



**ULEAM**  
UNIVERSIDAD LAICA  
ELOY ALFARO DE MANABÍ



EDITORIAL  
MAR ABIERTO

# PROGRAMACIÓN EN Java™

## ORIENTADA A OBJETOS A TRAVÉS DE EJEMPLOS

Colección  
(T.I.C.)

Edwin René Guamán Quinche  
José Oswaldo Guamán Quinche  
Daysi Mireya Erreyes Pinzón  
Hernán Leonardo Torres Carrión

Universidad Laica Eloy Alfaro de Manabí  
Ciudadela universitaria vía circunvalación (Manta)  
[www.uleam.edu.ec](http://www.uleam.edu.ec)

**Autoridades:**

Miguel Camino Solórzano, Rector  
Iliana Fernández, Vicerrectora Académica  
Doris Cevallos Zambrano, Vicerrectora Administrativa

**Programación en Java. Orientada a objetos a través de ejemplos**

©Edwin René Guamán Quinche  
©José Oswaldo Guamán Quinche  
©Daysi Mireya Erreyes Pinzón  
©Hernán Leonardo Torres Carrión

**Revisión pares académicos:**

Nombre: René Rolando Elizalde Solano  
Institución: Universidad Técnica Particular de Loja  
Tiempo completo  
Teléfono: 0994049481  
Email: [rrelizalde@utpl.edu.ec](mailto:rrelizalde@utpl.edu.ec)

Nombre: Freddy Patricio Ajila Zaquinaula  
Institución: Escuela Superior Politécnica de Chimborazo  
Tiempo completo  
Teléfono: 098057220  
Email: [freddy.ajila@epoch.edu.ec](mailto:freddy.ajila@epoch.edu.ec)

**Consejo Editorial:** Universidad Laica Eloy Alfaro de Manabí

**Director Editorial:** Hernán Murillo Bustillos

**Diseño de cubierta:** José Márquez

**Diseño y diagramación:** José Márquez

**Estilo, corrección y edición:** Alexis Cuzme (DEPU)

**ISBN:** 978-9942-775-05-4

Edición: Primera. Noviembre 2017

Departamento de Edición y Publicación Universitaria (DEPU)  
Editorial Mar Abierto  
2 623 026 Ext. 255  
[www.marabierto.uleam.edu.ec](http://www.marabierto.uleam.edu.ec)  
[www.depuproject.blogspot.com](http://www.depuproject.blogspot.com)  
[www.editorialmarabierto.blogspot.com](http://www.editorialmarabierto.blogspot.com)  
Manta - Manabí - Ecuador

## DEDICATORIA

A Dios que es mi fortaleza, a mis padres por el apoyo y amor recibido, a mi querida esposa e hijos que son mi fuente de inspiración y el motivo de superación.

René

A mi familia, quienes han estado a mi lado todo este tiempo, a mis padres y hermanos por apoyarme en todo este proceso y a todas aquellas personas que me prestaron su ayuda.

José

Con reverencia y admiración a Dios, porque es la luz que ha guiado mi existencia desde siempre; a mi esposo por su apoyo incondicional y a mis tres hijos quienes con su inocencia y amor se han convertido en la principal inspiración de todos mis logros y éxitos.

Mireya

A mi familia, especialmente a mis padres que son ejemplo de lucha y progreso, a mis hermanos Paúl y Renato, y a mi querida esposa Natalia y a mis hijos Nicolás y Felipe quienes son mi fuente de inspiración y el motor de mi superación.

Hernán

## **RESUMEN**

En este libro se ejemplifica como programar con Java, para ello ha sido necesario contar con el aporte de varios colaboradores en el campo de desarrollo de aplicaciones informáticas, lo que ha permitido que la realización del presente trabajo sea posible y que sobre todo sirva a las futuras generaciones de programadores.

En su estructura consta de varias secciones y al final de cada una los respectivos ejemplos y cuestionarios, que han permitido exponer de forma concreta conceptos esenciales en la vida de un buen desarrollador; en el capítulo uno se da a conocer la introducción de la programación orientada a java, encontrará características, versiones, jdk y su respectiva configuración; el capítulo dos está orientado a los fundamentos y requisitos básicos para iniciar un programa en java como palabras reservadas, manejo de comentarios, errores comunes, paquetes y la manera correcta de importarlos; en el capítulo tres se detalla el uso de tipos de datos primitivos, identificadores, variables y constantes como elementos claves en la creación de cualquier programa; el cuarto capítulo se orienta al manejo de todos los tipos de operadores que son posibles de manipular en java; en el capítulo cinco están descritas las estructuras de control necesarias para controlar el flujo de los datos; en el capítulo seis se encuentra una sección completa del manejo adecuado de los métodos y parámetros; las clases y los objetos se detallan en el capítulo siete; en el capítulo ocho se detalla la estructura de un tipo de dato enumerado como datos simple o como un objeto; en el capítulo nueve se da a conocer una característica muy interesante que es la herencia, con sus respectivos conceptos como jerarquía, redefinición y sobre escritura de elementos y la sentencia súper; en el capítulo diez se definen los conceptos relacionados con la programación de interfaces y clases abstractas; en el capítulo once se explica los mecanismos de implementación al establecer las relaciones entre clases. En el capítulo doce se introduce al manejo de excepciones y la diferencias entre errores de compilación y de ejecución; en el capítulo trece se describe las principales clases utilizada por los programadores para diseñar sus programas y las clases envolventes; en el capítulo catorce se explica el manejo de cadena a través de sus métodos más importantes. Para finalizar se realiza una explicación del manejo de arreglos y matrices a través de ejemplos.

Palabras clave: Java, objetos, programación.

## CONVENCIONES TIPOGRÁFICAS

Se ha adoptado un conjunto de convenciones tipográficas con el objetivo de mejorar la lectura de los conceptos y comprender los ejercicios planteados.

Los estilos empleados en el texto son:

- El texto general del libro se utiliza la letra Times New Roman.
- Las palabras reservadas se destacan con color morado. Ejemplo: **new, for, if**, etc.
- Los nombres de los programas, clase, métodos se destacan con tipo de letra Courier (11 puntos). Ejemplo: Estudiante, muestraMensaje, IPersona, etc
- Toda palabra que no sea en español, se destacan con cursiva. Ejemplo: *String Development Kit, swing*

Los estilos empleados en los ejemplos son:

- Se han omitido las tildes y eñes en los identificadores de los ejemplos propuestos al ser Java un lenguaje escrito en inglés. Ejemplo: anio, dias, numero, etc.
- Los ejemplos normalmente están completos y por lo tanto se pueden escribir y probar. Se los ha encerrado en cajas y están numerados. El texto de los ejemplos emplea un tipo de letra de paso fijo Courier. Ejemplo:

*Ejemplo 1 Estilo del código Java*

```
1 public class Computadora {  
2     String marca;  
3     double precio;  
4  
5     public void prender(){  
6         // bloque de código del método  
7         String mensaje = "Se prendió";  
8         System.out.println(mensaje);  
9     }  
10 }
```

- Los números de línea permiten hacer referencia a una instrucción concreta del código.

- Los ejemplos parciales tienen el mismo formato que un ejemplo completo, pero no se colocará su título.

```
1 | int numero = 9;  
2 | String mensaje = "Se prendió";
```

- La salida que genera un código de ejemplo o un programa se encierra en un cuadro y están numerados (el texto de las salidas emplea un tipo de letra de paso fijo Courier). Ejemplo:

```
1 | \programas_libro>java MayorTresNumeros  
2 | Ingrese el primer número  
3 | 7  
4 | Ingrese el segundo número  
5 | 8  
6 | Ingrese el tercer número  
7 | 6  
8 | El número mayor es = 8
```

Los estilos empleados en los cuadros de sintaxis y recomendaciones:

- Las sintaxis de cada elemento va encerrado en un cuadro sin numeración. Ejemplo:

```
import java.paquete.NombreClase1;  
import java.paquete.NombreClase2;
```

- Las recomendaciones o sugerencias va encerrado en un cuadro sin numeración. Ejemplo:

#### **Recomendación**

Se recomienda que al definir una clase, se establezca el inicio y final de la clase a través de la llave abierta y cerrada { }, los programadores que están aprendiendo este lenguaje de programación, suelen olvidar cerrar la clase, esto ayudará a evitar errores al momento de compilar sus aplicaciones.

## Contenido

INTRODUCCIÓN.....	15
1 FUNDAMENTOS DE JAVA .....	16
1.1 INTRODUCCIÓN .....	16
1.2 CARACTERÍSTICAS .....	17
1.3 VERSIONES.....	18
1.4 JDK.....	19
1.5 JVM.....	19
1.6 JRE.....	21
1.7 INSTALACIÓN Y CONFIGURACIÓN DE JAVA.....	21
1.7.1 WINDOWS.....	22
1.7.2 LINUX.....	25
2 FUNDAMENTOS DE LA PROGRAMACIÓN DE JAVA.....	27
2.1 INTRODUCCIÓN A LAS CLASES .....	27
2.2 PALABRAS RESERVADAS.....	32
2.3 COMENTARIOS.....	33
2.4 MÉTODO Main .....	34
2.5 PRIMER PROGRAMA.....	36
2.6 ERRORES COMUNES AL COMPILEAR UNA CLASE.....	38
2.7 PAQUETES E IMPORTACIONES.....	39
2.8 EJEMPLOS.....	44
2.9 CUESTIONARIO .....	45
3 TIPOS DE DATOS PRIMITIVOS, IDENTIFICADORES.....	48
3.1.1 TIPOS DE DATOS PRIMITIVOS.....	48
3.1.2 DATOS ENTEROS: .....	49
3.1.3 DATOS DECIMALES: .....	50
3.1.4 DATOS CARACTER.....	51
3.1.5 DATOS LÓGICOS.....	52
3.2 Variables, Identificadores y constantes.....	53
3.2.1 VARIABLE .....	53
3.2.2 IDENTIFICADORES.....	55
3.2.3 CONSTANTES.....	56
3.3 EJERCICIOS.....	56
3.4 CUESTIONARIO .....	58
4 OPERADORES .....	60
4.1 INTRODUCCIÓN .....	60

4.2	OPERADORES ARITMÉTICOS.....	61
4.3	OPERADORES DE RELACIÓN.....	63
4.4	OPERADORES CONDICIONALES .....	64
4.5	OPERADORES A NIVEL DE BITS.....	65
4.6	OPERADOR DE ASIGNACIÓN.....	66
4.7	OPERADOR DE INCREMENTO Y DECREMENTO .....	68
4.8	¿OPERADOR TERNARIO ?:.....	70
4.9	PRECEDENCIA DE OPERADORES.....	70
4.10	EJERCICIOS.....	71
4.11	CUESTIONARIO .....	73
<b>5</b>	<b>ESTRUCTURAS DE CONTROL .....</b>	<b>76</b>
5.1	INTRODUCCIÓN .....	76
5.2	ESTRUCTURAS CONDICIONALES O SELECTIVAS.....	77
5.2.1	ESTRUCTURA if – then.....	77
5.2.2	ESTRUCTURA if / else.....	78
5.2.3	ESTRUCTURAS CONDICIONALES ANIDADOS.....	82
5.2.4	OPERADOR TERNARIO.....	83
5.2.5	ESTRUCTURA switch.....	85
5.3	ESTRUCTURAS REPETITIVAS .....	89
5.3.1	ESTRUCTURA for .....	89
5.3.1.1	Bucles Infinitos.....	92
5.3.1.2	Términos múltiples en la declaración del bucle for.....	92
5.3.1.3	BUCLE FOR ANIDADOS.....	93
5.3.2	ESTRUCTURA while .....	94
5.3.3	ESTRUCTURA do while.....	97
5.3.4	SENTENCIAS DE SALIDA DE UN BUCLE.....	98
5.3.4.1	SENTENCIA break.....	98
5.3.4.2	SENTENCIA continue.....	101
5.4	EJERCICIOS.....	103
5.5	CUESTIONARIO .....	106
<b>6</b>	<b>MÉTODOS.....</b>	<b>112</b>
6.1	DEFINICIÓN DE MÉTODOS.....	112
6.2	MÉTODOS QUE NO DEVUELVE NINGÚN VALOR Y SIN PARÁMETROS .....	114
6.3	MÉTODOS QUE DEVUELVE UN VALOR.....	115
6.4	PARÁMETROS DE UN MÉTODO .....	117
6.5	EJERCICIOS.....	118
6.6	CUESTIONARIO .....	120

7	CLASES Y OBJETOS.....	124
7.1	CLASE .....	124
7.1.1	ATRIBUTOS .....	126
7.2	CONSTRUCTOR.....	128
7.2.1	CONSTRUCTOR POR DEFECTO.....	129
7.2.2	CONSTRUCTOR CON PARÁMETROS.....	130
7.3	OBJETOS.....	131
7.4	MODIFICADORES DE ACCESO.....	133
7.4.1	MODIFICADOR PÚBLICO .....	134
7.4.2	MODIFICADOR PRIVADO.....	135
7.4.3	MODIFICADOR PROTEGIDO.....	139
7.4.4	MODIFICADOR POR DEFECTO.....	142
7.5	SENTENCIA this .....	144
7.6	ATRIBUTOS Y MÉTODOS ESTÁTICOS.....	146
7.7	CUESTIONARIO .....	147
8	DATOS DE TIPO ENUMERADO.....	152
8.1	INTRODUCCIÓN .....	152
8.2	DEFINICIÓN .....	153
8.3	DATOS DE TIPO ENUMERADO AVANZADOS.....	157
8.4	EJEMPLOS.....	159
8.5	CUESTIONARIO .....	162
9	HERENCIA.....	165
9.1	DEFINICIÓN .....	165
9.2	JERARQUÍA DE CLASES.....	168
9.3	REDEFINICIÓN O SOBRE ESCRITURA DE ELEMENTOS.....	170
9.4	SENTENCIA súper .....	171
9.5	CLASE OBJECT .....	173
9.6	EJERCICIO .....	173
9.7	CUESTIONARIO .....	175
10	INTERFACES Y CLASES ABSTRACTAS.....	177
10.1	INTERFACES.....	177
10.1.1	ATRIBUTOS EN LAS INTERFACES .....	180
10.1.2	IMPLEMENTACIÓN DE VARIAS INTERFACES .....	181
10.2	CLASES ABSTRACTAS.....	183
10.3	EJEMPLO .....	184
10.4	CUESTIONARIO .....	189
11	RELACIONES ENTRE CLASE .....	192

11.1	DEFINICIÓN .....	192
11.2	ASOCIACIÓN .....	193
11.2.1	NAVEGACIÓN DE LAS ASOCIACIONES.....	193
11.3	AGREGACIÓN Y COMPOSICIÓN .....	197
11.3.1	AGREGACIÓN.....	197
11.3.2	COMPOSICIÓN.....	199
12	EXCEPCIONES.....	203
12.1	DEFINICIÓN .....	203
12.2	CAPTURA DE EXCEPCIONES.....	205
12.3	BLOQUE finally .....	207
12.4	LANZAR UNA EXCEPCIÓN .....	208
12.5	PROPAGACIÓN DE EXCEPCIONES.....	210
12.6	CREAR PROPIAS EXCEPCIONES.....	211
12.7	PREGUNTAS.....	212
13	CLASES FUNDAMENTALES.....	214
13.1	CLASE Object.....	214
13.1.1	MÉTODO equals.....	214
13.1.2	MÉTODO toString.....	216
13.2	CLASES ENVOLVENTE.....	217
13.2.1	CLASE Integer.....	217
13.2.2	CLASE Short .....	218
13.2.3	CLASE Long .....	219
13.2.4	CLASE Byte .....	219
13.2.5	CLASE Double .....	220
13.2.6	CLASE Float.....	221
13.2.7	CLASE Boolean.....	222
13.2.8	CLASE Character.....	222
13.2.9	CONVERSIONES ENTRE LOS TIPOS PRIMITIVOS Y SUS CLASES ENVOLVENTE .....	223
13.3	CLASE Math.....	224
13.4	CLASE Scanner.....	226
13.5	CLASE Date.....	228
13.6	EJEMPLOS.....	229
13.7	CUESTIONARIO .....	231
14	MANEJO DE CADENAS.....	233
14.1	DEFINICIÓN .....	233
14.2	COMPARACIÓN ENTRE CADENAS .....	235

14.2.1	MÉTODO equals.....	235
14.2.2	MÉTODO equalsIgnoreCase.....	236
14.2.3	MÉTODO compareTo.....	236
14.3	INFORMACIÓN BÁSICAS DE CADENAS.....	237
14.3.1	MÉTODO length.....	237
14.3.2	MÉTODO charAt.....	238
14.4	MANEJO DE CADENAS.....	240
14.4.1	MÉTODO concat.....	240
14.4.2	MÉTODO replace.....	240
14.4.3	MÉTODO trim.....	241
14.4.4	MÉTODO split.....	241
14.4.5	MÉTODOS toUpperCase y toLowerCase.....	242
14.5	MÉTODO CON SUBCADENAS.....	243
14.6	CONVERSIÓN A CADENAS.....	244
14.7	CLASE StringBuilder.....	244
14.8	EJEMPLOS.....	246
14.9	CUESTIONARIO.....	247
15	ARREGLOS.....	248
15.1	DEFINICIÓN.....	248
15.2	ARREGLOS UNIDIMENSIONALES.....	249
15.2.1	DECLARACIÓN Y CREACIÓN.....	250
15.2.2	LECTURA Y ESCRITURA.....	252
15.2.3	EJEMPLOS CON ARREGLOS.....	253
15.3	ARREGLOS BIDIMENSIONALES.....	257
15.3.1	DECLARACIÓN Y CREACIÓN.....	258
15.3.2	LECTURA Y ESCRITURA.....	261
15.3.3	EJEMPLOS.....	262
	BIBLIOGRAFÍA.....	265
	SOBRE LOS AUTORES.....	266

## ÍNDICE DE FIGURAS

Figura 1 Proceso de ejecución de una aplicación Java .....	20
Figura 2 Descarga del SDK .....	21
Figura 3 Instalación correcta del JDK .....	22
Figura 4. Propiedades de mipc .....	23
Figura 5 Propiedades del sistema - variables de entorno .....	23
Figura 6 Variable de entorno .....	24
Figura 7 Variable de Sistema .....	24
Figura 8 Configuración correcta de JDK .....	25
Figura 9 sudo update-alternatives .....	26
Figura 10 Representación abstracta del conocimiento .....	29
Figura 11 Representación de la clase Computadora en UML .....	29
Figura 12 Primera Clase .....	37
Figura 13 Archivos Computadora.java y Computadora.class .....	37
Figura 14 Error de compilación .....	38
Figura 15 Representación de paquetes .....	40
Figura 16 Sintaxis de una estructura selectiva simple .....	77
Figura 17 Sintaxis de la estructura selectiva doble .....	78
Figura 18 Sintaxis del operador ternario .....	83
Figura 19 Sintaxis de la estructura de selectiva múltiple .....	87
Figura 20 Sintaxis de la estructura repetitiva para .....	90
Figura 21 Estructura de una estructura repetitiva while .....	95
Figura 22 Estructura de la estructura repetitiva hasta que .....	97
Figura 23 Estructura de un método .....	113
Figura 24 Invocación a un método .....	114
Figura 25 Métodos en UML .....	114
Figura 26 Métodos de la clase en UML .....	115
Figura 27 Métodos de la clase en UML sentencia return .....	116
Figura 28 Métodos con parámetros .....	117
Figura 29 Clase en UML .....	125
Figura 30 Diagrama de clases básico .....	125
Figura 31 Estructura de una clase .....	126
Figura 32 Atributos en UML .....	127
Figura 33 Estructura de un atributo .....	128
Figura 34 Constructor por defecto .....	129
Figura 35 Estructura para crear un objeto .....	132
Figura 36 Modificador acceso público .....	134
Figura 37 Modificador acceso privado .....	136
Figura 38 Diagrama paquete proyecto libro .....	139
Figura 39 Diagrama de paquetes .....	141
Figura 40 this, diferencia entre atributo y parámetro .....	144
Figura 41 Representación de un tipo de dato enumerado un UML .....	155
Figura 42 Relación entre un dato de tipo enumerado y una clase .....	155
Figura 43 Dato de tipo enumerado avanzado .....	158
Figura 44 Relación entre una clase y un enum .....	160
Figura 45 Representación de herencia en UML .....	166
Figura 46 Relación de herencia .....	166

Figura 47 Herencia.....	167
Figura 48 Jerarquía de clases.....	168
Figura 49 Sobre escritura de métodos.....	170
Figura 50 Ejemplo de herencia .....	173
Figura 51 Interfaz en UML .....	178
Figura 52 Representación de la implementación de interfaces .....	179
Figura 53 Implementación de una interfaz.....	179
Figura 54 Implementación de varias interfaces.....	181
Figura 55 Clase Abstracta.....	183
Figura 56 Implementación de una clase abstracta .....	184
Figura 57 Diagrama interfaces y clases abstractas.....	185
Figura 58 Diagrama Clase Abstracta e Interfaces.....	187
Figura 59 Asociación unidireccional.....	193
Figura 60 Asociación unidireccional.....	194
Figura 61 Relación de asociación bidireccional.....	195
Figura 62 Asociación de 1..* .....	196
Figura 63 Relación *..*.....	196
Figura 64 Representación de agregación.....	197
Figura 65 Agregación entre clases.....	197
Figura 66 Relación de composición .....	200
Figura 67 Composición entre clases .....	200
Figura 68: Diagrama de clase de Excepciones .....	205
Figura 69 Estructura de una cadena.....	239
Figura 70 Representación de un arreglo.....	249
Figura 71 Partes de un arreglo.....	251
Figura 72 Valores por defecto en un arreglo de enteros.....	251
Figura 73 Valores por defecto en un arreglo de String .....	252
Figura 74 Escritura de un arreglo.....	252
Figura 75 Componentes de una matriz.....	258
Figura 76 Partes de un arreglo.....	259
Figura 77 Valores por defecto en una matriz de enteros.....	260
Figura 78 Valores por defecto en una matriz de String .....	260
Figura 79 Escritura de una matriz.....	261

# ÍNDICE DE TABLAS

Tabla 1 Palabras reservadas .....	32
Tabla 2 Tag para documentar clases con JavaDoc .....	34
Tabla 3 Paquetes principales en java .....	40
Tabla 4 Métodos principales de la clase Scanner .....	42
Tabla 5 Caracteres especiales.....	52
Tabla 6 Tipos de datos primitivos .....	52
Tabla 7 Operadores aritméticos .....	61
Tabla 8 Operadores de relación.....	63
Tabla 9 operadores condicionales .....	64
Tabla 10 Operadores de bit .....	65
Tabla 11 Operador de asignación =.....	66
Tabla 12 Operadores de asignación.....	67
Tabla 13 Operadores de incremento o decremento.....	68
Tabla 14 Precedencia de operadores .....	71
Tabla 15 Reglas para generar los métodos get y set .....	138
Tabla 16 Modificares de acceso .....	144
Tabla 17 Métodos de la clase Enum .....	156
Tabla 18 Métodos de la clase Object .....	173
Tabla 19 Cardinalidades .....	194
Tabla 20 Representación de la cardinalidad .....	195
Tabla 21 Diferencias entre las relaciones de agregación y composición .....	201
Tabla 22 Datos primitivos y Clases envolventes.....	217
Tabla 23 Atributos de la clase Integer.....	217
Tabla 24 Métodos principales de la clase Integer .....	217
Tabla 25 Atributos de la clase Short.....	218
Tabla 26 Métodos de la clase Short.....	218
Tabla 27 Atributos de la clase Long .....	219
Tabla 28 Métodos de la clase Long.....	219
Tabla 29 Atributos clase Byte .....	219
Tabla 30 Métodos de la clase Byte.....	220
Tabla 31 Atributos de la clase Double.....	220
Tabla 32 Métodos de la clase Double.....	220
Tabla 33 Atributos de la clase Float.....	221
Tabla 34 Métodos de la clase Float.....	221
Tabla 35 Atributos de la clase Boolean.....	222
Tabla 36 Métodos principales de la clase Boolean .....	222
Tabla 37 Atributos principales de la clase Character .....	222
Tabla 38 Métodos principales de la clase Character .....	223
Tabla 39 Métodos principales de la clase Scanner.....	224
Tabla 40 Métodos principales de la clase Scanner.....	226
Tabla 41 Métodos principales de la clase Date.....	228
Tabla 42 Métodos principales de la clase String.....	234
Tabla 43 Métodos principales de la clase StringBuilder.....	245

## INTRODUCCIÓN

El presente libro denominado “Introducción a la programación orientada a objetos en Java a través de ejemplos”, en el que se presentan los fundamentos del lenguaje Java a través de ejemplos, ha sido desarrollado por un grupo de profesionales de la Ingeniería en Sistemas que desean compartir sus conocimientos adquiridos en el quehacer de la enseñanza a nivel universitario.

La idea de desarrollar este libro, surge como fruto de la experiencia por varios años como docentes en el campo de la programación y por la imperiosa necesidad de colaborar con las personas que se inician en este fascinante mundo de la programación, y que al momento de implementar sus programas prefieren el lenguaje Java.

El libro está pensado, para todos aquellos principiantes que no tienen conocimientos de programación en general y de Java en particular, donde se explica mediante ejemplos términos que para la mayoría de personas resultan complejos. El propósito es para que puedan fundamentar sus primeros proyectos orientados a la resolución de problemas del entorno real y con ello mejorar sus destrezas y capacidades en el arte de la programación.

El método empleado para el desarrollo del presente libro se basa en la experiencia de los profesionales, comprometidos con la programación y deseosos de aportar en este ámbito muy importante de la ciencia, así mismo ha sido necesaria la revisión literaria de conceptos básicos en fuentes oficiales en la web y varios libros relacionados con la Informática y Computación.

# **1 FUNDAMENTOS DE JAVA**

---

En este capítulo se realiza una descripción de las características que tiene el lenguaje Java y las versiones del SDK. Además, se explica el proceso de instalación en los sistemas operativos window y linux. Por último, se indica la configuración del CLASSPATH.

