JAVA

APUNTES BÁSICOS

EDICIÓN SEPTIEMBRE 2021

by **Jorge A. López Vargas** / jorgaf@gmail.com / @jorgaf

Este libro, es una complemento didáctico y no debe usarse sin la guía de un docente, quien presente y profundice cada uno de los temas que aquí se desarrollan, debido a que únicamente se muestra de forma concreta como utilizar el lenguaje de programación Java y carece de explicaciones extensas que ayuden a un aprendiz a comprender cada uno de los temas. Si bien al final de cada apartado se muestran otros recursos, los mismos están en la misma línea, cómo usar Java.

Java y sus características 6

Generalidades y definición 6
Instalación del entorno de desarrollo 9
Comprobación de la correcta instalación 13
Consejos al momento de buscar recursos sobre Java 15

Variables, expresiones y sentencias 17

Valores y tipos 17
Variables 20
Nombres de variables y palabras reservadas 23
Entrada y salida de datos 25
Operadores y Operandos 28
Evaluación de expresiones 29
Precedencia de operadores 30
Sentencias 31
RECURSOS ADICIONALES 32

Métodos útiles 33

Métodos Matemáticos 33
Invocación de métodos 35
Conversión de tipos 36
RECURSOS ADICIONALES 37

Condicionales 39

Expresiones lógicas 39
Operadores lógicos 41
Ejecución condicionada 42
Ejecución alternativa 42
Condicionales encadenadas 44
Condiciones anidadas 45
Operador ternario 46
Switch 46
Switch como expresión 48
RECURSOS ADICIONALES 50

Iteración 51

Estructura do…while 51
Estructura for 53
Estructura while 55
RECURSOS ADICIONALES 56

Arreglos 58

Definición 58
Creación de arreglos 59
Accediendo a los elementos 60
Mostrar ARREGLOS 62
Recorrido de arreglos 63
Arreglos bidimensionales 64
Operaciones básicas en arreglos bidimensionales 65
RECURSOS ADICIONALES 67

Métodos 68

Escribiendo métodos 68
Métodos que No devuelven resultados 69
Métodos que devuelven resultados 71
Uso de métodos, Más allá de la asignación 73
Sobrecarga de métodos 76
Recursividad 77
RECURSOS ADICIONALES 79

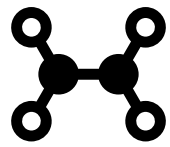
Cadenas de texto 81

Objeto 81
SECUENCIA DE Caracteres 83
Comparando cadenas de texto 84
MÉTODOS útiles 86
Bloques de texto 91
RECURSOS ADICIONALES 93

Archivos 95

Conceptos básicos 95
Usando file para Obtener información 96
Lectura y escritura de archivos de texto 101

Lectura de archivos vía URL 104
RECURSOS ADICIONALES.....105
Palabras finales.....106



JAVA Y SUS CARACTERÍSTICAS

GENERALIDADES Y DEFINICIÓN

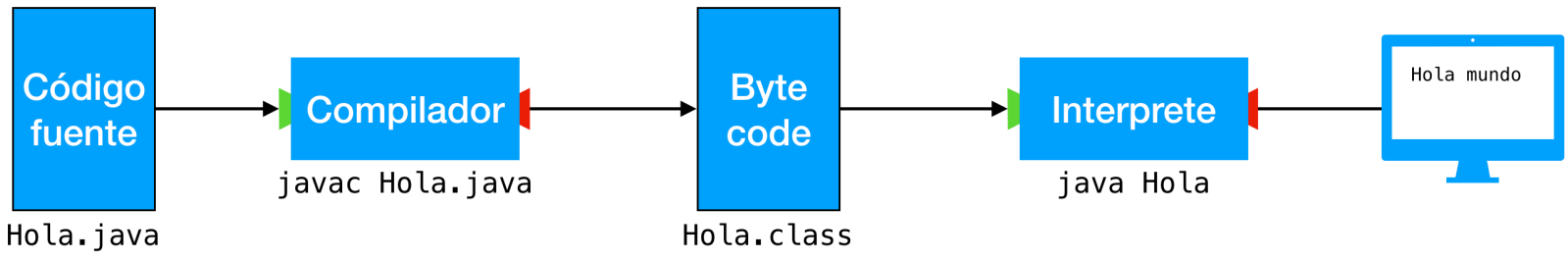
El lenguaje de programación Java es un lenguaje de tercera generación o de alto nivel creado en 1996 por la empresa Sun Microsystems, hoy extinta, y que fue adquirida por Oracle en el año 2009. Es decir, usted aprenderá un lenguaje de programación consolidado y en constante evolución que cuenta con una gran variedad de recursos (libros, vídeos, manuales, tutoriales, etc.) disponibles en la Web.

En la actualidad, Java es uno de los lenguajes de programación mas populares gracias a una extensa comunidad de desarrolladores y al gran número de aplicaciones desarrolladas utilizando dicho lenguaje. Esta popularidad ha permitido que existan diferentes implementaciones, así puede encontrar la implementación oficial/comercial desarrollada por Oracle (https://www.oracle.com/technetwork/es/java/javase/overview/index.html), otra implementación libre y abierta denominada OpenJDK (https://openjdk.java.net) desarrollada por una comunidad de software libre. También existe una implementación desarrollada y usada por Amazon que se denomina Corretto (https://aws.amazon.com/es/corretto/)

Java es un lenguaje de programación de **propósito general**, es decir que se pueden desarrollar diferentes tipos de aplicaciones, tales como aplicaciones de: consola, escritorio, Web, móviles, embebidas, etc.. Actualmente, Java es **multiparadigma**, soportando los paradigmas de programación estructurada, orientada a objetos y funcional. Otra de las características principales de este lenguaje es que es **multiplaforma**, es decir, un programa escrito en Java puede ejecutarse, sin cambio alguno, en diferentes sistemas operativos (Unix, Linux, Mac OS, Windows, Android, etc.).

El código escrito en Java generalmente se almacena en archivos que llevan la extensión *.java* y que se denominan código fuente. Una vez escrito el código fuente, se compila y se genera un código intermedio escrito en *byte code* (lo que le permite ser multiplataforma). Para su ejecución, el *byte code* es interpretado por la máquina virtual de Java. Es por ello que se considera a Java como un lenguaje **compilado e interpretado**. Este proceso se resume en la

siguiente imagen:



La imagen anterior muestra la forma tradicional de compilar y ejecutar un programa Java: escribir el código fuente, almacenarlo en un archivo *.java*, compilarlo utilizando el comando *javac* con el fin que se genere el código intermedio, almacenado en un archivo *.class*, luego ejecutarlo a través del comando *java* para finalmente ver el resultado en la pantalla de un computador.

Desde la versión 11 de Java, es posible ejecutar un programa Java sin la necesidad de una compilación previa, esa característica es parte de las mejoras del lenguaje y que responde al trabajo de una propuesta de mejora para Java o Java Enhancement Proposal (JEP) denominada Launch Single-File Source-Code Programs o JEP 330. Es necesario mencionar que no todas las clases pueden saltarse el proceso de compilación, está pensado para algunas clases que cumplen ciertas características, como por ejemplo que todo el código debe estar en un única clase. Es una alternativa para quienes están iniciando en la Orientación a Objetos.

Si bien el proceso anterior, compilar/ejecutar, sigue aún vigente, existen otras formas de trabajo más sencillas diseñadas especialmente para el aprendizaje, que hacen que se necesite una cantidad menor de comandos e interacciones con elementos que para un aprendiz resultan extraños.

Una de estas herramientas se denomina *JShell*. Esta herramienta pertenece al grupo de herramientas denominadas *REPL* existentes en otros lenguajes de programación y que permiten: *Read* (leer), *Evaluate* (evaluar), *Print* (imprimir) y *Loop* (bucle), que se usan para aprender y construir prototipos utilizando código de un lenguaje de programación, en este caso Java. Esta herramienta está disponible desde la versión 9 de Java y en cada versión se agregan algunas novedades que hacen mucho mas fácil la escritura de código, por ejemplo agregar sangrías o tabulado de forma automática. En las primeras secciones de este documento se utilizará ampliamente esta herramienta. La siguiente imagen muestra brevemente cómo trabajar con Jshell en la ejecución de sentencia que imprime un mensaje

en pantalla.

```
$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell> System.out.println("Hola mundo")
Hola mundo

jshell>
```

No quiero finalizar esta sección, sin mencionar algunas otras herramientas que permiten realizar el ciclo completo del desarrollo de una programa Java.

Los entornos de desarrollo integrado o también denominados IDEs por sus siglas en inglés (Integrated Development Environment), son herramientas que además de permitir escribir código fuente, compilar y ejecutar programas Java, ayudan en cada uno de esos procesos (especialmente la escritura del código fuente), haciendo la vida del programador mucho más sencilla con opciones como auto-completado, resaltado de líneas de código con error, depuración o debugging de código fuente y un largo etc..

Indudablemente los IDEs buscan que el proceso de desarrollo sea mucho más rápido, pero también pueden provocar malos entendidos, especialmente para quienes se inician en la programación, por ejemplo: algunos de mis alumnos cuando se les pregunta qué lenguaje de programación utilizas responden NetBeans, es decir, mencionan la herramienta y no el lenguaje de programación. Hay que tener claro que **aprenderá el lenguaje de programación Java** y usará algún IDE como NetBeans. Es como si en una clase de gramática y ortografía en español, alguien diga escribo en Microsoft Word, cuando en realidad escribe en español y usa un procesador de palabras como Word.

Dentro del mundo Java existen muchos IDEs, no pretendo hacer una revisión extensa y exhaustiva de cada uno de ellos, únicamente nombraré tres, ya que son los utilizados con mayor frecuencia.

- Eclipse - https://www.eclipse.org/downloads/packages/

- NetBeans - https://netbeans.apache.org
- IntelliJ IDEA - https://www.jetbrains.com/idea/

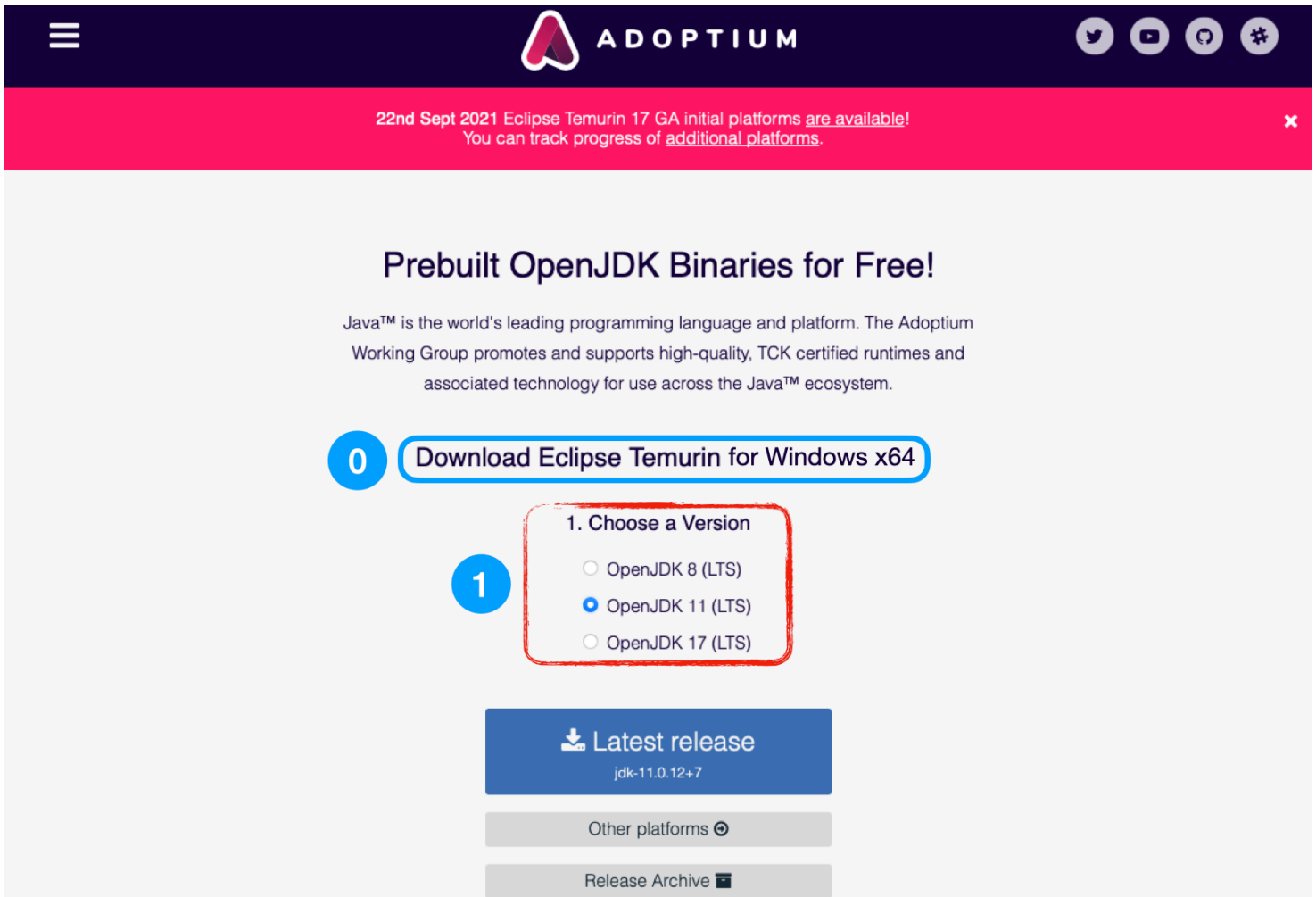
Las dos primeras opciones son gratuitas, mientras que la última tiene un modelo dual de licenciamiento, en donde, se incluye una versión comercial (denominada Ultimate) y otra gratuita (Community), su diferencia está alrededor de las herramientas adicionales que se pueden usar, pero para aprender lo básico, la versión gratuita es suficiente. También es necesario destacar que muchas instituciones de educación tienen acceso gratuito a la versión comercial únicamente para fines educativos. Es posible que tu institución tenga acceso a la versión comercial, aquí puedes encontrar el listado de instituciones por dominio de país (https://github.com/JetBrains/swot/tree/master/lib/domains).

INSTALACIÓN DEL ENTORNO DE DESARROLLO

Antes de realizar el proceso de instalación es necesario seleccionar cuál implementación de Java prefiere utilizar. Recuerde que se mencionaron tres implementaciones: Java SE de Oracle, OpenJDK y Corretto. En este documento se utilizará la implementación OpenJDK, pero eres tú quien puede elegir cualquier otra implementación.

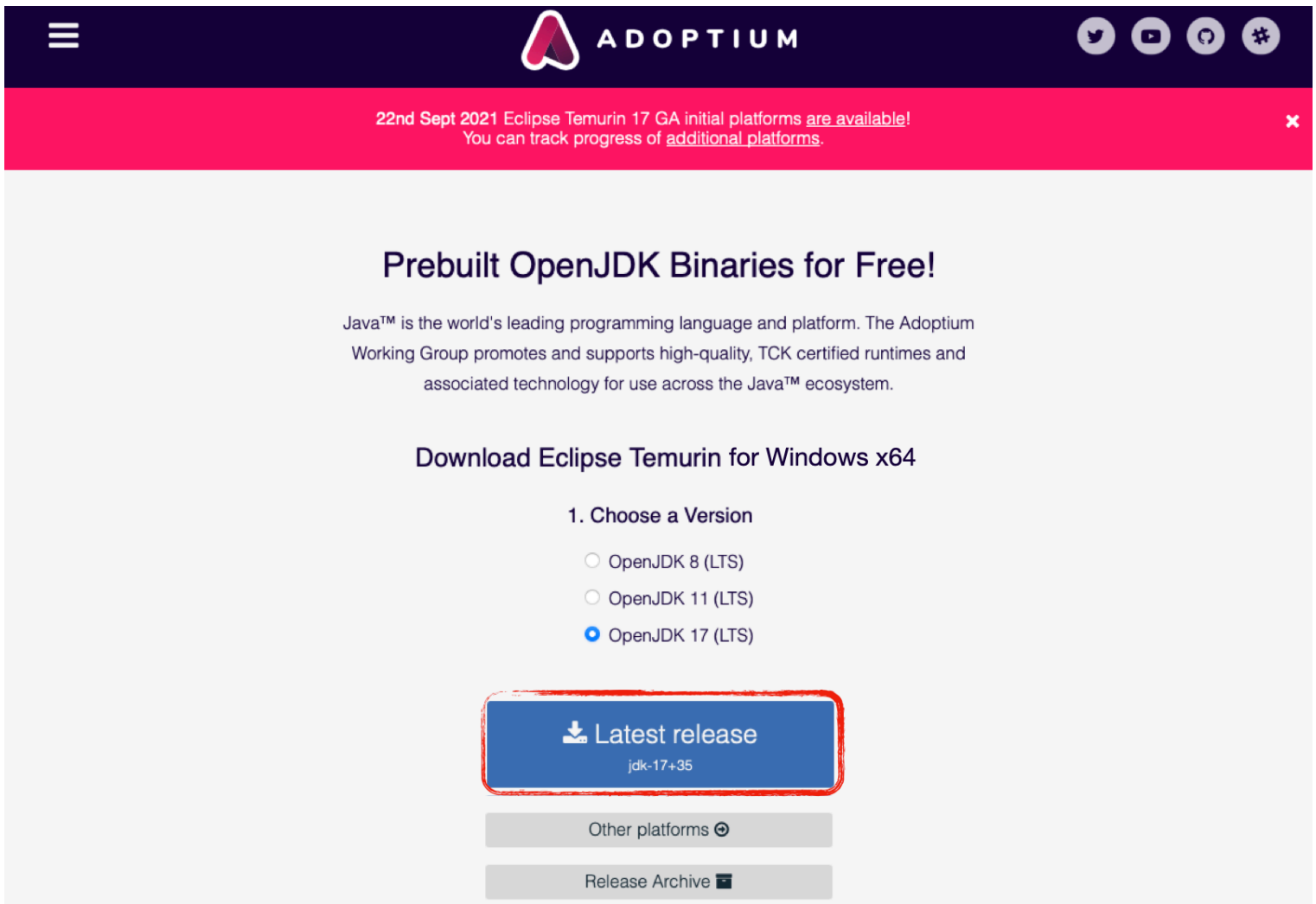
El proceso de instalación es un proceso que depende del sistema operativo con el que se trabaja, inclusive puede variar según la versión del sistema operativo que se tenga instalada. En este documento se mostrará la instalación para el sistema operativo Windows 10, es muy probable que el proceso cambie con el tiempo, pero tome el proceso de instalación con una referencia, más no como una guía definitiva.

La instalación comienza abriendo el navegador Web e ingresando a la siguiente dirección: https://adoptium.net, una imagen de esa página Web la puede ver a continuación.

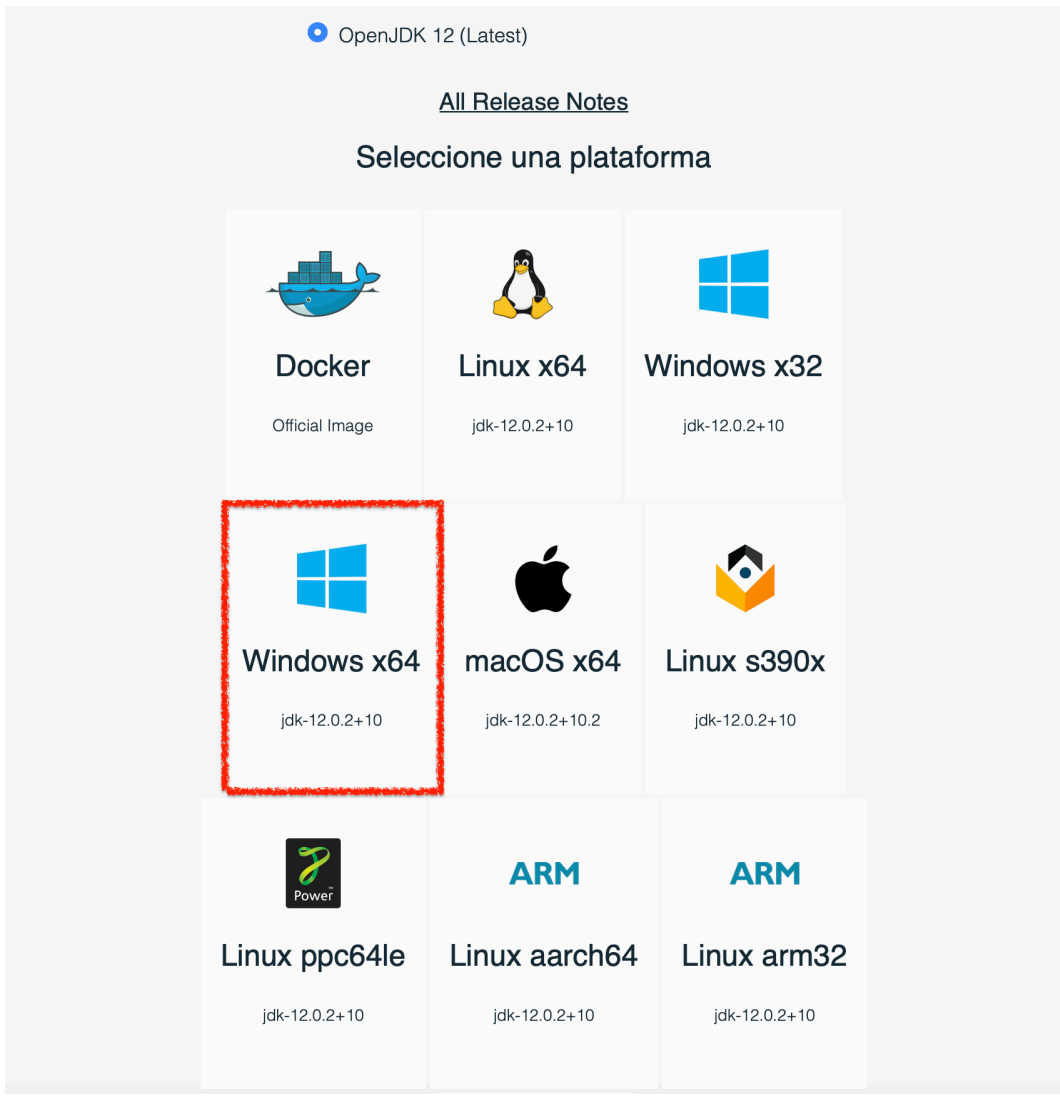


La página detecta automáticamente la versión de su sistema operativo, esto se puede ver en el recuadro marcado con la etiqueta 0. En mi caso estoy trabajando en Windows 10 y me encuentro en un equipo de 64 bits, esto por ello que se muestra Windows x64, esto podría cambiar según su sistema operativo.

