
     }
     }
}
```

Aunque la expresión iteradora se encuentra vacía, dentro del cuerpo del bucle la variable de control aumenta. Esto significa que el bucle se repite y va a comprobar que sea menor que **10**.

Bucle infinito

Un bucle que nunca termina se puede crear dejando vacía la expresión condicional, de la siguiente forma:

```
public class ForVariante3{
public static void main(String args[]) {
for(;;) // bucle intencionalmente infinito

{

// ...

}
```

Aunque existen algunas tareas de programación que requieren un bucle infinito, como los procesadores de comando del sistema operativo, la mayor parte de este tipo de bucles tan solo son bucles con requisitos especiales de terminación.

Bucle sin cuerpo

El cuerpo asociado a un bucle puede estar vacío, esto se debe a que una instrucción **null** es sintácticamente válida. Veamos un ejemplo:

```
  public class ForVariante4{
  public static void main(String args[]) {
       int i;
       int suma=0;
       for(i=1;i<=5;suma+= i++){ //No hay cuerpo en este
bucle
            System.out.println("La suma es: " +suma);
  }
  }
  }
```

Variables de control dentro del bucle

A menudo, la variable que controla un bucle **for** solo es necesaria para dicho bucle. Cuando esto sucede, es posible declarar la variable dentro de la inicialización del **for**. Veamos un ejemplo:

```
  public class ForVariante5{
  public static void main(String args[]) {
       int suma=0;
               int fact=1;
       //Calcula el factorial de los números del 1 al 5
           for(int i=1;i<=5; i++){
           suma += i;
           fact *= i;
           //i no se conoce hasta aquí
           }
         System.out.println("La suma es: " +suma);
         System.out.println("El factorial es: " +fact);
       }
  }
```

El resultado de este código es el siguiente: **La suma es 15 y el factorial 120.**

Uso de break para salir del bucle

Si empleamos la instrucción **break**, podemos forzar la salida inmediata de un bucle omitiendo cualquier código en su cuerpo, así como la prueba de condición del bucle. Esto terminará el bucle, y el control del programa se reanudará en la instrucción siguiente a él. Veamos un ejemplo:

```
public class ForVariante6{
public static void main(String args[]) {
     int num= 100;
           //Recorre el bucle si i*i es menor que num
          for(int i=0;i<num; i++){
            if(i*i>=num) break;
          //termina el bucle si i*i >=100
          System.out.println(i+" ");
          }
          System.out.println("bucle completo.");
     }
}
```

Una vez que ejecutemos este programa, veremos que se generará la siguiente salida: **0 1 2 3 4 5 6 7 8 9 bucle completo**.

Uso de break como goto

La instrucción **break** puede emplearse por sí sola para proporcionar una forma actual de la instrucción **goto**. Java no cuenta con una instrucción **goto** porque esta proporciona una manera no estructurada de modificar el flujo de ejecución del programa.

El punto flotante en los bucles

Los valores con punto flotante suelen traer problemas al ser aproximados. Controlar ciclos con variables con estos valores puede producir valores imprecisos del contador y pruebas de terminación no deseadas. Por esta razón, es necesario que procuremos utilizar números enteros para controlar los ciclos del contador.

Los programas que llevan a cabo un uso extenso de **goto** suelen ser difíciles de comprender y de mantener. Sin embargo, existen ciertos lugares en los que **goto** resulta útil y legítimo, por ejemplo, puede ser útil al salir de conjuntos profundamente anidados de bucles.

Para manejar estas situaciones, Java define una forma expandida de la instrucción **break**, la que nos permite salir de uno o más bloques de código. No es necesario que estos bloques sean parte de un bloque o un **switch**, puede tratarse de cualquier bloque.

Además, se puede especificar con precisión dónde se reanudará la ejecución, pues esta forma de **break** funciona con una etiqueta.

La forma general de la instrucción **break** etiquetada se muestra a continuación:

```
public class ForVariante7{
public static void main(String args[]) {
     hecho:
     for(int i=0;i<10;i++){
     for(int j=0;j<10;j++ ){
     for(int k=0;k<10;k++ ){
     System.out.println(k+" ");
     if(k==5) break hecho; //salta a hecho
     }
System.out.println("Tras bucle k");//no se ejecuta
     }
System.out.println("Tras bucle j");//no se ejecuta
     }
System.out.println("Tras bucle i");
 }
}
```

El uso de continue

Es posible forzar una iteración temprana de un bucle omitiendo la estructura de control normal del bucle. Esto se logra usando la instrucción **continue**, la que impone la ejecución de la siguiente iteración del bucle omitiendo cualquier código entre sí mismo y la expresión condicional que controla el bucle. Por lo tanto, **continue** es esencialmente el complemento de **break**. Por ejemplo, el siguiente

programa usa **continue** como ayuda para imprimir los números pares entre el **0** y el **100**.