## **Objetivos:**

- Identificar las fortalezas que brinda el lenguaje Java al construir aplicaciones de escritorio.
- Configurar la CLASSPATH de java en los sistemas operativos Window y Ubuntu.

## **1.1 INTRODUCCIÓN**

Dentro de las etapas del ciclo de vida de desarrollo de aplicaciones informáticas, está la programación, en la cual es necesario adoptar un determinado lenguaje. Un lenguaje de programación, es el mecanismo que permite al programador comunicarse con el computador mediante la codificación y ejecución de instrucciones para realizar determinado proceso o tarea.

Un lenguaje de programación, es la herramienta base para codificar las instrucciones que el programador escribe relacionando símbolos, palabras reservadas y reglas sintácticas según el lenguaje empleado, que le permite una comunicación adecuada con el computador y a su vez controlar la ejecución ordenada y secuencial de tareas para la obtención de un resultado que se conoce como programa.

Con la aparición de los lenguajes orientados a objetos la eficiencia y los resultados a obtener dependen de la forma como el programador modele e interprete las

características y acciones de los objetos como si se trataran de objetos del mundo real.

Existe una jerga extensa de lenguajes de programación orientados a objetos (POO), uno de ellos es Java, que apareció en el año 1992, y desde entonces se ha convertido en uno de los más populares y empleado para desarrollo de aplicaciones.

## 1.2 CARACTERÍSTICAS

Entre las más importantes tenemos:

- **Simple**, permite a los desarrolladores novatos lograr una curva de aprendizaje significativa en corto tiempo, así mismo programadores de C++, C# pueden migrar fácilmente por la simplicidad en la sintaxis.
- **Orientado a objetos**, soporta los pilares fundamentales de la POO, como encapsulación, herencia, polimorfismo.
- **Robusto**, permite validar el código tanto en tiempo de compilación como en ejecución, haciendo que las aplicaciones desarrolladas bajo este lenguaje sean confiables, facilitando al programador de los tediosos procesos de liberación de memoria y manejo de punteros.
- **Portable**, gracias a la máquina virtual de java, se puede tener aplicaciones independientes de la plataforma.
- **Interpretado**, el proceso de compilación e interpretación puede realizarse en una sola pasada y en tiempo real.
- **Multitarea**, permite sincronizar varios hilos de ejecución al mismo tiempo.
- **Distribuido**, a través del API de Java se tiene acceso a clases específicas, que permiten al programador desarrollar aplicaciones distribuidas de manera eficiente.
- **Difundido**, por ser un lenguaje muy conocido, ya que existen muchas comunidades, blogs, grupos de noticias que publican diariamente sus experiencias con este lenguaje en internet.

### 1.3 VERSIONES

Existen varias versiones del lenguaje de programación java desde la versión 1 creada en 1991 (conocido anteriormente con OAK) hasta la versión 8 creada en el 2014, en cada uno de ellas hay mejoras tanto a la máquina virtual como agregaciones de nuevas clases, las cuales se detalla a continuación:

- **Versión 1.0:** fue creada en 1996, contiene el api AWT, las clases iniciales y la máquina virtual de java, como fue lanzado demasiado pronto tenía muchos errores.
- **Versión 1.1** fue lanzada en 1997, incluye una mejora en la máquina virtual, API JDBC para conexión con base de datos, RMI para invocaciones a métodos remotos.
- **Versión 1.2:** fue lanzada en 1998, agrega el API de *swing* y *collections* para el manejo de estructuras de datos.
- **Versión 1.3:** fue lanzada en el 2000, incluye el nuevo compilador JIT (justo a tiempo) que permite compilar los métodos al ser llamados a código byte, además se añade soporte para JNDI (Interfaz de nombrado y directorios java) y el API *JAVA SOUND*.
- **Versión 1.4:** fue lanzada en 2002 y ofrece soporte de expresiones regulares, el nuevo API NIO (New I/O) para manejo de entrada / salida, criptografía JCE (*JAVA CRYPTOGRAPHY EXTENSION*), soporte para XML, *SOCKET JSSE* (*JAVA SECURE SOCKET EXTENSION*) y además incluye parte de seguridades JASS (*JAVA AUTENTIFICATION AND AUTHORIZATION SERVICE*).
- **Versión 1.5:** fue lanzada en 2004, permite hacer la conversión de datos primitivos a sus clases envolventes y viceversa (*WRAPPERS*), además del uso de *ENUMS* y bucles simplificados. Fue renombrado a java 5.
- **Versión 1.6:** fue lanzada en 2006, incluye libreras para gestionar servicios web mediante JAX-WS y nuevos drivers para la conexión de base de datos. Además *Sun Microsystem* lo convierte a *OPEN SOURCE* bajo la licencia GPL y en 2008 se lo conoce como *OpenJDK*.
- **Versión 1.7:** fue lanzado en 2011, incluye mejoras en la máquina virtual con respecto a los recolectores de basura y una nueva librería IO para gestión de archivos.

- **Versión 1.8:** esta versión fue lanzada en 2014, se incluyen expresiones lambda en el cual podemos crear métodos sin nombre (anónimos), eliminación del espacio de memoria *PermGem* y la creación del *Metaspace* haciendo el tamaño de la memoria dinámicamente y no permitirá la fuga de memoria, nueva API para *TIME* y *DATE*, soporte para codificación y decodificación Base64.

## 1.4 JDK

*Java Development Kit* es el entorno de desarrollo de Java, que provee las herramientas necesarias las cuales permiten compilar, ejecutar y desarrollar programas de escritorio basados en este lenguaje. El JDK tiene varias versiones, hasta la actualidad está en la versión 8.

Oracle distribuye de forma gratuita la máquina virtual de java (JDK) para los sistemas operativos Window, Linux o Mac. Los programas más importantes que incluye el JDK son:

- **Javac:** permite compilar una clase de java.
- **Java:** permite ejecutar un programa compilado.
- **Javadoc:** permite crear la documentación de un proyecto de software.
- **Jar:** permite ejecutar un archivo .jar que posea un claspath ejecutable.

Las variables de entorno del JDK son:

- **CLASSPATH:** es el directorio donde están las librerías o las clases del usuario.
- **PATH:** es la ruta del directorio donde está ubicado el JDK, lo utiliza el sistema operativo para buscar los ejecutables desde la línea de comandos.

## 1.5 JVM

La Máquina virtual de java es la encargada de ejecutar los programas basados en Java, y su objetivo principal es garantizar la portabilidad de las aplicaciones, es decir, que si un programa fue desarrollado y ejecutado en Windows deberá ejecutarse de la misma manera en Linux o Mac. Para hacer las aplicaciones Java

portables, se crea archivos con instrucciones específicas en *bytecodes*, el intérprete de java ejecuta estas instrucciones que se guardan en los archivos .class.

Las tareas principales del JVM son:

- Crear objetos y los asigna a la memoria.
- Limpia los objetos que ya no están siendo utilizados.
- Asignar variables registros y pilas.
- No permite realizar conversiones entre datos que sean de distinta naturaleza.
- En arreglos, listas y colecciones los punteros son gestionados directamente por la JVM.
- Gestiona de forma eficiente la memoria.

Del concepto expuesto, en la siguiente figura se ilustra el proceso de ejecución de un programa Java desde el ingreso del código fuente en un archivo .java, la compilación de los archivos por medio del comando **javac** y la ejecución de la clase ejecutada por medio del comando **java**, el JVM cumple la función de ejecutar una aplicación Java en cualquier sistema operativo a través de sus intérpretes.



Figura 1 Proceso de ejecución de una aplicación Java

#### Recomendación

El código hecho en Java es compilado por el comando **javac** para ser transformado en un archivo **.class** y estos archivos pueden ser ejecutados en cualquier sistema operativo por medio del intérprete de Java.

## 1.6 JRE

*Java Runtime Environment* es el que permite ejecutar las aplicaciones Java previamente compilados mediante el uso de librerías y utilidades, está conformada por la Máquina virtual de java (JVM) y las librerías estándar (API).

## 1.7. INSTALACIÓN Y CONFIGURACIÓN DE JAVA

El primer paso que se debe realizar previo al proceso de instalación es descargar la última versión del JDK de Java en la página oficial de Oracle, se acepta la licencia y se selecciona el JDK en el sistema operativo que se vaya a trabajar.

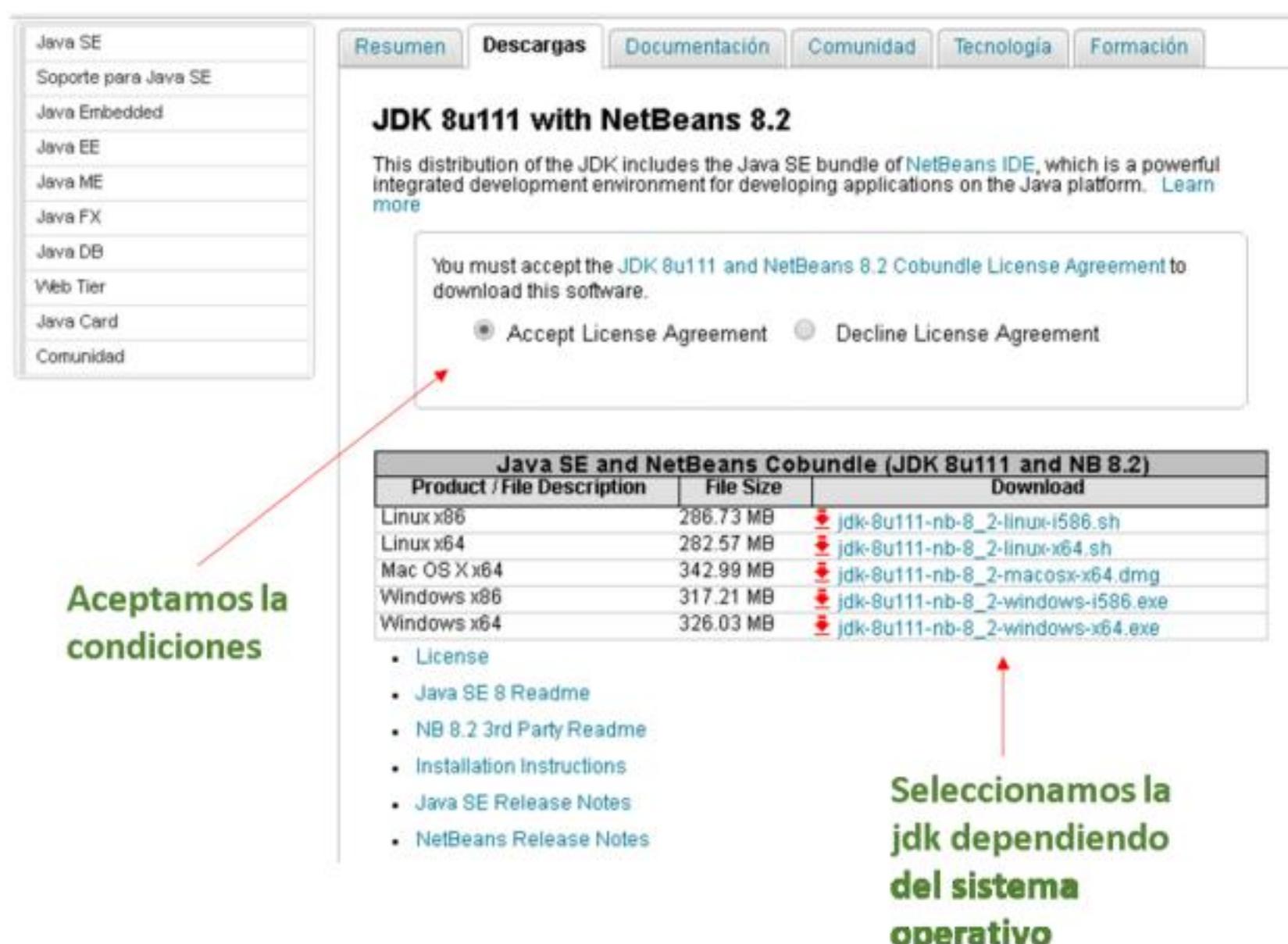
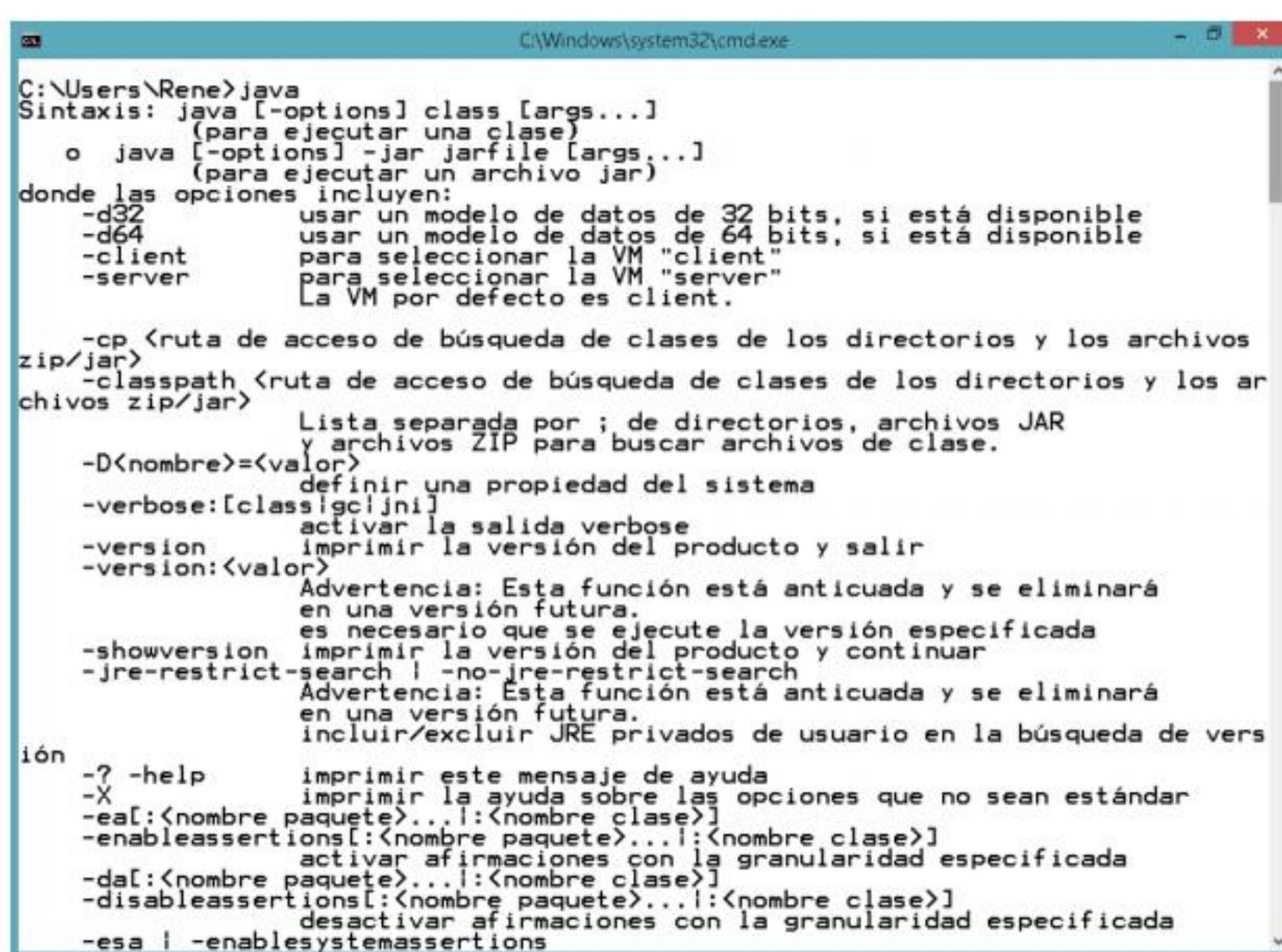


Figura 2 Descarga del SDK

Luego de haber instalado el JDK se procede a configurar las variables de entorno, estas variables son muy importantes ya que le informamos al Sistema operativo en donde está instalado el JDK.

### 1.7.1 WINDOWS

El proceso de instalación del JDK de Java es el de cualquier aplicación de escritorio. Pero es indispensable comprobar si la instalación fue la correcta o no. El primer paso es abrir la terminal de comando y ejecutar el comando **java**, si imprime el mensaje “*No se encuentra el comando*” significa que el SDK no se instaló correctamente y es necesario reinstalar. Si la instalación ha sido correcta imprime información del comando **java** como se ilustra en la siguiente figura:



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The command entered is 'java'. The output is the Java command-line interface help text, which includes options for running Java programs and Java Virtual Machine (JVM) settings. Key sections include:

- Sintaxis:** java [-options] class [args...]  
o java [-options] -jar jarfile [args...]
- donde las opciones incluyen:**
  - d32 usar un modelo de datos de 32 bits, si está disponible
  - d64 usar un modelo de datos de 64 bits, si está disponible
  - client para seleccionar la VM "client"
  - server para seleccionar la VM "server"  
La VM por defecto es client.
- zip/jar**
  - cp <ruta de acceso de búsqueda de clases de los directorios y los archivos zip/jar> Lista separada por ; de directorios, archivos JAR y archivos ZIP para buscar archivos de clase.
  - D<nombre>=<valor> definir una propiedad del sistema
  - verbose:[class|gc|jni] activar la salida verbose
  - version imprimir la versión del producto y salir
  - version:<valor> Advertencia: Esta función está anticuada y se eliminará en una versión futura.  
es necesario que se ejecute la versión especificada
  - showversion imprimir la versión del producto y continuar
  - jre-restrict-search | -no-jre-restrict-search Advertencia: Esta función está anticuada y se eliminará en una versión futura.  
incluir/excluir JRE privados de usuario en la búsqueda de versión
- ión**
  - ? -help imprimir este mensaje de ayuda
  - X imprimir la ayuda sobre las opciones que no sean estándar
  - ea[:<nombre paquete>...]:<nombre clase> activar afirmaciones con la granularidad especificada
  - enableassertions[:<nombre paquete>...]:<nombre clase> activar afirmaciones con la granularidad especificada
  - da[:<nombre paquete>...]:<nombre clase> desactivar afirmaciones con la granularidad especificada
  - disableassertions[:<nombre paquete>...]:<nombre clase> desactivar afirmaciones con la granularidad especificada
  - esa | -enablesystemassertions

Figura 3 Instalación correcta del JDK

Instalado el JDK, el siguiente paso es configurar la variable de entorno, esta variable es muy importante porque se informa al sistema operativo la ruta donde está instalado el JDK y se la denomina JAVA\_HOME.

Para configurar la variable de entorno se debe seguir los siguientes pasos:

1. Ir a propiedades del sistema en MiPc.
2. Seleccionar la opción “**Configuración avanzada del sistema**”.

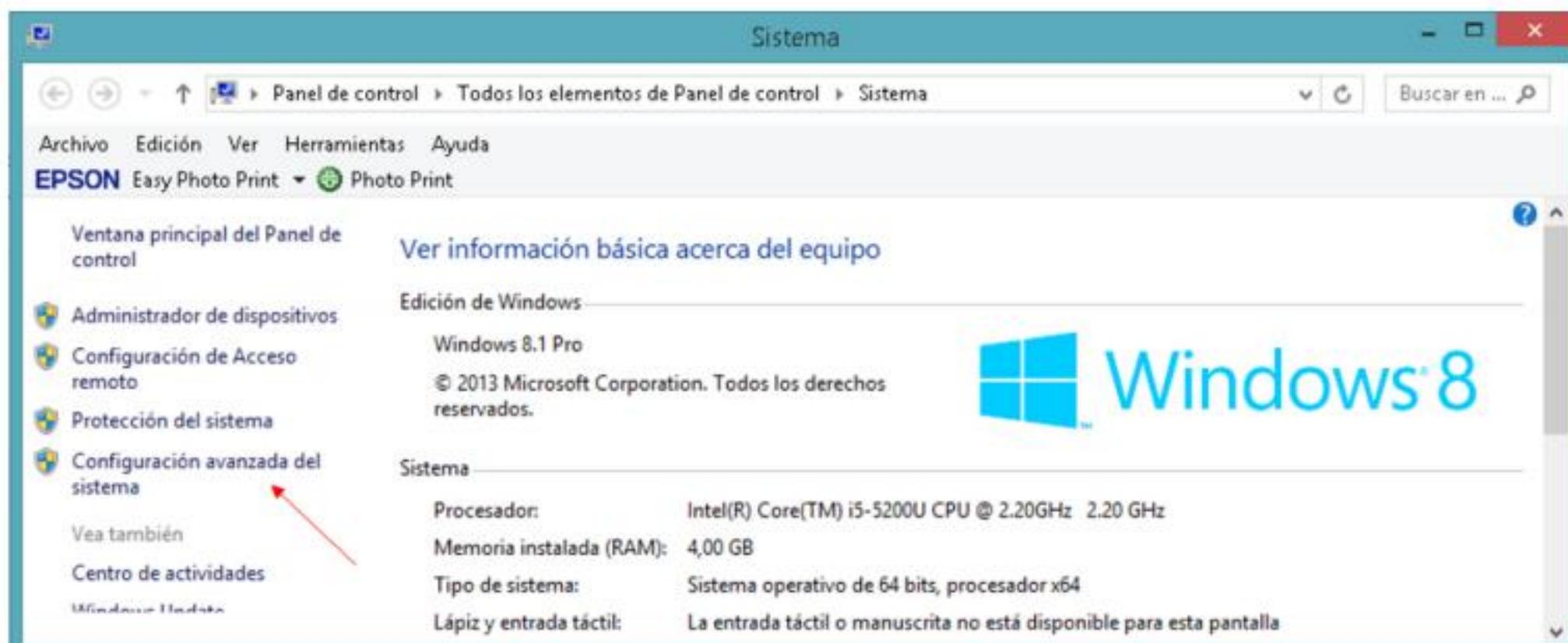


Figura 4. Propiedades de mpc

### 3. Seleccionar la opción “**variables de entorno**”.

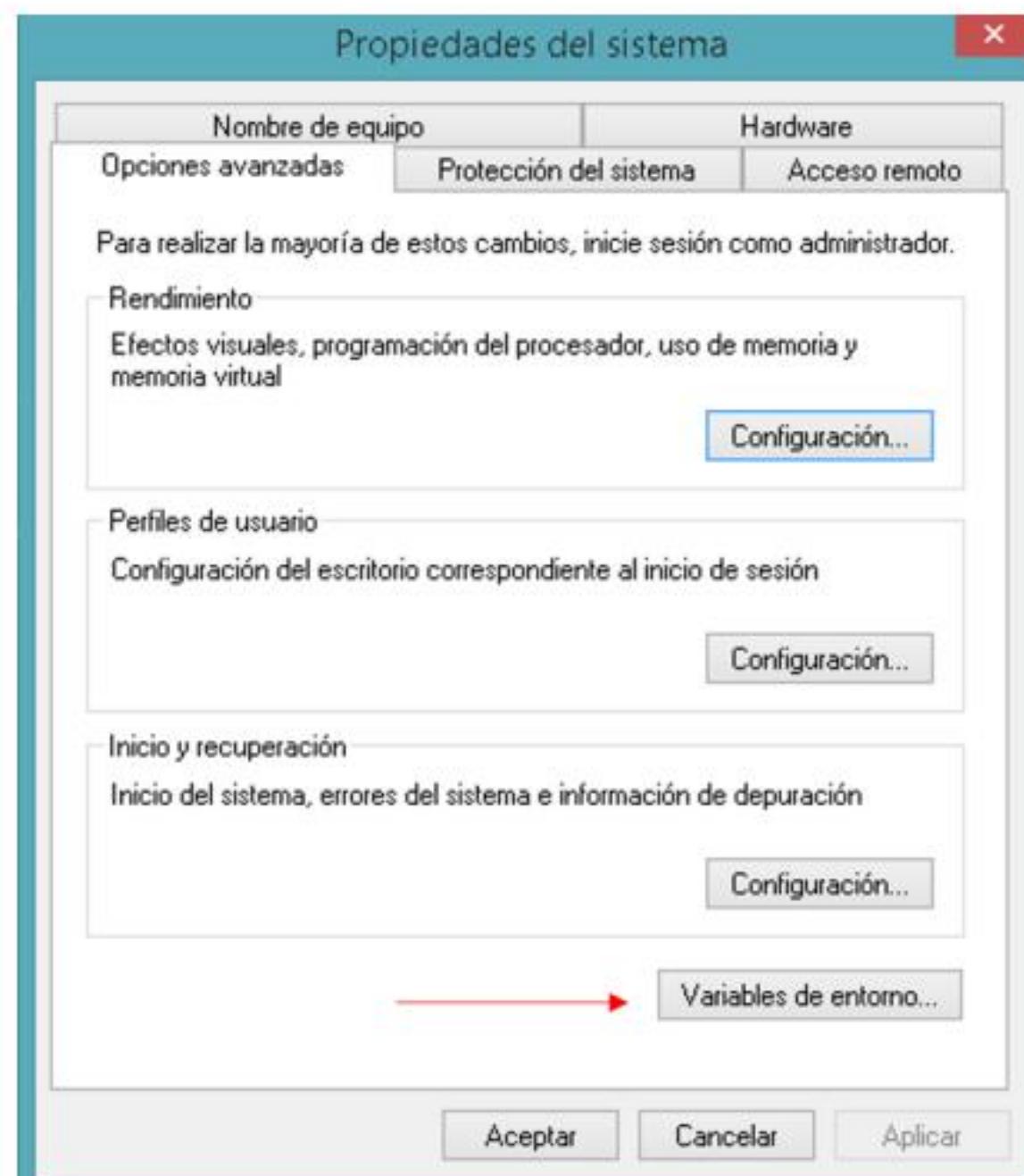


Figura 5 Propiedades del sistema - variables de entorno

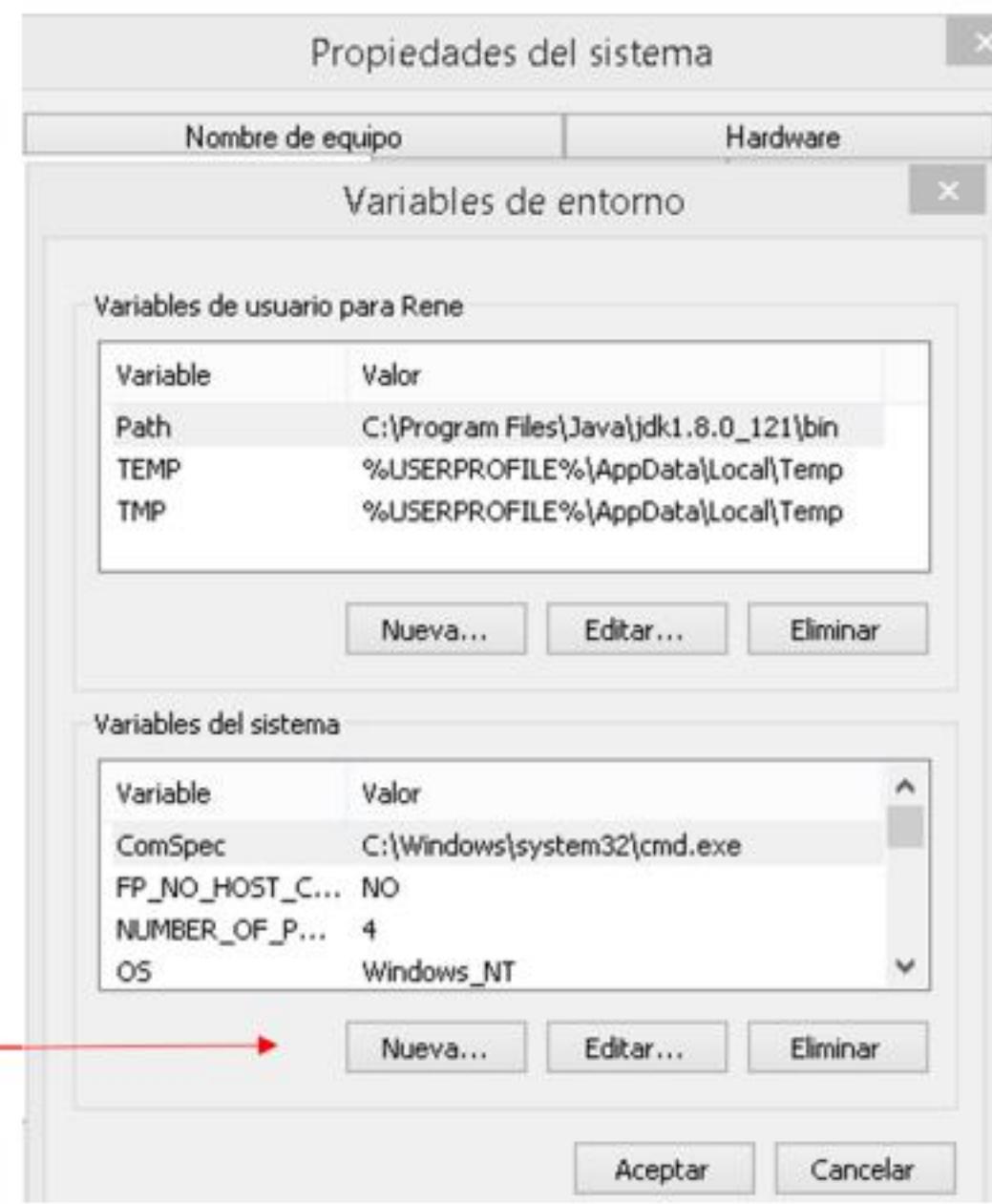


Figura 6 Variable de entorno

4. Crear un nueva variable de entorno en variables del sistema llamada JAVA\_HOME, se ubica la dirección del directorio donde está instalado el JDK. En Windows si la instalación fue por defecto, el directorio donde se instala el SDK es C:\Program Files\Java.

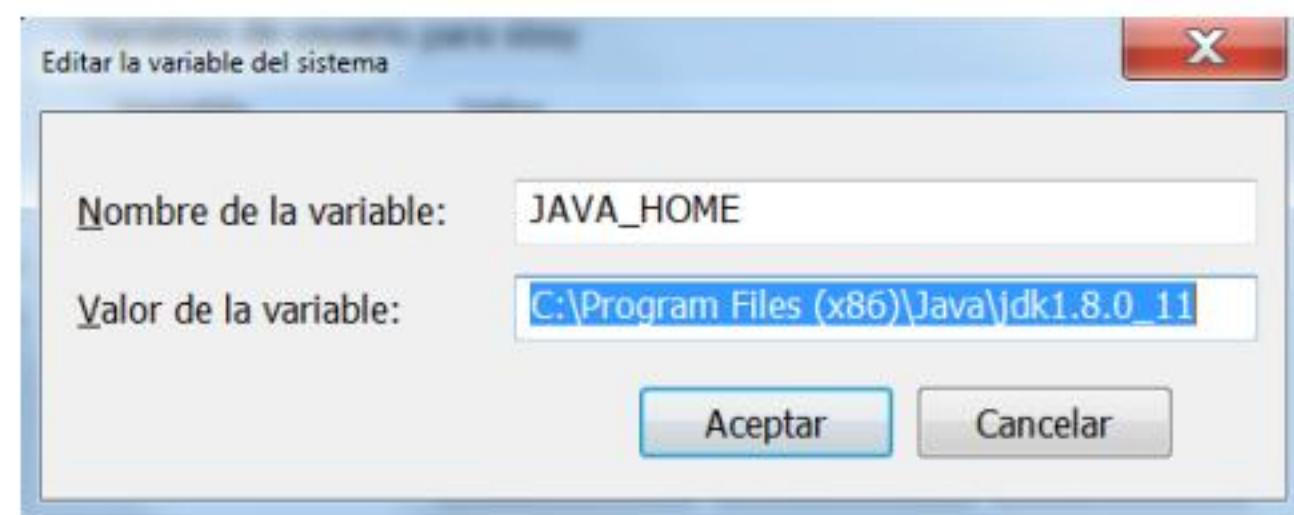


Figura 7 Variable de Sistema

5. Verificar si se configuró correctamente el SDK por medio de la terminal de comandos. Al digitar el comando `javac`, imprime el mensaje "**No se encuentra el comando**" significa que el JAVA\_HOME está mal direccionado y hay revisar si la dirección del SDK es la correcta.

Si la configuración ha sido correcta imprime información del `javac` como se ilustra en la siguiente figura:

The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. It displays the usage information for the Java compiler (javac). The output is as follows:

```
C:\Users\Rene>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                     Generate no debugging info
  -g:{lines,vars,source}       Generate only some debugging info
  -nowarn                     Generate no warnings
  -verbose                    Output messages about what the compiler is doing
  -deprecation                Output source locations where deprecated APIs are u
  sed
    -classpath <path>          Specify where to find user class files and annotati
  on processors
    -cp <path>                 Specify where to find user class files and annotati
  on processors
    -sourcepath <path>          Specify where to find input source files
    -bootclasspath <path>       Override location of bootstrap class files
    -extdirs <dirs>             Override location of installed extensions
    -endorseddirs <dirs>        Override location of endorsed standards path
    -proc:{none,only}           Control whether annotation processing and/or compil
  ation is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors
  to run; bypasses default discovery process
  -processorpath <path>        Specify where to find annotation processors
  -parameters                 Generate metadata for reflection on method paramete
  rs
  -d <directory>              Specify where to place generated class files
  -s <directory>              Specify where to place generated source files
  -h <directory>              Specify where to place generated native header file
  -s
    -implicit:{none,class}    Specify whether or not to generate class files for
  implicitly referenced files
    -encoding <encoding>       Specify character encoding used by source files
    -source <release>          Provide source compatibility with specified release
    -target <release>          Generate class files for specific VM version
    -profile <profile>         Check that API used is available in the specified p
  rofile
    -version                  Version information
    -help                     Print a synopsis of standard options
    -Akey[=value]              Options to pass to annotation processors
    -X                        Print a synopsis of nonstandard options
    -J<flag>                 Pass <flag> directly to the runtime system
    -Werror                   Terminate compilation if warnings occur
```

Figura 8 Configuración correcta de JDK

### 1.7.2 LINUX

Existen diferentes sistemas operativos bajo la distribución de GNU-LINUX, en este caso se utilizará el sistema operativo Ubuntu 16.04.

La instalación se hará utilizando la consola de comandos:

-Para instalar JSK en debían o Ubuntu se ejecuta el comando:

**apt-get install opendk-<versión>-jre**

-En fedora, Centos, Oracle el comando es:

**yum install java-<version>-opendk**

Previo a la instalación en Ubuntu se debe agregar y actualizar el repositorio de Oracle:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
```

La última versión estable de Java por el momento es el JDK 8:

```
$ sudo apt-get install oracle-java8-installer
```

Para instalar el JDK 9 se utiliza el siguiente comando:

```
$ sudo apt-get install oracle-java9-installer
```

Instalado el SDK de java, se debe configurar la variable de entorno JAVA\_HOME. El primer paso es averiguar dónde se encuentra instalado:

```
$ sudo update-alternatives --config java
```

```
~$ sudo update-alternatives --config java
sudo: unable to resolve host      Inspiron-3443
There is 1 choice for the alternative java (providing /usr/bin/java).

  Selection    Path                      Priority   Status
-----* 0           /usr/lib/jvm/java-8-oracle/jre/bin/java  1081     auto mode
* 1           /usr/lib/jvm/java-8-oracle/jre/bin/java  1081     manual mode

Press <enter> to keep the current choice[*], or type selection number: ■
```

Figura 9 sudo update-alternatives

Copiar la ruta y abrir el archivo *enviroment* con cualquier editor de texto:

```
$ sudo nano /etc/environment
```

Al final del documento se agrega la ruta del JDK:

```
JAVA_HOME="/usr/lib/jvm/java-8-oracle"
```

Guardado el documento, es necesario cargarlo de nuevo mediante el comando:

```
$ source /etc/environment
```

Para comprobar si la configuración es la correcta, se utiliza el comando [javac](#) en la consola de comandos descrito en la configuración en el sistema operativo Window.

## **2 FUNDAMENTOS DE LA PROGRAMACIÓN DE JAVA**

---

En este capítulo se describe el concepto de “clase”, su sintaxis y a través de ejemplos se define a los atributos y métodos. Se explica el comportamiento de los comentarios en el código. Se listan las palabras reservadas en el lenguaje Java. Además se indica el funcionamiento del método *main*, su compilación y su ejecución. Por último se explica algunos errores clásicos cometido por los desarrolladores principiantes.

### **Objetivos:**

- Reconocer las palabras reservadas del lenguaje Java.
- Utilizar los tipos de comentarios para documentar el código.
- Implementar el método *main* para ejecutar aplicaciones de escritorio.
- Evaluar los mensajes de error generados al compilar archivos .java.

### **2.1 INTRODUCCIÓN A LAS CLASES**

En la programación orientada a objeto, las clases son el punto neuronal de las aplicaciones de software porque permiten representar de forma abstracta cómo estará formado el problema que se pretende resolver. Por lo tanto una clase es una plantilla que permite establecer el esqueleto de las entidades que interactuarán en el sistema.

En Java se emplea la palabra reservada **class** para crear una clase. Por ejemplo, suponiendo que existe una clase llamada Computadora y se quiere representar en código, se define a continuación:

*Ejemplo 2 Definición de una clase básica*

```
1 | public class Computadora {  
2 | }  
3 | }
```

En la línea 1, se emplea la palabra reservada **public** y es un modificador de acceso que establece que la clase podrá ser utilizada por otras clases. La palabra Computadora es el nombre o identificador de la clase definida por el programador. Las llaves abierta “{” y cerrada “}” establece el inicio y el final de la clase.

**Recomendación**

Se recomienda que al definir una clase, se establezca el inicio y final de la clase a través de la llave abierta y cerrada {}, los programadores que están aprendiendo este lenguaje de programación, suelen olvidar cerrar la clase, esto ayudará a evitar errores al momento de compilar sus aplicaciones.

En Java, las clases se componen de dos partes fundamentales que son los atributos y métodos. Cuando definimos una clase, establecemos las características y acciones que tendrán los objetos y en un sentido más amplio permiten definir:

- **Sus atributos:** características que tendrán las clases y estarán definidos por los modificadores de accesibilidad, en otros lenguajes se las conoce como variables globales.
- **Sus métodos:** establecen las acciones que pueden realizar las clases, en otros lenguajes se llaman funciones o procedimientos.

Por ejemplo, piense en una computadora, el cerebro representa este conocimiento a través de una imagen y es la primera representación abstracta del conocimiento como se ilustra en la siguiente imagen. Cualquier persona al graficar una computadora puede definir y establecer características como su marca, precio, color, capacidad de disco, memoria, si es de escritorio o portátil etc.; y funcionalidades como procesar, calcular, imprimir, etc. Pero el programador debe representar este conocimiento en un diagrama de datos que posteriormente se pueda traducir en un lenguaje de programación.

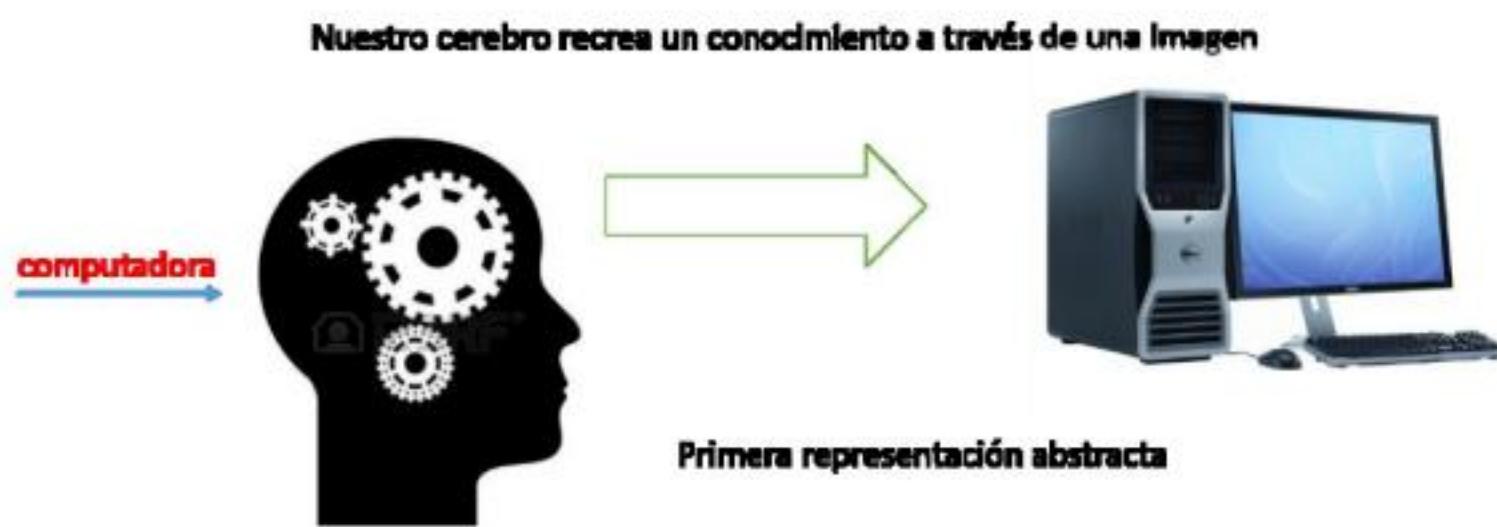


Figura 10 Representación abstracta del conocimiento

Para representar un conocimiento abstracto se utiliza diagramas en UML.

UML es un lenguaje unificado de modelado y es un estándar que sirve para crear esquemas, diagramas y documentación relativa a los desarrollos de software. Desde otro enfoque, UML permite diagramar los planos de las aplicaciones informáticas. En este caso se utilizará el diagrama de clases para representar las entidades que interactuarán en el problema y posteriormente implementarla a través del lenguaje de programación Java.

Obtenida la primera representación abstracta de la computadora, se representará a la clase Computadora en UML:

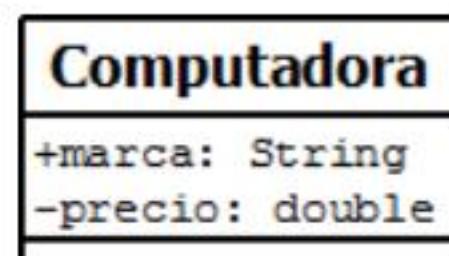


Figura 11 Representación de la clase Computadora en UML

En la clase Computadora se ha definido dos atributos, la marca y el precio. Al traducir del diagrama UML al lenguaje Java se genera el siguiente código:

Ejemplo 3 Definición de una clase con dos atributos

```

1 | public class Computadora {
2 |   public String marca;
3 |   private double precio;
4 |
  
```

En este ejemplo, en la línea 2 y 3, se establecen los atributos de la clase Computadora, para definir un atributo se ubica el tipo de dato y el identificador, el primero es de tipo *String* (es una cadena) y el segundo es de tipo *double* (decimal). Sus identificadores son marca y precio; y son establecidos por el programador.

Ahora a la clase Computadora, se la definirá un método llamado *prender()*:

*Ejemplo 4 Definición de una clase con un método*

```
1 public class Computadora {  
2     String marca;  
3     double precio;  
4     public void prender(){  
5         // bloque de código del método  
6     }  
7 }
```

En este ejemplo, en la línea 5 la palabra prender es el identificador definido por el programador, la palabra reservada **public** establece que es de carácter público y podrá ser llamado en cualquier clase, **void** expresa que el método no devolverá ningún tipo de datos. Tanto las clases como los métodos deben definir sus límites, su inicio y fin. Para ambos casos las llaves abierta y cerrada “{ }” establecen esos límites.

Las clases una vez al ser declaradas e implementadas, se guardan con la extensión .java y el nombre del archivo debe ser exactamente igual al nombre de la clase. En el ejemplo anterior, la clase Computadora debe ser guardada con el nombre Computadora.java.

**Recomendación**

Por convención todos los nombres de las clases se deben escribir como sustantivo propio, la primera letra debe ser escrita con mayúscula y cuando el nombre de una clase está conformada por más de dos palabras, la primera letra de cada palabra debe ser escrita con mayúsculas. Algunos ejemplos: EmpleadoGobierno, BancoPrivado, etc.

Es importante recordar que los lenguajes de programación establecen sus propias reglas para formar los programas, definen sintaxis que los programadores deben respetar. La sintaxis permiten crear elementos en las clases, tales como:

- **Identificadores:** los nombres que se dan a las clases, interfaces, clases abstractas, datos de tipo enumerado, arreglos, métodos, variables, etc.
- **Tipos de datos:** primitivos, enumerados o por referencia.
- **Palabras reservadas:** las palabras que utiliza el propio lenguaje y que no se pueden utilizar como identificadores.
- **Sentencias:** secuenciales, selectivas y repetitivas.

- **Bloques de código:** métodos.
- **Comentarios:** de una sola o múltiples líneas.
- **Expresiones.**
- **Operadores.**

Java es un lenguaje de programación muy riguroso en su sintaxis pero en el tema de indentación es pobre, porque se puede escribir el código en una sola línea y puede ser funcional. También el código puede ser escrito en la misma jerarquía, en otras palabras sin respetar los espacios para distinguir que bloques de código pertenece a la clase, a los métodos, o estructuras de control. Esto hace que el código no sea legible.

El siguiente ejemplo, se ilustra la implementación de la clase Computadora definida en el ejemplo anterior, no tiene errores de compilación, pero no respeta las reglas de indentación, haciendo que el código no sea legible, no permitiendo la lectura del código, para realizar modificaciones o mantenimiento del mismo.

*Ejemplo 5 Clase Computadora sin reglas de identación*

```

1  public class Computadora {
2    String marca;
3    double precio;
4
5    public void prender(){
6      // bloque de código del método
7    }
8 }
```

Ahora, el siguiente ejemplo no respeta los saltos de línea ni la indentación de código pero no tiene errores de compilación, así que el programa podrá ser funcional pero no será legible para su mantenimiento. En Java el separador de instrucciones es “;”, así que un programa puede ejecutarse en una sola línea de código.

*Ejemplo 6 Definición de una clase sin identación y sin saltos de línea*

```

1  public class Computadora { String marca;double precio; public void prender(){
2    // bloque de código del método
3  }
4 }
```

Para resolver este inconveniente, se debe emplear cuatro espacios como unidad de indentación. La construcción exacta de la indentación (espacios en blanco contra

tabuladores) no se especifica. Los tabuladores deben ser exactamente cada 4 espacios.

## 2.2 PALABRAS RESERVADAS

Para definir clases, atributos, métodos, tipos de datos, estructuras de control, etc., es imprescindible utilizar palabras reservadas. En Java son identificadores predefinidos que tienen un significado para el compilador, como su nombre indica, estas palabras están reservadas por el lenguaje de programación, y no se pueden usar como identificadores de usuario.

En la siguiente tabla se lista algunas palabras reservadas declaradas en Java:

Tabla 1 Palabras reservadas

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

Las palabras reservadas se pueden clasificar en las siguientes categorías:

- **Tipos de datos primitivos:** boolean, float, double, int, long, byte, short, char.
- **Valores booleanos:** true, false.
- **Sentencias condicionales:** if, else, switch.
- **Sentencias iterativas:** for, do, while, continue.
- **Tratamiento de las excepciones:** try, catch, finally, throw.
- **Estructura de datos:** class, interface, implements, extends.
- **Modificadores y control de acceso:** public, private, protected.

### Recomendación

Hay que tener en cuenta que todas las palabras reservadas se han definido en minúsculas.

## 2.3 COMENTARIOS

Son mecanismos en que se apoyan los lenguajes de programación para documentar las aplicaciones y son utilizados por los programadores para explicar cómo funciona el código. En Java, los comentarios pueden ser líneas de código no ejecutable o texto definido por el programador, en otras palabras el compilador no las procesa y no están sujetos a ninguna sintaxis del lenguaje. Hay tres tipos de comentarios:

- **Comentarios de una sola línea:** comienza con doble slash “//” y pueden ser colocados en cualquier parte de nuestro código y ocupan una sola línea.
- **Comentarios de múltiples líneas:** cuando el comentario ocupan más de una línea deben comenzar con “/\*”y terminar con “\*/”.
- **Comentarios de documentación con javadoc:** para la documentación del programa se necesita comentarios especiales, por ejemplo @author, @method, etc. Para definir el bloque del comentario se utilizará /\*\* para iniciar el comentario y \*/ para terminar el comentario.

En el siguiente ejemplo se emplea el concepto de comentario de una sola línea:

*Ejemplo 7 Comentario de una sola línea*

```
1 public class Animal {  
2     // se definirán los atributos de la clase  
3     String nombre; //atributo, define el nombre del animal  
4 }
```

En este ejemplo, existen dos comentarios definidos en la líneas 2 y 3, ambos son comentarios de una sola línea porque utiliza el doble slash //. En la línea 3 el comentario se ubica al final de la instrucción y se lo utiliza para explicar el significado de la instrucción. Java no toma en cuenta a los comentarios al momento de compilarlos o ejecutarlo, por esa razón, se los puede ubicar al inicio o final de la instrucción.

El siguiente tipo de comentario es el de múltiples líneas, y permite comentar un bloque de texto utilizando “/\* .... \*/”. En el siguiente ejemplo se ilustra su utilización:

*Ejemplo 8 Comentario de múltiples líneas*

```
1  /*
2   * To change this license header, choose License Headers
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  public class Animal {
7      String nombre;
8  }
```

Otra forma de definir comentarios es usando los caracteres especiales de javadoc y sirven para genera la documentación del programa. En la siguiente tabla se explica algunos tab usados en javadoc:

*Tabla 2 Tag para documentar clases con JavaDoc*

Tag	Descripción
@author	Indica el nombre del desarrollador.
@deprecated	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores.
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.
@return	Informa de lo que devuelve el método, no se puede usar en constructores o métodos "void".
@see	Asocia con otro método o clase.
@throws	Excepción lanzada por el método.
@version	Versión del método o clase.

## 2.4 MÉTODO Main

Un proyecto de software o programa está compuesto por un conjunto de clases que interactúan para prestar servicios a través de sus métodos. De toda esta colección de clases debe existir una que empiece con la ejecución del programa. Java ha definido un método especial que inicie con la ejecución del proyecto llamado "*main*". La clase que implemente este método se la denomina clase Ejecutora. Se define su sintaxis en el siguiente ejemplo:

```
1  public static void main (String [] args){
2      //código
3  }
```

En la línea 1 se define la cabecera del método *main* y se compone de los siguientes elementos:

- **public**: todo método tiene un modificador de acceso que puede ser público, protegido, privado o por defecto; esto depende del nivel de accesibilidad que se defina. Esta palabra reservada especifica que el método puede ser accedido desde cualquier otro método o clase. El método *main* siempre será declarado con modificador de acceso **public**.
- **static**: establece que el método podrá ser llamado sin la necesidad de una instancia u objeto. Un método estático es un método de clase que se lo llama directamente con el nombre de la clase. El método *main* se ejecuta automáticamente al convertirse en el ejecutor del programa.
- **void**: los métodos tienen dos comportamientos, el primero devuelve un valor y está obligado a utilizar la sentencia **return**, el segundo no devuelve ningún valor pero ejecuta todas las instrucciones. La palabra reservada **void** establece que no se devolverá ningún valor y solo ejecutará las instrucciones declaradas en el método.
- **main**: todo método debe definir un nombre, pero la máquina virtual de java busca en sus clases un método llamado *main*, este identificador le indica a Java que será el ejecutor del programa.
- **String**: es un tipo de dato que pertenece a la Clase String que permite manejar cadenas y proviene del paquete java.lang.
- **String [ ]**: los corchetes representan una estructura de datos lineal estática, quiere decir que es un arreglo de objetos. Este arreglo permite recibir parámetros al ejecutar la aplicación.

Además hay que diferenciar entre un método de instancia y uno de clase, el primero es el que se invoca siempre sobre una instancia, es decir, un objeto de una clase. Por ejemplo:

```
1 | Numero n = new Numero0;  
2 | n.sumar0;
```

Siendo n un objeto de la clase Numero quien es el encargado de invocar al método *sumar( )*. El segundo es el método de clase, es aquel que puede ser invocado sin existir una instancia, por ejemplo los métodos estáticos.

En este caso, para utilizar el método *main* no se necesita crear una instancia. El siguiente ejercicio se codifica la forma de implementar este método:

*Ejemplo 9 Método main*

```
1 public class Computadora {  
2     public static void main (String [] args){  
3         System.out.println("Computador digital");  
4     }  
5 }  
6 }
```

En la línea 2, el método inicia con el modificador de acceso **public**, además es estático, por lo que no podrá utilizar atributos o métodos dinámicos de la clase. Los métodos pueden o no devolver un dato, en este caso, el método *main* no retorna ningún tipo de datos al ser definido como **void** y su función será ejecutar las instrucciones que se encuentren en él.

Se definen solo un parámetro, es un *array* de caracteres, cada elemento del *array* es un parámetro enviado por el usuario desde la línea de comandos. A este argumento se le llama comúnmente “args”. El arreglo de caracteres “String [] args” es importante para el método.

Al igual que las clases, los métodos deben definir el inicio y final de cada uno de ellos a través de las llaves “{ }”.

En la línea 3, la instrucción `System.out.println()`, es la que permitirá imprimir por consola un mensaje “*Computador digital*”.

## 2.5 PRIMER PROGRAMA

Ahora, el primer paso para crear la clase *Computadora* es editar un documento de texto y nombrarlo como “*Computadora.java*”, hay que tener cuidado al crear el documento porque si el nombre del archivo es diferente al nombre de la clase, generará un error. En Java el nombre del archivo.java y de la clase deben ser idénticos.

En la siguiente figura se ilustra la relación entre el nombre de archivo y la clase.



Computadora.java

```
1 public class Computadora {  
2     public static void main (String [] args){  
3         System.out.println("Computador digital");  
4     }  
5 }
```

Figura 12 Primera Clase

Una vez codificada la clase y el método *main*, se debe realizar el proceso de compilación. Esto permite verificar que la sintaxis del documento en Java es la correcta. El comando para realizar este proceso es **javac**. La sintaxis para compilar una clase es:

```
$ javac NombreArchivo.java
```

Compilemos la clase Computadora con la siguiente instrucción:

```
1 | $ javac Computadora.java
```

Si la sintaxis definida en la clase Computadora es correcta, no mostrará mensajes de error:

```
1 programas_libro$ javac Computadora.java  
2 programas_libro$
```

Si en el proceso de compilación no se detectaron errores, el SDK de java genera un documento “.class” con el nombre de la clase. Este archivo contendrá el *bytecode*, y consiste en instrucciones que la JVM conoce y puede ejecutar. El archivo generado es “Computadora.class”, como se ilustra en la siguiente figura:



Figura 13 Archivos Computadora.java y Computadora.class

Se compila la clase y generado el archivo .class, se debe ejecutar la clase que contenga el método *main*, la sintaxis es:

```
$ java NombreClase
```

Se procede a ejecutar la clase Computadora:

```
1 | $ java Computadora
```

Java interpreta y ejecuta las instrucciones que se encuentran en el método *main*, y el resultado generado es la impresión por consola del mensaje “*Computación digital*”:

```
1 programas_libro$ java Computadora  
2 Computador digital
```

## 2.6 ERRORES COMUNES AL COMPIALAR UNA CLASE

Al iniciar a programar en cualquier lenguaje de programación se está a expensas de cometer errores en su sintaxis. Por ello no nos debe causar asombro cuando esto sucede, los errores de compilación son parte del aprendizaje del programador. El primer paso para aprender de los errores es conocer cómo se estructuran, su sintaxis es la siguiente:

NombreArchivo.java : línea\_del\_error : descripción\_del\_error  
                               ubicación\_del\_error

- **NombreArchivo.java:** indica en que archivo se ha generado el error.
- **Número de línea:** indica que línea se ha generado el error.
- **Descriptor:** hace una descripción del tipo de error generado.
- **Ubicación:** indica en que parte del código se ha generado el error.

En la siguiente figura se ilustra un ejemplo típico de un error de compilación:

```
\programas_libro>javac Computadora.java  
Computadora.java:8: error: ';' expected  
    System.out.println("Computador dígital")  
                                                                                ^  
1 error
```

Figura 14 Error de compilación

Si se analiza el mensaje que se genera al compilar una clase en java, en la primera parte indica que se encuentra en la clase Computadora.java, en la línea 8 y la descripción indica que se ha omitido “;”, la ubicación indica donde se ha generado el error, en este caso es al final de la instrucción System.out.println(“Computadora digital”).

Se lista algunos errores más comunes en los desarrolladores en java:

- **Una Clase no está en el directorio correcto.**

```
1 javac: file not found: Computadora.java
2 Usage: javac <options> <source files>
3 use -help for a list of possible options
```

- **No se ha cerrado correctamente un método.**

```
1 Computadora.java:27: error: reached end of file while parsing
2 }
3 ^
4 1 error
```

- **"string" No se encuentra bien definida una clase del API de java.**

```
1 Computadora.java:7: error: cannot find symbol
2 public static void main(string [] args)
3 ^
4 symbol: class string
5 location: class Computadora
6 1 error
```

- **El nombre de la Clase pública no coincide con el nombre del archivo.**

```
1 Computadora.java:1: error: class ComputadorA is public, should be declared in a file named
2 ComputadorA.java
3 public class ComputadorA {
4 ^
5 1 error
```

Los errores más comunes que comenten los desarrollados son:

- No colocar el punto y coma al final de una sentencia.
- No cerrar con llaves en alguna clase, método o estructura de control.
- Definir variables con el mismo nombre.
- Asignar valores diferentes en una variable, en Java solo se debe asignar un valor que corresponda al tipo de dato defino.
- Nombrar a los archivos .java con diferentes nombres al que se ha definido en las clases.

## 2.7 PAQUETES E IMPORTACIONES

El API de Java contiene miles de clases que permiten crear interfaces gráficas, conexiones a bases de datos, manejo de archivos, etc. Todas las clases deben estar organizadas de modo que los programadores puedan ubicarlas y hacer uso de los mismos, es por ello que los paquetes son mecanismos que permite organizar las

clases en grupos, considerando que estas deben tener afinidad y tienen que estar relacionadas.

Una aplicación en Java se compone de una colección de clases más o menos numerosa, se recomienda que se las divida por su función o afinidad en paquetes. Los paquetes también permiten resolver el conflicto de nombres que se puedan dar. En UML se representan gráficamente a través de carpetas, en la siguiente figura se ilustra dos paquetes:

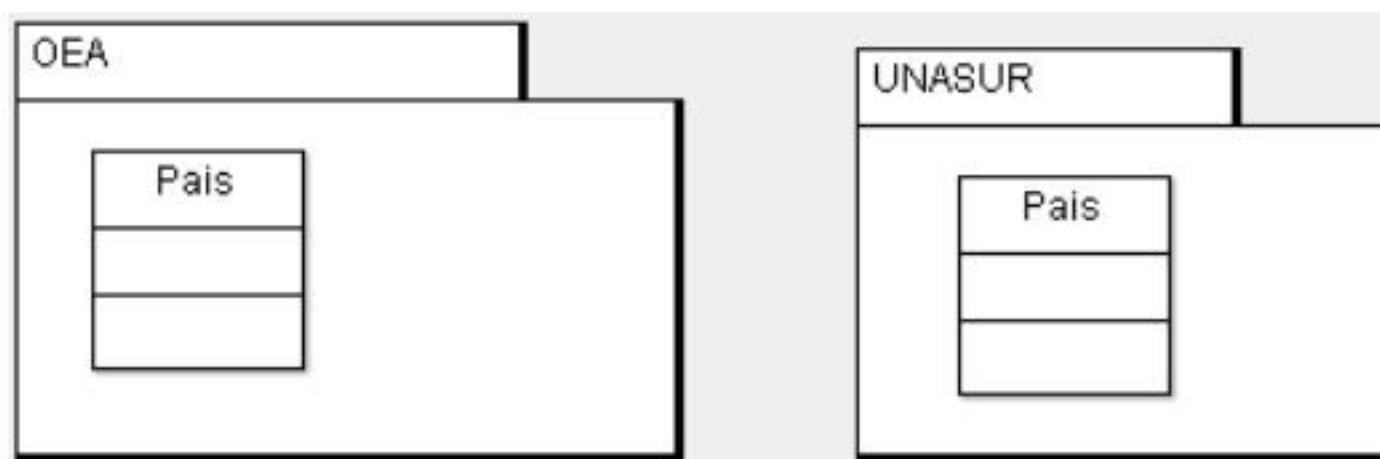


Figura 15 Representación de paquetes

En esta figura, se ilustra dos paquetes llamados OEA y UNASUR, cada una almacena un clase llamada País, si no existieran los paquetes, habrían dos clases llamadas por el mismo nombre y ocasionarían problemas al compilar un programa en Java, el compilador identificará que existe dos clases con el mismo nombre, lo que generaría un error de compilación y obviamente nunca se podría tener dos archivos identificados con el mismo nombre en un mismo directorio.

Los paquetes resuelven este problema, cada clase se la ubica en un paquete y Java las trata como clases diferentes porque están ubicadas en distintos paquetes, cada una es aislada una de la otra.

Java organiza sus clases e interfaces a través de paquetes, en la siguiente tabla se detalla los más importantes:

Tabla 3 Paquetes principales en java

<b>Paquete</b>	<b>Descripción</b>
java.lang	Es el más utilizado en las aplicaciones y contiene clases como String, Integer, Double, Float, System, Math, etc. Su importación es automática.
java.util	Contiene clases e interfaces de utilidades como Date, Calendar, List, ArrayList, Iterador, etc.
java.io	Contiene clases que permiten las operaciones de entrada y salida como OutputStream, InputStream, FileOutputStream, DataOutputStream, etc.
java.swing	Contiene clases para construir interfaces gráficas.

En las aplicaciones que se constituye, es indispensable que se utilice clases o interfaces que proporcionarán métodos definidas en el API de Java, para utilizar estos métodos es necesario realizar una importación. La palabra clave **import** es la que permite utilizar clases que se ubican en otros paquetes.

Existen dos maneras de importar clases de un paquete:

- **Importar clase por clase:** importar solo las clases que se utilizan en un paquete con .NombreClase. La sintaxis para importar una clase es:

```
import java.paquete.NombreClase1;  
import java.paquete.NombreClase2;
```

- **Todas las clases del paquete:** con el \* se importarán todas clases e interfaces que contenga el paquete. La sintaxis para importar todas las clases de un paquete es:

```
import java.paquete.*;
```

#### **Recomendación**

No se debe importar todas las clases e interfaces de un paquete para no sobrecargar a la clase que las importa, aunque se tenga que invertir más líneas de código es recomendable que se importe sólo las clases de paquete que se utiliza en la clase.

Con las importaciones se abre un abanico de clases que se pueden utilizar, por ejemplo existe el paquete llamado “util”, allí se puede utilizar los métodos de las clases *Scanner* y *Date*, la primera para la lectura por el teclado y la segunda para manipular la hora, fechas, etc.

*Scanner* es una clase que implementa el API de Java para la entrada de datos, tiene algunos métodos que se utilizan para el ingreso de datos desde teclado de acuerdo al tipo de dato. *Scanner* obtiene un flujo de datos y los convierte al tipo deseado de acuerdo al método utilizado.

Para acceder a los datos desde teclado se debe utilizar la clase *System.in*. También la clase *Scanner* puede analizar cadenas para extraer datos primitivos y cadenas

utilizando expresiones regulares. Algunos métodos que se pueden utilizar para la lectura de datos:

Tabla 4 Métodos principales de la clase Scanner

Métodos	Descripción
<code>hasNext()</code>	Método que permite saber si existe otro token de entrada. Devuelve un valor booleano.
<code>next()</code>	Método que devuelve un <code>byte</code> escaneado del token de entrada.
<code>nextByte()</code>	Método que devuelve un <code>byte</code> escaneado del token de entrada.
<code>nextInt()</code>	Método que devuelve un <code>int</code> escaneado del token de entrada.
<code>nextDouble()</code>	Método que devuelve un <code>double</code> escaneado del token de entrada.

En la siguiente clase, se ilustra un ejemplo de importación de la clase *Scanner*, la creación de un objeto y la utilización del método `nextInt()`.

Ejemplo 10 palabra reservada import

```
1  import java.util.Scanner;
2
3  public class Importaciones {
4
5      public static void main(String [] args){
6          Scanner lector = new Scanner(System.in);
7          System.out.println("Ingrese el número");
8          int numero = lector.nextInt();
9          System.out.println("El número ingresado es: " +
10                         numero);
11     }
12 }
```

En este ejemplo se importa la clase *Scanner*, y facilita la lectura de datos desde teclado. Para hacer uso sus métodos es imprescindible importarla porque pertenece a un paquete diferente de donde está ubicada la clase que lo invoca.

En la línea 1, la palabra reservada `import` establece que se va realizar una importación de la clase *Scanner* que pertenece al paquete `java.util`. En esta línea solo se realiza una importación simple, y no se ha utilizado la importación grupal a través de "\*" porque hay que evitar sobrecargar a la clase *Importaciones* con interfaces o clases que no se utilizarán.

Al establecer la importación se puede hacer uso de los métodos que implementa como nextInt(), nextLine(), nextDouble(), etc.

En la línea 6, se crea un objeto de la clase *Scanner* a través de la palabra reservada **new**, por ahora solo se limitará a crear un objeto de tipo *Scanner*. En la línea 8, se invoca al método nextInt( ), este permite capturar desde el teclado datos de tipo entero, y almacenarla en la variable número. En la línea 9, se imprime un mensaje “*El número ingresado es*” concatenado con el dato almacenado en la variable número.

Un aspecto a considerar es la existencia de errores comunes al importar clases o interfaces en los paquetes. La redundancia es una de ellas. En el siguiente código se ilustra una redundancia.

*Ejemplo 11 Redundancia de importaciones*

```
1 import java.util.Scanner;
2 import java.util.*;
3
4 public class Importaciones {
5
6     public static void main(String [] args){
7         //código
8     }
9 }
```

En este ejemplo existe una redundancia de importaciones. En la línea 1 se realiza la importación de una sola clase llamada *Scanner*, que proviene del paquete *java.util*.

En la línea 2 al utilizar el \*, se importa todas las clases del paquete *java.util*, incluido la clases *Scanner*.

Pero como se ha establecido, la clase *Scanner* pertenece al paquete *java.util*, ya no es necesario que realice importar dos veces la misma clase, la primera de forma directa y la segunda de forma implícita.

#### **Recomendación**

Por convención, se debe importar sólo las clases que se utilizarán y evitar la importación innecesaria de clases que no se utilizarán.

## 2.8 EJEMPLOS

**Ejercicio 1:** Diseñar un programa que lea dos números enteros por teclado y los imprima en pantalla.

*Ejemplo 12 Lectura de dos números por teclado*

```
1 import java.util.Scanner;
2
3 public class LecturaNumeros{
4
5     public static void main(String[] args) {
6         int numero1, numero2;
7         Scanner sc = new Scanner(System.in);
8         System.out.println("Ingrese el primer número entero: ");
9         numero1 = sc.nextInt();
10        System.out.println("Ingrese el segundo número entero: ");
11        numero2 = sc.nextInt();
12        System.out.println("Los números ingresados son: " +
13                           numero1 + " y " + numero2);
14    }
15 }
```

Se establece la importación de la clase *Scanner* en la línea 1 por medio de la instrucción *import*. Para la utilización del método *nextInt()*, es imprescindible crear un objeto, y lo realizamos a través de la palabra reservada *new*. En la línea 7 se crea el objeto y a partir del se puede utilizar el método para la lectura de datos enteros desde teclado.

Su salida es la lectura de dos números y la impresión en consola de cada uno de ellos:

```
1 \programas_libro> java LecturaNumeros
2 Ingrese el primer número entero:
3 9
4 Ingrese el segundo número entero:
5 7
6 Los números ingresados son: 9 y 7
```

**Ejercicio 2:** Diseñar un algoritmo que lea el nombre y apellidos de una persona e imprima un saludo “*Buenos días nombre\_apellidos\_ingresados*”.

*Ejemplo 13 Algoritmo para generar un saludo*

```
1 import java.util.Scanner;
2
3 public class Saludo {
4
5     public static void main(String[] args) {
6         Scanner lector = new Scanner(System.in);
```

```

7  String nombres, apellidos;
8  System.out.println("Ingrese sus nombres: ");
9  nombres = lector.nextLine();
10 System.out.println("Ingrese sus apellidos: ");
11 apellidos = lector.nextLine();
12 System.out.println("Buenos días " + nombres + " " +
13         apellidos);
14 }
15 }
```

Su salida es:

```

1 \programas_libro> java Saludo
2 Ingrese sus nombres:
3 Karina Patricia
4 Ingrese sus apellidos:
5 Zumba
6 Buenos Días Karina Patricia Zumba
```

## 2.9 CUESTIONARIO

### 1. Java no es:

- Un lenguaje de programación orientado a objetos.
- Un lenguaje de programación orientada a objetos.
- Un lenguaje que solo funciona en un sistema operativo UNIX.
- Un lenguaje de programación interpretado.

### 2. ¿Qué es un bytecode?

- Es formato donde se ubica el código fuente de nuestras clases en java.
- El formato que se obtiene tras compilar un archivo fuente archivo.java.
- Una palabra reservada.
- Es un archivo que guarda información de sus aplicaciones.

### 3. Para compilar la clase “Trabajador” ¿Cuál instrucción es la correcta?

- java Trabajador.java
- javac Trabajador.class
- javac Trabajador.java
- java Trabajador.class

### 4. Para compilar un conjunto de clases en un solo comando se debe utilizar la instrucción:

- javac \*.java
- javac Archivos.java
- java \*.java
- java Archivos.java

### 5. Al generar el siguiente código en java, ¿Cómo se debe llamar el fichero java que corresponda a este código fuente?

```

1 public class Cooperativa {
2     public static void main(String[] args) {
3         System.out.println("Es una cooperativa de transito");
4     }
5 }
```

- Cooperativa.class
- Cooperativa.txt
- Cooperativa.java
- cooperativa.java

**6. Para utilizar solo a la clase Date que proviene del paquete útil, qué instrucción permitirá utilizar sus métodos:**

- Date date = new Date();
- import java.util.Date;
- import java.Date;
- import java.util.\*;

**7. ¿Cuál enunciado es falso?**

- Al ejecutarse un programa, los comentarios se imprimen en pantalla después de utilizar los caracteres //.
- Los comentarios de una sola línea se pueden utilizar tanto al inicio de la línea o final de la instrucción.
- Los comentarios son utilizados para documentar el código en los programas.
- Existen tres tipos de comentarios, el de una línea, múltiples líneas y las etiquetas utilizadas por javadoc.

**8. Seleccione cuáles palabras reservadas son válidas en java:**

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>- void</li> <li>- main</li> <li>- public</li> <li>- for</li> </ul> | <ul style="list-style-type: none"> <li>- static</li> <li>- numero</li> <li>- string</li> <li>- char</li> </ul> |
|---|--|

**9. Seleccione cuáles enunciados son incorrectos:**

- En Java se considera que los siguientes identificadores “vehículo” y “VeHíCuLo” son idénticas.
- Todas las clases pueden utilizar el comando java al ejecutar una aplicación, independientemente si no han definido el método main.
- Por convención se debe definir las clases como sustantivos propios y cuando el nombre de la clase es formada por dos palabras, la primera letra de cada palabra es con mayúscula.
- Los siguientes identificadores son válidos para definir una clase: 3Numero, 87, 67Personas, hora22, Persona, y TipoCasa.

**10. Cuál de las siguientes extensiones corresponde al archivo fuente de un programa:**

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>- .java</li> <li>- .class</li> </ul> | <ul style="list-style-type: none"> <li>- .exe</li> <li>- .bat</li> </ul> |
|---|--|

- .stub

**11. ¿Cuáles de las siguientes líneas de código corresponden a un comentario?**

- "Es un comentario"
- /\*\* Es un comentario \*/
- (\* Es un comentario \*)
- /\* Es un comentario \*/
- { Es un comentario }
- // Es un comentario
- <>Es un comentario>>

**12. ¿Cuál es el método que especifica que una clase es Ejecutora?**

- public static void main(String [] args )
- public void main (String [] args )
- public static void main(String args )
- public void main (String args )

**13. Cómo se recomienda escribir el nombre de una clase:**

- Todo el nombre en minúscula.
- Debe ser en terminación verbal ejemplo la clase Correr.
- En sustantivo, la primera letra en mayúscula.
- Ninguna de las anteriores.

**14. Las importaciones permiten:**

- Utilizar clases que requieren tu proyecto de software y están en el mismo paquete.
- Utilizar clases que requieren tu proyecto de software y están en otro paquete.
- Utilizar clases que requieren tu proyecto de software y están hechas en otro lenguaje.
- Utilizar clases cuando establecen una jerarquía por medio de la herencia.

## 3 TIPOS DE DATOS PRIMITIVOS, IDENTIFICADORES

---

En este capítulo se explica el funcionamiento de cada uno de los operadores primitivos formando expresiones, se codifican ejemplos para explicar su comportamiento. Además se explica la función que cumple el identificador, las reglas para definirlos correctamente y los errores que se comete al nombrarlos. Para finalizar se define a los conceptos de variables y constantes, se establecen las diferencias que existen entre cada uno.

### Objetivos:

- Identificar y clasificar los tipos de datos primitivos.
- Establecer las reglas para seleccionar un identificador válidos en Java.
- Diferenciar cómo usar las variables y constantes.

### 3.1.1 TIPOS DE DATOS PRIMITIVOS

Son almacenados en variables, que ocupan un espacio de memoria en el ordenador. Además sirven para almacenar datos durante la ejecución de la aplicación y están en constante cambio. En java son elementos específicos o elementales que representan un único dato simple y se puede clasificar en cuatro grupos: enteros, decimales, carácter y boléanos. Para declarar una variable se sigue la siguiente sintaxis:

tipo\_dato *identificador\_variable*;

Al definir un valor de tipo entero, real, lógico, carácter, cadena de caracteres o un valor nulo (**null**), en Java se domina literal.

### 3.1.2 DATOS ENTEROS:

Son representados por: **byte**, **short**, **int** y **long**. Permite almacenar datos enteros en las variables o atributos de una clase.

- **byte**: es un entero de 8 bits o un **byte**. Su valor mínimo es -128 y el máximo 127.
- **short**: es un entero de 2. Su valor mínimo es -32,768 y el máximo 32,767.
- **int**: es un entero de 4 **byte**. Su valor mínimo es -2,147,483,648 y el máximo 2,147,483,649.
- **long**: es un entero de 8 **byte**. Su valor mínimo es -9,223,372,036,854,775,808 y el máximo 9,223,372,036,854,775,807. Para determinar que el dato pertenece a este tipo de dato, se le agrega L, un ejemplo 558765068L.

En el siguiente ejemplo se define cuatro variables locales para cada tipo de datos entero:

Ejemplo 14 Tipo de dato entero

```
1 public class TipoDatosPrimitivo {  
2     public static void main(String [] args){  
3         byte b = 9;  
4         short s = 60;  
5         int i = 9056;  
6         long l = 5680576695;  
7     }  
8 }
```

En este ejemplo, se definen cuatro variables con tipos de datos enteros diferentes. El comportamiento de cada uno al parecer es normal, pero existe un elemento que hay que tomar en cuenta, en la línea 6, se ha definido un literal de tipo entero, Java lo toma como tipo de dato **int** y su identificador es "l", al compilar se genera un error de compilación:

```
1 programas_libro>javac TipoDatosPrimitivo.java  
2 TipoDatosPrimitivo.java:6: error: integer number too large: 5680576695  
3     long l = 5680576695;  
4           ^  
5 1 error  
6
```

Este error de compilación, expresa que el valor almacenado en el identificador "l" es de tipo entero **int** y supera su límite en el rango de valores. Para resolver este

problema, hay que indicar que el literal debe ser de mayor extensión y se le agrega al final **L** que representa que es de tipo **long**. En un literal de tipo **long**, su valor mínimo y máximo no será como el de un **int** (Su valor mínimo es -2,147,483,648 y el máximo 2,147,483,649) sino como el de un **long** (valor mínimo es  $-9 \times 10^{18}$  y el máximo  $9 \times 10^{18}$ ).

La solución se representa en el siguiente ejemplo:

Ejemplo 15 Literal entero por defecto

```
1 | public class TipoDatosPrimitivo {  
2 |     public static void main(String [] args){  
3 |         byte b = 9;  
4 |         short s = 60;  
5 |         int i = 9056;  
6 |         long l = 5680576695L; // se corrige el error  
7 |     }  
8 | }
```

Java también permite dígitos en diferentes formatos, como:

- **Octal (dígitos 0-7):** todos los números que inicien con 0 representan este sistema.

```
1 | int octal = 020;
```

- **Hexadecimal (dígitos 0-9 y letras A-F):** todos los números que inicien 0 seguido de x o X.

```
1 | int hexadecimal = 0x1F;
```

- **Binarios (dígitos 0-1):** todos los números que inicien con 0 seguido de b o B.