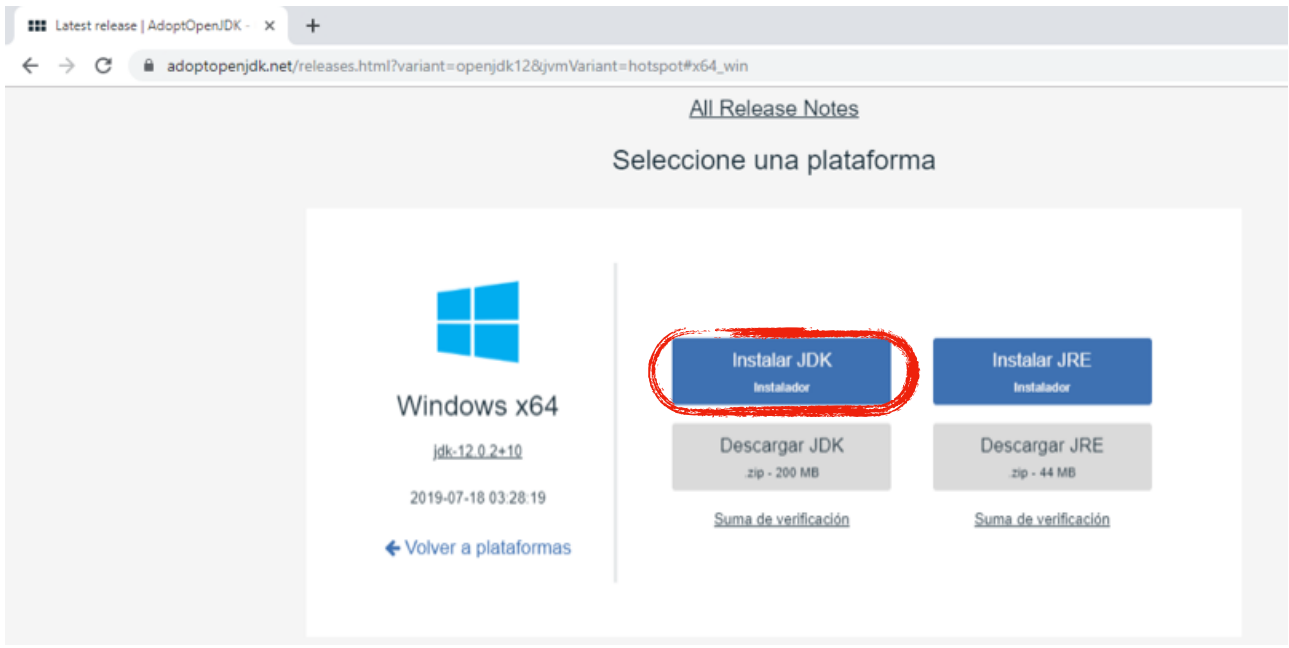
La primera acción a ejecutar se hace en el recuadro marcado como 1, debe elegir la última opción, aquella señalada con (***Latest,*** al momento de revisar esta sección era OpenJDK 17), tal como lo muestra la siguiente imagen.



La siguiente acción es dar clic en el botón que se ha encerrado en el recuadro de color rojo, esto le llevará a la siguiente página:

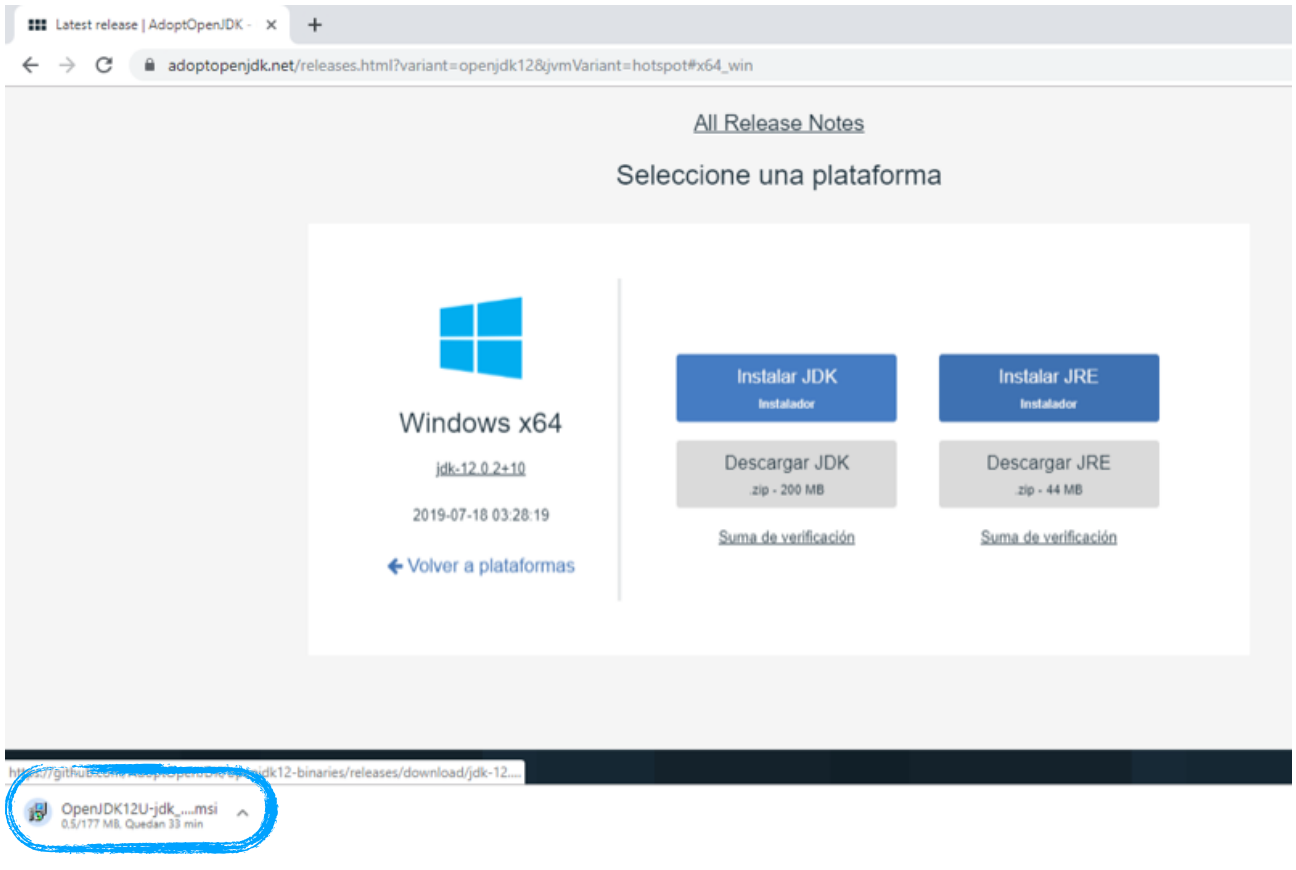


Aquí debe seleccionar su sistema operativo, el mismo que se le informó (ver la primera imagen). En mi caso debo dar clic en Windows x64. Esa acción le mostrará la siguiente página:

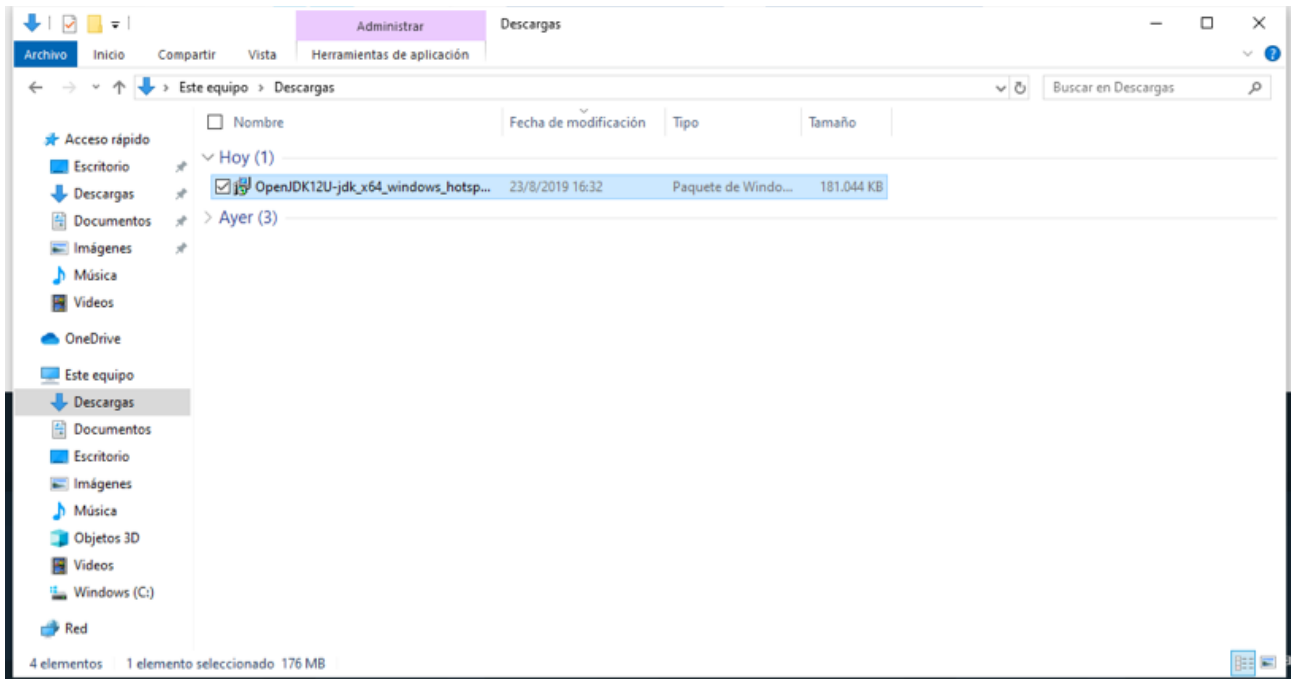


Debe dar clic en el botón encerrado en el recuadro rojo. Esto inicia el proceso de descarga del instalador, el tiempo que se necesite para la descarga dependerá de la velocidad de su conexión a internet.

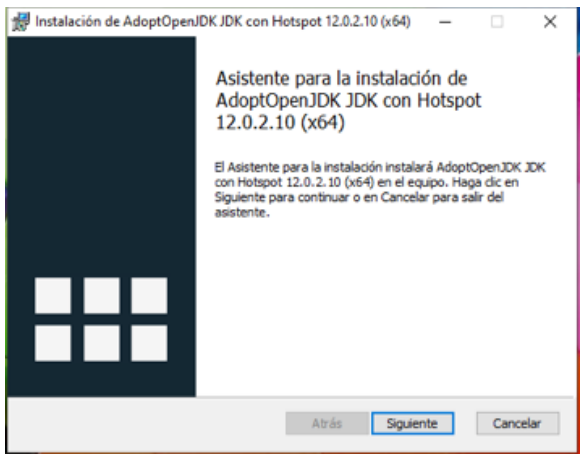
Una vez que concluya la descarga, haga clic sobre la pestaña de descarga y seleccione la opción Mostrar en carpeta, para ver el archivo descargado, como se muestra a continuación:

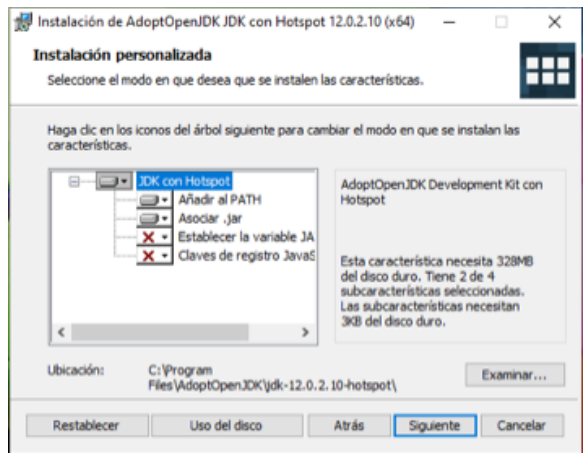


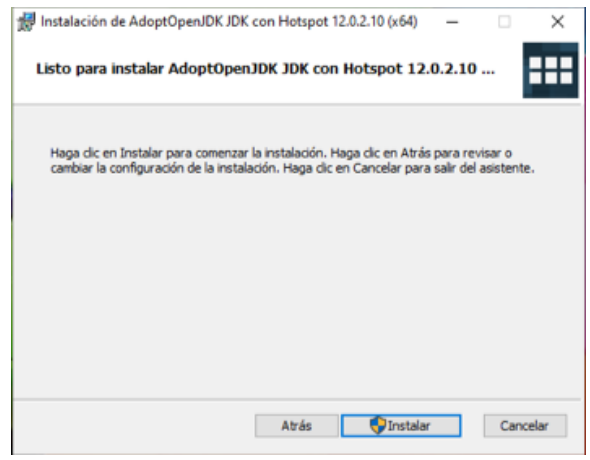
Doble clic en el archivo para que inicie el proceso de instalación.



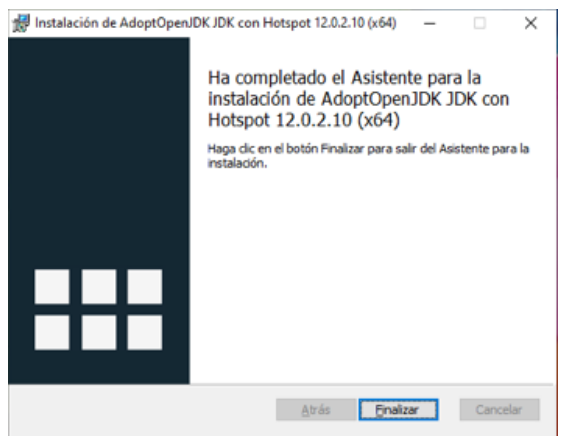
Las siguientes pantallas se generan dando clic en el botón siguiente, sin cambiar ninguna de las opciones de configuración.







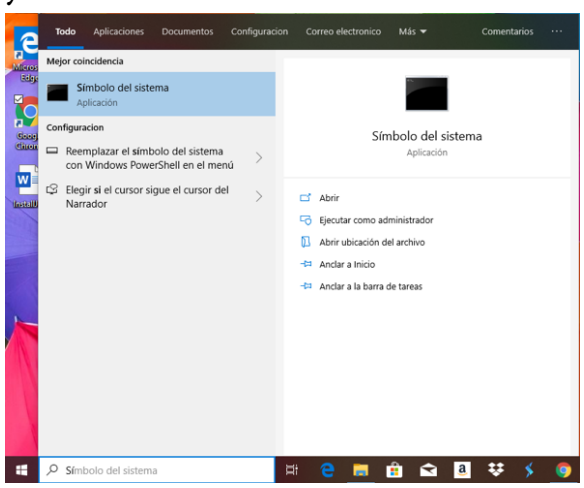
Al hacer clic en el botón instalar, de la última ventana, se debe autorizar la instalación en la primera ventana de la izquierda. Se inicia el proceso y una vez concluido se presenta la segunda imagen, finalizando así la instalación.

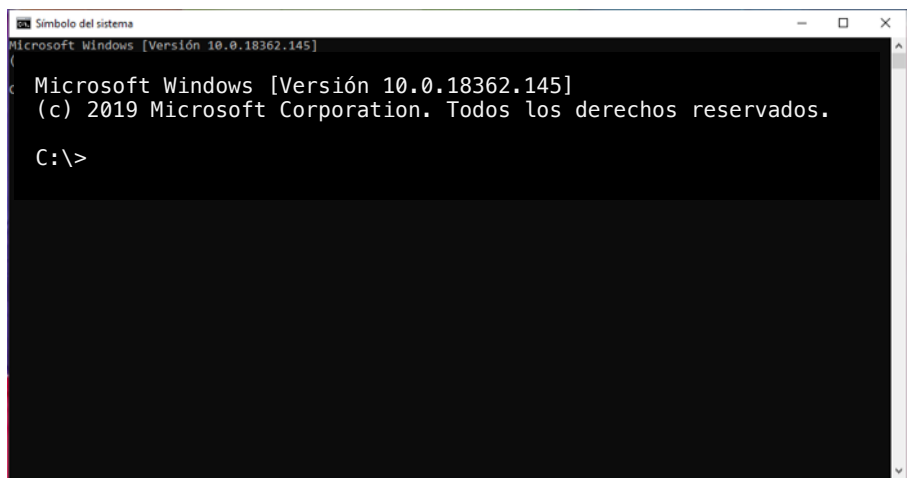


COMPROBACIÓN DE LA CORRECTA INSTALACIÓN

Una vez concluido el proceso de instalación descrito anteriormente se puede verificar escribiendo un primer programa sencillo, para ello es necesario que se ejecuten la siguientes acciones.

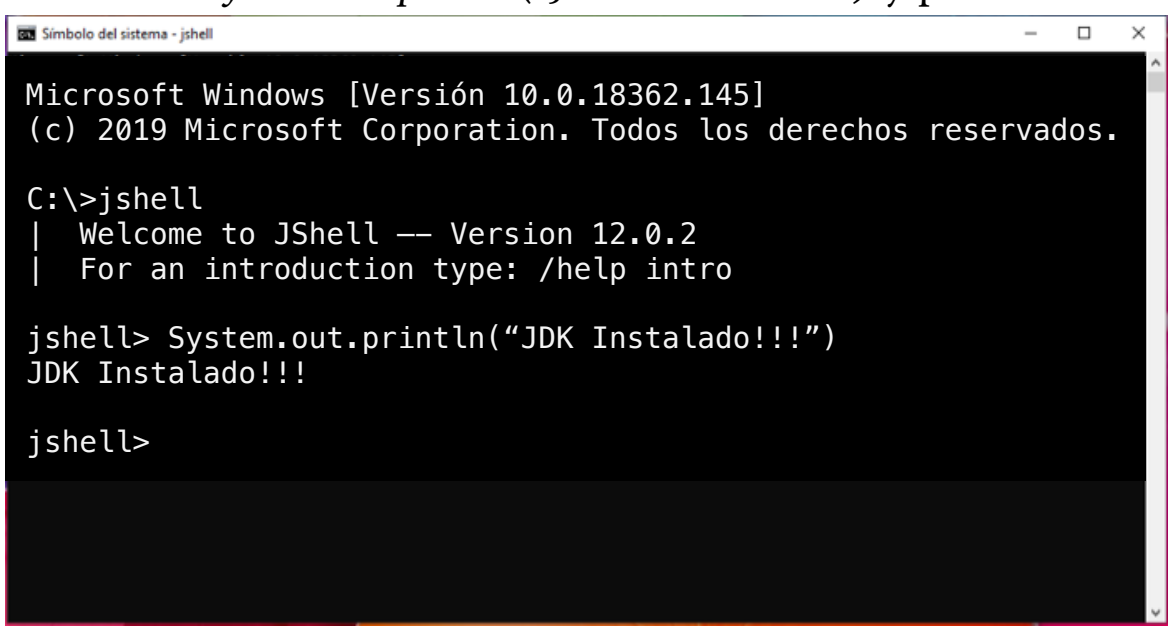
Lo primero es ejecutar el programa conocido como Símbolo del sistema, para ello, puede buscarlo como se lo hace en la primera imagen de la izquierda, selecciona la primera opción y como resultado de la acción anterior se muestra la segunda imagen.



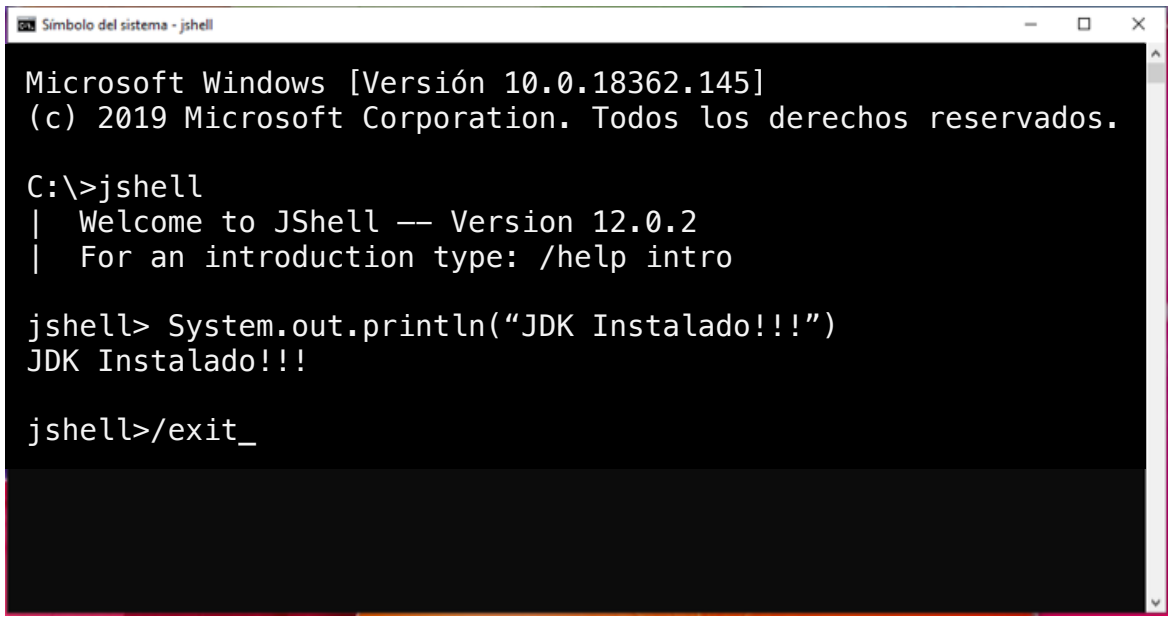


La siguiente acción es utilizar la herramienta JShell para ello es necesario escribir el comando *jshell* y presionar e*nter*.

Una vez que se encuentra dentro de JShell es necesario escribir y ejecutar el siguiente comando: *System.out.println(“JDK Instalado!!!”)* y presionar *enter*.