```
public class ForVariante8{
public static void main(String args[]) {
  int i;
//Imprime números pares entre 0 y 100
    for(i=0;i<=100; i++){
          if((i%2)!=0) continue;     //itera
        System.out.println(i);
    }
    }
}
```

Solo se imprimen los números pares, porque uno no causaría que el bucle se iterara antes, omitiendo la llamada a **println()**.

En los bucles **while** y **do while**, una instrucción **continue** hará que el control vaya directamente a la expresión condicional y luego siga el recorrido del bucle. En el caso de **for**, se evalúa la expresión de iteración del bucle, luego se ejecuta la expresión condicional y, más tarde, se sigue el bucle.

Bucles externos

Al igual que con la instrucción **break**, **continue** puede especificar una etiqueta para describir en cuál bucle incluido se deberá continuar. Aquí se presenta un programa de ejemplo que usa **continue** con una etiqueta:

```
public class ForVariante9{
public static void main(String args[]) {
    bucleexterno:
    for(int i=1;i<10; i++){
   System.out.println("\nPaso de bucle externo "+i+ ",
Bucle interno: ");

    for(int j=1;j<10; j++){
```

```
      if(j==5)continue bucleexterno; //continua el bucle
externo
      System.out.print(j+ " - ");
      }
       }
       }
  }
```

Bucles anidados

Como vimos en los ejemplos anteriores, es posible anidar un bucle dentro de otro. Los bucles anidados se usan para resolver una amplia variedad de problemas de programación, así que son parte esencial de esta. Veamos el siguiente ejemplo para encontrar los factores de los números del **2** al **100**:

```
  public class ForVariante10{
  public static void main(String args[]) {
     //Imprime los factores del 2 al 100
      for(int i=2;i<=100; i++){
            System.out.println("Factores de "+i+": ");
      for(int j=2;j<i; j++)
      if((i%j)==0)
      System.out.println(j);
      }
      }
  }
```

El goto actualizado

El lenguaje Java se toma muy en serio este tipo de instrucciones, usa **break** y **continue**. Es necesario tener en cuenta que, con el advenimiento de los procedimientos, es posible llamar a estos desde cualquier lado del programa, lo que facilitará la tarea se programar.

ARRAYS

Los **arreglos** o **arrays** son colecciones de datos que nos permiten agrupar variables relacionadas entre sí y que comparten el mismo tipo. Podemos crearlos para cualquier tipo de información que pueda ser almacenada en una variable. Cuando creamos un array, se reserva automáticamente el espacio en la memoria; al iniciarlo, debemos indicar su tipo y su tamaño; este tamaño debe ser un número entero positivo y no puede variar durante la ejecución.

Tipos de arreglos

La clasificación de los arreglos se realiza en torno a las dimensiones que presenten. Existen tres grupos:

Vector	Arreglos unidimensionales (1D), con un subíndice.
Matriz	Arreglos bidimensionales (2D), con dos subíndices.
Multidimensional	Tres o más dimensiones (3D), con tres o más subíndices.

Vector

Un **vector** o arreglo unidimensional es una estructura de datos que almacena un conjunto de datos de un mismo tipo. Se trata de un arreglo; es una lista de **n** elementos que posee las siguientes características:

- Se identifica por un único nombre de variable.
- Sus elementos se almacenan en posiciones contiguas de memoria.
- Se accede a cada uno de sus elementos en forma aleatoria.

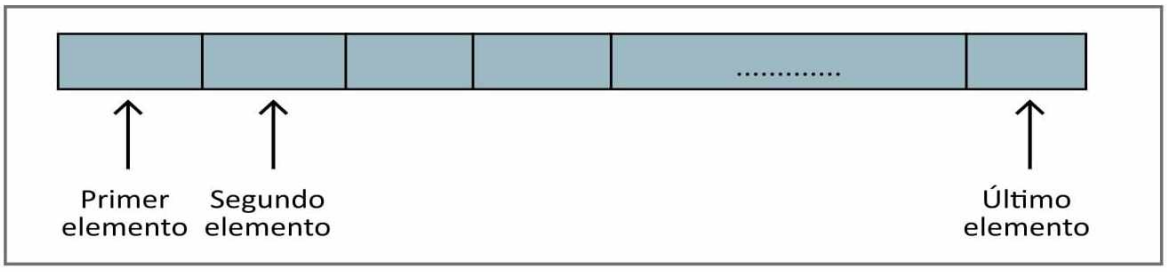


Figura 11. Gráfico que nos muestra el funcionamiento de un arreglo de una dimensión o vector.

La sintaxis de un vector es la siguiente:

```
Tipo_dato nombre_array[]=new tipo_dato[tamaño];
```

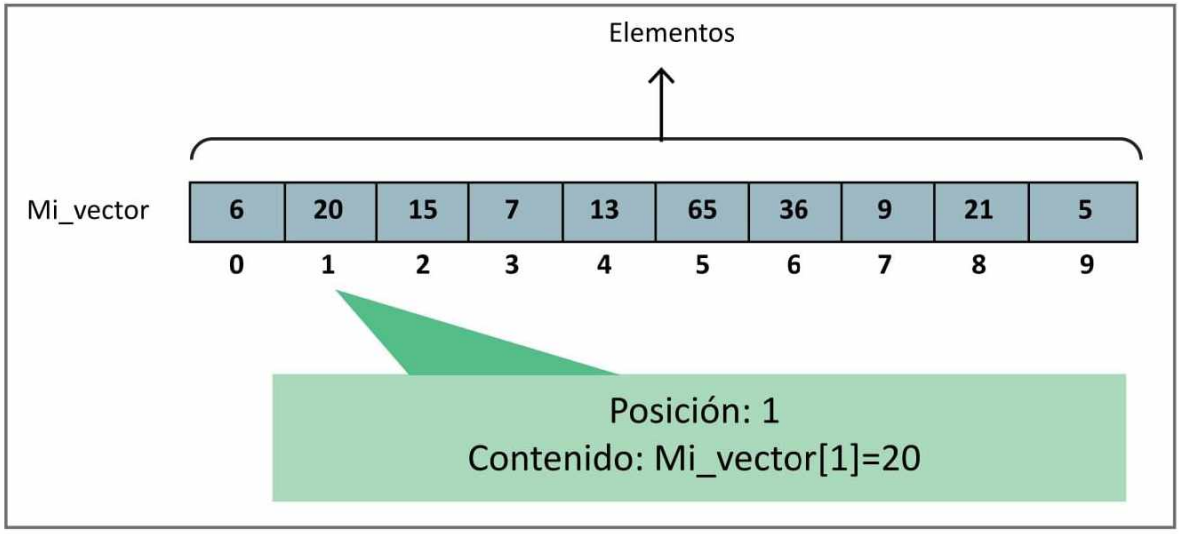


Figura 12. Ejemplo gráfico de un vector en el que verificamos el uso de índices para referenciar a los elementos. En Java debe empezar por cero (0).

Veamos un ejemplo en el que observaremos cómo se inicializa e instancia una variable dentro de los arreglos:

```
int num[]=new int[5];
```

Otra forma de escribirla es la siguiente:

```
int num[]={5, 6, 5, 2, 3};
```

Los arreglos se rellenan en forma automática. Por ejemplo, si se trata de un array de **int**, se completan con **0**; si es un array de **String**, se rellenan con **null**:

```
public static void main(String[] args) {
        //Definimos un array con 10 posiciones
        int num[]=new int [10];
        num[0]=5;
```

```
            num[1]=9;

            System.out.println(num[0]);
            System.out.println(num[1]);
        }
```

Para recorrer un array debemos usar un bucle, en el que lo único que cambie será la posición. Veamos un ejemplo de cómo rellenar un array de **10** posiciones con múltiplos de **5**:

```
  public class Array1 {

      public static void main(String[] args) {
        //Definimos un arreglo con 10 posiciones
          int num[]=new int [10];
        //Recorremos el array
        //array.lenght indica la longitud del arreglo, en
este caso, devuelve 10
          for (int i=0, multiplo=5;i<num.length;i++,
multiplo+=5){
              num[i]= multiplo;
              System.out.println(num[i]);
          }
      }
    }
```

Como vemos en el ejemplo anterior, para saber la longitud del array usamos el atributo **length**.

Matriz

El **array bidimensional** o **matriz** se puede considerar como un vector de vectores.

Por consiguiente, se trata de un conjunto de elementos, todos del mismo tipo, en el cual el orden de los componentes es significativo y en el que se necesita especificar los subíndices para que se identifique cada elemento del arreglo.

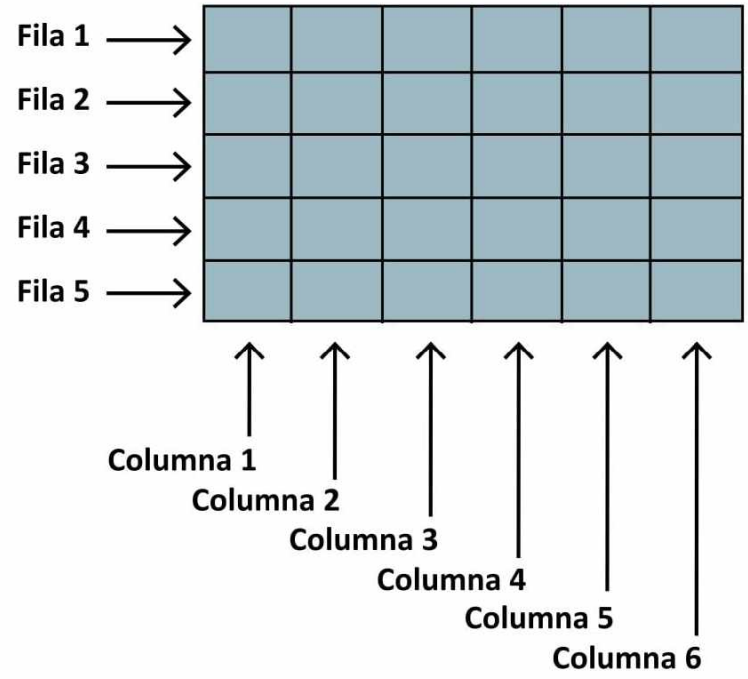


Figura 13. Representación gráfica que indica el funcionamiento de un arreglo de dos dimensiones o matriz.

Si se visualiza un arreglo unidimensional, se puede considerar como una columna de datos; un arreglo bidimensional es un grupo de columnas.

El funcionamiento de las matrices suele ser un tanto abstracto, pues es necesario entender su uso práctico.

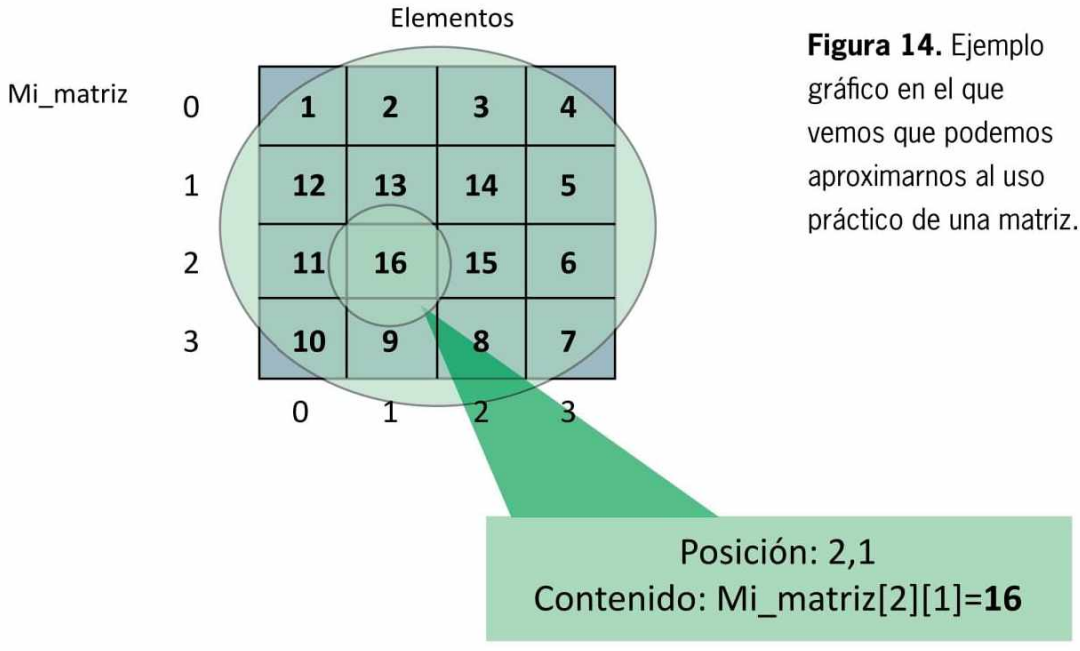


Figura 14. Ejemplo gráfico en el que vemos que podemos aproximarnos al uso práctico de una matriz.

Array multidimensional

Un **array multimensional** reúne dos o más arrays unidimensionales; podemos definirlo de tres dimensiones, cuatro dimensiones, hasta de n-dimensiones. En general, un array de n-dimensiones requiere que los valores de n-índices puedan ser especificados para identificar un elemento individual del array. Si cada componente de un array tiene n-índices, el array se dice que es solo de n-dimensiones, es decir, como si fuera una tabla.

Este tipo de array se crea igual que un array unidimensional, solo hay que añadir un corchete más con un tamaño en la definición del array.

	Punto		
Tiempo	1	2	3
1	65.5	68.7	62.0
2	68.8	68.9	64.5
3	70.4	69.4	66.3
4	68.5	69.1	65.8

Figura 15. Ejemplo práctico que nos presenta el funcionamiento de un arreglo con varias dimensiones.

Veamos un ejemplo en el que definimos un array multidimensional:

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

        //Definimos un array de 3 filas x 5 columnas
        int array[][]= new int [3][5];
//Asignamos un 5 al array, en la fila 0 columna 1
        array[0][1] = 5;

    }
```