```
1 | int binario = 0b110;
```

### 3.1.3 DATOS DECIMALES:

Son representados por: **float** y **double**. Permite almacenar datos decimales y están compuesto de la parte entera y decimal:

- **float:** es un decimal de 4 bytes. Su valor mínimo  $-3,4 \times 10^{38}$  y el valor máximo  $3,4 \times 10^{38}$ . Para determinar que el valor es de tipo float se coloca una f al final del valor, por ejemplo 54,7f.
- **double:** es un decimal de 8 bytes. Su valor mínimo  $-1,79 \times 10^{308}$  y su valor máximo  $1,79 \times 10^{308}$ . Se le puede ubicar la letra d al final del número para indicar que es un valor **double**, por ejemplo 54,7d.

El siguiente ejemplo indica la forma de declarar datos de tipo decimal:

*Ejemplo 16 Datos de tipo decimal*

```
1 public class TipoDatosPrimitivo {  
2     public static void main(String [] args){  
3         float f = 9f;  
4         double d = 60.76;  
5     }  
6 }
```

#### **Recomendación**

Se debe emplear el tipo de datos **double** cuando se trabaja con tipos de dato reales, debido a que muchas aplicativos para cálculo matemático trabajan con este formato.

#### **3.1.4 DATOS CARACTER**

Es representado por la palabra reservada **char**. Permite almacenar símbolos de escritura. Con el código UNICODE se puede almacenar cualquier carácter especial. En el siguiente ejemplo se indica la forma de declarar datos de tipo carácter:

*Ejemplo 17 Datos primitivos de tipo caracter*

```
1 public class TipoDatosPrimitivo {  
2     public static void main(String [] args){  
3         char numero = '9';  
4         char vocal = 'u';  
5         char letra = 'g';  
6         char dato = 65; // código UNICODE  
7     }  
8 }
```

Para definir caracteres se utiliza la comilla simple, las cadenas de caracteres utilizan doble comilla, para este caso si se declara una variable de tipo carácter y se le asigna un valor con doble comilla, dará un error de compilación porque la cadena de texto es un tipo de datos diferente a los caracteres.

En las líneas 3, 4 y 5, se definen tipos de datos primitivos carácter y son datos declarados de forma legal. En la línea 6 puede existir confusión porque se declara un valor de tipo entero en una variable de tipo carácter, Java asume que el valor 65 representa a un código Unicode y lo traduce como un carácter, en este caso la variable datos se le asignará el dato 'A'.

En Java, también hay una serie de caracteres especiales que van precedidos por el símbolo \, se presentan en la siguiente tabla:

Tabla 5 Caracteres especiales

Carácter	Significado
\b	Retroceso.
\t	Tabulador.
\n	Salto de línea.
\f	Alimentación de página.
\r	Retorno de carro.
\"	Dobles comillas.
\'	Comillas simples.
\udddd	Las cuatro letras d, son en realidad números en hexadecimal. Representa el carácter Unicode cuyo código es representado por las dddd.

### 3.1.5 DATOS LÓGICOS

Es representado por la palabra reservada **boolean**. Ocupa dos bytes y permite almacenar un solo valor, si es **true** (verdadero) o **false** (falso). El siguiente ejemplo indica la forma de declarar este tipo de datos:

Ejemplo 18 Datos Primitivos booleanos

```
1 public class TipoDatosPrimitivo {  
2     public static void main(String [] args){  
3         boolean estado = true;  
4         boolean condición = false;  
5     }  
6 }
```

En resumen, los datos primitivos son datos esenciales donde se basa todos los tipos de datos de Clase. En la siguiente tabla se indica los tipos de datos, los byte que ocupa, sus rangos y un ejemplo de cada uno:

Tabla 6 Tipos de datos primitivos

Tipo de datos primitivo	Bytes que ocupa	Límites de valores	Ejemplo
Byte	1	-128 a 127	12
Short	2	-32.768 a 32.767	12
Int	4	-2.147.483.648 a 2.147.483.649	12
Long	8	-9 · 10 a 9 · 10	12
Float	4	-3,4 · 1038 a 3,4 · 1038	12.56
double	8	-1,79 · 10308 a 1,79 · 10308	12.56f
Char	2	Caracteres (en Unicode)	'k'
boolean	2	true, false	True

### **Recomendación**

Las variables definidos en los métodos se deben ser inicializados o asignarles un valor, de lo contrario se generará un error de compilación.

## **3.2 Variables, Identificadores y constantes**

### **3.2.1 VARIABLE**

Es un nombre que se da a un valor y ocupa un espacio en memoria para que almacena datos. Cuando se declara una variable, debe indicar el tipo y su identificador. Por ejemplo se definirá dos variables con diferentes tipos de datos:

```
1 | double promedioNotas;  
2 | int numberEstudiantes;
```

Java es un lenguaje de programación fuertemente tipado y obliga a que se declare el tipo de dato antes de utilizar la variable o atributos, otros lenguajes de programación que no tipados permiten declarar sus atributos o variables sin definir su tipo de datos como ejemplo php, python, etc.

Se han definido estas dos variables sin valores, cuando se le agrega valores en su declaración se denomina **inicialización** de una variable. Para inicializar se usa el operador de asignación = y se le da un valor, mirar el siguiente ejemplo:

```
1 | double promedioNotas = 9.5;  
2 | int numberEstudiantes = 25;
```

Se puede declarar múltiples variables del mismo tipo en la misma instrucción utilizando como separador de las variables la coma (,). Por ejemplo:

```
1 | int numero, edad, cantidad;
```

Las tres variables fueron declaradas de tipo de dato entero **int** pero ninguna ha sido inicializada.

Se puede declarar e inicializar múltiples variables del mismo tipo en una misma línea de código. Por ejemplo:

```
1 | int numero, edad, cantidad = 444;
```

Las tres variables fueron declaradas (número, edad y cantidad) de tipo de dato entero **int** y solo una la variable cantidad ha sido inicializada con el valor 444.

Ahora se analiza las declaraciones en el siguiente ejemplo:

Ejemplo 19 Declaración e inicialización de variables

```
1 public class Variable {  
2     public static void main(String [] args){  
3         boolean estado, cuentaActiva; // legal  
4         int cantidad = 78, numero; // legal  
5         double salario, double precio; // no legal  
6         char letra; char vocal; // legal  
7     }  
8 }
```

En la línea 3, la primera declaración cumple con la sintaxis y es legal. Se declaran dos variables de tipo booleana sin inicializar.

En la línea 4, la segunda declaración cumple con la sintaxis y es legal. Se declaran la variable cantidad y se inicializa con un valor 78, mientras la variable número solo fue declarada sin inicialización.

En la línea 5, la tercera declaración no cumple con la sintaxis correcta y no es legal. Java no permite declarar dos tipos de datos en la misma instrucción. Es incorrecto porque “**double** salario” y “**double** precio” son dos variables declarada y separadas por la coma, cada declaración es una instrucción individual, para corregir el error se debe reemplazar la coma por el punto y coma (;) o a su vez eliminar la segunda palabra reservada **double**.

```
1 | double salario; double precio;  
1 | double salario, precio;
```

Estas dos sentencias cumplen con la sintaxis y son legales.

En la línea 6, la cuarta declaración, existen dos sentencias separadas por el punto y coma y son sentencias válidas pero si se rescata el concepto que los programas deben ser legibles y ordenados, esta sentencia no cumpliría con este aspecto.

Es preciso indicar que en Java tiene tres tipos de variables:

- **De instancia o miembros de dato:** se usan para guardar los atributos de un objeto particular.

- **De clase:** también llamada de datos estáticos, son valores que se guardan son los para todos los objetos de una determinada clase.
- **Locales:** son las que se utiliza dentro de los métodos.

#### **Recomendación**

Los programadores deben ser ordenados en sus códigos y declarar las variables en un solo lugar del programa, de preferencias al inicio del método.

### 3.2.2 IDENTIFICADORES

Es un nombre que identifica a una variable, método o clase en programas en Java.

Para definir un identificador se debe aplicar las siguientes reglas:

- El nombre del identificador debe iniciar con una letra, el carácter subrayado “\_” o el carácter dólar “\$”.
- Puede incluir números, pero nunca al inicio.
- Java distingue entre letras mayúsculas y minúsculas.
- No puede contener el carácter espacio en blanco.
- No se pueden utilizar las palabras reservadas.
- No se pueden incluir caracteres especiales.

#### **Recomendación**

Por convención, hay que nombrar a un identificador con un nombre que sea significativo, así el programa será más legible si se desea interpretar el código para dar mantenimiento. Los nombres de las variables siempre comienza con letra minúscula (mesa, edad, ciudad, etc.) y si el nombre está compuesto por dos o más palabras, la primera letra desde la segunda palabra va con mayúscula (centroComercial, empleadoAdministrativo, estudianteReprobado, etc.)

En el siguiente ejemplo se identifican algunos errores comunes al nombrar variables:

*Ejemplo 20 Identificadores erróneos*

```

1 public class Identificadores {
2     public static void main(String [] args){
3         int 5Materia; // ilegal y error
4         int micorreo@gmail; // ilegal y error
5         int #base; // ilegal y error
6         int static; // ilegal y error
7     }
8 }
```

En este ejemplo, en la línea 3 el identificador 5Materia es ilegal porque no se puede iniciar con un número.

En la línea 4 el identificador micorreo@gmail es ilegal porque @ no es una letra, dígito, \$ o \_.

En la línea 5 el identificador #base es ilegal porque inicia con #, solo se puede iniciar con letra, dígito, \$ o \_.

En la línea 6, es ilegal el nombre del identificador porque está utilizando una palabra reservada.

### 3.2.3 CONSTANTES

Es una variable del programa que mantiene un valor inmutable a lo largo de toda la vida del programa. Estos valores definidos en las constantes no cambian. Para definir una constante en Java se utiliza la palabra reservada **final**.

Su sintaxis es:

```
static final nombreConstante = valor;
```

La palabra reservada **static** se aplica a una constante cuyo valor es única para todas las instancias de la clase. La palabra reservada **final** indica que una variable es de tipo constante y no admitirá cambios después de su declaración y asignación de valor. Las constantes son declaradas e inicializadas en la zona de definición de los atributos, y no se definen en los métodos. El siguiente ejemplo se indica la forma de declarar una constante.

Ejemplo 21 Constantes

```
1 | public class Constantes {  
2 |  
3 |     private static final double PI = 3.1416;  
4 |  
5 |     public static void main(String [] args){  
6 |         System.out.println("constantes" + PI);  
7 |     }  
8 | }
```

## 3.3 EJERCICIOS

**Ejemplo 1:** Realizar un programa que permita ingresar el número de horas y presentará el número de semanas, días y horas que correspondas a las horas ingresadas.

Ejemplo 22 Convertir horas en semanas, días

```
1 import java.util.Scanner;
2
3 public class Horas {
4
5     public static void main(String[] args) {
6         Scanner lector = new Scanner(System.in);
7         int totalHoras, semanas, dias, horas;
8         System.out.print("Ingrese las horas:");
9         totalHoras = lector.nextInt();
10        semanas = totalHoras / (24 * 7);
11        dias = totalHoras % (24 * 7) / 24;
12        horas = totalHoras % 24;
13        System.out.println("El total de " + totalHoras + "
14            horas");
15        System.out.println("Se convierte a:");
16        System.out.println(semanas + " semanas");
17        System.out.println(dias + " días");
18        System.out.println(horas + " horas");
19    }
20 }
```

En este ejemplo se utiliza una combinación de operadores aritméticos: %, /, + para crear expresiones en estructuras lineales como “semanas = totalHoras / (24 \* 7);”, se inicializan cuatro variables de tipo entero y se realiza una entrada de datos por teclado mediante el método nextInt() de la clase *Scanner*. En la línea del 10 al 12 se almacenan datos en las variables semanas, días y horas, producto de una expresión aritmética y se presentan los valores de las variables en pantalla a través de la instrucción System.out.println();.

Su salida es:

```
1 \programas_libro>java Horas
2 Ingrese las horas:300
3 El total de 300 horas
4 se convierte a:
5 1 semanas
6 5 días
7 12 horas
```

**Ejemplo 2:** Realizar un programa que permita leer la cantidad de grados centígrados y permita traducir a grados Fahrenheit. La fórmula para realizar esta transformación es:

$$F = 32 + (9 * C / 5)$$

```

1 import java.util.Scanner;
2
3 public class Grados {
4
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7         double gradosC, gradosF;
8         System.out.println("Introduce grados Centígrados:");
9         gradosC = sc.nextDouble();
10        gradosF = 32 + (9 * gradosC / 5);
11        System.out.println(gradosC + " °C = " + gradosF + " °F");
12    }
13 }
```

Su salida es:

```

1 \programas_libro>java Grados
2 Introduce grados Centígrados:
3 600
4 600.0 °C = 1112.0 °F
```

### 3.4 CUESTIONARIO

- 1. ¿Cuáles de los siguientes identificadores son válidas en java?**
  - A\$B
  - java.util
  - helloWorld
  - public
  - true
  - 1980\_s
  
- 2. ¿Cuáles de las siguientes sentencias son correctas?**
  - int i1 = 1\_234;
  - double d2 = 1\_234.\_0;
  - double d4 = 1\_234.0;
  - double a = 1234,0;
  
- 3. Se tiene dos operandos entero a y b, y se requiere realizar una operación de división y almacenarlo en la variable r. ¿Cuál operador en Java podrían utilizarse en la expresión r = a operador b?**
  - \*
  - /
  - //
  - %
  - div
  - \
  
- 4. Selecciona que instrucción permite definir múltiples variables del mismo tipo en Java :**
  - int a, b, c, d;
  - double a; b; c; d;
  - boolean a, b; c, d;
  - char a = b = c = d;
  
- 5. ¿Cuántos byte ocupa un tipo de dato int?**
  - 4
  - 8
  - 16
  - 32
  - 64

**6. ¿Cuál es la precisión de un tipo de dato short?**

- 8 bit
- 16 bit
- 32 bit
- 64 bit

**7. ¿Cuál de los siguientes tipos no es primitivo?**

- int
- float
- Integer
- boolean

**8. Señale los datos primitivos de la siguiente lista:**

- String
- Double
- long
- char

**9. Seleccione las instrucciones que definen o inicializan variables de forma correcta:**

- boolean estado, cuentaActiva;
- int cantidad = 78, numero;
- double salario, double precio;
- char letra; char vocal;

**10. Indicar cuál o cuáles de los siguientes valores o constantes literales puede asignarse a una variable de tipo boolean:**

- "true"
- boolean
- false
- falso
- "false"
- true

**11. Seleccione la sentencia correcta para definir una constante para el valor de la velocidad de la luz:**

- int velocidadLuz = 300000;
- static int velocidadLuz = 300000;
- static final int velocidadLuz = 300000;
- final int velocidadLuz = 300000;

**12. ¿Cuáles de los siguientes tipos primitivos utiliza el formato en punto o coma flotante?**

- byte
- short
- float
- int
- double
- long

**13. ¿Cuáles de las siguientes instrucciones modifican sus valores?**

- int numero = i + j + k + 7;
- int valor = lector.nextInt()
- System.out.println( "valores modificados" );
- System.out.println( "i = 90" );

## 4 OPERADORES

---

En este capítulo describe la clasificación y funcionamiento de los operadores que soporta el lenguaje de programación Java. Además a través de ejemplos explica cuál es el comportamiento de cada uno de ellos.

### Objetivos

- Describir y explicar el funcionamiento de los operadores aritméticos, incrementales, de relación, lógicos, asignación y ternario.
- Evaluar expresiones utilizando cualquier operador.

### 4.1 INTRODUCCIÓN

Los lenguajes de programación permiten realizar operaciones entre los tipos de datos primitivos, por ejemplo: la suma, resta, producto, cociente, residuo, mayor que, menor o igual, etc. Un operador en java es un símbolo especial que puede ser aplicado para fijar valores en las variables o atributos.

Hay tres tipos de operadores en Java:

- Unario.
- Binario.
- Ternario.

Además pueden realizar acciones sobre uno o dos operandos. Un operador que actúa sobre un solo operando es un operador unario, y si actúa sobre dos operandos es un operador binario.

El valor devuelto tras la evaluación de una expresión depende del operador y del tipo de los operandos. Por ejemplo, los operadores aritméticos trabajan con operandos numéricos, llevan a cabo operaciones aritméticas básicas y devuelven el valor numérico |correspondiente. Por ello hay una estrecha relación entre el operador y los operandos.

En Java, la lectura de los operadores no necesariamente se realiza de izquierda a derecha, depende del operador para establecer la dirección de lectura.

## 4.2 OPERADORES ARITMÉTICOS

Java soporta varios operadores aritméticos que actúan sobre números enteros y en coma flotante. Los operadores binarios soportados por Java son:

Tabla 7 Operadores aritméticos

Operador	Descripción	Ejemplo	Resultado
	Suma	int a = 80 + 10	a tiene el valor de 90
+	Concatenar	String a = "Hola " + "amigos "	a tiene el valor de "Hola amigos "
-	Resta	int a = 80 - 10	a tiene el valor de 70
*	Multiplicación	int a = 80 * 10	a tiene el valor de 800
/	División	int a = 80 / 10	a tiene el valor de 8
%	Modulo o residuo	int a = 80 % 10	a tiene el valor de 0

El siguiente ejemplo hace uso de los operadores aritméticos para operaciones básicas.

Ejemplo 23 Operadores aritméticos

```
1 public class OperadoresAritmeticos {  
2     public static void main(String [] args){  
3         int suma = 8 + 5;  
4         int resta = 8 - 5;  
5         int producto = 8 * 5;  
6         int division = 8 / 5;  
7         int residuo = 8 % 5;  
8         System.out.println("Operador suma = " + suma);  
9         System.out.println("Operador resta = " + resta);  
10        System.out.println("Operador producto = " + producto);  
11        System.out.println("Operador division = " + division);  
12        System.out.println("Operador residuo = " + residuo);  
13    }  
14 }
```

La salida es:

```
1 \programas_libro> java OperadoresAritmeticos  
2 Operador suma = 13  
3 Operador resta = 3  
4 Operador producto = 40  
5 Operador division = 1  
6 Operador residuo = 3
```

Una mala práctica es colocar los operandos y operadores juntos, sin sus respectivos espacios, por ejemplo:

```
1 | int suma=8+5;
```

### Recomendación

Por convención los operadores deben estar separados por los operandos en un espacio para que el código sea legible, por ejemplo:

```
int suma = 8 + 5;
```

El operador + también cumple un rol diferente al de sumar y puede concatenar cadenas, esta rol que cumple el operador + se lo verá al tratar la sección de cadenas.

También es necesario tener en cuenta que el resultado de la operación aritmética / está relacionado con sus operandos. Por ejemplo si la división se da entre enteros su resultado será entre entero, y si su división se la realiza entre decimales su respuesta es en decimales. En el siguiente ejemplo ilustramos dos divisiones con diferentes tipos de datos:

Ejemplo 24 Operador división

```
1 public class OperadoresAritmeticos {  
2     public static void main(String [] args){  
3         double resultado;  
4         int a = 5 , b = 2;  
5         resultado = a / b;  
6         System.out.println(" División = " + resultado);  
7         resultado = (double) a / (double) b;  
8         System.out.println(" División = " + resultado);  
9     }  
10 }
```

En este ejemplo, se ha definido una variable de tipo decimal llamada resultado en la línea 3, en la línea 4 se definen e inicializan variables de tipo entero. En la línea 5 se asigna a la variable resultado una división de enteros, como resultado es de tipo double, se conoce que este tipo de datos soporta tanto la parte entera y decimal. En java la división entre entero da como resultado una división entera, por lo tanto a resultado se le asignará el valor de la división entre entero (2).

En la línea 7 se hace una conversión de tipo, se pasa el valor de a y b de tipo entero a decimal. La división entre decimales da como resultado un valor decimal y se le asignará a la variable resultado.

Su salida es:

```
1 \programas_libro>java OperadoresAritmeticos
2 División = 2.0
3 División = 2.5
```

El mismo principio que se aplica al operador / se aplica en %.

#### **Recomendación**

Recuerda que el operador % obtiene el residuo de una división y es aplicado a números enteros.

Tener muy cuenta las divisiones por cero genera un error de ejecución.

## **4.3 OPERADORES DE RELACIÓN**

Este tipo de operadores realizan comparaciones entre datos compatibles de tipos primitivos cuyo resultado siempre será booleano, **true** o **false**. En otras palabras sirven para comparar valores. En la siguiente tabla se detallan este tipo de operadores:

*Tabla 8 Operadores de relación*

<b>Operador</b>	<b>Descripción</b>	<b>Ejemplo</b>	<b>Resultado</b>
<code>==</code>	Igual	<code>boolean a = 80 == 10;</code>	a tiene el valor de <b>false</b> .
<code>!=</code>	Distinto	<code>boolean a = 80 != 10;</code>	a tiene el valor de <b>true</b> .
<code>&lt;</code>	Menor que	<code>boolean a = 80 &lt; 10;</code>	a tiene el valor de <b>false</b> .
<code>&gt;</code>	Mayor que	<code>boolean a = 80 &gt; 10;</code>	a tiene el valor de <b>true</b> .
<code>&lt;=</code>	Menor o igual que	<code>boolean a = 80 &lt;= 10;</code>	a tiene el valor de <b>false</b> .
<code>&gt;=</code>	Mayor o igual que	<code>boolean a = 80 &gt;= 10;</code>	a tiene el valor de <b>true</b> .

En el siguiente ejemplo se indica el uso de los operadores de relación.

*Ejemplo 25 Operadores de Relación*

```
1 public class OperadoresRelacion {
2     public static void main(String [] args){
3         boolean igual = 80 == 10;
4         boolean diferente = 80 != 10;
5         boolean menor = 80 < 10;
6         boolean mayor = 80 > 10;
7         boolean menorIgual = 80 <= 10;
8         boolean mayorIgual = 80 >= 10;
9         System.out.println("Operador igual = " + igual);
10        System.out.println("Operador diferente = " + diferente);
11        System.out.println("Operador menor que = " + menor);
12        System.out.println("Operador mayor que = " + mayor);
13        System.out.println("Operador menor igual = " +
14            menorIgual);
```

```

15     System.out.println("Operador mayor igual = " +
16         mayorIgual);
19 }
20 }
```

La salida por pantalla es:

```

1 \programas_libro> java OperadoresRelacion
2 Operador igual = false
3 Operador diferente = true
4 Operador menor que = false
5 Operador mayor que = true
6 Operador menor igual = false
7 Operador mayor igual = true
```

#### **Recomendación**

No se debe confundir el operador de asignación = con el operador de igualdad == porque con el primero asignamos un valor a una variable y con el segundo comparamos dos valores si son iguales.

## **4.4 OPERADORES CONDICIONALES**

Realizan operaciones sobre datos booleanos y tienen como resultado un valor booleano. Los operadores condicionales que soporta Java son:

*Tabla 9 operadores condicionales*

<b>Operador</b>	<b>Descripción</b>	<b>Ejemplo</b>	<b>Resultado</b>
&&	Y	boolean a = (8 > 10) && (12 < 5);	a tiene el valor de false
	O	boolean a = (8 > 10)    (12 < 5);	a tiene el valor de true
!	Negación	boolean a = (!true);	a tiene el valor de false

En el siguiente ejemplo se indica el uso de los operadores condicionales.

*Ejemplo 26 Operadores Condicionales*

```

1 public class OperadoresCondicionales {
2     public static void main(String [] args){
3         boolean y = (8 > 10) && (12 < 5);
4         boolean o = (8 > 10) || (12 < 5);
5         boolean negacion = (!true);
6         System.out.println("Operador y = " + y);
7         System.out.println("Operador o = " + o);
8         System.out.println("Operador negacion = " + negacion);
9     }
10 }
```

La salida por pantalla es:

```

1 \programas_libro>java OperadoresCondicionales
2 Operador y = false
3 Operador o = false
4 Operador negacion = false

```

## 4.5 OPERADORES A NIVEL DE BITS

Realizan operaciones con dígitos (ceros y unos) de la representación binaria de los operandos. Exceptuando al operador negación, los demás operadores son binarios.

Operadores que realizan operaciones sobre un solo bit. Los operadores que soporta Java son:

*Tabla 10 Operadores de bit*

Operador	Descripción	Ejemplo	Resultado
&	And	a = 20 & 5 ;	a tiene el valor de 4.
	Or	a = 20   5 ;	a tiene el valor de 21.
~	Not	a = ~20 ;	a tiene el valor de -21.
^	Xor	a = 20 ^ 10	a tiene el valor de 17.
>>	Desplazamiento a la derecha	a = a >> b	a tiene el valor de 0.
<<	Desplazamiento a la izquierda	a = a << b	a tiene el valor de 640.
>>>	Desplazamiento derecha con relleno de ceros	a = a >>>b	a tiene el valor de 0.

En el siguiente ejemplo se indica el uso de los operadores de bit.

*Ejemplo 27 Operadores a bit*

```

1 public class OperadoresBit {
2     public static void main(String [] args){
3         int a = 20, b = 5;
4         int and = a & b;
5         int or = a | b;
6         int not = ~a;
7         int xor = a ^ b;
8         int derecha = a>>b;
9         int izquierda = a<<b;
10        int rellenoDerecha = (a>>>b);
11        System.out.println("Operador and = " + and);
12        System.out.println("Operador or = " + or);
13        System.out.println("Operador not = " + not);
14        System.out.println("Operador xor = " + xor);
15        System.out.println("Operador desplazamiento a la derecha
16                    = " + derecha);
17        System.out.println("Operador desplazamiento a la
18                    izquierda = " + izquierda);
19        System.out.println("Operador desplazamiento a la derecha
20                    con relleno = " + rellenoDerecha);
21    }
22 }

```

La salida por pantalla es:

```
1 \programas_libro>java OperadoresBit
2 Operador and = 4
3 Operador or = 21
4 Operador not = -21
5 Operador xor = 17
6 Operador desplazamiento a la derecha = 0
7 Operador desplazamiento a la izquierda = 640
8 Operador desplazamiento a la derecha con relleno = 0
```

## 4.6 OPERADOR DE ASIGNACIÓN

Este operador se utiliza el símbolo `=`, es de tipo binario. Además, es el más importante y frecuentemente usado, se ha empleado para la inicialización de las variables.