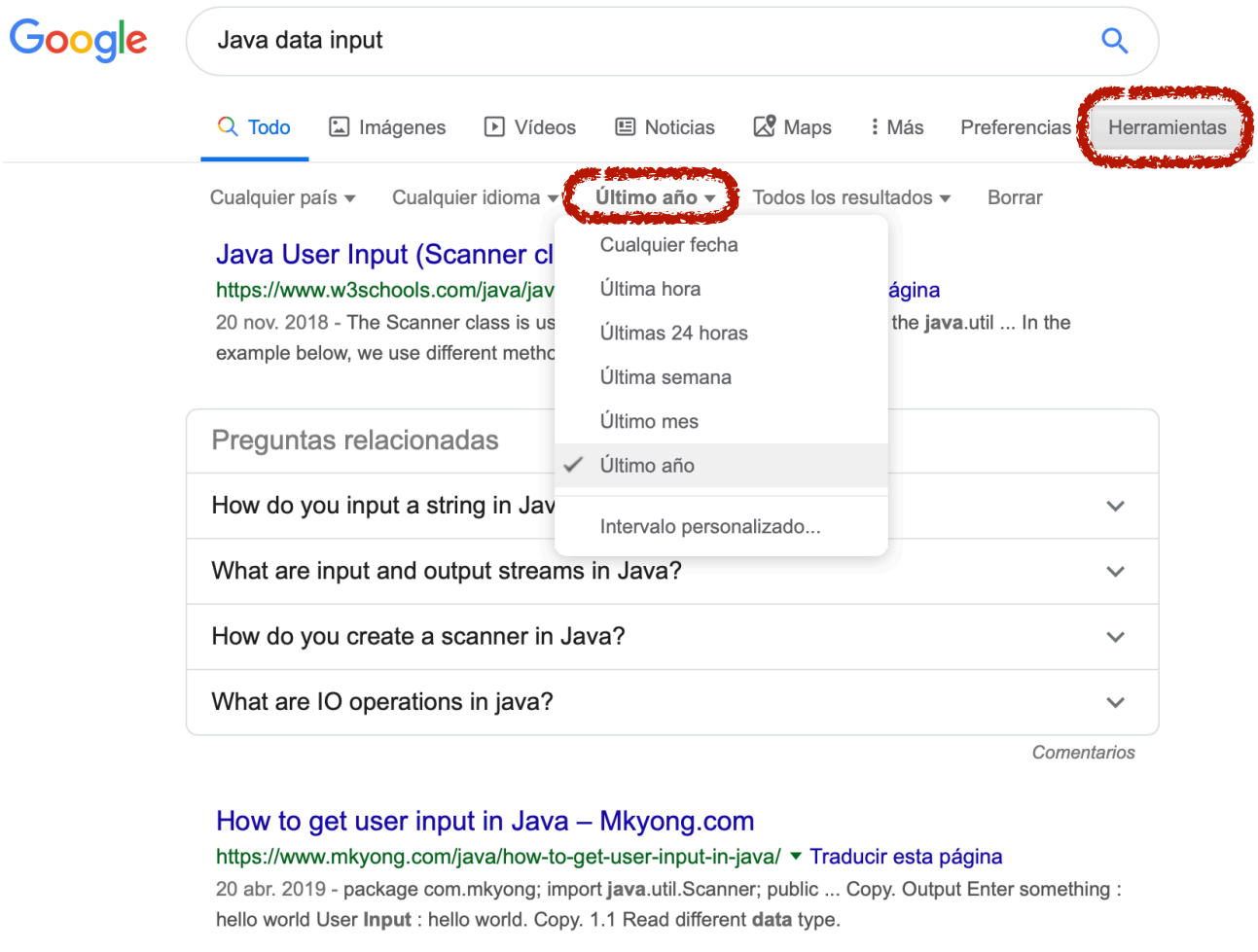
Si ve el mismo mensaje como muestra la pantalla anterior, ha instalado el entorno de desarrollo correctamente. Finalmente es necesario salir de *jshell*, para ello es necesario ejecutar el comando */exit* y presionar *enter*.



CONSEJOS AL MOMENTO DE BUSCAR RECURSOS SOBRE JAVA

Java es un lenguaje maduro y que ha evolucionado con el tiempo, así lo evidencian las diferentes versiones que tiene. Esto provoca que mucha de la información que se encuentra en la Web haga referencia a versiones anteriores del lenguaje, dejando de lado aquellas que se centran en las versiones actuales.

Tomando en cuenta lo anterior, mi recomendación es que al utilizar un buscador como Google para encontrar recursos sobre Java, lo configure para que devuelva los resultados del último año. La siguiente imagen muestra cómo hacer esto.



Esta configuración no siempre funciona ya que si tú búsqueda no devuelve resultados que hayan sido publicados en el último año, Google te mostrará los resultados más recientes, cuya fecha de indexación exceden el año, es decir podría mostrar resultados de más de 12 meses, pero esta es una mejor opción que dejar que se presenten los resultados sin ningún criterio.

VARIABLES, EXPRESIONES Y SENTENCIAS

Muchos de los lenguajes de programación utilizan los conceptos definidos en la arquitectura de von Newman para construir programas. En este capítulo conocerá algunos de esos elementos.

Todos en algún momento nos hemos encontrado resolviendo algún problema matemático o aplicando una fórmula para obtener algún valor. ¿Cómo se pueden representar esos valores o variables dentro de un lenguaje de programación?

VALORES Y TIPOS

Como seres humanos estamos acostumbrados a utilizar valores, es decir, números, letras, texto, fechas, horas, etc.. Estamos tan cómodos con los valores que a veces pasamos por alto algunos detalles, como por ejemplo que los números pueden ser enteros o reales, positivos o negativos, así como tampoco nos percatamos que el texto no es más que una secuencia de letras que puede incluir otros caracteres y signos.

También hemos aprendido a hacer operaciones utilizando esos valores, podemos sumar, restar, multiplicar o dividir, ordenar ascendente o descendente, etc. Es decir, los conceptos valores y operaciones son algunas de nuestras primeras nociones.

¿Cómo se representan en Java los valores? Para responder a esa pregunta inicie revisando la siguiente tabla.

Tipo	Representación
Entero	-1, 0, 1
Real	-1.3, 0.0, 1.7
Texto	“Loja capital musical del Ecuador”
Letra - caracter	‘A’, ‘b’, ‘1’

Los valores enteros pueden ser positivos o negativos, al igual que los reales, note que en Java se utiliza el punto (.) como separador decimal. El texto se debe encerrar entre comillas dobles (“”), mientras que los caracteres se encierran entre comillas simples (‘’).

Es momento de trabajar con valores en Java, para ello abra su terminal o ventana de comandos y ejecute el comando *jshell*[1]. Ese comando producirá una salida similar a la siguiente:

```
$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell>
```

Ha iniciado *jshell* y ahora puede empezar a ingresar algunos valores, como muestra la siguiente imagen. Ingrese un valor y presione la tecla *Enter*, así como se mencionó anteriormente utilice el punto para separar los decimales, comillas dobles para encerrar el texto y comilla siempre para letras.

```
jshell> 1
$1 ==> 1

jshell> 1.3
$2 ==> 1.3

jshell> "Hola mundo"
$3 ==> "Hola mundo"

jshell> 'A'
$4 ==> 'A'

jshell>
```

Ahora ejecute algunas operaciones básicas con valores numéricos, haga alguna suma entre dos valores, o alguna división, así como muestra la siguiente imagen. Ponga especial atención en la última operación. No olvide que no se puede hacer divisiones por 0 ya que

---

[1] Existe una versión Web de JShell, la puede encontrar en https://tryjshell.org

eso daría como resultado el infinito, o como muestra la imagen se lanzaría una excepción.

```
jshell> 1 + 5
$5 ==> 6

jshell> 3.1 + 4.4
$6 ==> 7.5

jshell> 6.0 / 1.5
$7 ==> 4.0

jshell> 4 / 0
|  Exception java.lang.ArithmeticException: / by zero
|        at (#8:1)

jshell>
```

Para salir de JShell es necesario ejecutar el comando */exit*, una vez que se ejecuta ese comando termina la ejecución del *jshell* y su terminal volverá a la normalidad.

Java es un lenguaje de programación que se caracteriza por ser fuertemente "*tipeado*", es decir que cualquier valor debe tener asociado un tipo de dato. Es por ello que en las siguientes líneas se estudiará los tipos de datos en Java.

Java tiene varios tipos de datos, pero empecemos por los siguientes cuatros tipos y cuando sea necesario se describirán el resto de tipos de datos

Tipo de dato	**Java**
Entero	int
Real	double
Caracter	char
Texto	String

Ahora comprobará que en el ejemplo anterior, Java determinó el tipo de dato de cada uno de los valores que ingresó, para ello en una ventana terminal ejecute el comando *jshell -v*. El

argumento *-v* señala que retro-alimente cada una de las acciones ejecutadas.

```
$ jshell -v
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell>
```

Vuelva a ingresar valores, enteros, reales, texto y caracteres, como lo hizo anteriormente y observará cómo aparece información adicional que entre otras cosas indica el tipo de dato de cada uno de los valores ingresados, *int*, *double*, *String* o *char*. Además de los tipos de datos de seguro notó el texto que señala que se han creado las variables *$1*, *$2*, *$3* y *$4*.

```
jshell> 1
$1 ==> 1
|  created scratch variable $1 : int

jshell> 1.3
$2 ==> 1.3
|  created scratch variable $2 : double

jshell> "Hola mundo"
$3 ==> "Hola mundo"
|  created scratch variable $3 : String

jshell> 'A'
$4 ==> 'A'
|  created scratch variable $4 : char
```

En el siguiente apartado aprenderá todo acerca sobre las variables utilizando el lenguaje de programación Java.

VARIABLES

Las variables son un mecanismo para almacenar los valores que un programa necesita para trabajar. Estos valores se almacenan en la memoria del computador, por lo que una variable es una forma sencilla de acceder a un espacio específico de la memoria RAM de un computador.

Las primeras acciones que aprenderá es a declarar y asignar valores a una variable. Iniciemos con la declaración.

Para declarar una variable en Java necesita de dos elementos, el tipo de dato de la variable y el nombre de la variable. La forma estándar se muestra a continuación.

```
<Tipo de dato> <nombre de la variable>;
```

Cuando se declara una variable de un tipo de dato, esa variable podrá almacenar valores que correspondan únicamente a ese tipo dato, si trata de almacenar otro valor se lanzará un error. Veamos la declaración de variables en JShell. Ejecute Jshell usando el comando revisando anteriormente.

```
jshell> int edad
edad ==> 0

jshell> double peso
peso ==> 0.0

jshell> String nombre
nombre ==> null

jshell> char genero
genero ==> ''
```

Dentro de JShell al declarar una variable se asigna un valor por defecto según el tipo de dato, así a las variables de tipo de dato entero se asignará 0, a las de tipo de dato real 0.0, al texto *null* y a las de tipo carácter el carácter vacío. Estos valores se pueden modificar, pero para ello necesita conocer la operación de asignación.

Asignar un valor a una variable es una operación que dentro del lenguaje de programación Java se hace utilizando el signo de igual (=). Para ello, necesita el nombre de la variable y el valor que será asignado a la variable. Si la variable tenía un valor anterior este se pierde. La forma general de asignar un valor a un variable es la siguiente:

```
<nombre de la variable> = <valor>
```

Para asignar un valor a la variable es necesario que esta haya sido declarada con anterioridad, además, el valor que se asignará deberá ser del mismo tipo de dato que se usó en su

declaración, si no se cumple con estos requisitos se lanzará un error. Veamos varios ejemplos de asignación.

```
jshell> edad = 28
edad ==> 28

jshell> peso = 74.3
peso ==> 74.3

jshell> nombre = "Jorge"
nombre ==> "Jorge"

jshell> genero = 'M'
genero ==> 'M'

jshell> edad = "18"
|  Error:
|  incompatible types: java.lang.String cannot be converted to int
|  edad = "18"
|         ^--^

jshell> isbn = 31123
|  Error:
|  cannot find symbol
|    symbol:   variable isbn
|  isbn = 31123
|  ^--^
```

Las últimas dos asignaciones lanzan un error. En el primero se trata de asignar una cadena de texto a una variable declarada como entero. Mientras que, el segundo mensaje de error señala que se trata de usar una variable que no ha sido declarada. Así que si tiene errores como estos ya conoce las causas y cómo resolverlos.

También es posible realizar la declaración y asignación de las variables a esto generalmente se lo conoce como inicialización de una variable. La siguiente imagen muestra un ejemplo de declaración e inicialización de 4 variables.

```
jshell> int edad = 28
edad ==> 28

jshell> double peso = 74.3
peso ==> 74.3

jshell> String nombre = "Jorge"
nombre ==> "Jorge"

jshell> char genero = 'M'
genero ==> 'M'
```

Por último es posible que Java infiera el tipo de dato de una variable, para ello es necesario asignar un valor inicial a la variable que previamente fue identificada con la palabra *var*, para comprender mejor esto vea la siguiente porción de código.

```
jshell> var edad = 28
edad ==> 28

jshell> var peso = 74.3
peso ==> 74.3

jshell> var nombre = "Jorge"
nombre ==> "Jorge"

jshell> var genero = 'M'
genero ==> 'M'
```

En cada una de las declaraciones no se especifica el tipo de dato de la variable, pero se utiliza el valor inicial para inferir cada uno de los tipos de datos. Esta forma de declaración e inicialización de variables es bastante útil y recuerde que únicamente se puede usar cuando se asigna un valor a una variable.

NOMBRES DE VARIABLES Y PALABRAS RESERVADAS

Java, así como todos los lenguajes de programación, posee estándares para nombrar todos los elementos que se declaran y se usan en un programa y las variables no son la excepción.

Una versión en español del estándar de programación para el lenguaje Java se puede encontrar en: https://www.um.es/docencia/vjimenez/ficheros/practicas/ConvencionesCodigoJava.pdf. En resumen, para las variables se sugiere las siguientes convenciones:

- Una variable se escribe en minúsculas. Java es un lenguaje “case sensitive” es decir que diferencia entre mayúsculas y minúsculas
- Debe iniciar con una letra, aunque podría iniciar con $ o _, pero generalmente se usan letras.
- En caso de estar formada para más de una palabra, la primera letra, a partir de la segunda palabra, se escribe en mayúscula.

En palabras de Robert C. Martin, los nombres de variables, además de seguir un estándar de programación, deben ser seleccionados de tal manera que representen la función que desempeñan dentro del programa, pero en especial que revelen por qué existen, qué es lo que hacen y cómo usarlas.

Si bien una variable puede tener cualquier nombre, existe un conjunto de nombres que no se puede utilizar ya que tienen un significado dentro del lenguaje de programación. Un listado de las palabras reservadas o Keywords se describen en la siguiente tabla:

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

Ninguna de esas palabras se pueden utilizar como nombres de variables ya que son empleadas por el lenguaje de programación para su funcionamiento.

Finalmente recordar que el uso de estándares de programación hace más legible un programa, de ninguna manera evita o genera errores en los programas.

ENTRADA Y SALIDA DE DATOS

Cuando nos iniciamos en programación, una de las primeras acciones que se busca hacer es ingresar datos y presentarlos.

Para la presentación de los datos, tarea que también se denomina salida de datos, el lenguaje Java proporciona los siguientes métodos:

Método	Descripción	Ejemplo	Salida
System.out.print	Imprime y luego ubica el cursor a continuación del texto	System.out.print(“Nombre: ”)	Nombre:
System.out.println	Imprime, da un salto de línea y ubica el cursor en la nueva línea	System.out.println(“Mi Libro ”)	Mi Libro
System.out.printf	Se utiliza cuando se debe mezclar texto con valores de variables.	System.out.printf(“Name: %s”, name)	Name: Loja

La primera sentencia, la recomiendo para construir interfaces de usuario tipo formulario, en donde aparece una etiqueta o texto para indicar que se debe ingresar cierta información. El segundo método, para presentar un mensaje o texto que describe algo. Finalmente el tercer método, para mezclar texto con valores de variables de una forma sencilla.

Es necesario detallar el funcionamiento del printf. En la documentación del método se dice que es una forma conveniente para escribir una mensaje con formato, utilizando una cadena de formato que incluye instrucciones para mezclar texto con valores de variables. Analice el siguiente código:

```
var nombre = “Jorge”
var edad = 42
var estatura = 1.77
var institucion = “j4loxa”

System.out.printf(“Hola %s, tienes %d años y mides %fm. Bienvenido a la %s\n",
    nombre,
    edad,
    estatura,
    institucion);
```

El método printf, recibe un número variable de parámetros. El primero es la cadena con formato en dónde se mezcla el texto con marcas de formato que serán reemplazados por los valores de los parámetros adicionales. Es decir que la salida del código anterior será: *Hola Jorge, tienes 42 años y mides 1.77m. Bienvenido a j4loxa*.

Existen diferentes marcas de formato, tantas como tipos de datos, pero por ahora vamos a utilizar las siguientes:

Marca	Usar con el tipo de dato
%s	String
%d	Enteros
%f	Reales

Es necesario tener en cuenta que deben existir tantas marcas de formato, como variables, si alguna falta, se lanzará un error. El tipo de dato de cada variable determina el tipo de marca, y no se admiten cambios.

También es necesario señalar que las marcas de formato pueden tener propiedades que ayudan a construir la salida. Esas propiedad se ubican entre el signo % y la letra que determina el tipo de dato. Por ejemplo la marca *%f* tiene un propiedad que determina el número de dígitos decimales que presentará, así: *%.2f*, señalaría que se presentarán dos dígitos en la parte decimal del número que se presentará.

En el texto del ejemplo anterior, también se observa una secuencia de escape un combinación de dos caracteres que inician con un backslash (\). Existen varias, pero por ahora se usaran únicamente dos:

- \n que señala un salto de línea.
- \t para señalar un tabulación hacia la derecha.

Es decir, cuando se encuentra una secuencia de escape, será reemplazada por su equivalente, salto de línea o tabulación.

Los siguientes párrafos se dedican al ingreso de datos. Aunque es menester hacer una observación, en entornos como Jshell, el ingreso de datos se debería hacer utilizando valores iniciales a las variables. Esto debido a que Jshell ejecuta cada línea de código que se ingresa, por lo que la presentación no trabaja como se espera.

Dentro del lenguaje Java, existen diferentes mecanismos para realizar el ingreso de los datos. Estos mecanismos han evolucionado con el tiempo, tratando de disminuir la número de

líneas de código que se necesitan. Es por ello que aquí se utilizará la clase *Scanner* para realizar dicha tarea.

Recuerde que Java es un lenguaje orientado a objetos, es por ello que también está tarea se realiza a través de clases y objetos. En términos generales, para el ingreso de datos se debe:

- Importar la clase Scanner
- Crear un objeto de dicha clase
- Y utilizar sus métodos.

Analice la siguiente porción de código que muestra un ejemplo:

```
import java.util.Scanner;

Scanner l = new Scanner(System.in);
int edad;
double estatura;
String nombre;

System.out.print("Ingresa tú nombre: ");
nombre = l.next();
System.out.print("Ingresa tú edad (años): ");
edad = l.nextInt();
System.out.print("Ingresa tú estatura (metros): ");
estatura = l.nextDouble();
```

En la primera línea usted encuentra cómo se importa la clase Scanner, en la segunda línea de código se crea un objeto, denominado *l*, que es del tipo de la clase Scanner. Para finalmente en las líneas posteriores encontrar algunos métodos de esa clase.

El resumen de los métodos es el siguiente:

Método	Uso
next	Permite leer cadenas de texto, hasta encontrar un espacio en blanco.
nextLine	Se usa cuando se necesita leer un texto que contiene espacios en blanco.
nextInt	Lee números enteros.
netxDouble	Para leer número reales

Como ya se mencionó, JShell no es la mejor herramienta para realizar el ingreso de datos, una solución alternativa que puede servir, es integrar el mensaje con la lectura de datos de la siguiente manera:

```
import java.util.Scanner;

Scanner l = new Scanner(System.in);
int edad;
double estatura;
String nombre;

System.out.print("Ingresa tú nombre: “); nombre = l.next();
System.out.print("Ingresa tú edad (años): “); edad = l.nextInt();
System.out.print("Ingresa tú estatura (metros): “); estatura = l.nextDouble();
```

Eso hará que ambas sentencias (*System.out.println y l.next…*) se ejecuten como si fueran una, mejorando así la presentación. Aunque esta forma de trabajo no la recomiendo y sugiero que se use la inicialización de variables como forma de ingreso de datos.

OPERADORES Y OPERANDOS

Los operadores son símbolos que representan cálculos, mientras que los valores que estos utilizan se denominan operandos. Así como la mayoría de seres humanos usted debe conocer algunos de los operadores que se utilizan comúnmente. La siguiente tabla muestra un resumen de los operadores aritméticos de uso común en Java.

Operación	Operador
Suma	+
Resta	-
Multiplicación	*
División	/
Asignación	=
Módulo	%

En Java, como en la mayoría de lenguajes de programación, se utiliza un el asterisco (*) para señalar multiplicación. El operador módulo devuelve el residuo de una división entera, es decir sin parte decimal.

Un ejemplo del operador módulo se puede ver en la siguiente imagen. En ese caso se divide 7 para 4 y se devuelve el residuo que es 3.

```
jshell> 7 % 4
$1 ==> 3
```

Otra característica que puede resultar "inexplicable", es la división de dos número enteros, si se divide 5 para 2, en la aritmética tradicional la respuesta sería 2.5, pero como puede ver más abajo, esto no es así en Java. La razón se encuentra en los tipos de datos de los operandos, es decir, que cuando ambos operandos son del tipo de dato entero, el resultado de la división será un valor entero.

```
jshell> 5 / 2
$4 ==> 2
```

Para obtener un valor con decimales uno de los operandos debe ser del tipo de dato real, puede ser el numerador o del denominador. Analice las siguientes porciones de código:

```
jshell> var a = 5.0
a ==> 5.0

jshell> var b = 2
b ==> 2

jshell> a / b
$10 ==> 2.5
```

```
jshell> var a = 5
a ==> 5

jshell> var b = 2.0
b ==> 2.0

jshell> a / b
$7 ==> 2.5
```

```
jshell> var a = 5
a ==> 5

jshell> var b = 2
b ==> 2

jshell> (double)a / b
$13 ==> 2.5
```

```
jshell> var a = 5
a ==> 5

jshell> var b = 2
b ==> 2

jshell> a / (double)b
$16 ==> 2.5
```

En las primeras dos (empezando por la izquierda), se cambia el numerador o el denominador a *double*, mientras que, en las dos últimas se utiliza el *casting* de datos para transformar la variable de entero a real. El *casting* se puede utilizar en cualquiera de los dos operandos y el resultado será el mismo.

Java posee otros operadores, pero, por ahora, los que revisó son suficientes para realizar un sin número de programas. Cuando sea necesario se comentarán los nuevos operadores.

Un recurso que le ayudará a ampliar este tema se encuentra aquí: https://vimeo.com/426869946

EVALUACIÓN DE EXPRESIONES

Una expresión es una construcción compuesta por variables y operadores que se forma de acuerdo a la sintaxis del lenguaje y que dan como resultado un único valor.

Se utilizaron varias expresiones para asignar valores a variables o para realizar cálculos, recuerde que el tipo de dato del resultado de la expresión debe ser del mismo tipo de dato de la variable, caso contrario existirá un error. En el caso de la asignación, primero se resuelve la expresión y su resultado se asigna a la variable.

```
jshell> int a = 10
a ==> 10

jshell> a = a - 2
a ==> 8
```

En el código anterior, a la variable *a* se le asigna el valor de 10. En la siguiente expresión, se resta 2 unidades a esa variable y el resultado de esa expresión se asigna nuevamente a la variable *a*.

En la siguiente porción de código se muestra un ejemplo que calcula cuántos Kilo bytes existen en 16732 bytes.

```
jshell> int totalBytes = 16732
totalBytes ==> 16732

jshell> double kiloBytes = totalBytes / 1024.0
kiloBytes ==> 16.33984375
```

Algunas expresiones pueden ser ambiguas, como por ejemplo 21 + 79/100 o 8/2 * (2 + 2) esta última se hizo viral en agosto de 2019 en la red social Twitter. En el siguiente apartado conocerá la respuesta a esas expresiones y sobretodo la razón de los valores que obtuvo.

PRECEDENCIA DE OPERADORES

Cuando más de dos operadores aparecen en una expresión, el orden de evaluación depende de las reglas de precedencia de los operadores. Estas reglas determinan el orden en el que se deben ejecutar las operaciones, para ello asignan un valor de precedencia a cada operador y siguen la siguiente regla: el operador de mayor precedencia se ejecuta en primer lugar.