Siguiendo esta dinámica, podemos crear arreglos tridimensionales o de cuantas dimensiones necesitemos. Con esto ahorraremos espacio en la codificación y en la memoria. En el ejemplo anterior, nos hemos ahorrado la declaración de 15 variables.

Ordenar un array

La forma de ordenar un array es muy importante a la hora de trabajar con estos elementos. Hay varias maneras de hacerlo, tanto con números como con cadenas; a continuación las conoceremos.

Intercambio

Consiste en comparar el primer valor con el resto de las posiciones posteriores, cambiando el valor de las posiciones en caso de que el segundo valor sea menor que el primer valor comparado. Veamos un ejemplo utilizando números:

```
public static void intercambio(int lista[]){

        //Usaremos un bucle anidado
        for(int i=0;i<(lista.length-1);i++){
            for(int j=i+1;j<lista.length;j++){
                if(lista[i]>lista[j]){
         //Aquí intercambiamos los valores
                    int variableAuxiliar=lista[i];
```

Los arreglos y la memoria

Los arreglos ocupan espacio de memoria y, al igual que los demás objetos, se crean con la palabra clave **new**. También debemos especificar el tipo de cada elemento y el número de ellos que se va a requerir. Esta expresión devuelve una referencia que puede almacenarse en una variable tipo **arreglo**. Recordemos la importancia que tiene el uso de la memoria en los programas; aunque Java hace uso de un **garbage**, hay que tender a ser ahorrativos con él.

```
                    lista[i]=lista[j];
                    lista[j]=variableAuxiliar;

                }
            }
        }
    }
```

Ahora veamos un ejemplo utilizando cadenas:

```
public static void intercambioTextos(String lista[]){
     //Usaremos un bucle anidado
         for(int i=0;i<(lista.length-1);i++){
             for(int j=i+1;j<lista.length;j++){
        if(lista[i].compareToIgnoreCase(lista[j])>0){
            //Intercambiamos los valores
          String variableAuxiliar=lista[i];
                lista[i]=lista[j];
                lista[j]=variableAuxiliar;
                  }
            }
        }
    }
```

Burbuja

Este método, conocido como **bubble sort**, consiste en comparar el primero con el segundo; si el segundo es menor que el primero, se intercambian los valores. Después, el segundo con el tercero y así sucesivamente; cuando no haya ningún intercambio, el array estará ordenado. Este método tiene un inconveniente: mientras más elementos haya que ordenar, más tardará en hacerlo. Veamos un ejemplo de código:

```
  public static void burbuja (int lista[]){
         int cuentaIntercambios=0;
     //Se usará un bucle anidado y saldrá cuando el
arreglo esté ordenado
```

```
            for (boolean ordenado=false;!ordenado){
                for (int i=0;i<lista.length-1;i++){
                    if (lista[i]>lista[i+1]){
                     //Acá intercambiamos valores
                        int variableAuxiliar=lista[i];
                        lista[i]=lista[i+1];
                        lista[i+1]=variableAuxiliar;
                     //indicamos que hay un cambio
                        cuentaIntercambios++;
                    }
                }
           //Si no hay intercambios, es que está ordenado
                if (cuentaIntercambios==0){
                    ordenado=true;
                }
                //Se inicializa la variable para que empiece
a contar nuevamente
                cuentaIntercambios=0;
            }
        }
```

Quicksort

Este método consiste en ordenar un arreglo mediante un pivote, que es un punto intermedio en el arreglo. En otras palabras, se ordenan pequeños trozos del arreglo: a la izquierda están los menores a ese pivote y, a la derecha, los mayores a este. Después se vuelve a calcular el pivote de trozos de listas.