```
1 | int valor = 90;
```

En la sentencia anterior, el operador `=` asigna el valor de tipo entero (90) que se está en la derecha al operando. El operando de la izquierda suele ser el identificador de una variable o atributo. Para que la operación de asignación sea la correcta tanto el valor como el identificador deben ser del mismo tipo.

Tabla 11 Operador de asignación =

Operador	Descripción	Ejemplo	Resultado
<code>=</code>	Operador de asignación	<code>int a = 80</code>	a tiene el valor de 80

En Java las asignaciones múltiples son también posibles:

```
1 | int c = a = b = 100;
```

En la instrucción anterior se definen e inicializan una variable de tipo entero, tanto `a`, `b` y `c` se les asignará el valor de 100.

Además, en Java existen operadores aritméticos y de asignación en un solo componente. En otras palabras se puede combinar con operadores aritmético. En la siguiente tabla se resumen los diferentes operadores:

Tabla 12 Operadores de asignación

Operador	Descripción	Ejemplo	Resultado
<code>+=</code>	Suma combinada	<code>x+=y</code>	<code>x = x + y</code>
<code>-=</code>	Resta combinada	<code>x-=y</code>	<code>x = x - y</code>
<code>*=</code>	Producto combinado	<code>x*=y</code>	<code>x = x * y</code>
<code>/=</code>	División combinada	<code>x/=y</code>	<code>x = x / y</code>
<code>%=</code>	Residuo combinado	<code>x%=y</code>	<code>x = x % y</code>

Si tiene definido la siguiente sentencia:

```
1 | int x = 1;
2 | x += 15; //x = x + 15
```

Java evalúa las sentencias y reconoce que es una expresión de asignación, a la variable x se le asigna el nuevo valor de la suma entre x y el valor 15.

En el siguiente ejemplo se indica el uso de los operadores de asignación.

Ejemplo 28 Operadores a bit

```
1 public class OperadoresAsignacion {
2     public static void main(String [] args){
3         int x = 20;
4         int y = 5;
5         System.out.println(" Operador += es: " + (x+=y));
6         x = 20;
7         System.out.println(" Operador -= es: " + (x-=y));
8         x = 20;
9         System.out.println(" Operador *= es: " + (x*=y));
10        x = 20;
11        System.out.println(" Operador /= es: " + (x/=y));
12        x = 20;
13        System.out.println(" Operador %= es: " + (x%=y));
14    }
15 }
```

La salida por pantalla es:

```
1 \programas_libro>java OperadoresAsignacion
2 Operador += es: 25
3 Operador -= es: 15
4 Operador *= es: 100
5 Operador /= es: 4
6 Operador %= es: 0
```

## 4.7 OPERADOR DE INCREMENTO Y DECREMENTO

Son operadores unarios, el operando puede ser de tipo numérico o carácter y pueden emplearse de dos formas dependiendo de su posición con respecto al operando.

Tabla 13 Operadores de incremento o decremento

Operador	Descripción	Ejemplo	Resultado
++	Incremento: suma 1 al operando, como prefijo o sufijo.	int x = 2; x++	x contiene el valor de 3.
--	Decremento: como prefijo o sufijo.	int x = 2; x--	x contiene el valor de 1.

Este tipo de operadores son los más confusos para el programador, ya que el resultado de la operación depende de que el operador esté a la derecha o a la izquierda. Hay dos formas de utilizar el operador de incremento y el decremento: **x++ o x++.**

En la versión sufijo, el operando aparece a la izquierda del operador y hace que el operador incremente en uno al valor de la variable. En el siguiente código se explica la instrucción **x++ y x--.**

Ejemplo 29 Operadores de Incremento y decremento sufijo

```
1 public class OperadoresIncremento {  
2     public static void main(String [] args){  
3         int a = 5 , b = 0;  
4         b = a++;  
5         System.out.println(" Operador de incremento = " + a);  
6         System.out.println(" La variable b = " + b);  
7         a = 5;  
8         b = a--;  
9         System.out.println(" Operador de decremento = " + a);  
10        System.out.println(" La variable b = " + b);  
11    }  
12 }
```

En este ejemplo, en la línea 3, se declara dos variables a de tipo entero e inicializa a con valor 5 y b con 0.

En la línea 4, se utiliza un operador de incremento sufijo aplicado a la variable a, esta instrucción a++ primero utiliza la variable y luego se incrementa su valor. Significa que a b se le asigna el valor de a (5) y luego a se incrementará en uno. Los valores definidos son a = 6 y b = 5.

El mismo procedimiento se efectúa en el decremento.

La salida por pantalla al ejecutar la clase es:

```
1 \programas_libro>java OperadoresIncremento
2 Operador de incremento = 6
3 La variable b = 5
4 Operador de decremento = 4
5 La variable b = 5
```

En la versión prefijo, el operando aparece a la derecha del operador, en otras palabras `++x` primero se incrementa el valor de la variable y luego se utiliza. En el siguiente código se explica la instrucción `++x` y `--x`.

*Ejemplo 30 Instrucciones de incremento y decremento prefijo*

```
1 public class OperadoresIncremento {
2     public static void main(String [] args){
3         int a = 5 , b = 0;
4         b = ++a;
5         System.out.println(" Operador de incremento = " + a);
6         System.out.println(" La variable b = " + b);
7         a = 5;
8         b = --a;
9         System.out.println(" Operador de decremento = " + a);
10        System.out.println(" La variable b = " + b);
11    }
12 }
```

En este ejemplo en la línea 3, se declara dos variables a de tipo entero e inicializa a con valor 5 y b con 0.

En la línea 4, se utiliza un operador de incremento prefijo aplicado a la variable a, esta instrucción `++a` incrementa su valor en uno y luego se le asigna este valor a b. Significa que la b se le asigna el valor de a (a vale 6 y ahora b vale a). Los valores definidos son a = 6 y b = 6.

El mismo procedimiento se efectúa en el decremento.

La salida por pantalla al ejecutar la clase es:

```
1 \programas_libro>java OperadoresIncremento
2 Operador de incremento = 6
3 La variable b = 6
4 Operador de decremento = 4
5 La variable b = 4
```

## 4.8 OPERADOR TERNARIO:

Es conocido como if de una línea y permite ejecutar una instrucción u otra según el valor de la expresión. Su sintaxis es:

```
(expresión_lógica) ? valor_verdadero : valor_falso;
```

La expresión lógica es un condición que al ser evaluada devuelva un valor booleano (**true** o **false**), como toda condición selectiva se tiene dos opciones, las si la expresión lógica es verdadera entonces se ejecuta el valor verdadero y en caso contrario se ejecuta el valor false y esto se lo puede asignar a una variable. Su sintaxis es:

```
1 | String mensaje = (numero > 0) ? "positivo" : "negativo";
```

En la instrucción anterior, primero se evalúa la expresión lógica si el número es positivo o negativo, si el número tiene un valor de 5 la expresión será verdadera (**true**) y se le asignará a la variable “mensaje” el valor de “verdadero”. Este operador se explicará más detenidamente en el tema de estructuras selectivas.

## 4.9 PRECEDENCIA DE OPERADORES

Cuando se crea aplicaciones, siempre se codifica expresiones aritméticas, condiciones, etc. Una expresión es una combinación de operadores y operandos que se evalúa y se resuelve de acuerdo a precedencia de operadores, generando un determinado resultado. Si se tiene se define esta expresión:

```
1 | int resultado = a + b * c;
```

Un usuario normal, evaluaría la expresión de izquierda a derecha, primero sumaría los operando a y b, y luego la suma lo multiplicará con c. Pero en programación no necesariamente se resuelven las expresiones de izquierda a derecha sino evaluando la precedencia de operadores En el caso de esta expresión el operador \* tiene mayor procedencia que +, por ello el resultado de la expresión cambiaría. Si se tiene el caso de que dos operadores se encuentran en la misma expresión y tienen de la misma prioridad se evalúan de izquierda a derecha dentro de la misma expresión.

El orden de precedencia de los operadores en Java es:

Tabla 14 Precedencia de operadores

		Operadores			
Mayor precedencia	( )	[ ]			
	++	--	~	!	
	*	/	%		
	+	-			
	>>	>>>	<<		
	>	>=	<	<=	
	==	!=			
	&				
	^				
Menor precedencia					
	&&				
	?:				
	=	+ = , - =	* = , / =	% =	

### Recomendación

Los paréntesis cambian el orden de precedencia de operadores en las expresiones. Primero se ejecuta lo que está dentro de los paréntesis.

## 4.10 EJERCICIOS

**Ejemplo 1:** Realizar un programa que transforme la velocidad de km/h a m/s.

Ejemplo 31 Algoritmo de transformación de velocidades

```
1 import java.util.Scanner;
2
3 public class Velocidad {
4
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7         double velocidad;
8         System.out.println("Ingrese la velocidad en Km/h: ");
9         velocidad = sc.nextDouble();
10        System.out.println(velocidad + " Km/h -> " + velocidad *
11                           1000 / 3600 + " m/s");
12    }
13 }
```

En este ejercicio se hace la lectura por teclado de un dato decimal a través del método `nextDouble()` de la clase `Scanner`. En la línea 10, se define la expresión

"velocidad \* 1000 / 3600", como el operador \* y / tienen el mismo nivel de precedencia, la lectura se realiza de izquierda a derecha.

Su salida es:

```
1 \programas_libro> java Velocidad
2 Ingrese la velocidad en Km/h:
3 90
4 90.0 Km/h -> 25.0 m/s
```

**Ejercicio 2:** Diseñar un programa que dado el peso y la altura de una persona, calcule su índice de Masa Corporal.

*Ejemplo 32 Cálculo índice masa corporal*

```
1 import java.util.Scanner;
2
3 public class IndiceMasaCorporal {
4
5     public static void main(String[] args) {
6         Scanner lector = new Scanner(System.in);
7         System.out.println("Cálculo del índice de masa
8             corporal");
9         System.out.println("Ingrese su peso:");
10        double peso = lector.nextDouble();
11        System.out.println("Ingrese su altura:");
12        double altura = lector.nextDouble();
13        double indice = peso/(altura*altura);
14        System.out.println("Para un peso de "+ peso +" kilogramos
15            y");
16        System.out.println("una altura de "+ altura + " metros");
17        System.out.println("el índice de masa corporal es de "+
18            indice);
19    }
20 }
```

Su salida es:

```
1 \programas_libro> java IndiceMasaCorporal
2 Cálculo del índice de masa corporal
3 Ingrese su peso:
4 79,9
5 Ingrese su altura:
6 1,5
7 Para un peso de 79.9 kilogramos y
8 una altura de 1.5 metros
9 el índice de masa corporal es de 35.51111111111111
```

## 4.11 CUESTIONARIO

### 1. Evalúe la siguiente expresión y seleccione la respuesta correcta:

```
1 | int x1 = 6, x2 = 5, x3 = 89, x4 = 56, x5 = 8;
2 | boolean expresion = (x1 > x2) && (x1 > x3) || (x1 > x4) &&
3 |           (x1 > x5);
    - null
    - Error de compilación
    - false
    - 6
    - 70
    - true
```

### 2. Analice el siguiente código y seleccione la respuesta correcta:

```
1 | int i = 2;
2 | int j=1;
3 | System.out.println ((i >= 10) && (j < 40));
    - true
    - false
    - undefined
    - Error de compilación
```

### 3. Analice el siguiente código y seleccione la respuesta correcta:

```
1 | int x = 5;
2 | int y = 5;
3 | y /= ++x;
4 | System.out.println(y);
    - 0
    - 1
    - 5
    - 6
```

### 4. Analice el siguiente código y seleccione la respuesta correcta:

```
1 | int i=2;
2 | int j=3;
3 | int k=2;
4 | System.out.println((j + k < i) || (33 -i >= j));
    - true
    - false
    - undefined
    - Error de compilación
```

### 5. Analice el siguiente código y seleccione la respuesta correcta:

```
1 | int x = 5;
2 | int y = 5;
3 | y *= x++;
4 | System.out.println(y);
    - 6
    - 25
    - 30
    - 35
```

### 6. Analice el siguiente código y seleccione la respuesta correcta:

```
1 | int x = 5;
2 | int y = 5;
3 | y *= x++;
4 | System.out.println(x);
```

- 6
- 25
- 30
- 35

**7. ¿Qué operador tiene mayor precedencia?**

- /, %
- +, -
- &&, ||
- <, >

**8. Analice la siguiente instrucción y seleccione la respuesta correcta:**

- 1 | System.out.println(2 + 6 % 2);
- 2
  - 4
  - 6
  - 8

**9. Analice el siguiente código y seleccione la respuesta correcta:**

- 1 | int x=2;  
 2 | int y=2;  
 3 | int z;  
 4 | z = x++ + y;  
 5 | System.out.println(x);
- 2
  - 3
  - 4
  - 5

**10. Analice el siguiente código y seleccione la respuesta correcta:**

- 1 | int total = 6;  
 2 | int contador = 4;  
 3 | total -= --contador;  
 4 | System.out.println(contador);
- 1
  - 2
  - 3
  - 4

**11. ¿Qué sentencias permiten sumar una unidad a x?**

- x = 1
- x = x + 1;
- x += 1;
- x++;

**12. ¿Qué operador tiene mayor precedencia?**

- ^
- +, -
- ||
- <=

**13. ¿Qué hace la siguiente sentencia de código?**

- 1 | total -= --contador;
- No compila.
  - Decrementa contador a total y luego resta uno a contador.
  - Decrementa uno al contador y luego se lo resta a total.
  - Decrementa uno al contador y luego se lo asigna a total.

**14. Analice el siguiente código y seleccione la respuesta correcta:**

- ```
1 | int i=1;
2 | int j=2;
3 | int k=3;
4 | int m=2;
5 | System.out.println ((j >= i) || (k == m));
```
- true
  - false
  - undefined
  - Error de compilación

**15. ¿Qué realiza la siguiente instrucción?**

```
1 | contador += x;
```

- Suma el valor de x a contador.
- Asigna x al valor de contador.
- Suma uno al valor de x y se lo asigna al contador.
- Ninguna de las tres anteriores es correcta.

**16. ¿Qué realiza la siguiente instrucción?**

```
1 | z = x++ + y;
```

- No es correcta, no compila.
- Suma el valor de X+Y a Z y luego incrementa en uno la X.
- Incrementa en uno el valor de la X y lo suma a Y para asignárselo a Z.
- Suma uno a X y se lo asigna a Z, luego suma y a Z.

**17. ¿Cuál es el resultado del siguiente programa?**

```
1 | int x=1;
2 | System.out.println ( x == 1 );
```

- true
- false
- Error de compilación
- 1
- "x == 1"

**18. La sintaxis del operador ternario es:**

- (expresión\_lógica) ? valor\_verdadero : valor\_falso
- (expresión\_lógica) : valor\_verdadero ? valor\_falso
- (expresión\_lógica) ? valor\_falso : valor\_verdadero
- (expresión\_lógica) :<<valor\_falso>> ? <<valor\_verdadero>>

## 5 ESTRUCTURAS DE CONTROL

---

En este capítulo se realiza una descripción de cada estructura de control que el lenguaje de programación Java soporta. Además se detalla su sintaxis y se realizan ejemplos para conocer su funcionamiento.

### Objetivos

- Identificar las estructuras de control y describir su funcionamiento.
- Verificar la lógica de las estructuras de control y cuáles son sus posibles resultados.
- Realizar ejemplos utilizando cada estructura de control.

### 5.1 INTRODUCCIÓN

Por ahora se ha revisado instrucciones simples o secuenciales, por ejemplo:

- Definición e inicialización de variables.
- Imprimir mensajes a consola.
- Lectura de datos desde teclado.
- Operaciones de suma, resta, etc.

Pero existen otros tipos de instrucciones más complejas que contienen un bloque de sentencias, que permiten controlar la secuencia lógica de los programas, se clasifican en:

- Estructuras condicionales.
- Estructuras repetitivas.

Las estructuras de control pueden ejecutar una o varias instrucciones y se deben definir sus fronteras a través de las llaves abiertas y cerradas “{ }”. Además, no necesitan finalizar con un punto y coma “;”.

## 5.2 ESTRUCTURAS CONDICIONALES O SELECTIVAS

Se utilizan para evaluar una condición o expresión y dependiendo del resultado se realiza determina la acción. Las estructuras que se ofrecen son: “**if( )**” y “**switch( )**”.

### 5.2.1 ESTRUCTURA if - then

Esta estructura está compuesta solo por la parte verdadera del **if**. Evalúa la expresión booleana, si el resultado es verdadero, ejecuta las instrucciones de la parte verdadera, si la condición resultó falsa no se ejecuta ninguna instrucción porque no se ha definido las parte falsa de la estructura.

“**if**” es una palabra reservada. Su sintaxis es la siguiente:

```
if (expresión_booleana) {  
    //instrucciones que se ejecutan si la condición se cumple  
    (true)  
}
```

La siguiente figura detalla las partes que componen la estructura if – then:

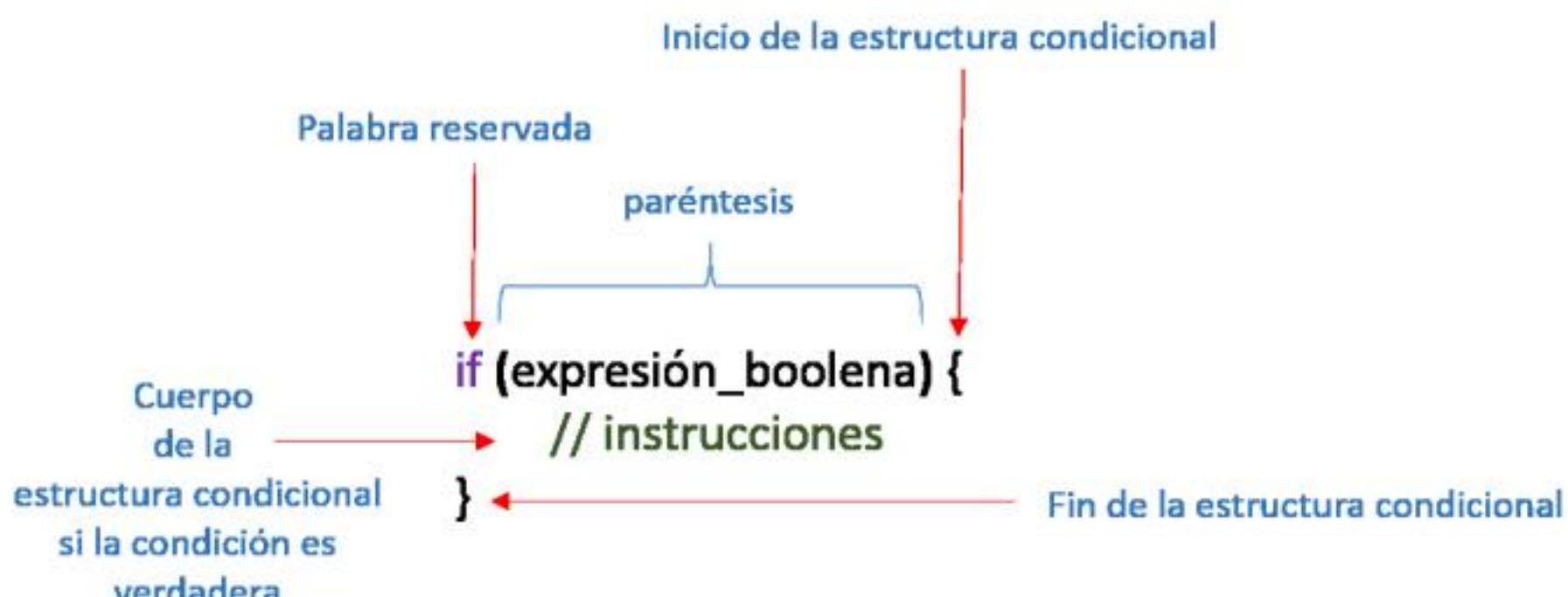


Figura 16 Sintaxis de una estructura selectiva simple

Analizar el siguiente ejemplo:

```
1 public class ProyectoLibroJava {  
2  
3     public static void main(String[] args) {  
4         int a = 15;  
5         if(a > 10){  
6             System.out.println("Es mayor a 10");  
7         }  
8     }  
9 }
```

En este ejemplo en la línea 5 se utiliza una estructura if -then para evaluar una expresión booleana y determinar si un número es mayor a 10, si la evaluación booleana es verdadera, imprime el mensaje “*Es mayor a 10*”, en caso de ser falsa finaliza la estructura if.

### 5.2.2 ESTRUCTURA if / else

Esta estructura está compuesta por dos caminos, el verdadero o falso. Evalúa una expresión booleana, si el resultado es verdadero, ejecuta las instrucciones que se definen en el bloque de **if**, caso contrario se ejecutan las instrucciones definidas en el **else**. La expresión booleana está vinculada con operadores booleanas o relacionales.

“**if**” y “**else**” son palabras reservadas. Su sintaxis es la siguiente:

```
if (expresión_booleana) {  
    //instrucciones que se ejecutan si la condición se cumple (true)  
} else {  
    //instrucciones que se ejecutan si la condición no se cumple  
    (false)  
}
```

En la siguiente figura se detallan las partes que componen la estructura **if/else**.

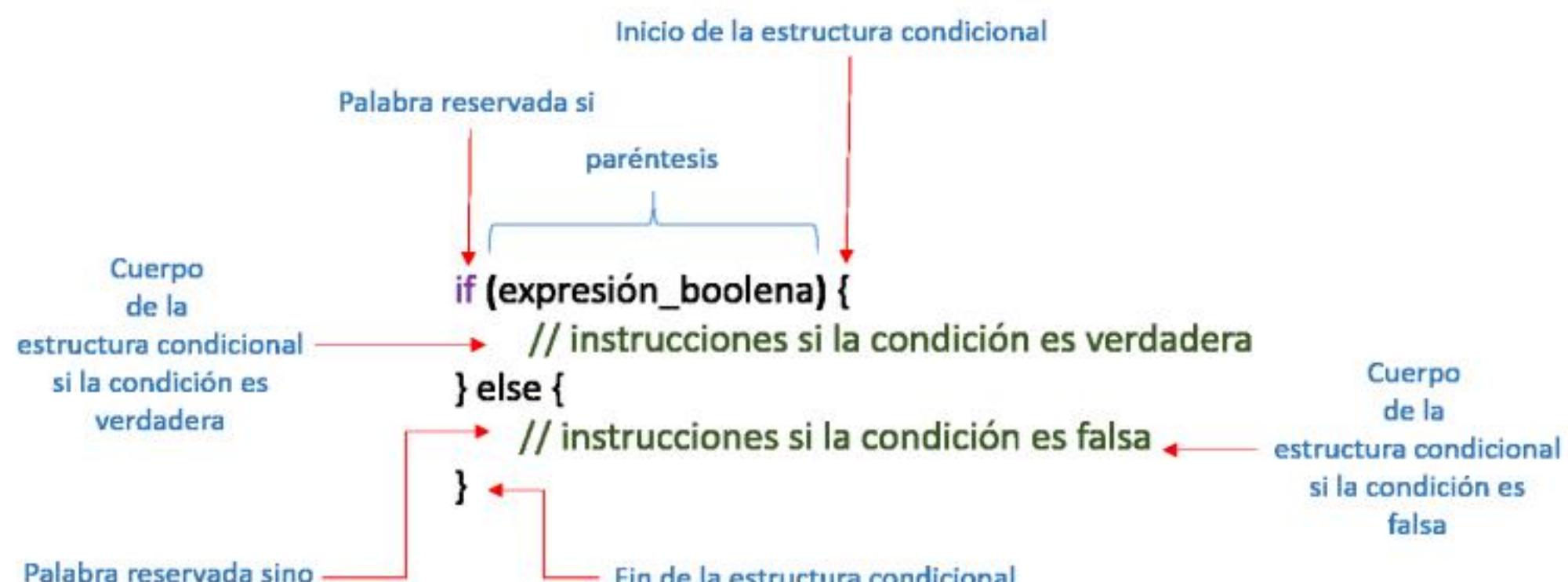


Figura 17 Sintaxis de la estructura selectiva doble

Analizar el siguiente ejemplo:

Ejemplo 33 Estructura Condicional if

```
1 import java.util.Scanner;
2
3 public class EstructuraSelectiva{
4
5     public static void main(String [] args){
6         Scanner lector = new Scanner(System.in);
7         System.out.println("Por favor, ingrese su edad");
8         int edad = lector.nextInt();
9         if(edad >= 18){
10             System.out.println("La persona es mayor de
11             edad");
12         }else{
13             System.out.println("La persona es menor de
14             edad");
15         }
16     }
17 }
18 }
```

En este ejemplo, se desea construir un programa que permita evaluar la edad de una persona para determinar si es mayor o menor de edad. Si la edad es mayor o igual a 18 se dice que es mayor de edad, caso contrario es menor de edad. Este tipo de estructuras solo tiene dos posibles caminos: el verdadero o falso.

En la línea 8, se ingresa un valor desde teclado a través del método nextInt( ) de la clase Scanner y se lo almacena en la variable “edad” de tipo entero.

En la línea 9, la estructura **if** evalúa una expresión booleana (edad >= 18), supóngase que la edad tiene un valor de 20, la expresión queda así (20 >= 18). Al evaluar esta expresión el resultado es verdadera y se ejecuta las instrucciones que están definidas en la sección del **if** y se imprime un mensaje en la consola “La persona es mayor de edad”.

En la misma línea, supóngase que la edad tiene un valor de 17, la expresión queda así (17 >= 18), al evaluar esta expresión el resultado es falso y se ejecutan las instrucciones que están definidas en la sección **else** cuando la condición es falsa. Se imprime un mensaje en consola “*La persona es menor de edad*”.

Otra característica que tiene esta estructura es cuando el bloque de instrucciones solo contiene una instrucción se puede omitir la llave abierta y cerrada “{ }”. Al omitir las llaves, Java asume que la estructura **if** del bloque verdadero o falso contiene una sola instrucción.

Analizar el siguiente ejemplo:

Ejemplo 34 Estructura if con una sola instrucción

```
1 import java.util.Scanner;
2
3 public class EstructuraSelectiva{
4
5     public static void main(String [] args){
6         Scanner lector = new Scanner(System.in);
7         System.out.println("Por favor, ingrese su edad");
8         int edad = lector.nextInt();
9         if(edad >= 18)
10             System.out.println("La persona es mayor de
11             edad");
12         else
13             System.out.println("La persona es menor de
14             edad");
15     }
16 }
```

En la línea 9, se observa que la estructura **if** no se han definido sus fronteras, esto se debe a que cuando los bloques de código solo contienen una sola instrucción se pueden omitir las llaves abiertas y cerradas { }, aunque no es obligatorio.

Sus salidas son:

```
1 \programas_libro>java EstructuraSelectiva
2 Por favor, ingrese su edad
3 8
4 La persona es menor de edad
```

```
1 \programas_libro>java EstructuraSelectiva
2 Por favor, ingrese su edad
3 78
4 La persona es mayor de edad
```

### Recomendación

Las reglas de identación son importantes en los lenguajes de programación, permiten dar elegancia y estilo al código y lo importante es que sea legible, en el ejemplo anterior se ha dado los espacios o sangrías necesarias a las instrucciones para que queden debidamente organizadas. En java los espacios no son parte de las instrucciones.

Ahora analícese un ejercicio que contenga una estructura condicional mixta, un bloque que contenga más de una instrucción y el otro solo una. El objetivo del programa es calcular el salario de las horas extras y presentarlos en pantalla. El

usuario debe ingresar el número de hora trabajadas por el empleado, si sobrepasa las 40 realiza el cálculo del salario extra, sino no realiza ningún cálculo.

Ejemplo 35 Estructura condicional de una y varias instrucciones

```
1 import java.util.Scanner;
2
3 public class SalarioTrabajador {
4
5     public static void main(String [] args){
6         Scanner lector = new Scanner(System.in);
7         int horasExtras = 0, horasLímite = 40;
8         double valorHora = 5, salario = 0;
9         System.out.println("Ingrese el número de horas
10            trabajadas por el trabajador");
11         int horasTrabajadas = lector.nextInt();
12         if(horasTrabajadas > horasLímite){
13             horasExtras = horasTrabajadas - horasLímite;
14             salario = (horasExtras * valorHora);
15             System.out.println("El salario extra recibido por
16               el empleado es "+salario);
17         }else
18             System.out.println("El empleado no tiene horas
19               extras");
20
21     }
22 }
```

En la línea 7 y 8, se declaran e inicializan las variables “horasExtras”, “horasLímite”, “valorHora” y “salario”. En la línea 11 se define una estructura condicional **if-else**, se evalúa la expresión booleana (horasTrabajadas > horasLímite), se tiene dos alternativas:

- **La primera:** si las horas trabajadas sobrepasan las 40, se ejecutan las tres instrucciones el bloque verdadero, en este caso el programador está obligado a definir las fronteras a través de las llaves de apertura y cierre “{ ”}. Si no se definen las fronteras, se genera un error de compilación al quedar desconectada las sentencias de la parte falsa. El siguiente ejemplo se ha omitido las fronteras en el parte verdadera del **if-else**.