El orden de los operadores, en Java, que conoce, hasta el momento se resume en la siguiente tabla:

Orden	Operador	Descripción	Asociatividad
16	()	Paréntesis	Izquierda a derecha
13	()	Casting	Derecha a izquierda
12	* / %	Multiplicativos	Izquierda a derecha
11	+ -	Adición	Izquierda a derecha
1	=	Asignación	Derecha a izquierda

El resto de la tabla lo puede encontrar en: https://introcs.cs.princeton.edu/java/11precedence/

Siguiendo el orden de precedencia, la expresión: 21 + 79/100 da como resultado 21.79 ya que primero se ejecuta la división (orden 12) y luego la suma (orden 11). ¿Y la otra expresión 8/2 * (2 + 2)? Siguiendo la misma regla podríamos decir que primero paréntesis y luego ¿qué resuelve multiplicación o división? Ya que ambas tienen igual precedencia. El siguiente párrafo le ayudará

La asociatividad señala que cuando dos operadores tienen el mismo orden de precedencia, la expresión se evalúa de acuerdo a la asociatividad señalada en esa columna. Volviendo a la expresión 8/2 * (2 + 2) por precedencia queda 8/2 * 4, primero paréntesis, en segundo lugar y por asociatividad (izquierda a derecha) división, 4 * 4 dando como resultado 16. Lo invito a usar JShell para verificar que el resultado es correcto.

Todo esto puede ser complicado, sobretodo memorizar la asociatividad. Para evitar errores y confusiones es mejor utilizar paréntesis para agrupar los operadores en el orden necesario que demanda el problema, ya que los paréntesis cambian el orden de precedencia de los operaciones que agrupa.

SENTENCIAS

Las sentencias son más o menos equivalentes a las oraciones en existen en los lenguajes naturales. Una sentencia forma una unidad completa de ejecución. Las expresiones pueden convertirse en sentencias si terminan con punto y coma (;), así por ejemplo $a = 23;$ es una

sentencia ya que termina con punto y coma. A expresiones como las anteriores suelen llamarse como sentencias de expresión.

Adicionalmente a las este tipo de sentencias, existen las sentencias de declaración y de control de flujo. Una sentencia de declaración declara una variable, una sentencia de control de flujo, regula el flujo de ejecución de un programa (estas se verán mas adelante).

También existe el concepto de bloque de sentencias, un bloque de sentencias está formado por cero o más sentencias que se encuentran entre llaves balanceadas (llave de apertura { como de cierre }).

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeo**: Práctica Guiada #1 - Variables y operaciones fundamentales básicas (https://vimeo.com/426869946).
- **Laboratorio**: Variables, entrada y salida (http://j4loxa.com/courses/java101/blog/2017/lab1.html).

MÉTODOS ÚTILES

Hasta el momento conoce los operados aritméticos y cómo usarlos. Pero, no son suficientes para resolver todos los problemas, por ejemplo si necesita encontrar las raíces de una ecuación cuadrática, debe calcular una raíz cuadra y elevar un número al cuadrado. Alguien podría decir que puede usar el método de Newton y la multiplicación para la primera y segunda operación respectivamente. Sin duda sería un buen ejercicio para mejorar las habilidades de programación, pero existen un grupo de métodos pre-construidos que vienen en los lenguajes de programación y que es necesario aprender a utilizar.

MÉTODOS MATEMÁTICOS

En matemáticas todos hemos visto funciones tales como seno, coseno u otras como los logaritmos naturales. Aprendimos a evaluar expresiones tales como $seno(180°)$ y $log(1/x)$, en las cuales lo primero que se hace es evaluar la expresión que se encuentra entre paréntesis, que se denomina argumento de la función. Lo segundo es evaluar la función en sí misma, generalmente utilizando una calculadora.

Dentro del lenguaje de programación Java existe una clase que contiene muchos de estos métodos matemáticos. Esa clase se denomina *Math*. Dicha clase no sólo posee métodos sino también cuenta con los valores de las constantes *PI* y *E*.

En la documentación de Java se dice que la clase *Math* contiene métodos para realizar operaciones numéricas básicas, tales como las funciones exponenciales, logarítmicas, raíces cuadradas y trigonométricas.

Son muchos los métodos que posee la clase como para nombrarlos a todos. Es por ello que se mostrarán únicamente aquellos que considero como los mas utilizados. El detalle de todos los métodos los pueden encontrar en la documentación del lenguaje. Así:

- Ir a https://docs.oracle.com/en/java/
- Seleccionar el link Java SE documentation
- API Documentation (en la parte izquierda)
- En la columna Module, seleccionar *java.base*
- En la la columna Package, seleccionar *java.lang*

- Y buscar la clase *Math*, en la tabla Class Summary.

La siguiente tabla muestra algunas de las funciones de la clase *Math*.

Método	**Descripción**
abs(x)	Devuelve el valor absoluto del número x
cos(x)	Devuelve el coseno del ángulo x cuya unidad son los radianes
sin(x)	Devuelve el seno del ángulo x cuya unidad son los radianes
tan(x)	Devuelve la tangente del ángulo x cuya unidad son los radianes
log(x)	Para obtener el logaritmo de base *E* del argumento x
log10(x)	Obtiene el logaritmo de base 10 de x
max(x, y)	Devuelve el número mayor entre x e y
min(x, y)	Devuelve el número menor entre x e y
pow(x, y)	Eleva la base x a la potencia y
sqrt(x)	Calcula la raíz cuadrada del número x
toDegrees(x)	Convierte al ángulo x de radianes a grados
toRadians(x)	Convierte el ángulo x de grados a radianes

¿Qué hacer si necesito calcular una expresión que incluya un elemento así $\sqrt[3]{9}$? En el listado de la tabla anterior no se muestra un método que permita calcular la raíz cúbica de un número. Generalizando no existe un método que permita calcular la raíz *y* de un número *x* $\sqrt[y]{x}$.

Para ello es necesario recurrir a la siguiente equivalencia matemática: $\sqrt[y]{x} = x^{\frac{1}{y}}$ En términos de métodos sería: $\sqrt[y]{x} = pow(x,1.0/y)$ Observe como ubico 1.0 para señalar que el resultado sea real.

Dentro de los métodos en general, en donde se incluye a los matemáticos, existen algunos conceptos que se deben aclarar, estos se estudiarán a detalle más adelante (existe un apartado en donde se estudia a detalle los métodos). Ahora mismo es necesario comprender que los métodos matemáticos poseen argumentos y devuelven valores.

Los argumentos son los valores que suministramos al método. En la tabla anterior *x* e *y* son los **argumentos** que se envían a los métodos. En la misma tabla la gran **mayoría de métodos devuelven valores de tipo real**. Tres **excepciones** son los métodos *abs*, *max* y *min*.

Esos métodos devuelven o un argumento (el valor absoluto) o uno de los argumentos *x* o *y*, por lo que el tipo de dato que devuelve depende del tipo de esos argumentos. Es decir si *x* e *y* son enteros, los métodos *abs*, *max* o *min*, devolverán un entero, en cambio si son reales, esos mismos métodos, devolverán un real. ¿Qué devolverían si *x* es entero e *y* es real? Más adelante conocerá cómo invocar a esos métodos y podrá responder a esa pregunta.

INVOCACIÓN DE MÉTODOS

Dentro de los lenguajes que siguen el paradigma de la orientación a objetos, el concepto de clase es fundamental. Si bien este documento no tiene el objetivo explicar programación orientada a objetos, es necesario mencionar algo sobre las clases.

Una clase es una abstracción del mundo real que incluye propiedades y métodos. Es de nuestro interés, por ahora, el tema de métodos y en particular aquellos que ya incorpora el lenguaje de programación.

Como se mencionó anteriormente, una de las clases que posee muchos métodos útiles es la clase *Math*. Para invocar a un método dentro de cualquier clase se debe seguir la siguiente notación:

```
<Nombre de la clase>.<método>(<argumentos>)
Math.sin(1)
```

El estándar del lenguaje de programación Java, permite diferenciar cada uno de sus elementos. Así, el nombre de una clase inicia con un letra mayúscula, mientras que para un método con minúscula. El nombre de una clase es un sustantivo (en la medida de lo posible), el de un método es un verbo. Si usted recuerda esto podrá diferenciar entre variables, clases y métodos.

Las siguientes porciones de código muestran cómo invocar a varios métodos de la clase *Math*, desde *Jshell*.

```
jshell> Math.max(4, 6)
$7 ==> 6
```

```
jshell> Math.min(4, 6)
$8 ==> 4
```

```
jshell> Math.pow(2, 3)
$9 ==> 8.0
```

```
jshell> Math.pow(9, 1.0/3)
$12 ==> 2.080083823051904
```

```
jshell> Math.max(1, 1.0)
$11 ==> 1.0
```

En las expresiones anteriores puede encontrar la respuesta a una pregunta que se planteó anteriormente sobre el tipo de dato que se devuelve cuándo los parámetros son de diferente tipo de dato (ver último código).

```
jshell> var valueMax = Math.max(10, 12)
valueMax ==> 12

jshell> var root = Math.pow(1, 1.0/valueMax)
root ==> 1.0

jshell> var root = Math.pow(8, 1/Math.max(10, 12))
root ==> 1.189207115002721

jshell> System.out.println(Math.toDegrees(root))
68.13654865658464
```

El invocar a un método de la clase *Math* devuelve un valor, ese valor se puede asignar a una variable o utilizarlo en alguna operación o presentarlo o etc. Imagínelos como variables, en donde puede ir una variable puede ir un método de la clase *Math*. La porción de código de arriba muestra parte de los diferentes usos de los métodos, la de abajo muestra un error al tratar de asignar un valor real a un variable entera.

```
jshell> int value = Math.abs(-2.1)
|  Error:
|  incompatible types: possible lossy conversion from double to int
|  int value = Math.abs(-2.1);
|              ^-----------^
```

Es momento de ver otras clases y sus métodos para realizar tareas como la conversión de tipos de datos.

CONVERSIÓN DE TIPOS

Dentro de los lenguajes de programación es posible transformar valores entre diferentes tipos de datos, es decir a un entero lo puedo transformar en real o a una cadena de texto en número o viceversa. No es recomendable transformar un real a un entero, ya que se pierde precisión, aconsejo utilizar el método *round* de la clase *Math* que ayuda en estos casos.

El primer mecanismo de transformación entre valores numéricos fue el casting que se resume así:

```
(<Tipo de dato destino>)<valor>
(double)100
```

En el caso anterior, el resultado sería 100.0, es decir que cambió de entero a real.

Para transformar de real a entero vea los casos que se muestran en la siguiente porción de código:

```
jshell> Math.round(100.0)
$18 ==> 100

jshell> Math.round(301.5)
$19 ==> 302

jshell> Math.round(301.3)
$20 ==> 301
```

Convertir un número a cadena de texto se puede hacer con el método *valueOf* de la clase *String*, como se muestra en la siguiente porción de código.

```
jshell> var value = 1265.987
value ==> 1265.987

jshell> String.valueOf(value)
$22 ==> "1265.987"

jshell> String.valueOf(123)
$23 ==> "123"

jshell> String.valueOf(87.87612)
$24 ==> "87.87612"
```

La conversión de cadenas de texto (String) a datos numéricos (entero o real) se cubrirá mas adelante en el apartado de cadenas de texto.

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeo**: Práctica Guiada #2 - La secuenciación (https://vimeo.com/410467485).

- **Laboratorio**: Secuenciación (http://j4loxa.com/courses/java101/blog/2017/lab2.html).

CONDICIONALES

La ejecución de los programas es de forma secuencial, una sentencia a continuación de otra, a esto se lo conoce como el flujo normal de ejecución de un programa, pero este flujo se puede alterar.

Una de las primeras formas de manipular el flujo normal de ejecución del programa es a través de las estructuras condicionales o también denominadas de selección, ya que estas permiten seleccionar qué sentencias se ejecutaran y cuáles no. Esa decisión se toma luego de evaluar si una condición se cumple (es verdadera) o no (es falsa).

Antes de empezar con el estudio de las estructuras condicionales, es necesario comprender cómo se construyen condiciones, es por ello que el siguiente apartado se dedica a su estudio.

EXPRESIONES LÓGICAS

Una expresión lógica, también conocida como expresión booleana, es una expresión que al evaluarse produce como resultado un valor que puede ser verdadero (true) o falso (false).

Esto nos conduce a la presentación de un nuevo tipo de dato. En Java existe un tipo de dato denominado *boolean*. Las variables declaradas con ese tipo de dato únicamente pueden contener dos valores *true* (verdadero) o *false* (falso). El siguiente código muestra cómo se pueden crear variables de ese tipo de dato.

```
jshell> var esCorrecto = false
esCorrecto ==> false

jshell> boolean existeError;
existeError ==> false

jshell> boolean isOK = true
isOK ==> true
```

Las expresiones lógicas trabajan con dos tipos operadores, los relacionales y los lógicos. A continuación estudiará los primeros.

Los operadores relacionales son los operadores que nos permiten realizar comparaciones. En las siguientes líneas conocerá a estos operadores y sus símbolos.

$x == y$ // x es igual a y

$x != y$ // x es diferente de y

$x > y$ // x es mayor que y

$x < y$ // x es menor que y

$x >= y$ // x mayor o igual que y

$x <= y$ //x menor o igual que y

De seguro estos operadores no son extraños para usted, únicamente hay que considerar que los símbolos son diferentes a los que generalmente se usan en matemáticas (Ejemplo mayor o igual y menor o igual que se representan así: $\geq$, $\leq$). Un error bastante común es confundir el símbolo de asignación (=) con el de igualdad (==). Otro no muy común es usar símbolos que no existen como =< o =>.

Recuerde que el resultado de evaluar una expresión que posee una operador relacional es un valor lógico (*true* o *false*). Aquí un ejemplo: que compara los valores de las variables x e y.

```
jshell> var x = 10
x ==> 10

jshell> var y = 12
y ==> 12

jshell> x == y
$8 ==> false

jshell> x != y
$9 ==> true

jshell> x > y
$10 ==> false

jshell> x < y
$11 ==> true

jshell> x >= y
$12 ==> false

jshell> x <= y
$13 ==> true
```

Al devolver un resultado lógico, los operadores relacionales se pueden utilizar para asignar valores a variables de ese tipo de dato, así como se muestra en la siguiente porción de código:

```
jshell> var resA = x == y
resA ==> false

jshell> boolean resB = x > y
resB ==> false
```

Una observación más antes de continuar. Los operadores relacionales NO se deben utilizar con variables de tipo *String*, ya que ese tipo de dato posee otras mecanismos para realizar comparaciones entre cadenas de texto.

OPERADORES LÓGICOS

Tres son los operadores lógicos de uso generalizado: *and* (y), *or* (o) y *not* (no). Dentro del lenguaje de programación Java, estos operadores se representan con los símbolos *&&*, || y *!*. Tanto el and(&&) como el or (||) son operadores binarios, es decir que necesitan de dos operandos. No así el operador not(!) el cual necesita de un único operando ya que es unario. Los resultados de estos operadores son similares a sus significados en inglés.

Por ejemplo, $x > 0 \;\&\; \&\; x < 10$ es verdadera sí y sólo sí, x es mayor que 0 y menor que 10. En cambio la expresión $n \,\%\, 2 == 0 \,||\, n \,\%\, 3 == 0$ es verdadera si cualquiera de las dos expresiones es verdadera, es decir, si n es divisible por 2 o por 3. Finalmente, el operador not(!) niega la expresión booleana, así $!(x > y)$ es verdadera si $(x > y)$ es falso, es decir, si x es menor o igual qué y.

Un error bastante común es tratar de escribir una expresión lógica, en la que se incluyen al menos un operador lógico, tal y como lo pronunciamos. Así por ejemplo: qué x sea mayor que 10 y menor o igual que 20, erróneamente se escribe así: $x > 10 \;\&\; \&\; <= 20$. No olvide que los operadores *and* y *or* son binarios, es decir necesitan de dos operandos. La forma correcta sería $x > 10 \;\&\; \&\; x <= 20$.

En Java los operadores && y || evalúan la segunda expresión solamente cuando es necesario. Por ejemplo, con el operador *and* se evalúa la segunda expresión únicamente cuando la primera es verdadera, no así cuando es falsa, ya que si la primera es falsa, toda la expresión será falsa. Lo mismo hace el operador *or*, si al evaluar la primera expresión el resultado es verdadero, ya no evalúa la segunda, ya que toda la expresión será verdadera.

Con estos operadores es posible crear condiciones que son la base de casi todas las estructuras de control, como lo verá en el siguiente apartado.

Antes de continuar, es necesario mencionar que el código que se mostrará no incluirá el texto *jshell>*, ya que por razones de espacio ha sido eliminado, pero el código funciona en esta herramienta.

EJECUCIÓN CONDICIONADA

Para escribir programas útiles, casi siempre es necesario verificar condiciones y consecuentemente reaccionar. Las sentencias condicionales nos dan esa habilidad. La forma más simple es la sentencia *if*, que en Java se puede escribir así:

```
if(edad >= 18) {
   result = "Es mayor de edad";
}
```

De seguro ya reconoció que la expresión entre los paréntesis es una expresión lógica. A dicha expresión se la conoce con el nombre de condición.

Si se llega cumplir dicha condición, es decir, la expresión es verdadera, se ejecuta el bloque de sentencias que están dentro del *if*, (la sentencia que está entre llaves). De lo contrario, si es falsa, se salta la ejecución de ese bloque.

Generalizando la forma de un if es la siguiente:

```
if(<condición>) {
   <bloque de sentencias>;
}
```

Es necesario que las sentencias que forman el bloque se separen utilizando un punto y como (;), caso contrario se mostrará un error. Esta regla se aplica para todos los bloques de sentencias que encontrará de aquí en adelante..

No existe un número límite para las sentencias que pueden estar dentro del bloque del *if*, pero al menos debe ir una. En el caso de ser una única sentencia, se puede obviar el uso de las llaves, pero para fines didácticos se recomienda utilizarlas siempre.

Muchas veces es necesario contar con otro bloque de sentencias que se ejecuten cuando la condición no se cumple. Este tema se detalla a continuación.

EJECUCIÓN ALTERNATIVA

Una segunda sentencia condicional es aquella que tiene dos posibilidades, dos bloques de código, que se indican con *if* y *else*, la condición determina cuál de esos bloques se ejecutan.

La forma general de esta estructura de selección es la siguiente:

```
if( <condición> ) {
   <bloque de sentencias del if>
} else {
   <bloque de sentencias del else>
}
```

Un ejemplo se puede ver a continuación:

```
var nro = 134
if( nro % 2 == 0 ) {
   result = "Es par";
} else {
   result = "No es par";
}
```

En el código anterior si el residuo de *nro* dividido por 2 es igual a cero se ejecuta el bloque de código que pertenece al *if*, caso contrario se ejecutará el que pertenece al *else*. Así como se mencionó anteriormente cuando el bloque de código, tanto del *if* como del *else*, está formado por una única sentencia, el uso de las llaves es opcional. Sin embargo, es mejor utilizarlos para evitar confusiones, tales como el denominado else suelto.

El siguiente código muestra ese problema. ¿A cuál de los 2 *if* pertenece el *else*?

```
if (x > 5)
   if(y > 5)
      result = "x e y son mayores que 5”;
else
   result = “x es menor que 5”;
```

A pesar de la tabulación, se asociará el *else* con el último *if*. Este problema se puede evitar cuando se utilizan las llaves para marcar los bloques. El uso de las llaves ayuda a dejar clara la asociación *if/else*, es por ello que aquí siempre se empleará llaves en las estructuras de

control.

```
if (x > 5) {
   if(y > 5) {
      result = "x e y son mayores que 5";
   }
} else {
   result = "x es menor que 5";
}
```

CONDICIONALES ENCADENADAS

Algunas veces es necesario verificar un grupo de condiciones relacionadas y poder elegir una de varias acciones. Una forma de hacerlo es encadenando un conjunto de declaraciones *if/else*.

```
if ( x > 0 ) {
   result = "x es positivo";
} else if ( x < 0 ) {
   result = "x es negativo";
} else {
   result = "x es cero";
}
```

Observe detenidamente el código anterior y note cómo inmediatamente después del primer *else* se coloca un *if* y su condición. Esta combinación, también suele llamarse *if-else-if*.

Cadenas de condiciones como la anterior puede ser tan extensa como sea necesarias, aunque pueden ser difíciles de leer si son demasiadas.

Cada condición es verificada en el orden en el que aparecen. Si la primera es falsa, la siguiente será verificada y así sucesivamente. Si alguna es verdadera, esas acciones serán ejecutadas y se termina la ejecución de la estructura de control. Incluso si más de una condición es verdadera, solo se ejecuta el primer bloque de sentencias.

Si alguna de las condiciones no se cumple, se podría ejecutar el bloque de sentencias que pertenecen al *else*, aquel que no tiene una nueva condición.

En resumen, la sintaxis general de esta estructura es la siguiente:

```
if( <condición> ) {
   <bloque de sentencias del if>
} else if ( <condición> ){
   <bloque de sentencias del else if>
} else {
   <bloque de sentencias del else>
}
```

CONDICIONES ANIDADAS

Adicionalmente al encadenamiento de los estructuras de selección estas se pueden anidar, es decir, tanto en el bloque de sentencias del *if* como en las del *else*, se puede ubicar otra estructura condicional.

El siguiente código de ejemplo muestra dos estructuras condicionales anidadas.

```
if ( x == 0 ) {
   result = "x es cero";
} else {
   if ( x > 0 ) {
      result = "x es positivo";
   } else {
      result = "x es negativo”;
   }
}
```

En este ejemplo, dentro del *else* de la condición *x == 0* se ha colocado una nueva estructura de selección que a su vez tiene dos posibles acciones, que a su vez cada una podría contener una nueva estructura condicional.

El anidar estructuras de control es un práctica de común uso cuando está elaborando programas y así como se puede hacer con las estructuras condicionales se puede hacer con las estructura iterativas que estudiará más adelante.

En ambos casos, el uso de llaves y la indentación es necesario para poder leer correctamente el programa que está elaborando.

Dentro del lenguaje de programación Java existen otros estructuras condicionales como: el operador ternario y el switch. Estos se analizan brevemente a continuación.

OPERADOR TERNARIO

También conocido como *if* de una sola línea, se trata de una estructura condicional que se utiliza principalmente para asignar un valor a una variable. Así como lo muestra la siguiente porción de código.

```
result = edad >= 18 ? "Mayor de edad" : “Menor de edad”
```

El propósito de ese código es asignar un valor a la variable *result* dependiendo del valor de la variable *edad*. Si edad es mayor o igual que 18 se asignará el valor *“Mayor de edad”*, caso contrario se asignará *“Menor de edad”*.

Note como esta estructura, al igual que las anteriores tiene una condición, en el ejemplo *edad >= 18*. A diferencia de las estructuras anteriores, aquí no se ejecuta un bloque de sentencias cuando la condición se cumple y otras cuando no, sino que selecciona un valor u otro que se asignará a la variable *result*.

El valor que se asignará cuando la condición se cumple es ubica luego del signo de interrogación (?). Mientras que el valor cuando no se cumple se ubica luego de los dos puntos (:). Este operador siempre tendrá dos posibles valores y este comportamiento no se puede cambiar, es por ello que se se asemeja a un *if/else*.