Para usar este método debemos recurrir a la recursividad; pasamos como parámetro el arreglo, su posición inicial y su posición final. El uso de este método mejora el rendimiento aun teniendo muchos valores que ordenar. Veamos un ejemplo de su uso:

```
  public static void quicksort (int lista1[], int izq, int der){
      int i=izq;
      int j=der;
      int pivote=lista1[(i+j)/2];
```

```
        do {
            while (lista1[i]<pivote){
                i++;
            }
            while (lista1[j]>pivote){
                j--;
            }
            if (i<=j){
                int aux=lista1[i];
                lista1[i]=lista1[j];
                lista1[j]=aux;
                i++;
                j--;
            }
        }while(i<=j);
        if (izq<j){
            quicksort(lista1, izq, j);
        }
        if (i<der){
            quicksort(lista1, i, der);
        }
    }
```

Método sort de java.util.Arrays

Para ejecutarlo escribimos **arrays.sort(array_a_ordenar);** insertando como parámetro el arreglo que necesitamos ordenar.

Posee varios métodos para distintos tipos. Veamos un ejemplo de su implementación:

```
import java.util.Arrays;
public class sortArray {

    public static void main(String[] args) {

        final int LONGITUD_ARREGLO =10;
        int lista[]=new int [LONGITUD_ARREGLO];
```

```
          rellenarArray(lista);
          String lista_String[]={"Argentina", "Colombia",
"Chile", "Brasil", "Bolivia"};

          System.out.println("arreglo de números sin ordenar:");
          imprimirArray(lista);

          //Ordenamiento de las listas
          Arrays.sort(lista);

          System.out.println("arreglo de números ordenado: ");
          imprimirArray(lista);

          System.out.println("Array de String sin ordenar: ");
          imprimirArray(lista_String);

   //Ordenamiento del arreglo, ordenará primero las
mayúsculas y luego las minúsculas
          Arrays.sort(lista_String);
          System.out.println("arreglo de String ordenado:");
          imprimirArray(lista_String);
      }
   //Método que imprimirá la lista de números enteros aleatorios
      public static void imprimirArray (int lista[]){
          for(int i=0;i<lista.length;i++){
              System.out.println(lista[i]);
          }
      }
   //Método que imprime el listado de países
     public static void imprimirArray (String lista[]){
          for(int i=0;i<lista.length;i++){
              System.out.println(lista[i]);
          }
      }
   //Método que rellenará la lista del array con el
resultado que encuentre en el método aleatorio.
      public static void rellenarArray (int lista[]){
```

```
            for(int i=0;i<lista.length;i++){
                lista[i]=numeroAleatorio();
            }
        }
    //Método matemático que buscará una cantidad de números
aleatorios hasta 1000
        private static int numeroAleatorio (){
            return ((int)Math.floor(Math.random()*1000));
        }
    }
```

En el código anterior realizamos dos arreglos. En primer lugar un listado de números aleatorios menores que 1000 (método **numeroAleatorio**), que se listan de manera desordenada y luego en forma ordenada, gracias al método **Arrays.sort(lista)**. En segundo lugar tenemos un vector con nombres de países no ordenados y, mediante el método **Arrays.sort(lista)**, se ordenarán alfabéticamente.

Métodos java.util.Arrays

Para acceder a estos métodos, debemos importar el paquete **java.util.Arrays** antes de la clase.

Indicaremos el nombre, la descripción, los parámetros y los datos que devuelve, considerando la siguiente sintaxis:

```
Arrays.nombre_metodo(parámetro);
```

Vectores y la clase Vector

Estos dos conceptos suelen confundirse, pero debemos entender que la clase Vector crece automáticamente hasta alcanzar la dimensión inicial máxima. Los utilizaremos en las estructuras dinámicas. Un vector es similar a un array, la diferencia está en que un vector crece automáticamente cuando alcanza la dimensión inicial máxima. Además, proporciona métodos adicionales para añadir, eliminar elementos, e insertar elementos entre otros dos existentes.