La forma general de este operador es la siguiente:

```
variable = <condición> ? <valor si se cumple > : <valor sino se cumple>
```

SWITCH

Si el operador ternario se asemeja a un *if/else*, este operador se asemeja a una condición encadenada, con la diferencia que el *switch* únicamente evalúa igualdad. El siguiente código

muestra un ejemplo del switch que trata de determinar el nombre del mes según su número.

```
var month = 8;
String monthName;
switch ( month ) {
   case 1:  monthName = "Enero";
            break;
   case 2:  monthName = "Febrero";
            break;
   case 3:  monthName = "Marzo";
            break;
   case 4:  monthName = "Abril";
            break;
   case 5:  monthName = "Mayo";
            break;
   case 6:  monthName = "Junio";
            break;
   case 7:  monthName = "Julio";
            break;
   case 8:  monthName = "Agosto";
            break;
   case 9:  monthName = "Septiembre";
            break;
   case 10: monthName = "Octubre";
            break;
   case 11: monthName = "Noviembre";
            break;
   case 12: monthName = "Diciembre";
            break;
   default: monthName = "Mes no válido";
            break;
}
```

La sintaxis de esta estructura condicional es mucho más compleja que las anteriores ya que necesita de varios elementos como case, *break* y *default*.

Lo importante es entender su funcionamiento. El *switch* evaluará el valor de la variable *month* con cada uno de los valores que se encuentra en cada caso buscando el valor igual. Si este no se encuentra, se ejecutará el caso por defecto.

En cada caso se pueden ubicar una o mas sentencias (en el ejemplo asignar un valor a la variable *monthName*), pero, siempre la última de estas sentencias debe ser *break* para señalar

que se termina la ejecución del caso. Sin la sentencia *break*, se ingresará a varios casos, independientemente si su valor es igual o no, lo que es un error.

El caso *default*, se ejecutará únicamente cuando el valor a comparar, en el ejemplo *month*, no sea igual a ninguno de los casos anteriores. Si bien este caso es opcional es una buena práctica incluirlo siempre, ya que es una forma de conocer que el valor posiblemente no está en el rango permitido.

Para las últimas versiones de Java, esta sentencias está evolucionando para convertirse en un expresión, es decir asignar un valor, pero sin tantos elementos sintácticos como actualmente se hace, el siguiente apartado puede observar un ejemplo.

SWITCH COMO EXPRESIÓN

Una de los cambios que se introdujeron recientemente al lenguaje Java, es convertir a el switch de una sentencia condicional a una expresión, recuerde que una expresión devuelve un valor. Además, se ha simplificado su sintaxis. Veamos el mismo ejemplo de el nombre del mes, utilizando ahora una expresión.

```
var month = 8;
String monthName = switch( month ) {
    case 1 -> "Enero";
    case 2 -> "Febrero";
    case 3 -> "Marzo";
    case 4 -> "Abril";
    case 5 -> "Mayo";
    case 6 -> "Junio";
    case 7 -> "Julio";
    case 8 -> "Agosto";
    case 9 -> "Septiembre";
    case 10 -> "Octubre";
    case 11 -> "Noviembre";
    case 12 -> "Diciembre";
    default -> "Mes no válido";
}
```

Visualmente es mucho más compacta que su anterior versión, observe cómo se han eliminado las sentencias *break* y por cada caso se usa una flecha en lugar de los dos puntos.

Pero esto es únicamente sintaxis, lo más interesante es que el *switch* ahora devuelve un valor que puede ser asignado a una variable, en este caso *monthName*.

Es posible que cada caso tenga más de una sentencia y se necesite construir un bloque de sentencias, esto se puede hacer usando llaves. Como muestra el siguiente ejemplo:

```
var month = 8;
String monthName = switch( month ) {
    case 1 -> "Enero";
    case 2 -> "Febrero";
    case 3 -> "Marzo";
    case 4 -> "Abril";
    case 5 -> "Mayo";
    case 6 -> "Junio";
    case 7 -> "Julio";
    case 8 -> "Agosto";
    case 9 -> "Septiembre";
    case 10 -> "Octubre";
    case 11 -> "Noviembre";
    case 12 -> "Diciembre";
    default -> {
        System.out.printf("Error %d no es un mes válido\n", month);
        yield "Mes no válido";
    }
}
```

En este ejemplo se muestra como el caso por defecto está compuesto por más de una sentencia. La primera imprime un mensaje de error, mientras que la segunda señala el valor que devuelve el caso, es decir, que si el valor de *month* es por ejemplo 100, se imprimirá el mensaje “Error 100 no es un mes válido” y la variable *monthName* tendrá el valor de “Mes no válido”.

Esta es la estructura que se debe usar en cada uno de los casos que estén formados por más de una sentencia, encerrarlas entre llaves y señalar el valor que se asignará usando la palabra reservada *yield*.

Hasta aquí el estudio de las estructuras de selección, recuerde que estas estructuras permiten escoger cuál o cuáles sentencias se ejecutaran y para ello deben evaluar una condición. El siguiente tema a estudiar son las estructuras de iteración o repetición. Antes de pasar a un nuevo contenido, lo invito a revisar algunos recursos adicionales que de seguro le ayudarán.

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeo**: Práctica Guiada #3 - La selección (https://vimeo.com/410475762).

- **Laboratorio**: Selección (http://j4loxa.com/courses/java101/blog/2017/lab3.html).

ITERACIÓN

Muchas de las tareas que realizamos son repetitivas, es decir, que debemos aplicar el mismo proceso varias veces sobre un conjunto de datos diferentes. Por ejemplo, imagine que debe calcular la nota promedio de cada estudiante que pertenece a un paralelo de una asignatura. El proceso es idéntico por cada estudiante, lo único que cambian son las notas que cada uno de ellos obtuvo.

Ejecutar el mismo código múltiples veces se denomina iteración. Dentro de muchos lenguajes de programación existen varias estructuras de control iterativas que reciben el nombre de ciclos repetitivos o estructuras de repetición. Java no es la excepción, y posee varios ciclos repetitivos, tales como: *do…while*, *for* y *while*.

Estas estructuras comparten una característica con las de selección, también utilizan una condición que determina cuándo detener su ejecución. Además, se utilizan llaves para encerrar la(s) sentencia(s) que la forman, recuerde a este agrupamiento se lo denomina bloque.

A continuación encontrará el detalle de cada uno de ellos.

ESTRUCTURA DO…WHILE

Es una estructura de repetición en donde la condición se evalúa posterior a la ejecución del bloque de sentencias que lo forman. Es decir, que el bloque de sentencias se ejecutaran al menos una vez. Esta es una característica importante al momento de seleccionar uno de los ciclos repetitivos. La estructura básica de esta estructura es la siguiente:

```
do {
   <bloque de sentencias>
} while(<condición>);
```

Una estructura repetitiva ejecutará un bloque de sentencias mientras su condición sea verdadera deteniendo la repetición cuando deje de serla. Este tipo de estructura de repetición es útil cuando se necesita calcular dinámicamente su condición.

Analice el siguiente ejemplo: suponga que dentro de un programa necesita procesar ciertos datos, mientras que el valor sea menor o igual que 5.

```
import java.util.Random;
var random = new Random();
int value;

do {
   value = random.nextInt(10);
   System.out.printf("%d\n", value);
} while(value <= 5);
```

Note la posición de la condición, además fíjese que termina en un punto y coma (;). Por ahora no se preocupe de la generación aleatoria de números, lo importante es conocer el ciclo repetitivo. El ciclo repetitivo terminará cuando *value* sea mayor que 5, es decir que dicha variable es la que controla el ciclo repetitivo.

Un ejemplo adicional que se usará para todas las estructuras repetitivas es el siguiente: desarrolle un programa que sume los 3 primeros números naturales. Un solución se puede ver en el siguiente código:

```
var suma = 0;
var i = 1;
do {
   suma = suma + i;
   i = i + 1;
} while( i < 4 );
System.out.printf("suma= %d\n", suma);
```

Analizando el código, este se puede resumir así:

Sentencia	**suma**	**i**
Se declara y se asigna un valor inicial a la variable *suma*	0	
Se declara y asigna un valor inicial a la variable *i*		1
Se agrega *i* a *suma*, el nuevo valor se asigna a *suma*, *suma = suma + i => suma = 0 + 1 => suma = 1*	1	
Se incrementa, en una unidad, el valor de la variable *i => i++ => 1 + 1 = 2*		2
Se evalúa la condición: i < *4 => 2 < 4 = true. Se continúa la ejecución del ciclo repetitivo*		
Se agrega *i* a *suma*, el nuevo valor se asigna a *suma*, *suma = suma + i => suma = 1 + 2 => suma = 3*	3	
Se incrementa, en una unidad, el valor de la variable *i => i++ => 2 + 1 = 3*		3
Se evalúa la condición: *i < 4 => 3 < 4 = true. Se continúa la ejecución del ciclo repetitivo*		

Sentencia	suma	i
Se agrega *i* a *suma*, el nuevo valor se asigna a *suma*, *suma = suma + i => suma = 3 + 3 => suma = 6*	6	
Se incrementa, en una unidad, el valor de la variable *i => i++ => 3 + 1 = 4*		4
Se evalúa la condición: *i < 4 => 4 < 4 = false*. Se termina la ejecución del ciclo repetitivo		
Se presenta el valor de *suma*, es decir 6		

Esta misma tabla se utilizará en los ciclos restantes y debe utilizarla para ver la diferencia entre estas estructuras.

Un recurso para ampliar este tema, está disponible aquí: https://vimeo.com/412833210

ESTRUCTURA FOR

Esta estructura se podría interpretar así: para una determinada cantidad de veces ejecuta este bloque de sentencias. Un ciclo *for* tiene la siguiente sintaxis:

```
for( <inicialización> ; <condición> ; <incremento> ) {
   <bloque de sentencias>
}
```

La primera sección, inicialización, se declara y se le asigna un valor inicial a la variable que contralora el ciclo. Esta sección se ejecuta por una sola vez, al inicio del ciclo repetitivo.

La condición, determina cuándo finaliza el ciclo repetitivo, al igual que la estructura anterior, esta se ejecutará mientras la condición sea verdadera y dejará de hacerlo cuando el resultado de evaluarla sea falsa. En la condición debe estar la variable definida en la primera sección.

Por último el incremento, es decir, el cambio de valor a la variable que controla el ciclo. Este cambio puede incrementar o disminuir el valor de la variable de control. Lo importante es que el cambio de valor permita que en algún momento se termine la repetición. Esta sección se ejecuta una vez ejecutado el bloque de sentencias del ciclo.

Una recomendación importante es NO modificar el valor de la variable que controla el ciclo, dentro de su bloque de sentencias, esto se considera una mala práctica de programación, ya que genera confusión.

Analice el siguiente ejemplo: se necesita sumar los primeros tres números naturales, para ello se ha desarrollado el siguiente código:

```
var suma = 0;
for(var i = 1; i < 4; i ++) {
   suma = suma + i;
}
System.out.printf("suma= %d\n", suma);
```

Analizando el código, este se puede resumir así:

Sentencia	**suma**	**i**
Se declara y se asigna un valor inicial a la variable *suma*	0	
Se declara y asigna un valor inicial a la variable *i*		1
Se evalúa la condición: *i < 4 => 1 < 4 = true. Se continúa la ejecución del ciclo repetitivo*		
Se agrega *i* a *suma*, el nuevo valor se asigna a *suma*, *suma = suma + i => suma = 0 + 1 => suma = 1*	1	
Se incrementa, en una unidad, el valor de la variable *i => i++ => 1 + 1 = 2*		2
Se evalúa la condición: i *< 4 => 2 < 4 = true. Se continúa la ejecución del ciclo repetitivo*		
Se agrega *i* a *suma*, el nuevo valor se asigna a *suma*, *suma = suma + i => suma = 1 + 2 => suma = 3*	3	
Se incrementa, en una unidad, el valor de la variable *i => i++ => 2 + 1 = 3*		3
Se evalúa la condición: *i < 4 => 3 < 4 = true. Se continúa la ejecución del ciclo repetitivo*		
Se agrega *i* a *suma*, el nuevo valor se asigna a *suma*, *suma = suma + i => suma = 3 + 3 => suma = 6*	6	
Se incrementa, en una unidad, el valor de la variable *i => i++ => 3 + 1 = 4*		4
Se evalúa la condición: *i < 4 => 4 < 4 = false*. Se termina la ejecución del ciclo repetitivo		
Se presenta el valor de *suma*, es decir 6		

Note como a pesar de que el último valor de la variable *i* es 4, este nunca se suma, ya que la condición no se cumple y no se llega a ejecutar el bloque de sentencias.

Una característica final del ciclo *for*, al evaluar la condición antes de ejecutar el bloque de sentencias, determina que estas se ejecutan cero o más veces.

Para ampliar este tema, revise el recurso disponible aquí: https://vimeo.com/415385290

ESTRUCTURA WHILE

Esta última estructura básica de repetición se podría considerar una mezcla de las dos anteriores, ya que al igual que el ciclo *for*, la condición se evalúa al inicio, por lo tanto, este ciclo se ejecuta 0 o más veces. Se asemeja el ciclo *do … while*, ya que se puede usar cuando se necesita calcular dinámicamente su condición.

Su sintaxis se puede ver en la porción de código que se muestra más abajo.

```
while(<condición>) {
   <bloque de sentencias>
}
```

Al igual que los otros dos ciclos repetitivos, el bloque de sentencias se ejecutará mientras la condición sea verdadera.

Al implementar el ejemplo de la suma de los primeros tres números naturales el código sería:

```
var suma = 0;
var i = 1;
while( i < 4 ) {
   suma = suma + i;
   i = i + 1;
}
System.out.printf("suma= %d\n", suma);
```

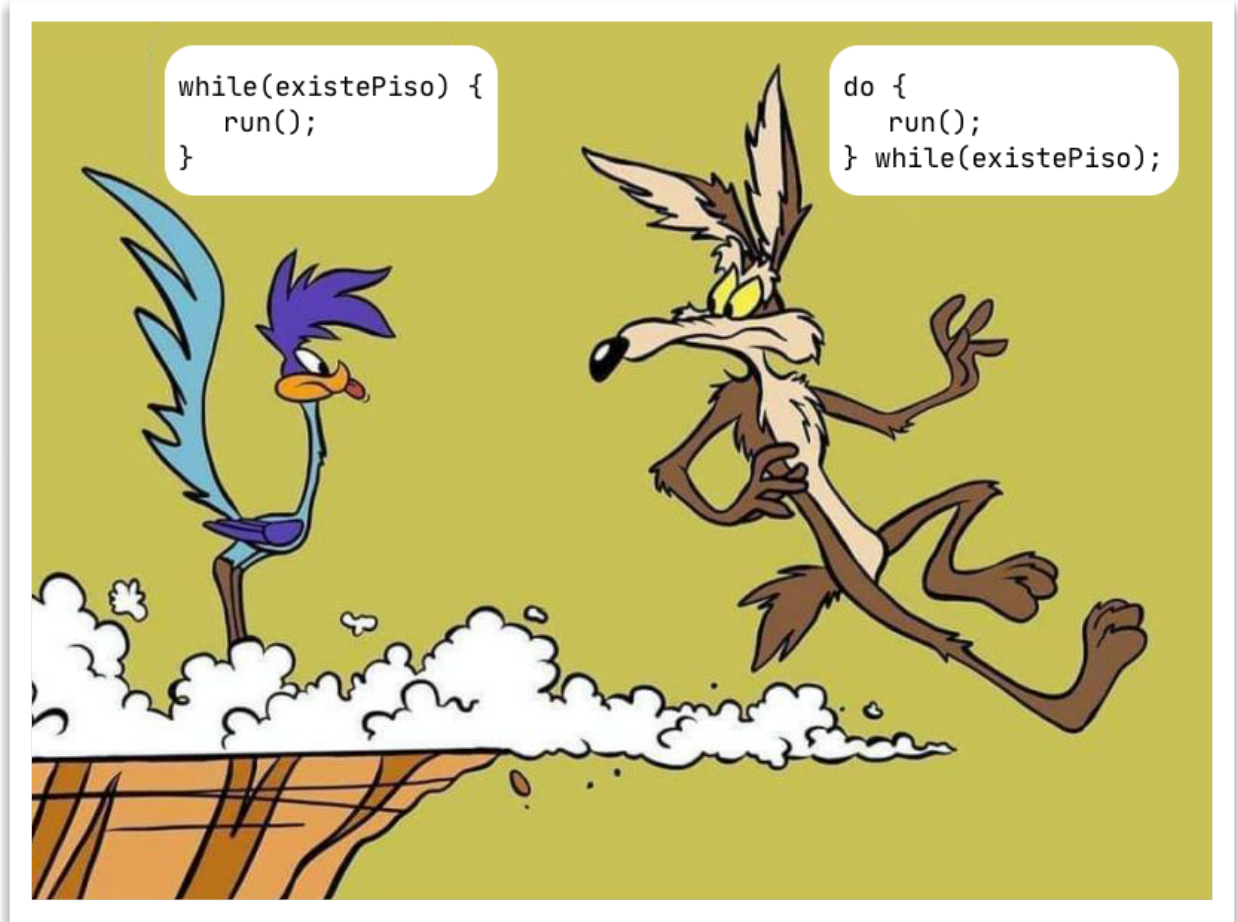
El análisis del código sería el mismo que se hizo para la estructura *for*.

Uno de los errores más comunes al utilizar estructuras repetitivas, son los denominados ciclos repetitivos infinitos, es decir, ciclos repetitivos en los que la condición nunca deje de ser verdadera. Este error no puede ser detectado por el compilador y se presenta en tiempo de ejecución, es decir cuándo se ejecuta el programa.

Los ciclos descritos hasta ahora comienzan inicializando una variable; tienen una condición, que depende de esa variable; y dentro del ciclo hacen algo con esa variable, como por ejemplo incrementarla. Más adelante conocer otro ciclo repetitivo que difiere de estos y que se utiliza para el trabajo con estructuras de datos.

Para ampliar este tema, revise el recurso disponible aquí: https://vimeo.com/417707392

Para finalizar una caricatura que muestra la diferencia entre un ciclo *do…while* y un *while*.



Adaptado de: https://devrant.com/rants/1939155/while-vs-do-while

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeos**:
  - Práctica Guiada #4 - La estructura de repetición do … while (https://vimeo.com/412833210).
  - Práctica Guiada #5 - La estructura de repetición FOR (https://vimeo.com/415385290).
  - Práctica Guiada #5 - La estructura de repetición WHILE (https://vimeo.com/417707392).
- **Laboratorios**:

- Ciclo do … while (http://j4loxa.com/courses/java101/blog/2017/lab4.html).
- Ciclo for (http://j4loxa.com/courses/java101/blog/2017/lab5.html).
- Ciclo while (http://j4loxa.com/courses/java101/blog/2017/lab6.html).

Adicionalmente y buscando crear programas en donde se utilicen todas las estructuras estudiadas al momento, recomiendo, revisar los siguientes recursos que le ayudarán a comprender cómo mezclar las diferentes estructuras de control para resolver un problema de complejidad media.

- Explicación teórica # 1. Series numéricas (https://vimeo.com/420537190)
- Práctica guiada #7. Series numéricas del grupo 1 (https://vimeo.com/421375375)
- Práctica guiada #8. Series numéricas del grupo 2 (https://vimeo.com/423358993)
- Práctica guiada #9. Series donde se mezclan elementos (https://vimeo.com/425354105)
- Laboratorio: Series numéricas (http://j4loxa.com/courses/java101/blog/2017/lab7.html)

ARREGLOS

Otra noción que comúnmente adquirimos es asociar varios elementos en grupos o conjuntos[2], esos elementos comparten una o más características comunes, además el grupo o conjunto posee también algunas características y una forma de trabajo. Esto es una forma de estructurar los datos. En este apartado se estudiará una estructura datos que se denomina arreglos.

DEFINICIÓN

Hasta ahora, ha estudiado variables que pueden almacenar valores individuales, tales como números o cadenas de texto, pero es posible almacenar varios valores de un mismo tipo en una única variable, para ello, debe utilizar arreglos.

Una arreglo es una colección de valores a los que se denomina elementos. Imagine una edificio de departamentos, cada departamento sería un elemento del arreglo edificio. Cada departamento tiene un número que permite identificarlo y tener acceso al mismo. Así como los edificios, un arreglo tiene un número finito de elementos, al número de elementos de un arreglo se lo denominada longitud o tamaño. Es posible resumir las características de los arreglos así:

- Son estructuras estáticas, es decir, no pueden crecer o decrecer sin perder sus elementos.
- El tamaño del arreglo es un número entero, no existen arreglos con tamaños que poseen partes decimales.
- Cada elemento se puede identificar a través de un índice que señala su posición dentro del arreglo. El índice de un arreglo es un número entero comprendido entre 0 (incluido) y la longitud o tamaño del arreglo (excluido). Acceder a un elemento fuera de ese rango, provoca errores.
- Todos los elementos que forman un arreglo deben ser del mismo tipo de dato. No es posible que, por ejemplo, exista un arreglo con números y texto mezclados, tampoco es posible que exista un arreglo que mezcle elementos que sean números enteros y reales.

---

[2] Un arreglo puede contener valores repetidos, mientras que un conjunto no.

Es momento de pasar al código y estudiar las formas que existen para crear arreglos, así como también se reflejan cada una de las características antes mencionas.

CREACIÓN DE ARREGLOS

Al igual que los valores individuales, en el lenguaje de programación Java, los arreglos se representan a través de variables, pero se utilizan los corchetes - `[]` - para diferenciar uno de otro. La forma de declarar un arreglo, sintácticamente, se parece a la declaración de una variable.

```
int []nums;
```

El código anterior señala que se ha declarado un arreglo de números enteros denominado *nums*. Aún no es posible usar el arreglo, ya que un arreglo necesita crearse previamente. Esto se debe a que en Java los arreglos son objetos (un tipo *especial* de dato que estudiará a futuro).

La creación de un arreglo se hace así:

```
int []nums;
nums = new int[10];
double []values = new double[30];
```

La declaración de un arreglo y su creación se pueden realizar en dos sentencias como sucede con el arreglo *nums* o en una sólo sentencia como es el caso de *values*. Los números que están entre corchetes en la creación de los arreglos representan al tamaño. Entonces, el arreglo *nums* podrá almacenar 10 números enteros, mientras que *values* 30 reales. Hasta ese punto cada posición de los arreglos tienen el valor de 0 para *nums* y 0.0 para *values*.

Así como con las variables un arreglo también puede tener un valor inicial como muestra la siguiente porción de código:

```
int []nros = { 3, 7, 2, 10, 1, 2 };
String []names = { "Ma. Vero", "Josué", "Sophía",
"Francisco", "Jorgito", "Ayelen", "Milca" };
```

El arreglo *nros*, es un arreglo de 6 números enteros, mientras que *names* es un arreglo de 7 cadenas de texto. Los valores iniciales se separan por comas. Observe cómo los valores iniciales guardan concordancia con el tipo de dato del arreglo.

Hasta aquí con la declaración, creación e inicialización de los arreglos, es momento de aprender a acceder a cada uno de los elementos del arreglo.

ACCEDIENDO A LOS ELEMENTOS

Como ya se mencionó anteriormente, cada elemento del arreglo se puede identificar a través de un índice que señala su posición. Entonces, para acceder a un elemento de un arreglo se necesita de la variable que representa al arreglo y el índice que señala la posición a la que se quiere acceder.

```
int []nros = new int[6];
nros[0] = 1;
nros[10] = 100;
```

En el código anterior, se declara el arreglo de entero denominado *nros* y en las siguientes sentencias se trata de acceder, para asignar un valor, a las posiciones 0 y 10 respectivamente. Observe cómo el índice se encierra entre corchetes. La primera sentencia asigna el valor 1 en la posición 0 del arreglo *nros*. La segunda, mostrará un error (*Exception java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 6*) ya que el índice está fuera del rango permitido, que en este caso es 0 (incluido) y 6 (excluido) o 0 y 5 incluidos ambos valores.

La combinación, identificador e índice del arreglo, es equivalente a una variable, cuyo valor es aquel que está en la posición del arreglo señalada por el índice. Esta combinación se puede usar en cualquier sentencia en donde se utiliza una variable, así como se puede ver en las siguientes sentencias:

```
int []nros = { 3, 7, 2, 10, 1, 2 };
nros[0] = 1;
nros[1] = nros[1] * 2;
System.out.printf("%d\n", nros[4])
if ( nros[3] > 10 ) {
   cont = cont + 1;
} else {
   cont2 = cont2 + 1;
}
```

Es poco práctico el tratar de acceder a los elementos de un arreglo usando tantas sentencias como elementos tiene, es por ello que generalmente se utiliza alguna estructura de iteración, aunque de forma casi generalizada se utiliza un ciclo *for*, así como muestra el siguiente código que utiliza el arreglo declarado anteriormente.

```
Scanner l = new Scanner(System.in);
for ( var i = 0; i < 6 ; i ++ ) {
   System.out.print("Ingrese un valor: ");
   nros[i] = l.nextInt()
}
```

Note también como el rango del índice del arreglo se refleja en la variable que controla el ciclo *for*, *i* toma el valor de 0 y se ejecutará hasta 5.

Como ya se mencionó un arreglo es un objeto y los objetos tienen propiedades, en el caso de los arreglos una de ellas es su longitud, es por ello, que se puede reescribir el ciclo for anterior, para consultar directamente al arreglo su longitud.

```
…
for ( var i = 0; i < nros.length ; i ++ ) {
   nros[i] = l.nextInt()
}
```

Prefiera está opción (consultar la longitud al propio arreglo) antes que usar un número, ya que si en algún momento cambia la longitud del arreglo, no deberá cambiar la condición de los ciclos repetitivos que trabajen con ese arreglo.

MOSTRAR ARREGLOS

Usted ya conoce que para mostrar variables, se utiliza sentencias *print* y sus variaciones *println* o *printf* y podría pensar en utilizar una de ellas para imprimir un arreglo, así como se muestra en la siguiente porción de código:

```
int []arr = {3, 9,12, 45, 31, 113};
System.out.println(arr);
```

La sentencia anterior produce la salida: *[I@533ddba*, pero, el resultado que se obtiene, no es lo que esperaba, imprimir todos los elementos. Hay que recordar que un arreglo es un objeto y lo que se imprime es una descripción del objeto en donde el *[* señala que es un arreglo, *I* que sus elementos son enteros (Integer) y el resto representa la dirección del arreglo en la memoria del computador.

Lo que se necesita es imprimir los elementos del arreglo, para ello existen varias alternativas que se muestran a continuación.

Es posible usar un ciclo for para imprimir los valores del arreglo, así:

```
int []arr = {3, 9,12, 45, 31, 113};
for ( var i = 0; i < arr.length; i ++ ) {
   System.out.printf("%d\n", arr[i]);
}
```

No olvide los límites del índice del arreglo, que serán los valores que tomará la variable *i*., tampoco que debe usar correctamente las marcas de formato según el tipo de dato de los elementos del arreglo.

Existe otra forma de recorrer arreglos utilizando una estructura de repetición que se denomina *for each* que es una simplificación al ciclo *for* normal. Un ejemplo:

```
String []nombres = { "Ma. Vero", "Josué", "Sophía",
"Francisco", "Jorgito", "Ayelen", "Milca" };
for ( var nombre : nombres ) {
   System.out.println(nombre);
}
```

En esta simplificación se puede leer como: por cada nombre que están dentro del arreglo nombres. Es una práctica común usar plural para nombrar arreglos y singular para representar a cada elemento. El tipo de dato de la variable *nombre* será del mismo tipo del arreglo(*nombres*), para el ejemplo, será *String*.

El *for each*, no utiliza un índice para recorrer las posiciones del arreglo, el recorrido lo hará automáticamente posición por posición. A esta estructura de control se la conoce como *azúcar sintáctico* ya que, en realidad, no es una nueva estructura, sino un simplificación del ciclo *for*, el compilador es quien convierte el *for each* en un *for*.

Otra forma de imprimir los valores del arreglo es utilizar la clase *Arrays* e invocar al método *toString*, como lo muestra la siguiente porción de código que utiliza el arreglo *nombres*, declarado anteriormente.

```
System.out.println(Arrays.toString(nombres));
```

La sentencia anterior produce la siguiente salida:

```
[Ma. Vero, Josué, Sophía, Francisco, Jorgito, Ayelen, Milca]
```

La salida que produce el método *toString*, no se puede modificar, así que, siempre los valores del arreglo estarán separados por comas y encerrados entre corchetes. Esto se podría considerar como una desventaja, ya que no permite personalizar la salida.

RECORRIDO DE ARREGLOS

Existen muchos cálculos que se pueden hacer con los elementos de un arreglo, todos se hacen utilizando una estructura de repetición y operando sobre cada uno de sus elementos, por ejemplo, contar los elementos de un arreglo que son números primos, el siguiente código implementa una solución a ese problema:

```
int []arr = {3, 9, 12, 45, 31, 113};
var cont = 0;
boolean esPrimo;
for ( var nro : arr ) {
   esPrimo = true;
   for ( var i = 2 ; i < nro ; i ++ ) {
      if ( nro % i == 0 ) {
         esPrimo = false;
      }
   }
   if ( esPrimo == true ) {
      cont = cont + 1;
   }
}
System.out.println(cont);
```

Este programa utiliza dos ciclos *for*, el primero recorre el arreglo y el segundo verifica si cada elemento del arreglo es o no un número primo, buscando otro divisor en el rango comprendido entre 2 y el número menos una unidad. Si no se encuentra otro divisor se incrementa el contador.

Así como el problema anterior existen otros muchos que se pueden aplicar a los arreglos, así como buscar un elemento, ordenamiento u otros conocidos como reducción, por ejemplo sería obtener la suma de todos los elementos de un arreglo. Estas operaciones están fuera del alance de este texto, pero, se invita al lector a buscar información sobre los mismos.

ARREGLOS BIDIMENSIONALES

También es posible crear arreglos de varias dimensiones, es decir es posible crear un arreglo cuyos elementos son otros arreglos.

Otra estructura bastante utilizada son los arreglos de dos dimensiones, también conocidos como matrices, ya que estos poseen filas y columnas. Una matriz se puede declarar así:

```
double[][] matrix = new double[5][7];
int[][] mat = {
   { 4, 7, 10, 34 -1 },
   { 9, 35, -8, 90, 120 },
   { 0, 5, 6, 12, 0 }
   { -5, -8, 3, -8, 2 }
};
```

La primera sentencia declara una matriz de números reales que tiene 5 filas y 7 columnas, es decir la segunda dimensión se define usando otro par de paréntesis. Entonces, el primer par de paréntesis representaran a las filas y el segundo par, a las columnas. La segunda sentencia crea e inicializa una matriz de números enteros que tiene 4 filas y 5 columnas, es decir, que el número de pares de llaves, de apertura y cierre, determinan las filas y el número de elementos, entre llaves, corresponde al número de columnas.

Dentro de las matrices, generalmente se habla de celdas, que es una combinación de tres elementos, el nombre de la variable que representa a la matriz, la fila y la columna. Esta triada, es similar a una variable, cuyo valor corresponde al que se encuentra en la intersección fila y columna.

```
double[][] matrix = new double[5][7];
matrix[0][0] = 2.5;
matrix[3][5] = 3.4;
matrix[6][1] = 7.8;
```

La segunda y tercera sentencia de la porción de código anterior, asigna un valor a matriz. Pero la cuarta sentencia lanza un error, ya que el rango de filas de la matriz no es correcto. El error que se produce es el siguiente: *Exception java.lang.ArrayIndexOutOfBoundsException: Index 6 out of bounds for length 5*.

OPERACIONES BÁSICAS EN ARREGLOS BIDIMENSIONALES

En este apartado se va a revisar dos operaciones básicas de los arreglos bidimensionales: el ingreso de datos y el recorrido. Así como con los arreglos existen otras operaciones que se pueden realizar, pero están fuera del alcance de este documento.

Para ingresar valores a un matriz, es necesario tener una bucle que recorra las filas y por cada fila otro que recorra sus columnas. Analice la siguiente porción de código:

```
import java.utilScanner;

Scanner l = new Scanner(System.in);
int[][] mat = new int[3][4];
for ( var i = 0; i < mat.length; i ++ ) {
   for ( var j = 0; j < mat[i].length; j ++ ) {
      System.out.print(“Ingrese un número”);
      mat[i][j] = l.nextInt();
   }
}
```

El primer *for*, es el encargado de recorrer cada una de las filas de la matriz, observe como se hace uso de la propiedad *length*, es decir, que *mat.length* devuelve el número de filas de la matriz. El segundo *for*, se usa para recorrer cada una de las columnas, note cómo se obtiene el número de columnas, *mat[i].length*, devuelve la cantidad de columnas que tiene la fila *i*. Finalmente *mat[i][j]* permite el acceso a la celda para que se asigne el valor que ingresa el usuario.

Para recorrer a una matriz, el procedimiento es similar al ingreso, por lo menos en las estructuras que se necesita. Dos ciclos *for*, uno para las filas y el segundo para las columnas. Luego se accede a la celda con la combinación del identificador de la matriz más los índices para las filas y las columnas. Aquí la porción de código que muestra cómo se puede recorrer una matriz.

```
int[][] mat = {
   { 4, 7, 10, 34 -1 },
   { 9, 35, -8, 90, 120 },
   { 0, 5, 6, 12, 0 }
   { -5, -8, 3, -8, 2 }
};
for ( var i = 0; i < mat.length; i ++ ) {
   for ( var j = 0; j < mat[i].length; j ++ ) {
      System.out.println(mat[i][j]);
   }
}
```

Es posible usar el *for each* para recorrer una matriz. Al igual que con el *for* tradicional se necesitan dos de estas estructuras, el primero que obtendrá una a una las filas y el segundo

para recorrer los elementos de una fila. Analice la siguiente porción de código, que hace lo mismo que la anterior, recorrer una matriz.

```
for ( var fila : mat ) {
   for ( var value : fila ) {
      System.out.println(value);
   }
}
```

No olvide que una matriz dentro del lenguaje Java es un arreglo de arreglos, entonces el primer *for each* devuelve cada fila, es decir, un arreglo y el segundo obtiene los valores de cada fila, es decir, recorre el arreglo que se obtuvo anteriormente.

Hasta aquí el tema de los arreglos. Luego aprenderá que existen otras estructuras de datos que tienen otros usos y mecanismos de trabajo, pero para empezar con la programación, arreglos y matrices son un buen punto de inicio.

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeo**: Práctica Guiada #10 - Arreglos (https://vimeo.com/431651126).
- **Laboratorios**:
  - Arreglos (http://j4loxa.com/courses/java101/blog/2017/lab8.html).
  - Matrices (http://j4loxa.com/courses/java101/blog/2017/lab9.html).

MÉTODOS

Siempre es posible dividir un problema en varias partes o sub-problemas y resolviendo cada una de esas partes se resuelve el problema inicial. A este principio se lo denomina divide y vencerás.

En programación, ese principio se aplica desde hace mucho tiempo atrás y aunque existen diferentes denominaciones, funciones, métodos, procedimientos, etc., todas pertenecen a lo que se denomina programación modular.

Una de las primeras formas, que posee Java, para construir módulos son los métodos. Es momento de estudiar este tema.

ESCRIBIENDO MÉTODOS

Hasta ahora ha utilizado métodos que pertenecen a la clase *Math*, pero, como cualquier otro lenguaje de programación, Java permite que los programadores escriban sus propios métodos.

El propósito de un método es ejecutar un grupo de sentencias y devolver un valor, aunque también existen otros métodos que ejecutan un grupo de sentencias, pero que retornen un valor.

La forma de un método es la siguiente:

```
<tipo_de_dato_retorno> <nombre> ( <parámetros> ) {
   <bloque de sentencias>
}
```

De dónde <*tipo_de_dato_retorno*> señala el tipo de dato del resultado que devuelve un método, puede ser cualquiera de los estudiados hasta el momento, incluidos arreglos de una o más dimensiones.

El nombre (<*nombre*>) de un método, según el estándar propuesto para Java, señala que debe ser un verbo, que se escribe por completo en minúsculas, en caso de estar compuesto por más de una palabra, la primera letra de cada palabra, a partir de la segunda, en mayúscula.

Los parámetros (<*parámetros*>) son los valores que necesita el método para hacer su trabajo. Tienen la forma de una variable y se separan por comas. Es posible que existan métodos sin parámetros, en ese caso se abre y cierra paréntesis.

Finalmente, el bloque de sentencias que ejecutará el método se encierran entre llaves.

Estos son los elementos que se necesitan para construir un método. Antes de continuar es necesario recordar que en este documento se utiliza *Jshell* como herramienta y la forma de los métodos responde a las características de esa herramienta. Existen otros elementos, tales como: modificadores de acceso, que no se toman en cuenta.

Los métodos se pueden clasificar en dos grupos, considerando si retornan o no resultados. En las siguientes secciones estudiará cada una de ellos.

MÉTODOS QUE NO DEVUELVEN RESULTADOS

Ha utilizado varios métodos que no devuelven resultados, por ejemplo los métodos *print* y sus variaciones *println* y *printf*. Si bien esos métodos ejecutan una serie de acciones, no devuelven resultado alguno. La utilidad de estos métodos se reduce a casi realizar tareas de impresión en pantalla.

Para definir este tipo de métodos es necesario cambiar un poco la forma de la declaración de métodos, así:

```
void <nombre> ( <parámetros> ) {
   <bloque de sentencias>
}
```

Tomando en cuenta que este tipo de métodos no devuelven resultados se reemplaza la sección del tipo de dato de retorno, por la palabra reservada *void*. Es necesario detallar que *void*, no es un tipo de dato, es una palabra reservada que indica que el método no devolverá ningún resultado.

El siguiente ejemplo, muestra un método que puede considerarse con una abreviatura del método *print*, ya que busca imprimir un mensaje en pantalla, sin la necesidad de otros elementos como *System.out*, únicamente necesita el nombre del método.

```
void print(String message) {
   System.out.print(message);
}
```

Observe que utiliza la palabra reservada *void*, señalando que no devuelve resultados, el método se llama *print* y recibe un parámetro denominado *message* que contiene el texto que desea mostrar. La única sentencia que posee, muestra en pantalla el mensaje utilizando el mecanismo ya conocido.

Para invocar a un método de esta categoría lo único que hace falta es agregar al sentencia que llama al método. Esto se podría hacer así:

```
print(“Java apuntes Básicos”);
```

Para invocar al método es necesario hacerlo usando su nombre y enviando los argumentos que necesita, que en este caso es un texto o también podría ser una variable de tipo *String*, o cualquier expresión que devuelva como resultado de su evaluación un valor de ese tipo de dato, incluyendo a otro método.

De seguro notó el cambio de argumento por parámetro, aunque generalmente se usan como sinónimos, el término parámetro se usa cuando se está construyendo el método, mientras que argumento, cuando se lo invoca. Sutiles diferencias que son necesarias conocer.

Dentro de las sentencias que forman el cuerpo del método, es posible que se invoque a otros métodos. Por ejemplo, se puede usar algún método de la clase *Math* para realizar algún cálculo, inclusive se puede invocar a métodos creados por usted mismo. El siguiente código muestra un ejemplo así:

```
void imprimirContorno() {
   System.out.println("****************************************");
   System.out.println("**                                    **");
   print("**        Java apuntes básicos        **\n");
   System.out.println("**                                    **");
   System.out.println("****************************************");
}
```

El método *imprimirContorno*, es un método que no recibe parámetros, entre las sentencias que forman su bloque se llama al método *print* que se escribió anteriormente, de esta forma se combinan dos métodos para completar una tarea.

MÉTODOS QUE DEVUELVEN RESULTADOS

Si bien los métodos anteriores son útiles, lo son mucho más aquellos que devuelven resultados. En este apartado se estudiará el detalle de estos métodos.

La forma de los métodos es la siguiente:

```
<tipo_dato_de_retorno> <nombre> ( <parámetros> ) {
   <bloque de sentencias>
   return resultado;
}
```

Para este tipo de métodos, no se puede usar la palabra reservada *void*, ya que estos devuelven un resultado que pertenece a un tipo de dato.

En estos métodos se debe utilizar la palabra *return* para formar una sentencia que devuelve un resultado. El resultado está representando por una variable o cualquier expresión que al evaluarla devuelve un valor cuyo tipo de dato sea el mismo que se utilizó en la declaración del método, es decir, corresponda con el tipo de dato que devuelve el método, caso contrario, se producirá un error.

Dentro de está categoría, un método puede tener varias sentencias que devuelven un valor, pero solo una se ejecutará según la lógica del programa, es decir, se puede usar varias veces la palabra *return*, pero sólo una de ellas se ejecutará. Es más el compilador de Java, es capaz de determinar si hace falta una sentencia *return*.

Analice el siguiente método en el que se puede ver un método que devuelve un valor. Observe cómo el método devuelve un valor lógico (*boolean*), además el método utiliza dos sentencias *return*, de las cuales sólo una se ejecutará. De esta forma se puede tener un método con varias sentencias *return*. Aunque el ejemplo es bastante sencillo, inclusive pude considerarse como trivial, sin embargo, es un método didáctico, ya que permite comprender la estructura y funcionamiento de este tipo de métodos.

```
boolean esPar( int nro ) {
   if ( nro % 2 == 0 ) {
      return true;
   } else {
      return false;
   }
}
```

Al igual que los métodos que no devuelven valor, para invocar a un método de esta categoría se debe utilizar su nombre y proporcionar los argumentos que necesita. Para el método anterior sería:

```
esPar( 1451 );
```

Al devolver un valor, un método de esta categoría se puede usara para asignar un valor a una variable, es decir, el valor que devuelve el método se asignará a una variable. Lo anterior implica que primero se ejecuta el método y luego se asigna su valor a una variable. Un ejemplo de esto se muestra a continuación:

```
var result = esPar( 1451 );
```

Una vez que se ejecute el código anterior, la variable *result*, tendrá el valor que devuelve el método *esPar*, así se puede utilizar el valor devuelto por ese método. Esto únicamente se puede hacer con métodos de esta categoría.

Antes de continuar, propongo el siguiente ejercicio: construya un método que ayude a determinar si un número es primo o no. Algunas características del método, recibe un argumento que será el número a analizar y devuelve *true* si el parámetro es primo, caso contrario devuelve *false*. Finalmente, el método se llamará *esPrimo*. Una implementación del método es la que se muestra a continuación.

```
boolean esPrimo( int nro ) {
   for ( var i = 2 ; i < nro / 2 ; i ++ ) {
      if ( nro % i == 0 ) {
         return false;
      }
   }
   return true;
}
```

A continuación se muestra un un ejemplo del uso del método anterior (*esPrimo*), observe cómo se asigna el valor que devuelve el método a la variable *resp*, para luego utilizarlo para construir la condición de una estructura de selección.

```
var value = 1521
var resp = esPrimo( value )

if ( resp == true ) {
   System.out.printf( "El %d ES primo\n", value );
} else {
   System.out.printf( "El %d NO es primo\n", value );
}
```

Con los ejemplos que se han desarrollado, el resultado de la invocación de un método se asigna a una variable, pero es posible realizar otras acciones. En el siguiente apartado estudiará algunas de las acciones que se puede hacer cuando se invoca a un método.

USO DE MÉTODOS, MÁS ALLÁ DE LA ASIGNACIÓN

Hasta ahora, los métodos que devuelven valor se han utilizado para asignar valores a variables, pero el uso de los métodos va mas allá de la asignación. En esta sección estudiará otras formas de uso de los métodos. Recuerde que esto es exclusivo para los métodos que devuelven un valor.

Una vez que se define y construye un método que devuelve un valor, ese método se puede utilizar para construir expresiones e inclusive construir nuevos métodos. Para comprender esto, es necesario que cada vez que vea la invocación a un método piense que a futuro, será

reemplazada por un valor, cuyo tipo de dato corresponde al tipo de dato que devuelve el método.

Un ejemplo, con el método *esPrimo* que se construyó anteriormente, ese método se puede emplear directamente para construir una expresión que devuelve un valor *booelan,* ya que se trata de un método que devuelve un valor de ese tipo de dato. En código se muestra así:

```
var value = 1521

if ( esPrimo( value ) ) {
   System.out.printf( "El %d ES primo\n", value );
} else {
   System.out.printf( "El %d NO es primo\n", value );
}
```

Note como la invocación del método se hace en la condición de una estructura de selección, esto conlleva a que primero se ejecuté el método y su resultado sea parte de la condición de la estructura de selección.

De la misma manera que se utilizó en una estructura de selección es posible que se utilice en otras estructuras, como las de repetición. El punto clave es ver a la invocación del método como un futuro valor.

Es posible también utilizar un método para construir otro método, para comprender este tema propongo el siguiente ejercicio: desarrolle un método que permita calcular la distancia entre dos puntos dados por las coordinas (x_1, y_1) y (x_2, y_2), para ello se debe usar la fórmula:

$$distancia = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

El método que implementa la fórmula anterior es el siguiente:

```
double calcDistance( double x1, double y1, double x2, double y2 ) {
   var dX = x2 - x1;
   var dY = y2 - y1;
   var dCuadrado = dX * dX + dY * dY;
   var resultado = Math.sqrt(dCuadrado);

   return resultado;
}
```

Para complementar el problema anterior, ahora propongo otro, suponga que tiene dos puntos, el primero será el centro de un círculo y el segundo un punto que pertenece a su perímetro. Con esa información se pide calcular el área del círculo.

Para calcular el área de un círculo se debe aplicar la fórmula

$area = \Pi * r^2$

De donde *r* corresponde al radio del círculo. Un método que calcule el área de un círculo dado su radio sería el siguiente:

```
double calcArea ( double r ) {
   var area = Math.PI * r * r;

   return area;
}
```

El problema ahora se reduce en encontrar el radio de un círculo. Para resolver el problema se ha construido el siguiente método:

```
double calcArea2 ( double xc, double yc, double xp, double yp ) {
   var r = calcDistance( xc, yc, xp, yp );
   var area = calcArea( r );

   return area;
}
```

El código anterior señala que para encontrar el radio del círculo se calcula la distancia que existe entre el punto central (dado por *xc* e *yc*) y el punto que pertenece al perímetro (dado por las coordenadas *xp* e *yp*). Con ese valor se invoca al método *calcArea*, para que devuelva el valor del área según el radio del círculo.