Revisemos los métodos que podemos utilizar en esta clase:

binarySearch	Busca un valor que le pasamos por parámetro y devuelve su posición. El arreglo debe estar ordenado.
copyOf	Copia un array y lo devuelve en un nuevo array.
copyOfRange	Copia un array y lo devuelve en un nuevo array. Le indicamos la posición de origen y de destino.
equals	Indica si dos arrays son iguales.
fill	Rellena un array con un valor que le indiquemos como parámetro.
sort	Ordena el array.
toString	Muestra el contenido del array pasado como parámetro.

Para entender los distintos métodos, veamos el siguiente ejemplo:

```
  import java.util.Arrays;
  public class Array2 {
       public static void main(String[] args) {
          int num[]={8, 10, 15, 20, 21, 25, 30, 32, 40, 41};

          //Devuelve un 4
          System.out.println("Metodo binarySearch:
"+Arrays.binarySearch(num, 21));
          //Copia el array num hasta la quinta posición
(este último no incluido), devuelve un array
          int num2[]=Arrays.copyOf(num, 4);

          System.out.println("Metodo copyOf ");
          //Lo recorremos para ver que lo realiza
correctamente
```

```
            muestraArray(num2);

            //Copia el array num de la tercera hasta la
octava posición, devuelve un array
            int num3[]=Arrays.copyOfRange(num, 2, 6);
            System.out.println("Metodo copyOfRange");
            muestraArray(num3);

            //Compara si num1 y num2 no son iguales
            System.out.println("Metodo equals: "+Arrays.
equals(num, num2));
                    System.out.println("Metodo fill");
            Arrays.fill(num3, 5);
            muestraArray(num3);

            System.out.println("Metodo toString");
            System.out.println(Arrays.toString(num));
        }

        public static void muestraArray(int num[]){
                for (int i=0;i<num.length;i++){
                System.out.println(num[i]);
            }
        }
    }
```

En el código anterior tenemos un vector con 10 elementos que son analizados con los distintos métodos **java.util.Arrays**:

La memoria en arreglos

Se sabe que Java no maneja punteros como lo hace C, solo se maneja con apuntadores al espacio de memoria donde estos objetos van a ir a parar. Java sí tiene un tipo de apuntadores o referencias, lo que no tiene es operadores que haya que poner explícitamente, como *, ^ o **&&**.

En el caso **binary Search** se encuentra la posición del vector.

En **copyOf** se devuelve un listado de 4 números del array.

Mediante **copyOfRange** crea un listado que va desde la posición 2 (el número 15) hasta la posición 6 (el número 30) del vector.

En el caso **equals**, dados dos vectores (**num** y **num2**), verifica si ambos son iguales (**TRUE** o **FALSE**).

Mediante **fill** rellenará, según el número del argumento 2, la cantidad de elementos que tenga el array **num3**.

En **toString** muestra el vector original pero como un parámetro.

El método **muestraArray** muestra todos los métodos, pero para ello debemos utilizar **for**; así lista los resultados según los valores guardados en las variables.

Arrays en métodos

Es importante saber cómo debemos definir un array como parámetro; de esta manera, realizaremos muchas tareas con su contenido:

```
public static void imprimirArray (int lista[]){
        for(int i=0;i<lista.length;i++){
            System.out.println(lista[i]);
        }
    }
```

Arrays en los métodos

Una de las situaciones más comunes es que operemos con arreglos en los métodos, así podemos mostrarlos, recorrerlos o cargarles información. La clase **Arrays** contiene varios métodos para manipular arrays (por ejemplo, para ordenar un array o buscar un valor u objeto dentro de él) y para comparar arrays.

En el código anterior definimos el tipo de dato, el nombre y, al añadir los corchetes, indicamos que se trata de un array. Se copia la dirección del array pasado por parámetro al array del método, y se puede modificar su contenido. Otra manera de devolver un array es la siguiente:

```
public static int[] rellenarArrayDesde(int a){
        int num[]=new int [10];
        for(int i=0;i<num.length;i++){
            num[i]=a;
            a++;
        }
        return num;
    }
```

Entonces, para devolver un array, debemos añadir dos corchetes al tipo devuelto y, en el **return**, solo indicamos el nombre del array sin los corchetes. Veamos cómo debemos invocar el método:

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

        int num[]=rellenarArrayDesde(5);
        imprimirArray(num);
    }
```

Ahora bien, solo es necesario pasar el nombre del array; también vemos que, si el método devuelve un array, debemos guardar la dirección en otro array. Veamos el ejemplo completo:

```
public class array3 {

    public static void main(String[] args) {

        int num[]=rellenarArrayDesde(5);
        imprimirArray(num);
    }
```

```
public static void imprimirArray (int lista[]){
    for(int i=0;i<lista.length;i++){
        System.out.println(lista[i]);
    }
}

public static int[] rellenarArrayDesde(int a){
    int num[]=new int [10];
    for(int i=0;i<num.length;i++){
        num[i]=a;
        a++;
    }
    return num;
}
```

RESUMEN CAPÍTULO 04

Este capítulo fue bastante exhaustivo pues, en un comienzo, revisamos estructuras de control simples hasta presentar algunas más complejas. Conocimos estructuras tales como while, do while y for, este último, con todas sus variantes y casos en los que podemos aplicarlo. También nos aproximamos a los arreglos, un grupo muy importante de elementos que es necesario conocer cuando aprendemos un lenguaje de programación como Java. Vimos qué son y para qué sirven, además presentamos diversos ejemplos de su uso y conocimos las formas de ordenamiento más relevantes.

Actividades 04

Test de Autoevaluación

1. ¿Qué es una estructura secuencial?
2. Explique la diferencia entre if e if else.
3. ¿En qué momento se ejecuta un while y cuándo termina el bucle?
4. ¿Cómo construimos un for?
5. ¿Cuál es la diferencia entre while y la estructura do while?
6. ¿Qué es un bucle infinito?
7. ¿De qué manera se implementa un foreach?
8. ¿Qué diferencia hay entre un array y un vector?
9. ¿Qué importancia tienen los arreglos en la programación?
10. ¿Qué maneras existen para ordenar arreglos?

Ejercicios prácticos

1. Escriba un programa que solicite la carga de un valor positivo y muestre desde 1 hasta el valor ingresado, de uno en uno.
2. Escriba un programa que lea 5 notas de alumnos e informe cuántos tienen notas mayores o iguales a 7, y cuántos, menores (aprobados y reprobados). Refactorice el programa para que aparezca el mensaje de “Diciembre” si tiene la nota mayor que 4 y menor que 7.
3. Mediante el ingreso de varios nombres de personas en un arreglo, implemente su ordenamiento alfabético.

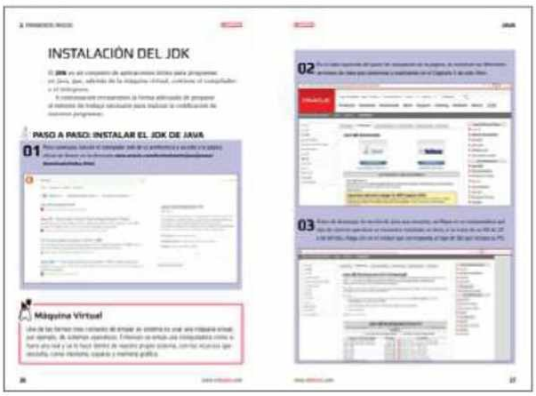
USERS

Programación en Java™

Vol. I

ACERCA DE ESTE CURSO

Java es uno de los lenguajes más robustos y populares en la actualidad, existe hace más de 20 años y ha sabido dar los giros adecuados para mantenerse vigente. Este curso de Programación en Java nos enseña, desde cero, todo lo que necesitamos para aprender a programar y, mediante ejemplos prácticos, actividades y guías paso a paso, nos presenta desde las nociones básicas de la sintaxis y codificación en Java hasta conceptos avanzados como el acceso a bases de datos y la programación para móviles.



ACERCA DE ESTE VOLUMEN

En este primer volumen de los cuatro que componen el curso, exploramos los conceptos básicos necesarios para enfrentarnos al mundo del desarrollo de aplicaciones, analizamos las características más importantes de Java y preparamos el entorno de desarrollo. También revisamos la sintaxis, elementos y estructuras de control de este lenguaje.

SOBRE EL AUTOR

Carlos Arroyo Díaz es programador, escritor especializado en tecnologías y docente. Se desempeña como profesor de Informática General, Java y Desarrollo Web. También ha trabajado como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires.

RedUSERS

En nuestro sitio podrá encontrar noticias relacionadas y participar de la comunidad de tecnología más importante de América Latina.

RedUSERS PREMIUM

RedUSERS PREMIUM la biblioteca digital de USERS. Accederás a cientos de publicaciones: Informes; eBooks; Guías; Revistas; Cursos. Todo el contenido está disponible online - offline y para cualquier dispositivo. Publicamos, al menos, una novedad cada 7 días

ISBN 978-987-4958-04-4