Como muestra el código anterior es posible invocar a uno o más métodos dentro de otro y utilizar los valores que devuelve en otras sentencias que forman el cuerpo del método. Algunos autores denominan a esto como una composición de métodos, ya que un método está compuesto por la invocación de uno o más métodos.

Otra forma de escribir el método anterior sería la siguiente:

```
double calcArea2 ( double xc, double yc, double xp, double yp ) {

   return calcArea( calcDistance( xc, yc, xp, yp ) );
}
```

En este caso, primero se calcula la distancia entre los dos puntos, para luego invocar al método que calcula al área. A primera vista puede parecer complejo, pero, lo que básicamente se hizo fue reemplazar las variables por las invocaciones de los métodos que se usaron para asignar valores.

SOBRECARGA DE MÉTODOS

Con los ejemplos anteriores que calculan el área de un círculo se pudo haber percatado que los métodos *calcArea* y *calcArea2* hacen lo mismo, lo único que cambia son la cantidad de parámetros que utilizan, 1 y 4 respectivamente.

Si dos métodos hacen lo mismo, deberían tener el mismo nombre, en nuestro caso ambos métodos deberían llevar el nombre *calcArea*. En Java cuando varios métodos llevan el mismo nombre se dice que se hace una sobrecarga de los métodos, pero, ¿cómo diferenciarlos? Para que sean válidos deben tomar diferentes parámetros y no se refiere a los nombres de los parámetros, sino a sus tipos de dato.

El método sobrecargado quedaría así:

```
double calcArea( double xc, double yc, double xp, double yp ) {
   var area = calcArea( calcDistance( xc, yc, xp, yp ) );

    return area;
}
```

Observe como dentro del método *calcArea* (con 4 argumentos), se invoca al método *calcArea* que usa un único argumento[3] que se obtiene luego de ejecutar al método *calcDistnace*.

---

[3] Esta característica no se puede usar en *Jshell*, al parecer aún no es soportada en *OpenJDK* 14.

La sobrecarga de métodos es un mecanismo que busca la claridad del código y evitar la duplicación de código que es una fuente de problemas, especialmente, cuando se realizan cambios en el código. Aunque, también puede llegar a causar confusión.

No olvide que la sobrecarga se puede obtener únicamente variando el número de parámetros que recibe un método, así como también el tipo de dato de los mismos. Sino se cambia uno de estos dos elementos, se producirá un error.

RECURSIVIDAD

En el apartado anterior se mencionó que un método puede estar compuesto por otros métodos, pero ¿es posible que un método se invoque a sí mismo? Es decir que una de las líneas que forman el bloque de código de un método se encuentre una invocación a sí mismo, como se muestra en el siguiente ejemplo:

```
void printAndSubstract( int value ) {

   System.out.println( value );

   printAndSubstract( value - 1 );
}
```

Como pudo ver, el método tiene únicamente dos sentencias, la primera imprime el valor del parámetro y en la segunda se invoca a sí mismo. A esto se le denomina recursividad.

Si usted trata de ejecutar el código anterior, se producirá un conjunto de llamadas que van hasta el infinito y esto es algo que deberíamos evitar. Para ello es necesario estudiar más acerca de las invocaciones recursivas y cómo manejarlas.

Para crear un método recursivo es necesario saber que, el método tendrá dos partes, la primera será el caso base y la segunda el caso recursivo. El caso base pone fin a la ejecución recursiva y devuelve un valor en caso de método que devuelven valores. Mientras que el caso recursivo, como de seguro ya se imagina, invoca nuevamente al método, pero cambia el valor de los parámetros.

Tomando el párrafo anterior, el método *printAndSubstract* únicamente tiene el caso recursivo, le hace falta el caso base. Para corregir ese método, ahora suponga que imprimirá y restará mientras el parámetro sea mayor o igual a cero. La implementación es la siguiente:

```
void printAndSubstract( int value ) {

   if( value >= 0 ) {
      System.out.println( value );
      printAndSubstract( value - 1 );
   }
}
```

Hasta aquí la recursividad se muestra como una compleja imitación de los ciclos repetitivos y es una verdad a medias. Por un lado, si tiene algo de iteración (repetición), pero por otra lado su utilidad no está en reemplazar a los ciclos repetitivos, sino en permitir escribir programas de una forma sencilla.

Un ejemplo más. Dentro de las matemáticas la función factorial se define así:

$0! = 1$

$n! = n * (n - 1)!$

La definición señala que el factorial de 0 es 1, mientras que para cualquier otro número n es n multiplicado por el factorial de $n - 1$, es decir se utiliza recursividad para definir a la función factorial. Así por ejemplo el factorial de 4 es:

$4! = 4 * (4 - 1)! = 4 * 3!$

$3! = 3 * (3 - 1)! = 3 * 2!$

$2! = 2 * (2 - 1)! = 2 * 1!$

$1! = 1 * (1 - 1)! = 1 * 0!$

$0! = 1$

Para obtener el resultado se debe reemplazar *n!*, desde la última línea hasta la primera. La implementación en código demanda la identificación del caso base y recursivo, que afortunadamente están en su definición.

Caso base → $0! = 1$

Caso recursivo → $n! = n * (n - 1)!$

El código quedaría así:

```
int factorial( int n ) {
   if( n == 0 ) {
      return 1;
   } else {
      return n * factorial( n - 1 );
   }
}
```

La recursividad es una técnica de programación que se utiliza principalmente para recorrer estructuras de datos complejas como los árboles, debido a que las soluciones recursivas son mucho más fáciles de comprender, aunque como desventajas están que son más complejas de implementar y algunas veces más lentas que su contra parte iterativa.

Resumiendo, para implementar recursividad es necesario identificar el caso base y el caso recursivo, que generalmente se representan como una estructura de selección. En el caso recursivo se debe incluir una sentencia que sea la invocación al mismo método, eso sí, el valor de los parámetros que se envían deben cambiar. El caso base, pone fin a la recursividad devolviendo un valor. En ambos casos, se debe usar la palabra *return*, ya que en ambos casos se devuelve o devolverá un valor.

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeos**:
  - Práctica Guiada #11 - Métodos (https://vimeo.com/455552462).
  - Práctica Guiada #12 - Métodos recursivos (https://vimeo.com/455554419).

- **Laboratorios**:
  - Métodos (http://j4loxa.com/courses/java101/blog/2017/lab10.html).
  - Recursividad(http://j4loxa.com/courses/java101/blog/2017/lab11.html).

CADENAS DE TEXTO

Hasta el momento se ha mencionado poco del tipo de dato *String*, únicamente se ha dicho que se utilizará para representar al texto, pero es necesario mostrar un poco más de lo que representa este tipo de dato y cómo se puede trabajar con él.

OBJETO

Dentro de Java a los tipos de datos: *int*, *double*, *boolean* y *char* se los denomina como tipos de datos primitivos, mientras que al tipo de dato String, como un tipo de referencia. Es necesario comentar que por cada tipo de dato primitivo le corresponde un tipo de referencia, así: *Integer*, *Double*, *Boolean*, *Character*. Note el uso de la primera letra mayúscula para los tipos de referencia, ya que eso es parte del estándar de programación de Java - los nombres de las clases inician con mayúscula.

La distinción anterior, se basa principalmente en que a diferencia de los otros tipos de dato que ha estudiado, una variable de tipo *String* es un objeto de esa clase, así como lo son los arreglos, o los objetos *Scanner* para el ingreso de datos. Pero, es un objeto un tanto particular ya que recibe cierta ayuda del compilador de Java para que luzca mas como cualquier variable (*int*, *double*, *boolean* o *char*) que como un arreglo o *Scanner*. Si bien esto ayuda a que el trabajo con las variables *String* sea más sencillo, también causa cierta confusión.

En los siguientes párrafos estudiará algunas de las “ayudas” que recibe este tipo de dato.

A pesar de ser un objeto, no necesita del operador *new* en su creación ya que los valores literales, es decir, valores encerrados entre comillas dobles, son convertidos en objetos del tipo *String* por el compilador, aunque aún se puede usar el operador new para crear objetos de ese tipo.

```
var titulo = "Java Apuntes Básicos”
String continente = new String( "América del sur" )
char []ciudad = { 'L', 'o', 'j', 'a' }
String ciudadS = new String(ciudad)
```

Las variables *titulo* , *continente* y *ciudadS* son objetos del tipo dato *String*. La primera no utiliza el operador *new* y será el compilador quien se encargará de ello, mientras que las otras sí utilizan ese operador. En la declaración de la segunda variable, el texto que contendrá se

envía como un parámetro, mientras que en la tercera, se envía como parámetro un arreglo de caracteres para que sean representados como texto.

Otra de las ayudas que recibe del compilador tiene que ver con la sobrecarga de operadores (que un mismo operador realice acciones diferentes), concretamente de los operadores + y +=, que para variables de tipo *String* son concatenar y concatenar y asignar. El siguiente ejemplo le ayudará a comprender mejor esto:

```
var pais = "Ecuador"
var provincia = "Loja"
System.out.println( provincia + ", " + pais )
```

La porción de código anterior producirá la salida: Loja, Ecuador, ya que el signo + fue sobrecargado para que cuando sus operandos sean cadenas de texto las concatene, que consiste en agregar al final del primero el texto del segundo.

El siguiente operador sobrecargado es el += que sirve para concatenar y asignar el resultado a otra variable. El mismo ejemplo similar al anterior, pero el resultado es asignado a una de las variables.

```
var pais = " Ecuador"
var provincia = "Loja, "
provincia += pais
```

El nuevo valor de la variable *provincia* es Loja, Ecuador, note también cómo se modifico au valor para agregar los caracteres coma y espacio.

Esta sobrecarga de operadores genera muchas confusiones, especialmente al momento de utilizar los operadores relaciones, ya que los estudiados hasta ahora funcionan con tipos de datos primitivos, no así con objetos o tipos de referencia.

Otra de las características de un objeto, es que los objetos tienen métodos asociados que permiten realizar operaciones sobre ellos. Para invocar a los métodos, debe utilizar el operador punto (.), de hecho muchos entornos de desarrollo integrados (IDE) al presionar el punto, luego del nombre de un objeto, muestran los métodos asociados a ese objeto. Más adelante aprenderá a utilizar algunos métodos de la clase *String*.

En la sección de métodos útiles conocerá cuáles son y cómo trabajar con los operadores relacionales, así como también, conocer otras acciones propias del tipo de dato *String*. Mientras tanto, continue estudiando otra de las características de este tipo de dato.

SECUENCIA DE CARACTERES

Una variable String es una cadena de caracteres y por tanto es posible acceder a cada uno de los caracteres que la forman. Es posible pensar que se trata de un arreglo de caracteres como sucede en otros lenguajes, pero no es así, en Java un cadena de caracteres es un objeto, como ya se lo dijo anteriormente.

Un *String*, representa a un cadena de caracteres. Internamente cada carácter se representa con 16 bits siguiendo el estándar *Unicode*, esto abre la posibilidad de contar con 65535 posibles combinaciones, cada una representando a un carácter, representación que es superior a la *ASCII* que únicamente permitía 255 caracteres. Para mayor información consulte aquí: http://www.unicode.org.

Considerando lo anterior es posible escribir directamente caracteres utilizando su código *Unicode*. Un ejemplo es el siguiente:

```
var msg = "\u004c\u006f\u006a\u0061\u0020\u2661"
System.out.println(msg)
```

El código anterior señala que la cadena de texto está compuesta por 6 caracteres expresados en su formato *Unicode*, ya que empiezan con la secuencia de escape *\u*. Al imprimir el mensaje, no se muestra el código *Unicode*, sino que se mostrará el carácter que representan, así el mensaje que se muestra es: Loja ♡. Para conocer el código de cada letra puede usar el siguiente buscador: https://www.compart.com/en/unicode/

Es posible acceder a cada uno de los caracteres que forman un *String*, para ello es necesario conocer que cada carácter ocupa una posición, la primera posición será la 0 y la última la longitud o número de caracteres que tiene la cadena de texto menos una unidad, similar a lo que sucede con los arreglos.

Un ejemplo, el siguiente código permite acceder a cada uno de los caracteres que forman una cadena de texto:

```
var mensaje = "Loja, capital musical del Ecuador."

for( var i = 0; i < mensaje.length(); i ++ ) {
   char caracter = mensaje.charAt( i );
   System.out.println( caracter );
}
```

Lo más interesante del código anterior, ocurre en el ciclo *for*. Observe la condición del ciclo, ahí se utiliza el método *length* de la clase *String*, que devuelve su longitud, es decir el número de caracteres que la forman. Dentro del cuerpo de ese ciclo repetitivo, se asigna valor a la variable *caracter*, empleando otro método, en este caso *charAt* que devuelve el carácter que se encuentra en la posición *i*.

Considerando el código anterior, cuando se trabaja con objetos, es necesario conocer sus métodos, no todos, pero sí algunos, posiblemente los más utilizados, pero esto lo encontrará en un apartado más adelante. En el siguiente apartado aprenderá la forma correcta de comparar cadenas de texto.

COMPARANDO CADENAS DE TEXTO

Para comparar cadenas de texto, se debe utilizar algunos métodos de la clase *String*. Los métodos que se deben utilizar permitirán saber si dos cadenas son iguales o, utilizando el orden alfabético, saber si una cadena precede o antecede a otra, para este fin debe utilizar los métodos *equals* y *compareTo*, respectivamente.

Empecemos con la igualdad, el siguiente ejemplo muestra el uso del método *equals*.

```
var cadena1 = "Orillas del Zamora tan bellas"
var cadena2 = "Orillas del Zamora tan bellas”

cadena1.equals( cadena2 )
```

El código anterior compara si los valores de las variables *cadena1* y *cadena2* son iguales, para ello invoca al método *equals* del objeto *cadena1* y le envía como parámetro el objeto *cadena2*. El método *equals* devuelve *true* si ambas cadenas son exactamente iguales (incluyendo mayúsculas) o *false* si no lo son, en nuestro ejemplo devolverá true. También se podría cambiar el orden así: *cadena2.equals(cadena1)* y el resultado será igual.

Existe una variante del método equals. El método *equalsIgnoreCase* hace lo mismo que *equals*, pero sin considerar si estas son mayúsculas, es decir considera que todas las letras son minúsculas. El siguiente código es un ejemplo:

```
var cadena1 = "Orillas del Zamora tan bellas"
var cadena3 = "orillas del zamora tan bellas”

cadena1.equals( cadena3 )
cadena1.equalsIgnoreCase( cadena3 )
```

La variable *cadena3*, se diferencia de *cadena1* debido a todas las palabras están escritas en minúsculas. Si ambas cadenas se comparan, usando el método *equals*, el resultado será *false*. Mientras que si se utiliza el método *equalsIgnoreCase* el resultado será *true*, debido a que no toma en cuenta la diferencia que surge por el uso de mayúsculas y minúsculas.

Si la tarea es ordenar alfabéticamente dos cadenas de texto, es necesario usar los métodos *compareTo* y *compareToIgnoreCase*. El siguiente ejemplo muestra su uso:

```
var apellido1 = "Cuenca"
var apellido2 = "Correa"

apellido1.compareTo( apellido2 )
```

El resultado de ese método, es un entero que puede estar en tres rangos negativo, cero y positivo que se resume en la siguiente tabla

Posible valor	Interpretación	Ejemplo
Negativo	Si el objeto se ubica antes que el argumento.	Si apellido1 se ubica antes que apellido2
Cero	Si el objeto es igual al argumento.	Si apellido1 es igual a apellido2
Positivo	Si el objeto se ubica después que el argumento.	Si apellido1 se ubica después que apellido2

Para el ejemplo concreto el resultado es un entero positivo señalando que *apellido1* va después de *apellido2* y es así, ya que el apellido Cuenca iría luego de Correa.

El uso de *compareToIgnoreCase* es similar, aunque no considera el uso de las mayúsculas, considera que todas las letras están en minúsculas, para comprender este tema analice el siguiente ejemplo:

```
var apellido1 = "Ayala"
var apellido2 = "AYora"

apellido1.compareTo( apellido2 )
apellido1.compareToCaseIgnoreCase( apellido2 )
```

En la primera comparación, con *compareTo*, el resultado es un número positivo, indicando que Ayala se ubica después de AYora lo que no es correcto. En la segunda comparación, usando *compareToCaseIgnoreCase*, el resultado es un número negativo, que señala que Ayala va antes que AYora, lo cual es correcto.

En la siguiente sección conocerá algunos otros métodos que serán útiles para trabajar con cadenas de texto.

MÉTODOS ÚTILES

Son muchos los métodos que están asociados a la clase *String*, para consultarlos puede ir a la documentación de la plataforma Java en su versión estándar, que se encuentra disponible en: https://docs.oracle.com/en/java/javase/, seleccione el link que está bajo *Latest Realease*, luego *API Documentation*, en la tabla que se muestra seleccionar *java.base* en la columna *Module*, ahora es momento de seleccionar el link *java.lang* que está en la columna *Package*, finalmente buscar la clase *String* en la tabla *Class Summary*, en esa página encontrará una tabla con el título *Method Summary*, ahí están todos los métodos de la clase.

Una característica de los métodos de esta clase es que, sus nombres, son sensibles al uso de mayúsculas y minúsculas, es decir, se consideran diferentes las letras mayúsculas de las minúsculas, generalmente a esto se lo conoce como case sensitive. Hay que tomar en cuenta esta característica ya que puede traer más de un problema si se ignora.

Aquí se mostrarán únicamente un grupo reducido de esos métodos, aquellos que, según el criterio del autor, son posiblemente los más utilizados. En una sección anterior ya se mencionaron 4 métodos de la clase String, para realizar comparaciones, pero existen muchos más.

Concatenación (*concat*):

```
"Catacocha la ".concat("tierra de los Paltas”)
//Catacocha la tierra de los Paltas
```

Este método une ambas cadenas de texto y devuelve una nueva cadena, el resultado se muestra como un comentario. El mismo resultado se puede obtener con el operador sobrecargado “+”.

Contiene (*contains*):

```
"Catacocha la tierra de los Paltas”.contains("tierra")
//true

"Catacocha la tierra de los Paltas”.contains("Tierra")
//false
```

Este método devuelve *true*, si su parámetro está dentro del objeto que lo invoca, caso contrario devuelve *false*. Este método es sensible al uso de mayúsculas y minúsculas, es por ello que en la primera línea devuelve *true*, mientras que en la segunda devuelve *false*.

Finaliza con (*endsWith*):

```
"Catacocha la tierra de los Paltas”.endsWith("tas")
//true
```

Verifica si la cadena de texto termina con el parámetro del método, devuelve verdadero sí es así, caso contrario devuelve falso.

Formato (*format*):

```
String.format("%s: %d (%s)", "López Andrea", 10, "Ok")
//López Andrea: 10 (Ok)
```

Este es un método estático y no se necesita de un objeto para invocarlo, sino que se utiliza la clase para invocarlo. Su estructura y funcionamiento es igual al del método *printf*.

Longitud (*length*):

```
"Catacocha la tierra de los Paltas".length()
//33
```

Devuelve el número de caracteres que están dentro de la cadena de texto

Está vacío (*isEmpty*):

```
"".isEmpty()
//true
" ".isEmpty()
//false
```

Devuelve true sí y sólo sí, la longitud (length) es cero. En la segunda invocación al método, la cadena de texto contiene un espacio en blanco.

Está en blanco (*isBlank*):

```
"".isBlank()
//true
" ".isBlank()
//true
"Hola".isBlank()
//false
```

Devuelve verdadero si la cadena está en blanco o contiene únicamente espacios en blanco, en cualquier otro caso devuelve falso.

Indice de (*indexOf*):

```
"Catacocha la tierra de los Paltas”.indexOf("ta")
//2
```

Devuelve el índice o posición en la que se encuentra la primera coincidencia del parámetro del método, en caso de no existir coincidentes el método devolverá -1.

Último indice de (last*IndexOf*):

```
"Catacocha la tierra de los Paltas”.indexOf("ta")
//30
```

Devuelve el índice o posición en la que se encuentra la última coincidencia del parámetro del método, en caso de no existir coincidentes el método devolverá -1.

Repetir (repeat):

```
"¡Java apuntes básicos! ".repeat(3)
//"¡Java apuntes básicos! ¡Java apuntes básicos! ¡Java apuntes básicos! "
```

Devuelve una nueva cadena de texto cuyo valor es la concatenación de si misma repetida el número de veces que indica su parámetro.

Reemplazar (replace):

```
"J@v@ @puntes básicos".replace("@", "a")
"Java apuntes básicos"
```

Devuelve una nueva cadena de texto en donde se reemplazó todas las apariciones de el primer parámetro con el segundo parámetro.

Split (dividir):

```
"Java apuntes básicos".split(" ")
String[3] { "Java", "apuntes", "básicos" }
```

Divide la cadena de texto en donde aparece el parámetro y almacena cada una de las partes en un arreglo.

Inicia con (*startsWith*):

```
"Catacocha la tierra de los Paltas".endsWith("Cat")
//true
```

Verifica si la cadena de texto inicia con el parámetro del método, devuelve verdadero sí es así, caso contrario devuelve falso.

Subcadena (*substring*):

```
"Catacocha la tierra de los Paltas".substring(13, 19)
//"tierra"
```

Devuelve una nueva cadena de texto con los caracteres que están entre el primer y segundo parámetro.

A minúsculas (*toLowerCase*):

```
"Catacocha la tierra de los Paltas".toLowerCase()
//catacocha la tierra de los paltas
```

Convierte todos los caracteres a minúsculas y devuelve esa nueva cadena de texto.

A mayúsculas (*toUpperCase*):

```
"Catacocha la tierra de los Paltas".toLowerCase()
//catacocha la tierra de los paltas
```

Convierte todos los caracteres a mayúsculas y devuelve esa nueva cadena de texto.

Podar (*trim*):

```
" Catacocha la tierra de los Paltas\t".trim()
//"Catacocha la tierra de los Paltas"
```

Devuelve una nueva cadena de texto, en la cual todos los espacios en blanco o caracteres que lo representen han sido eliminados.

Como ya se dijo, son muchos los métodos que posee la clase *String*, y aquí sólo se mostraría algunos de ellos. Siga las instrucciones que encontrará al inicio de este tema y podrá tener acceso a la documentación de Java y revise los métodos que se quedaron fuera.

El siguiente apartado muestra una característica que está disponible a partir de la versión 14 de Java de forma experimental, pero que pronto estará de forma definitiva, y que permite escribir de una mejor manera las cadenas de texto.

BLOQUES DE TEXTO

Hasta el momento ha revisado cadenas de texto que están formadas por una única línea, es decir no hay saltos de línea, pero, son bastante comunes las cadenas de texto que incluyan varios saltos de línea y su representación dentro de Java se torna engorrasa. Veamos un ejemplo: suponga que necesitar almacenar, en una variable, el siguiente texto, pero con la condición de conservar los saltos de línea.

```
Orillas del Zamora tan bellas
de verdes saucedales tranquilos
campiña de mi tierra risueña
casita de mis padres mi amor.
```

Una forma podría ser:

```
var almaLojana = "Orillas del Zamora tan bellas\nde verdes
saucedales tranquilos\ncampiña de mi tierra risueña\ncasita
de mis padres mi amor."
```

La representación anterior se escribe en una línea, agregando las secuencias de escape (\n) en la posición, en donde van los saltos de línea. Si bien es una representación válida, la legibilidad está comprometida, ya que la apariencia es diferente al texto original.

Tomando en cuenta la crítica anterior, una nueva forma de representar la variable es:

```
var almaLojana =
   "Orillas del Zamora tan bellas\n" +
   "de verdes saucedales tranquilos\n" +
   "campiña de mi tierra risueña\n" +
   "casita de mis padres mi amor."
```

Indudablemente la lectura del contenido de la variable ahora es mucho mas fácil y se asemeja al texto original, pero, ahora además de agregar el salto de línea, se concatena varias cadenas de texto (4 cadenas), que es un esfuerzo adicional.

Para resolver este tipo de problemas y otros, se ha diseñado el concepto de bloque de texto, que no es más que una representación de una cadena de texto que tiene varias líneas y que evita la necesidad de usar secuencias de escape, permitiendo que el programador tenga el control sobre el forma del texto.

Los bloques de texto empezaron a estar disponibles como una característica preliminar desde Java 13 y continuó en ese estado en Java 14, eso quiere decir que aún no se encuentran directamente disponibles, sino que es necesario realizar ciertas acciones previas para poder usarlas.

En Java 15 cambiaron de estado y se pueden utilizar directamente, pero, si usted trabaja con las versiones anteriores, deberá realizar algunas acciones adicionales antes de poder usar esta característica, así por ejemplo para jshell deberá activar las características preliminares, usando el siguiente comando:

```
jshell --enable-preview
```

Para utilizar los bloques de texto, se debe encerrar el texto entre tres comillas dobles en la apertura y cierre. El siguiente código muestra su uso

```
var almaLojana = """
   Orillas del Zamora tan bellas
   de verdes saucedales tranquilos
   campiña de mi tierra risueña
   casita de mis padres mi amor."""
```

Ahora se puede escribir el texto sin utilizar ni secuencias de escape ni concatenación de variables. La variable sigue siendo una cadena de texto y se podría usar todos los métodos que esa clase posee.

Todas las variables que se crean a través de bloques de texto, son del tipo de dato *String*, y se puede usar cualquier método de esa clase. Como se muestra en la siguiente porción de código

```
almaLojana.length()
almaLojana.repeat( 3 )
```

También se puede usar en cualquier método que reciba como parámetro un *String*, como lo muestra el siguiente ejemplo:

```
System.out.println("""
   Primera línea
   Segunda línea
   Tercera línea
""")
```

Existen algunas recomendaciones para el trabajo con los bloques de texto, así tenemos:

- Se deberían usar cuando mejoren la claridad y legibilidad del código, particularmente en cadenas de texto con varias líneas.
- Sí el texto cabe en una línea sin concatenación ni secuencias de escape, no use bloques de texto.

Existen otras recomendaciones que las puede encontrar aquí: https://cr.openjdk.java.net/~jlaskey/Strings/TextBlocksGuide_v11.html#style-guidelines-for-text-blocks

Hasta aquí el estudio de la clase *String* y sus métodos. En la última sección estudiará el manejo básico de archivos dentro de Java y aprenderá a leer y agregar contenido a los mismos.

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeos**:
  - Práctica Guiada #13 - Cadenas de texto (https://vimeo.com/455555954).
  - Práctica Guiada #14 - Bloques de texto (https://vimeo.com/455556340).

- **Laboratorio**: Funciones sobre tipos de dato String (http://j4loxa.com/courses/java101/blog/2017/lab12.html).

ARCHIVOS

Mientras un programa está ejecutándose los valores de las variables que utiliza se encuentran almacenados en memoria, pero cuando termina su ejecución todos esos valores desaparecen, es por esto que a las variables se las considera como un medio volátil de almacenamiento.

Si lo que busca es que las variables que maneja un programa dejen de ser volátiles, estas se deben almacenar en un medio de persistencia no volátil, como por ejemplo un archivo que se aloja en un disco duro o una memorias USB o en la nube.

En este apartado revisará los conceptos que le permitirán almacenar la información (variables) de un programa en un archivo, pero previo a ello es necesario conocer algunas características del manejo de archivos dentro del lenguaje de programación Java.

CONCEPTOS BÁSICOS

Los principales conceptos que debe tener claro cuando se habla de archivos es que Java asume que dentro de un disco duro existirán archivos y directorios.

Los directorios son contenedores de archivos e inclusive de otros directorios. En un sistema operativo como Windows los directorios reciben el nombre de carpetas, aunque formalmente Java menciona directorios, como concepto general e independiente de los sistemas operativos.

En cambio un archivo no es un contenedor, un archivo no puede incluir otros archivos. Un archivo únicamente puede contener datos. Adicionalmente un archivo tiene una extensión que está generalmente asociada con la aplicación que lo genera y que puede trabajarlo, por ejemplo, los archivos con extensión *txt* son archivos de texto, mientras que los archivos con extensión *docx* son archivos manejados por Microsoft Word.

Dentro de Java, tanto los archivos como los directorios son manejados por la clase denominada *File*, que es una representación abstracta de los nombres de rutas de archivos y directorios que permite tener acceso a los metadatos de archivos y directorios, no es una clase para realizar la lectura o escritura de archivos, para esas tareas existen otras clases.

Dentro de la clase *File*, los nombres de rutas de archivos están formados por dos partes:

- Un *prefijo* opcional dependiente del sistema operativo que especifica la unidad de disco. Para sistemas basados en UNIX es "/" para especificar el directorio raíz, mientras que Windows, definió Universal Naming Convention (UNC) que es un concepto que abarca varios elementos, pero que aquí se va resumir así: el prefijo se forma por una letra que señala la unidad, seguido por dos ":" y "\\".

- Una secuencia de cero o más cadenas que representan el *nombre*. Generalmente el primer nombre es un directorio, mientras que los siguientes pueden señalar un directorio o un archivo. Cada nombre se separa del siguiente por un carácter separador predeterminado que se puede obtener a través los atributos *separator* y *charSeparator*.

Otra característica, cuando se crea un objeto con la clase *File*, es que este puede apuntara un elemento, como un directorio o archivo, real o no del sistema de archivos, es decir, el nombre de ruta que se emplea no necesariamente debe señalar algo que existe en mi computador, puede que exista o puede que no. En caso de no existir, es posible que la tarea sea crear, en caso de existir es posible que las tareas sean: obtener información o leer o escribir datos.

Con estas consideraciones iniciales, lo primero que aprenderá es a consultar la información de archivos y directorios, para luego aprender a leer y escribir datos en un archivo.

USANDO FILE PARA OBTENER INFORMACIÓN

Antes de empezar a obtener información, es necesario crear un objeto de la clase *File*, para realizar esa actividad existen diferentes alternativas que se pueden ver en esta primera porción de código:

```
import java.io.File

File demoFile = new File("/tmp/demo.txt")
File imgsDir = new File("/tmp/images")
File fileTest = new File("test.txt")
```

Lo primero que se debe hacer para trabajar con archivos es importar la clase *File*, ya que esta no pertenece al paquete estándar de Java, sino que se encuentra en un paquete denominado *io*, que deriva de entrada (Input) y salida (Output). Describiendo de forma general a las sentencias, es posible afirmar que la primera y tercera apunta a un archivo, mientras que la segunda a un directorio. La descripción a detalle se hace en el siguiente párrafo.

Ahora que ya conoce algo de la clase *File*, es momento de entrar en los detalles, lo primero que debe haber notado es que es necesario crear objetos de la misma, para los dos primeros objetos(*demoFile* y *imgsDir*) se usaron rutas absolutas, es decir que incluyen prefijo y nombres. El tercer objeto utiliza una nombre de ruta relativa, es decir solo usa el nombre del archivo que incluye la extensión y es relativa al actual directorio de trabajo, este directorio se puede consultar así:

```
System.out.println(System.getProperty("user.dir"))
```

Ninguna de las acciones anteriores crea archivos o directorios y no generan errores si realmente no existen. Para hacer más didáctico este punto, suponga que está trabajando con la siguiente estructura de directorios

```
tmp
|____demo.txt
|____images
| |____Duke-Guitar.png
| |____281px-Duke-Guitar.png
| |____702px-Duke-Guitar.png
|____test.txt
```

En la estructura anterior existen 2 directorios (*tmp* e *images*) y 5 archivos (*demo.txt*, *Duke-Guitar.png*, *281px-Duke-Guitar.png*, *702Duke-Guitar.png* y *test.txt*). La jerarquía muestra que el directorio raíz será *tmp*, dentro están el directorio *images* y los archivos *demo* y *test*. Mientras que todos los archivos con extensión *png* están dentro del directorio *images*.

La estructura es independiente del sistema operativo, pero para los siguientes ejemplos usaré prefijos y separador de un sistema operativo *Unix*, para quienes están trabajando con Windows se debe cambiar los nombres de rutas. Con estas características, es momento de aprender a utilizar varios métodos de la clase *File*.

Existe (*exists*)

Este método verifica si el archivo o directorio que se usó para crear el objeto de la clase *File* existe realmente en el disco duro. En caso de no existir, devolverá falso. El siguiente código muestra su uso.

```
File tmpDir = new File("tmp")
tmpDir.exists() //true

File tmp2File = new File("tmp/tmp2.txt")
tmp2Dir.exists() //false

File dukeGuitar = new File("tmp/images/Duke-Guitar.png")
dukeGuitar.exists() //true
```

La porción de código muestra que se crean 3 objetos de la clase *File*, la invocación al método *exists* y como comentario, el valor que devuelve. Note que las rutas son relativas al directorio de trabajo. El objeto *tmp2File* utiliza una ruta que no existe, ya que el archivo *tmp2.txt* no esta en la estructura propuesta, es por ello que devuelve *false*. Para los otros dos objetos, devuelve *true* ya que tanto el directorio como el archivo si son parte de la estructura.

Es directorio (*isDirectory*)

```
File tmpDir = new File("tmp")
tmpDir.isDirectory() //true

File tmp2File = new File("tmp/tmp2.txt")
tmp2Dir.isDirectory() //false

File dukeGuitar = new File("tmp/images/Duke-Guitar.png")
dukeGuitar.isDirectory() //false
```

Este método devuelve *true*, si en la ruta que usó para crear el objeto de la clase *File*, se apunta a un directorio o tiene la forma de un directorio, carece de extensión, caso contrario devolverá *false*.

El código anterior, únicamente devuelve *true* para el objeto *tmpDir*, para los otros dos devuelve *false*, ya se trata de archivos y no directorios, inclusive para *tmp2File* que es un archivo que no existe, pero su nombre de ruta tiene la forma de un archivo (posee una extensión).

Es archivo (*isFile*)

```
File tmpDir = new File("tmp")
tmpDir.isFile() //false

File tmp2File = new File("tmp/tmp2.txt")
tmp2Dir.isFile() //true

File dukeGuitar = new File("tmp/images/Duke-Guitar.png")
dukeGuitar.isFile() //true
```

Es un método que devuelve *true* si el nombre de ruta apunta o tiene la estructura de un archivo y *false* si apunta a un directorio o su nombre de ruta carece de extensión, es decir, tiene la forma de un directorio.

En el ejemplo anterior el único objeto de la clase *File* que no tiene forma de archivo es el primero, además de no tener forma, físicamente es un directorio que existe, según la estructura propuesta.

Longitud (length)

```
File tmpDir = new File("tmp")
tmpDir.length() //192

File tmp2File = new File("tmp/tmp2.txt")
tmp2Dir.length() //0

File dukeGuitar = new File("tmp/images/Duke-Guitar.png")
dukeGuitar.length() //2417408
```

Si se trata de un archivo el método devuelve el tamaño en bytes del archivo que señala la ruta, en caso de que el objeto apunte a un archivo que no existe, devuelve 0.

Cuando se trata de directorios el resultado del método no corresponde al tamaño, de hecho la documentación dice que devuelve un valor no especificado. Por lo que no se debe usar en caso de directorios.

En el código el objeto *tmp2File*, no existe, por lo tanto el resultado que se obtuvo fue 0, mientras que el objeto *dukeGuitar*, el tamaño que es obtuvo fue de 2.4 MB aproximadamente.

Listar (list)

```
File tmpDir = new File("tmp")
tmpDir.list() //{ "demo.txt", "images", "test.txt" }

File tmp2File = new File("tmp/tmp2.txt")
tmp2Dir.list() //null

File dukeGuitar = new File("tmp/images/Duke-Guitar.png")
dukeGuitar.list() //null
```

En caso de ser un directorio, este método devuelve un arreglo con todos los nombres de los archivos y directorios que están dentro del directorio especificado en el nombre de ruta, como cadenas de texto. Si el nombre de ruta apunta a un archivo, el método devolverá *null*.

En el código, el único directorio corresponde al objeto *tmpDir* y lo que devuelve el método es el contenido del mismo, como se puede verificar en la estructura propuesta al inicio.

Listar como archivo (listFiles)

```
File tmpDir = new File("tmp")
tmpDir.listFiles() //{ tmp/demo.txt, tmp/images, tmp/test.txt }

File tmp2File = new File("tmp/tmp2.txt")
tmp2Dir.list() //null

File dukeGuitar = new File("tmp/images/Duke-Guitar.png")
dukeGuitar.list() //null
```

En caso de ser un directorio, este método devuelve un arreglo que contiene archivos y directorios que están dentro del directorio especificado en el nombre de ruta como objetos de la clase *File*. Si el nombre de ruta apunta a un archivo, el método devolverá *null*.

En el código, el único directorio corresponde al objeto *tmpDir* y lo que devuelve el método es el contenido del mismo, pero como objetos de la clase *File*, como se puede verificar en la estructura propuesta al inicio.

Hasta aquí esta revisión de unos pocos métodos de la clase *File*, si está interesado en conocer más acerca de estos métodos y muchos otros más, debe recurrir a la documentación de Java, para ello debe seguir las siguientes instrucciones:

- Visitar el sitio https://docs.oracle.com/en/java/javase/
- Seleccionar el enlace que está bajo *Latest Release*, por ejemplo *JDK 14*.
- En el menú de la izquierda seleccionar *API Documentation*.
- En la tabla seleccionar el módulo *java.base*.
- En la siguiente página, en la tabla packages, seleccionar el paquete *java.io*.
- Lo anterior lo llevará a una nueva página en la que debe buscar la tabla *Class Summary*, dentro de esa tabla encontrará la clase *File* sobre la cual debe dar clic, y podrá conocer sobre la clase y más métodos que posee la clase.

Antes de finalizar comentarles que un tema que no se mencionó al usar los métodos de la clase *File*, fueron los permisos que los usuarios tienen sobre los archivos o directorios reales. No olviden que los sistemas operativos modernos soportan varios usuarios y cada usuario tiene algunos permisos sobre archivos y directorios.

Al utilizar los métodos de la clase *File*, es posible que se pretenda infringir esos permisos lo que impedirá que el método devuelve el resultado descrito. Además, el tema permisos se debe considerar cuando se trabaja con archivos o directorios reales.

En el siguiente apartado estudiará acerca de la lectura y escritura de archivos de tipo texto utilizando Java. Si bien las tareas pueden parecer triviales, hay algunos aspectos a considerar y que se revisan en ese apartado.

LECTURA Y ESCRITURA DE ARCHIVOS DE TEXTO

Antes de iniciar es necesario aclarar que aquí se hará una presentación muy limitada de el trabajo con archivos, utilizando el lenguaje de programación Java. Principalmente por dos razones. La primera, la diversidad de formatos de archivos, usted conocen que casi por cada aplicación, existe un formato de archivos específico. La segunda, en Java existen diferentes formas para trabajar con archivos, algunas son bastante antiguas y tediosas.

Tomando en cuenta las consideraciones anteriores, se ha limitado este contenido al trabajo con archivos de texto plano y únicamente se presentará sólo dos formas, de las muchas, que existen en Java para leer y escribir archivos que resultan compactas y fáciles de comprender.

Antes de iniciar comentar que se usará la estructura de archivos que se presentó en el apartado *Usando File para obtener información*. Es momento de revisar una primera forma para leer archivos de texto usando Java, para ello analice el siguiente código:

```
import java.io.File;
import java.util.Scanner;

File testFile = new File( "tmp/test.txt" )
Scanner textOfFile = new Scanner( testFile )

while( textOfFile.hasNextLine() ) {
   System.out.println( textOfFile.nextLine() );
}
textOfFile.close()
```

Las clases que se utilizarán son la clase *File* y *Scanner*, esto justifica los *imports* que se realizan al inicio. Para crear el objeto *textOfFile* se envía como parámetro el objeto *testFile*, así se asocian ambos objetos y se conoce que archivo leer.

El ciclo repetitivo itera mientras el archivo tiene líneas y presenta a cada una de ellas con los métodos *hasNextLine* y *nextLine*. La última sentencia cierra el recurso que se abrió anteriormente.

La segunda forma de leer el contenido de un archivo se muestra a continuación:

```
import java.util.stream.Stream;
import java.nio.file.Files;
import java.nio.file.Paths;

var pathToData = "tmp/test.txt"
Stream<String> fileStream = Files.lines(Paths.get( pathToData ))
fileStream.forEach(System.out::println)
```

Esta manera es mucho más sencilla que la anterior, aunque utiliza algunas clases adicionales y conceptos de programación funcional, pero se puede resumir así: crea un flujo con cada una de las líneas que están dentro del archivo, luego imprime cada una de ellas. Si bien es

una forma más sencilla, puede resultar compleja de comprender considerando los conceptos adicionales que emplea.

Al igual que con la lectura, se presentarán dos métodos para realizar la escritura de datos en archivos de texto. La primera se muestra en la siguiente porción de código:

```
import java.nio.file.Files;
import java.nio.file.Paths;

var contenido = "\nHola, Java apuntes básicos”
var pathToData = "tmp/test.txt"
Files.write(
   Paths.get( pathToData ),
   contenido.getBytes(),
   StandardOpenOption.APPEND )
```

Nuevamente se hace uso de las clases *Files* y *Paths* del paquete *nio*, que se podría traducir cómo el nuevo Input/Output debido a que posee algunas clases que simplifican y mejoran las existentes. Observe cómo se utiliza únicamente el método *write* y se envía como parámetro el nombre de ruta del archivo, el contenido a escribir, como bytes y la operación que se hará, en este caso agregar el nuevo contenido al existente.

Si bien el código anterior agrega un nuevo contenido a uno que ya existía, el siguiente código no agrega sino que reemplaza el texto existe con uno nuevo.

```
import java.io.File;
java.io.FileWriter

File testFile = new File( "tmp/test.txt" )

FileWriter out = new FileWriter( testFile )
out.write("Hola mundo")
out.write("Adiós")
out.close()
```

En este caso se crea un objeto de la clase *FileWriter* utilizando el objeto *testFile*, luego se usa el método *write* para escribir en el archivo. Finalmente se cierra el recurso. Así se escribe un nuevo contenido en el archivo.

Para agregar contenido al archivo es necesario señalar que se va agregar contenido o lo que se denomina modo “append”, eso se consigue, agregando un parámetro cuando se crea el objeto *FileWriter*, así:

```
import java.io.File;
java.io.FileWriter

File testFile = new File( "tmp/test.txt" )

FileWriter out = new FileWriter( testFile, true )
out.write("Hola mundo")
out.write("Adiós")
out.close()
```

De esa manera el contenido se agregará al final del archivo. La clase *FileWriter* es ideal para el trabajo con archivos de texto, ya que entre cosas permite trabajar con las diferentes formas de codificación de los archivos.

LECTURA DE ARCHIVOS VIA URL

Antes de concluir con los contenidos de este documento se muestra un ejemplo de cómo leer un archivo que se encuentra publicado en una sitio Web, es decir se leerá su código html.

La forma de hacerlo será a través de un objeto de la clase Scanner, y su forma es bastante similar a la lectura de archivos, como se mostró anteriormente. El código es el siguiente:

```
import java.net.URL
import java.util.Scanner

var j4loxaHomePage = "http://j4loxa.com/index.html"
URL webPageURL = new URL( j4loxaHomePage )
Scanner webPage = new Scanner( webPageURL.openStream() )

while ( webPage.hasNextLine() ) {
   System.out.println( webPage.nextLine() );
}
webPage.close();
```

El código trabaja con la clase URL (Uniform Resource Locator), es decir apunta a un recurso de la World Wide Web y necesita de una cadena de texto para crear objetos, en este la URL de una página Web. Una vez creada la URL, se abre una conexión a esa recurso a través del método *openStream* y se usa para crear el objeto de la clase *Scanner*. La lectura y presentación es la misma como si tratará de un archivo local.

Así se cierra esta apartado que mostró de una forma resumida el trabajo con archivos.

RECURSOS ADICIONALES

Para complementar los contenidos de este apartado te invito a revisar los recursos, que se muestran a continuación. Generalmente encontrará, vídeos que muestra cómo emplear lo aprendido para construir un programa Java y un laboratorio de programación que describe paso a paso cómo resolver un problema y escribir un programa Java todo sin salir de la Web.

- **Vídeos**:
  - Práctica Guiada #15 - Obtener información de un archivo (https://vimeo.com/455557514).
  - Práctica Guiada #16 - Escribir y leer archivos de texto (https://vimeo.com/455558651).
- **Laboratorio**: Archivos (http://j4loxa.com/courses/java101/blog/2017/lab13.html).

PALABRAS FINALES

Una recomendación final, mucho más importante que aprender el lenguaje de programación de moda o el framework más utilizado para construir aplicaciones, es aprender sobre los paradigmas de programación, ya que los lenguajes ganan y pierdan popularidad, los mejores frameworks están por construirse, es decir, son conceptos dinámicos y cambiantes, pero los paradigmas no y sobre estos se sientan las bases que los lenguajes de programación tratan de asimilar de mejor o peor manera.

En este pequeño documento se hizo una revisión del paradigma de programación estructurado, es decir, revisó algunas estructuras de control que permiten construir programas. También conoció brevemente sobre la programación modular, representada por métodos y clases, que aunque no ha construido ha aprendido a utilizar y por último ha visto pocos ejemplos del paradigma funcional.

El futuro de los programadores es ser políglotas, en todo ámbito de los lenguajes, tanto el que entienden los humanos como el que entienden las máquinas. Su tarea, aunque ahora mismo parece compleja, es aprender más de un paradigma de programación y luego seleccionar algún o algunos lenguajes que le permitan aplicarlos.

Hoy en día, muchos lenguajes de programación soportan ya varios paradigmas, y los que aún no lo hacen, de seguro están en camino de implementarlo. Entonces, hay que estar preparados. La tendencia es orientación a objetos y funcional. Tal vez y la próxima vez que nos encontremos sea para estudiar algún de esos paradigmas.

Gracias por llegar hasta aquí y leer estas últimas palabras, como se pudo percatar, esta no es una obra terminada, esta en proceso de construcción y revisión, es una obra en versión beta perpetua. Y si quiere ayudar a mejorarla, no dude en escribirme a mi correo: jorgaf@gmail.com.

¡Mil gracias!