```
10 if (horasTrabajadas > horasLímite)
11     horasExtras = horasTrabajadas - horasLímite;
12     salario = (horasExtras * valorHora);
13     System.out.println("El salario extra recibido por el
14       empleado es "+salario);
15 else
16     System.out.println("El empleado no tiene horas extras");
```

Al compilar genera un error que describe que el bloque falso no está definida en la estructura condicional:

```
1 \programas_libro>javac SalarioTrabajador.java
2 SalarioTrabajador.java:15: error: 'else' without 'if'
3     else
4         ^
5 1 error
```

*Salida 1 Ejemplo 30*

- **La segunda:** si las horas trabajadas no sobrepasan las 40, se define una sola instrucción (línea 16) por ello se ha omitido las llaves de apertura y cierre y es perfectamente legal. En este caso Java solo considera que existe una sola instrucción.

### 5.2.3 ESTRUCTURAS CONDICIONALES ANIDADOS

Todas las estructuras condicionales o repetitivas forman un bloque de instrucciones que según la necesidad de los algoritmos se puede anidar o no otras estructuras de control.

El siguiente ejemplo permite evaluar el mayor de tres números:

*Ejemplo 36 Algoritmo mayor de tres números*

```
1 import java.util.Scanner;
2
3 public class MayorTresNumeros{
4     public static void main(String [] args){
5         Scanner lector = new Scanner(System.in);
6         System.out.println("Ingrese el primer número");
7         int primero = lector.nextInt();
8         System.out.println("Ingrese el segundo número");
9         int segundo = lector.nextInt();
10        System.out.println("Ingrese el tercer número");
11        int tercero = lector.nextInt();
12
13        if(primer > segundo && primero > tercero){
14            System.out.println("Número mayor es = " +
15            primero);
16        }else if(segundo > tercero){
17            System.out.println("Número mayor es = " +
18            segundo);
19        }else{
20            System.out.println("Número mayor es = " +
21            tercero);
22        }
23    }
24 }
```

En la línea 13, se define una estructura **if-else**, se evalúa la condición (primero > segundo && primero > tercero), se tiene dos alternativas:

- **Alternativa verdadera:** no existe anidación porque solo compone de una instrucción secuencial.
- **Alternativa falsa:** se anida otra estructura condicional y su sintaxis es perfectamente legal porque todas las estructuras de forma implícitas son instrucciones. Al aplicar una anidación se establece jerarquía de código.

Su salida es:

```
1 \programas_libro>java MayorTresNumeros
2 Ingrese el primer número
3 7
4 Ingrese el segundo número
5 8
6 Ingrese el tercer número
7 6
8 El número mayor es = 8
```

#### 5.2.4 OPERADOR TERNARIO

Se lo conoce como **if** de una línea u operador condicional. Para poder utilizar el operador ternario, la estructura condicional debe contener una sola instrucción tanto en el bloque verdadero como falso. Se lo representa con el carácter "?".

Su sintaxis es:

```
(expresión_boolena) ? valor_verdadero : valor_falso;
```

En la siguiente figura se ilustra los elementos que componen al operador ternario:

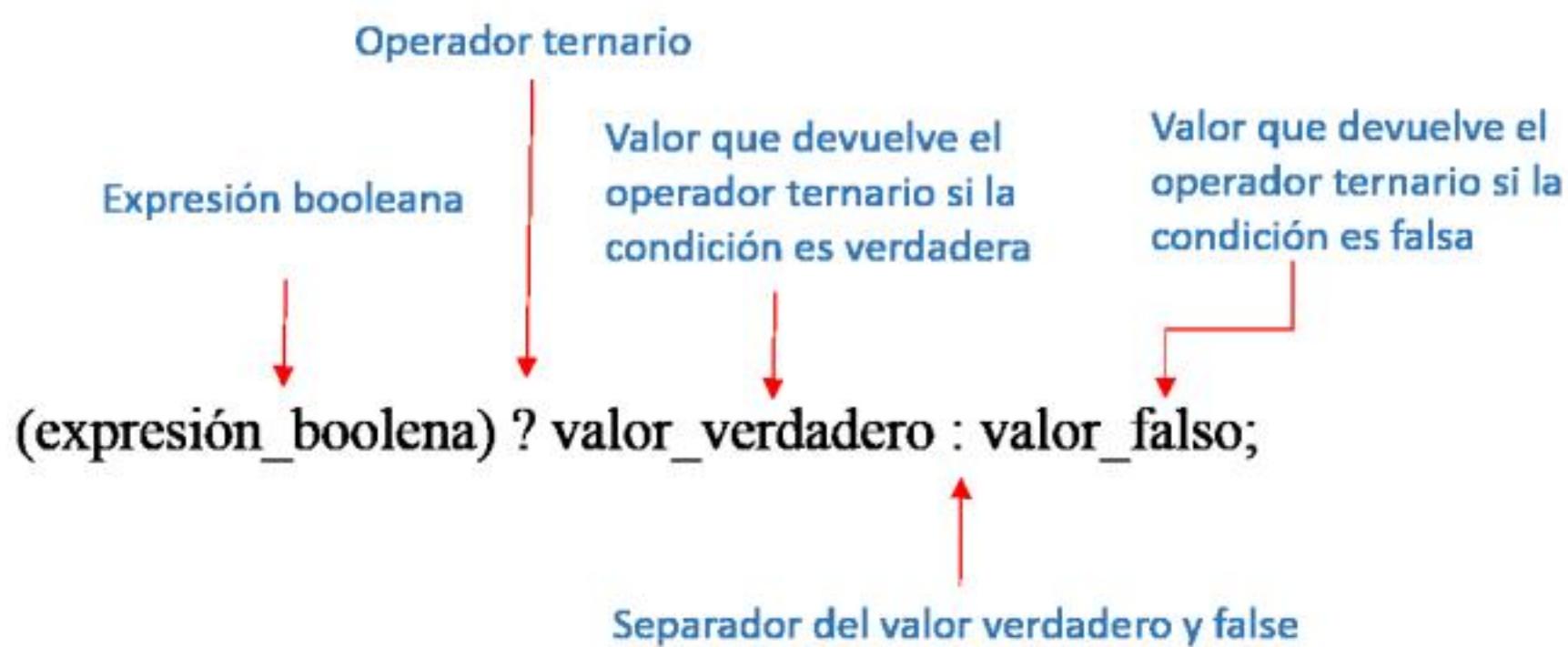


Figura 18 Sintaxis del operador ternario

La condición al ser evaluada devuelve un valor booleano falso o verdadero, si devuelve verdadero entonces se ejecuta la sección del valor verdadero y caso contrario se ejecuta el valor falso. El resultado de la operación con el operador ternario se lo puede almacenar a una variable. Analizar el siguiente ejemplo:

Ejemplo 37 Operador ternario

```
1 public class OperadorTernario{  
2     public static void main(String [] args){  
3         int x = 8, y = 4, mayor;  
4         /*  
5             if (x > y)  
6                 mayor = x;  
7             else  
8                 mayor = y;  
9             */  
10            mayor = (x > y) ? x : y;  
11            System.out.println("El número mayor es: " + mayor);  
12        }  
13    }
```

En este ejemplo, desde la línea 4 a la 9 se ha comentado el código de una estructura condicional **if - else**, lo que se pretende es cambiar el código de la estructura **if** por una operador ternario. El primer paso se debe analizar si la estructura condicional cumple con un requisito básico, que tanto en el bloque verdadero y falso solo contengan una sola instrucción.

Superado este requisito, en la línea 10 se define usa sentencia usando el operador ternario y se almacena en una variable de tipo entero llamada “mayor”, la primera parte es evaluar la expresión booleana ( $x > y$ ), la segunda y tercera parte son los valores que devuelve el operador al evaluar la condición. Para separar el bloque verdadero y el false se utiliza el carácter “:”.

El resultado es:

```
1 \programas_libro> java OperadorTernario  
2 El número mayor es: 8
```

Para hacer legible la definición del operador ternario se recomienda poner a la condición entre paréntesis, aunque no sea necesario.

Ahora analizar el error común que se generan al utilizar este operador en el siguiente ejemplo.

### Ejemplo 38 Operador ternario

```
1 public class OperadorTernario{  
2     public static void main(String [] args){  
3         int x = 8, y = 4, mayor;  
4         mayor = (x > y) ? x : "Es menor";  
5         System.out.println("El número mayor es: " + mayor);  
6     }  
7 }
```

En la línea 4 se define un operador ternario y por concepto las expresiones de la parte verdadera y falsa deben ser del mismo tipo de datos, caso contrario, generará un error de compilación. En este caso, en la parte verdadera devuelve “x” que es una variable de tipo entera, en la parte falsa devuelve una cadena de texto.

### ¿Dónde se genera el error?

El error se genera al tratar de asignarle un valor de la variable “mayor” que no le corresponde. Si la condición resulta verdadera el tipo de dato que se devuelve es entero y si es falso se devuelve una cadena, allí se genera el problema, porque se le está asignando una cadena a una variable de tipo entero, dos tipos de datos totalmente diferentes.

El resultado es:

```
1 \programas_libro>javac OperadorTernario.java  
2 OperadorTernario.java:4: error: incompatible types: bad type in conditional expression  
3     mayor = (x > y) ? x : "es menor";  
4                     ^  
5     String cannot be converted to int  
6 1 error  
7
```

### 5.2.5 ESTRUCTURA switch

Es una sentencia condicional de selección múltiple que evalúa el valor de una variable o expresión de un tipo entero (`byte`, `short` o `int`), carácter, cadena o enumerada, pero no real. Cada caso se define dentro del `switch` y dependiendo de la expresión ejecuta las instrucciones definido en uno de ellos. Para finalizar cada caso se utiliza la palabra reservada `break`. Si no se define la sentencia `break`, el programa salta a la sección de `default` y ejecuta las instrucciones definidas allí.

Al evaluar una expresión y no se encuentra definida en el **switch** se ejecuta las instrucciones que están en el **default**, aunque esta sección no es obligatoria y se omite la utilización de la sentencia **break**.

Su sintaxis es la siguiente:

```
switch (expresión) {  
    case valor1:  
        //sentencias caso valor1;  
        break;  
    case valor2:  
        //sentencias caso valor 2;  
        break;  
    ...  
    case valorN:  
        //sentencias caso valor N;  
        break;  
    default:  
        //se ejecutan si el valor evaluado no está en el switch  
}
```

En esta estructura de control se utilizan las siguientes palabras reservadas:

- **switch**: define la estructura de múltiple selección.
- **case**: define los casos de la estructura **switch**, y define las instrucciones que se ejecutarán dependiendo del valor de la expresión.
- **break**: termina el caso y hace que finalice la estructura **switch**.
- **default**: al evaluar la expresión no se encuentra definida en los casos se ejecuta en la zona por defecto.

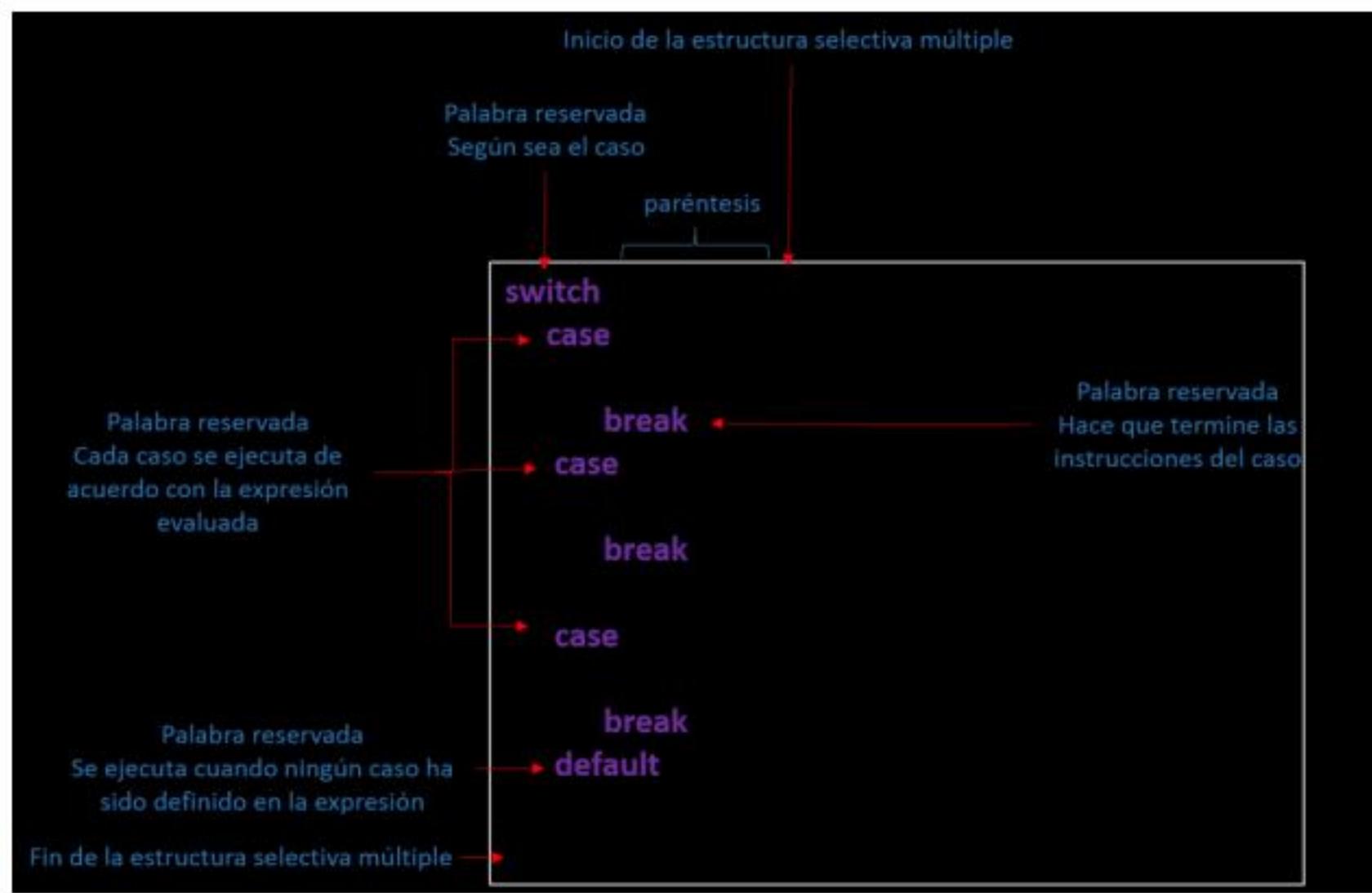


Figura 19 Sintaxis de la estructura de selectiva múltiple

En el siguiente ejemplo se evalúa el valor de la variable “día” en una estructura múltiple, si está definida en algún caso se ejecutarán determinadas instrucciones, cuando no se encuentra definido, salta a la sección `default`.

Ejemplo 39 Estructura switch

```

1 import java.util.Scanner;
2
3 public class EstructuraSwitch{
4     public static void main(String [] args){
5         int opcion = 3;
6         String dia = "";
7         switch (opcion){
8             case 1:
9                 dia = "Lunes";
10                break;
11            case 2:
12                dia = "Martes";
13                break;
14            case 3:
15                dia = "Miércoles";
16                break;
17            case 4:
18                dia = "Jueves";
19                break;
20            case 5:
21                dia = "Viernes";
22                break;
23            default:
24                dia = "No es un día laborable";
25        }
26        System.out.println(dia);
27    }
28 }
```

Su resultado es:

```
1 programas_libro> java EstructuraSwitch  
2 Miércoles
```

Ahora analizar el siguiente ejemplo donde no se usa la instrucción **break**:

*Ejemplo 40 case sin break*

```
1 import java.util.Scanner;  
2  
3 public class EstructuraSwitch{  
4  
5     public static void main(String [] args){  
6         int opcion = 1;  
7         String dia = "";  
8         switch(opcion){  
9             case 1:  
10                dia = "Lunes";  
11            case 2:  
12                dia = "Martes";  
13            case 3:  
14                dia = "Miércoles";  
15            default:  
16                dia = "No es un día de la semana";  
17        }  
18        System.out.println(dia);  
19    }  
20 }
```

En este ejercicio, en todos los casos definidos en la estructura **switch** se ha omitido la sentencia **break**, su sintaxis es totalmente correcta pero cambiará el significado en la ejecución del 'programa. Por ejemplo, si la "opción" tiene un valor de 1, se ubica en la **case 1**, se le asigna el valor de "Lunes" a "día", pero al no encontrar la instrucción **break**, salta a la sección "**default**" y ejecuta todas sus instrucciones definidas en esta sección, el valor de "día" se modifica por "No es un día de la semana", cambiando totalmente el significado de la variable "día". Lo mismo pasará con todos los casos que no hayan invocado a la sentencia **break**.

La sentencia **break** es importante porque pone fin al **case** y lo obliga a terminar a la estructura **switch**.

El resultado es:

```
1 programas_libro> java EstructuraSwitch  
2 No es un día de la semana
```

En java 7 la estructura **switch** soporta datos de tipo cadena *String* y las clases que representan a los datos primitivos.

Los tipos de datos que soporta la estructura `switch` son:

- `int` y *Integer*.
- `byte` y *Byte*.
- `short` y *Short*.
- `char` y *Character*.
- `String`.
- Datos de tipo enumerado.

## 5.3 ESTRUCTURAS REPETITIVAS

Este tipo de estructura permite ejecutar de forma repetida un bloque específico de instrucciones. Los bucles ejecutan un conjunto de instrucciones de forma reiterada mientras o hasta se cumpla una determinada condición.

En Java hay tres tipos diferentes de bucles:

- `for`
- `while`
- `do-while`

### 5.3.1 ESTRUCTURA for

Es una estructura repetitiva, se la utiliza cuando el número de iteraciones es conocido y es controlado por una variable contadora. Además es usado para recorrer estructuras de datos lineales como los arreglos, listas, pilas, colas, etc. Su sintaxis es:

```
for (inicialización; condición; incremento/decremento)
{
    // instrucciones;
}
```

La palabra reservada `for` establece una estructura condicional “para”, se definen tres zonas o instrucciones dentro del ciclo, cada una está separada por “;” y son:

- **Zona de inicialización:** se define la expresión inicial y se ejecuta una sola vez antes de evaluar la primera iteración, se define e inicializa el contador del **for**.
- **Zona de condición:** se define una expresión booleana. Al evaluar esta expresión resulta verdadera se ejecuta las instrucciones que están dentro del **for**, caso contrario no ejecuta ninguna instrucción más y termina su función. Por cada iteración, el ciclo **for** evalúa la expresión booleana antes de ejecutar las instrucciones.
- **Zona de Incremento o decremento:** cuando se ejecutan todas las instrucciones que están dentro del **for**, se incrementa o decrementa el valor del contador.

La estructura **for** debe definir sus fronteras de inicio y finalización, se las realiza a través de la llave de apertura y la llave de cierre “{ }”.

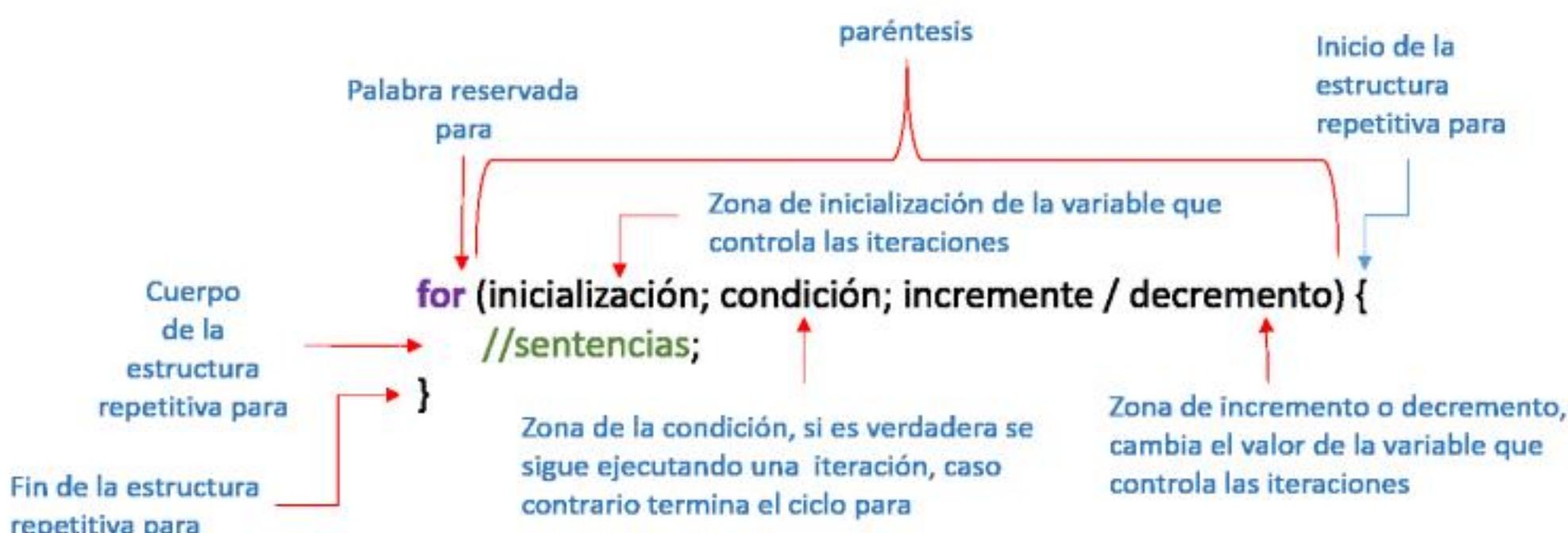


Figura 20 Sintaxis de la estructura repetitiva para

Analizar el siguiente ejemplo:

Ejemplo 41 Estructura for

```
1 public class EstructuraPara {
2     public static void main(String [] args){
3         int limite = 10;
4         for(int i = 0; i < limite; i++){
5             System.out.println("Iteración " + i);
6         }
7     }
8 }
```

En este ejercicio, en la línea 4, en la zona de inicialización, la variable contador “i” se define e inicializa con un valor 0 y solo se ejecutará al iniciar el ciclo **for**. En la

zona de la condición se define una expresión booleana ( $i < \text{límite}$ ), como “ $i$ ” vale 0 y “ $\text{límite}$ ” vale 10, entonces  $0 < 10$ , la condición se cumple y se ejecutan las instrucciones definidas en el ciclo **for** e imprime un mensaje. Al terminar de ejecutar las sentencias en cada iteración en la zona de incremento, la variable contador incrementa su valor en uno.

Las iteraciones irán entre 0 a 9. Cuando el valor del contador es 10 la expresión será  $(10 < 10)$  y al evaluarla dará como resultado un valor falso, en ese momento al no cumplir la condición, no ejecuta ninguna instrucción dentro de la estructura **for**.

El resultado generado es:

```
1  \programas_libro> java EstructuraPara
2  Iteración 0
3  Iteración 1
4  Iteración 2
5  Iteración 3
6  Iteración 4
7  Iteración 5
8  Iteración 6
9  Iteración 7
10 Iteración 8
11 Iteración 9
```

Ahora analizar el siguiente código:

Ejemplo 42 Estructura for con una sola instrucción

```
1  public class EstructuraPara {
2      public static void main(String [] args){
3          int limite = 10;
4          for(int i = 0; i < limite; i++)
5              System.out.println("Iteración " + i);
6      }
7  }
```

En la línea 4 se define la estructura del bucle **for** sin sus fronteras, esto se debe a que sola contiene una instrucción, en Java se omite las llaves de apertura y cierre “{ ”. Por consiguiente es una definición legal. Pero cuando define más de una sentencia en el bucle **for**, el programador está obligada en definir sus fronteras.

### 5.3.1.1 Bucles Infinitos

Al definir una estructura repetitiva es necesario tener en cuenta que se pueden generar bucle infinitos, por ejemplo cuando la condición siempre resultará verdadera o la variable contadora no incrementa su valor.

En el siguiente ejemplo es un claro caso de bucle infinito:

*Ejemplo 43 Bucle for infinito*

```
1 public class EstructuraPara {  
2     public static void main(String [] args){  
3         int limite = 10;  
4         for(int i = 0; i < limite;)  
5             System.out.println("Iteración " + i);  
6     }  
7 }
```

En la línea 4 se ha definido la sección de inicialización con “int i = 0”, la zona de la condición evalúa la expresión booleana (*i < limite*), claramente se ha definido sus dos primeras zonas pero se ha omitido la sentencia de incremento o decremento, con ello la variable contador jamás incrementará o decrementará su valor, siempre tendrá un valor de 0 y se generará un bucle infinito.

Otro ejemplo de bucle infinito es:

*Ejemplo 44 Bucle infinito, sin definir las zona del ciclo for*

```
1 public class EstructuraPara {  
2     public static void main(String [] args){  
3         int limite = 10;  
4         for(;;)  
5             System.out.println("Iteración ");  
6     }  
7 }
```

En este ejemplo, en la línea 4 ninguna zona ha sido implementada, a simple vista pareciera un error de compilación o ejecución, de hecho se compilará y ejecutará sin ningún problema porque es un bucle definido de forma legal con “;”. Las zonas en un bucle **for** son obligatorias aunque no se las implemente como en ejemplos anteriores.

### 5.3.1.2 Términos múltiples en la declaración del bucle for

En el siguiente código se definen variante en un ciclo **for** clásico.

*Ejemplo 45 Variante de declaraciones en un for*

```
1 public class EstructuraPara {  
2     public static void main(String [] args){  
3         int j = 10;  
4         for(long i = 0, h = 10; i < 10 ; i++, j--) {  
5             System.out.print(i + " ");  
6         }  
7         System.out.print("\nj = "+j + " ");  
8     }  
9 }
```

En este ejemplo, existen algunas variante con respecto a un ciclo **for** convencional. En la línea 4, la primera variante está ubicada en la zona de inicialización, se puede observar que se definen e inicializan dos variables ( “i” y “j”), en Java estas variantes son legales, siempre y cuando las definiciones sean del mismo tipo de datos, si se definen con dos diferentes tipos de datos, al momento de compilar generará un error.

La segunda variante se define en la zona de incremento/decremento, se puede definir varias instrucciones como “i++” y “j++”, estas siempre deben estar separadas por una “,” y al final de cada iteración las instrucciones definidas en la zona de incremento se ejecutan.

El resultado generado es:

```
1 \programas_libro>java EstructuraPara  
2 0 1 2 3 4 5 6 7 8 9  
3 j = 0
```

### 5.3.1.3 BUCLE FOR ANIDADOS

La anidación de estructuras de control es un proceso normal pueden implementar en todas las sentencias de control: selectivas o repetitivas. Los bucles anidados son aquellos que incluyen otros ciclos **for** dentro del cuerpo del ciclo **for** base. Las variables de inicialización deben ser diferentes.

El siguiente ejemplo genera las tablas de multiplicar y se ilustra las anidación de ciclos **for**:

*Ejemplo 46 Anidación de bucles for*

```
1 public class TablaMultiplicacion{  
2     public static void main(String [] args){  
3         for (int i = 1; i < 12; i++){  
4             for (int j = 1; j < 12; j++){  
5                 System.out.println(i + " * " + j + " = " + (i *  
6                             j));  
7             }  
8         }  
9     }  
10 }
```

Las variables de inicialización de los ciclos **for** tanto en la línea 3 o 4 son diferentes para evitar duplicidad de las mismas. Esta definición de anidación es legal.

### 5.3.2 ESTRUCTURA while

Es un bucle o sentencia repetitiva que evalúa una condición antes de iniciar la iteración. Para definir esta estructura se utiliza la palabra reservada **while**.

Al evaluar la expresión booleana y su resultado es verdadera, se ejecutan las instrucciones definidas dentro del ciclo **while**. Si la condición es falsa ya no ejecutará ninguna iteración. Se debe definir sus fronteras a través de las llaves de apertura y de cierre “{ }”.

Su sintaxis es:

```
while (expresión_booleana) {  
    //sentencias;  
}
```

La siguiente figura ilustra las partes que componen una estructura repetitiva **while**.

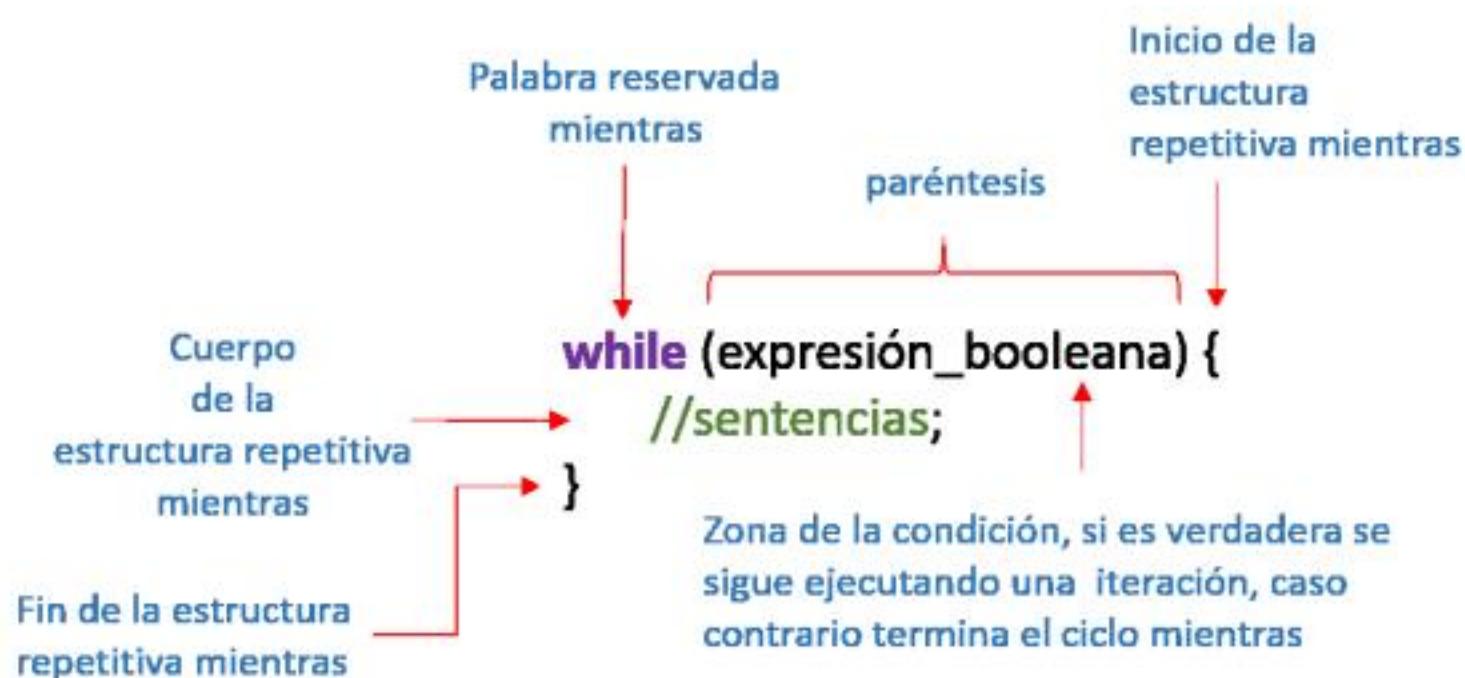


Figura 21 Estructura de una estructura repetitiva while

Analizar el siguiente ejemplo:

Ejemplo 47 Estructura while

```

1 public class EstructuraMientras {
2     public static void main(String [] args){
3         int contador = 1, factorial = 1;
4         while (contador <= 5) {
5             factorial *= contador;
6             contador++;
7         }
8         System.out.print("Factorial es " + factorial);
9     }
10 }
```

El objetivo del siguiente ejemplo es calcular el factorial de 5. En la línea 3, se define e inicializar una variable “contador” con el valor de 1, esta permite controlar las iteraciones en el ciclo, y es evaluada en la expresión booleana (contador <= 5), en otras palabras se evalúa  $1 \leq 5$ , la expresión da como resultado verdadero y se ejecuta las instrucciones definidas en el cuerpo del ciclo **while**. En la línea 6 se modifica “contador” a través de un operador de incremento y para ser evaluada en la siguiente iteración.

Como en los anteriores casos, de bucle **while** está compuesto por una sola instrucción se puede omitir las llaves de apertura y cierre { }.

Analizar el siguiente ejemplo:

*Ejemplo 48 Tabla de multiplicar con ciclo while*

```
1 public class TablaMultiplicacion{  
2     public static void main(String [] args){  
3         int serie = 5, limite = 12, contador = 1;  
4         while(contador <= limite){  
5             System.out.println(serie + " * " + contador + " = "  
6                         + (serie * contador));  
7             contador++;  
8         }  
9     }  
10 }
```

En la línea 3, se definen e inicializan tres variables de tipo entero, serie, límite y contador. En la línea 4, se declara un ciclo **while** y están controlados por una expresión de tipo booleana (contador <= límite), contador controla las iteraciones del bucle, dependiendo de su valor al evaluarlo se ejecutará o no las instrucciones. En la línea 5, por cada iteración se imprime la multiplicación entre la serie y el contador.

En la línea 6, la variable contador modifica, incrementa su valor en uno por cada iteración.

El resultado generado es:

```
1 \programas_libro>java TablaMultiplicacion  
2 5 * 1 = 5  
3 5 * 2 = 10  
4 5 * 3 = 15  
5 5 * 4 = 20  
6 5 * 5 = 25  
7 5 * 6 = 30  
8 5 * 7 = 35  
9 5 * 8 = 40  
10 5 * 9 = 45  
11 5 * 10 = 50  
12 5 * 11 = 55  
13 5 * 12 = 60
```

### 5.3.3 ESTRUCTURA do while

Es un bucle o ciclo repetitivo con una condición al final. En este ciclo al menos una sentencia se ejecuta y luego evalúa la expresión booleana al final, mientras sea cierta una condición se ejecuta otro ciclo y cuando la expresión booleana es falsa deja de ejecutar las instrucciones.

Su sintaxis es:

```
do {  
    //instrucciones;  
} while (expresión_booleana);
```

En la siguiente figura se ilustra las partes que componen a la estructura repetitiva **do while**.

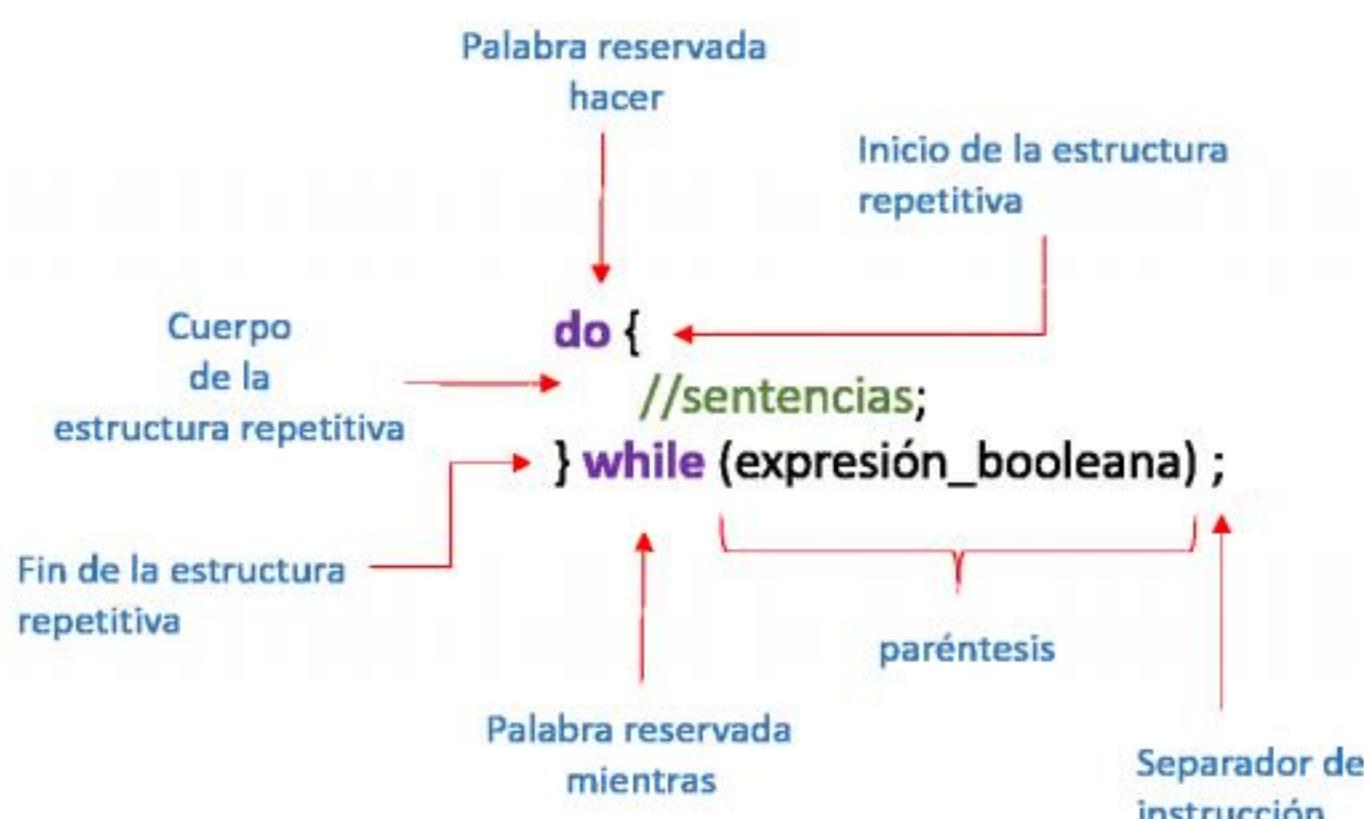


Figura 22 Estructura de la estructura repetitiva hasta que

Analizar el siguiente ejemplo:

Ejemplo 49 bucle do while

```
1 public class EstructuraHacerMientras {  
2     public static void main(String [] args){  
3         int i = 1;  
4         System.out.println("Numeros del 1 al 10: ");  
5         do{  
6             System.out.println(i);  
7             i++;  
8         }while(i <= 10);  
9     }  
10 }
```

En la línea 5, inicia el ciclo y se ejecutan las instrucciones definidas en las líneas 6 y 7 y al final evalúa la expresión booleana ( $i \leq 10$ ), cuando la expresión booleana es

verdadera, inicia de nuevo otra iteración, cuando la expresión es falsa termina el ciclo **do while**.

### 5.3.4 SENTENCIAS DE SALIDA DE UN BUCLE

En Java existen dos sentencias que nos permiten forzar la salida de un bucle, y son:

- **break**: permite salir completamente del bucle.
- **continue**: permite salir de la iteración actual y saltar directamente a la siguiente.

Estas sentencias modifican el comportamiento de los ciclos y se recomienda evitar su uso ya que produce malos hábitos al programar.

#### 5.3.4.1 SENTENCIA break

La sentencia **break** se lo trabajó en la estructura **switch** que permitía finalizar cada **case**. En las estructuras repetitivas hacen que termine inmediatamente el bucle repetitivo, todas las instrucciones que estén seguidas de la sentencias **break** no se ejecutarán. Además es una palabra reservada de Java.

En el objetivo del siguiente ejemplo es imprimir un mensaje siete, analicemos su comportamiento:

Ejemplo 50 Sentencia break

```
1 public class SentenciaBreak {  
2     public static void main(String[] args) {  
3         for (int i = 1; i <= 7; i++) {  
4             if (i == 4){  
5                 System.out.println("Dia " + i + ":"  
6                     " Descansando");  
7                 break;  
8             }  
9             System.out.println("Dia " + i + ": Trabajando ");  
10        }  
11    }  
12 }
```

En la línea 4 se define una estructura **if** que evalúa la (condición “*i == 4*”), cuando la expresión booleana es verdadera imprime el mensaje “Dia 4: Descansando”, seguido ejecuta la sentencia “**break**” haciendo que termine de forma automática el

ciclo **for** independientemente si la condición del ciclo **for** sigue siendo verdadera. En este caso, la iteración 5, 6 y 7 no se ejecutarán.

### El resultado es:

```
1 \programas_libro> java SentenciaBreak
2 Dia 1: Trabajando
3 Dia 2: Trabajando
4 Dia 3: Trabajando
5 Dia 4: Descansando
```

Normalmente la sentencia **break** termina de forma automática al ciclo repetitivo que lo contiene, pero que pasa si el ciclo está anidado, en el siguiente código imprime las series de las tablas de multiplicar del 1 al 12 y analizaremos el comportamiento de la sentencia **break**:

*Ejemplo 51 sentencia break en un ciclo anidado*

```
1 public class SentenciaBreak {
2     public static void main(String[] args) {
3         for (int i = 1; i <= 12; i++) {
4             for (int j = 1; j <= 12; j++) {
5                 if (i == 6){
6                     break;
7                 }
8                 System.out.println(i + " * " + j + " = " +
9                         (i*j));
10            }
11        }
12        System.out.println("Salimos del bucle...");
```

En este ejemplo, existen dos estructuras repetitivas anidadas. En la línea 6 se ha definido una sentencia **break** y se ubica en el segundo ciclo, ahora al momento de ejecutar el algoritmo, imprime las series del 1 al 5, pero en la línea 5 al evaluar la condición (*i == 6*) y resulte la condición verdadera, se ejecuta la sentencia **break** y termina automáticamente el segundo ciclo **for**, pero el primer ciclo continúa de forma normal ejecutando cada iteración e imprime la serie de 7 al 12.

En el ejemplo anterior sólo se detuvo un ciclo repetitivo, que pasaría si quiere detener ambos ciclos.

## ¿Cómo detener los dos ciclos de forma directa?

Java puede trabajar a la sentencia **break** como etiqueta. Una etiqueta debe ir colocada justo antes del bucle al que va etiquetando y se forma con un identificador válido seguido de dos puntos ":". También son útiles cuando se tiene bucles anidados y se quiere especificar en cuál de ellos se debe hacer un **break**. Se sigue la siguiente sintaxis:

*etiqueta\_break : estructura  
repetitiva*

En el siguiente ejemplo se explica el uso de la etiqueta **break**:

*Ejemplo 52 Etiqueta break*

```
1 public class SentenciaBreakEtiqueta {  
2     public static void main(String[] args) {  
3         etiquetaFor : for (int i = 1; i <= 12; i++) {  
4             for (int j = 1; j <= 12; j++) {  
5                 if (i == 6){  
6                     break etiquetaFor;  
7                 }  
8                 System.out.println(i + " * " + j + " = " +  
9                             (i*j));  
10            }  
11        }  
12        System.out.println("Salimos del bucle...");  
13    }  
14 }
```

El programa comienza con la ejecución en el primer bucle **for** "i" ignorando la etiqueta "etiquetaFor", seguido entra al segundo bucle anidado "j". Realiza la primera iteración y evalúa la condición (*i == 6*), cuando el primer ciclo esté en la iteración 6, la estructura **if** del segundo ciclo devolverá **true**, ejecuta la sentencia **break** con etiqueta y termina todos los ciclo donde ha sido etiquetada la sentencia **break**, provocando que los dos ciclo **for** dejen de ejecutar iteraciones. Las series del 6 al 12 no se ejecutan.

La salida del programa es:

|   |                                      |           |           |            |            |
|---|--------------------------------------|-----------|-----------|------------|------------|
| 1 | \programas_libro>java SentenciaBreak |           |           |            |            |
| 2 | 1 * 1 = 1                            | 2 * 1 = 2 | 3 * 1 = 3 | 4 * 1 = 4  | 5 * 1 = 5  |
| 3 | 1 * 2 = 2                            | 2 * 2 = 4 | 3 * 2 = 6 | 4 * 2 = 8  | 5 * 2 = 10 |
| 4 | 1 * 3 = 3                            | 2 * 3 = 6 | 3 * 3 = 9 | 4 * 3 = 12 | 5 * 3 = 15 |

|    |                      |             |             |             |             |
|----|----------------------|-------------|-------------|-------------|-------------|
| 5  | 1 * 4 = 4            | 2 * 4 = 8   | 3 * 4 = 12  | 4 * 4 = 16  | 5 * 4 = 20  |
| 6  | 1 * 5 = 5            | 2 * 5 = 10  | 3 * 5 = 15  | 4 * 5 = 20  | 5 * 5 = 25  |
| 7  | 1 * 6 = 6            | 2 * 6 = 12  | 3 * 6 = 18  | 4 * 6 = 24  | 5 * 6 = 30  |
| 8  | 1 * 7 = 7            | 2 * 7 = 14  | 3 * 7 = 21  | 4 * 7 = 28  | 5 * 7 = 35  |
| 9  | 1 * 8 = 8            | 2 * 8 = 16  | 3 * 8 = 24  | 4 * 8 = 32  | 5 * 8 = 40  |
| 10 | 1 * 9 = 9            | 2 * 9 = 18  | 3 * 9 = 27  | 4 * 9 = 36  | 5 * 9 = 45  |
| 11 | 1 * 10 = 10          | 2 * 10 = 20 | 3 * 10 = 30 | 4 * 10 = 40 | 5 * 10 = 50 |
| 12 | 1 * 11 = 11          | 2 * 11 = 22 | 3 * 11 = 33 | 4 * 11 = 44 | 5 * 11 = 55 |
| 13 | 1 * 12 = 12          | 2 * 12 = 24 | 3 * 12 = 36 | 4 * 12 = 48 | 5 * 12 = 60 |
| 14 | Salimos del bucle... |             |             |             |             |

### 5.3.4.2 SENTENCIA continue

Contrario al concepto de **break** que forzaba finalización de los ciclo en las estructuras repetitivas, la sentencia **continue** hace que se detenga la ejecución de la iteración actual y saltar a la siguiente iteración. Además es una palabra reservada y debe ser definida dentro de un bucle.

En el siguiente ejemplo se ilustra el uso de la sentencia **continue**:

Ejemplo 53 Sentencia continue

```

1 public class SentenciaContinue {
2     public static void main(String[] args) {
3         for (int i = 1; i <= 7; i++) {
4             if (i == 2 || i == 3){
5                 continue;
6             }
7             System.out.println("Dia "+ i + ": Trabajando ");
8         }
9     }
10 }
```

En la línea 3, se inicia la primera iteración, y empieza evaluando la condición en la estructura selectiva en la línea 4, cuando el contador del ciclo *i*, tenga el valor de 4, la condición del **if** será verdadera ingresando y ejecutando la sentencia **continue**, esta sentencia no forza al ciclo a terminar sino ignora la iteración actual y no permite que se sigan ejecutando las instrucciones que están debajo de ella y continúa con la siguiente iteración.

La salida del programa es:

```

1 programas_libro>java SentenciaContinue
2 Dia 1: Trabajando
3 Dia 2: Trabajando
4 Dia 3: Trabajando
```

5 Dia 5: Trabajando  
6 Dia 7: Trabajando

Sentencia **continue** como etiqueta.

Normalmente la instrucción **continue** afecta sólo a la estructura repetitiva que lo contenía, en ciclos anidados el concepto es el mismo pero con la diferencia es que según donde se ubique la etiqueta, se generará el salto de iteración. Las etiquetas son válidas solo en estructuras de control repetitivos.

Se sigue la siguiente sintaxis:

etiqueta\_continue : estructura  
repetitiva

En el siguiente ejemplo se tiene un ejemplo de la sentencia **continue** como etiqueta:

*Ejemplo 54 Etiqueta continue*

```
1 public class SentenciaContinue {  
2     public static void main(String[] args) {  
3         etiquetaFor : for (int i = 1; i <= 4; i++) {  
4             for (int j = 1; j <= 12; j++) {  
5                 if (i == 2 || i == 3){  
6                     continue etiquetaFor;  
7                 }  
8                 System.out.println(i + " * " + j + " = " + (i*j)  
9                         + " ");  
10            }  
11        }  
12        System.out.println("Salimos del bucle...");  
13    }  
14 }
```

El objetivo del ejemplo es imprimir las series de las tablas de multiplicar del 1 al 4, al usar el etiqueta **break** se fuerza a terminar el ciclo donde se ubica la etiqueta, en cambio al utilizar **continue** como etiqueta solo se ignorará las iteraciones 2 y 4 del primer ciclo. Después de que se ejecute la sentencia **continue**, el flujo de ejecución continúa ejecutando la siguiente iteración del bucle **for** identificado por la etiqueta.

```
1 \programas_libro>java SentenciaContinue  
2 1 * 1 = 1 4 * 1 = 4  
3 1 * 2 = 2 4 * 2 = 8  
4 1 * 3 = 3 4 * 3 = 12  
5 1 * 4 = 4 4 * 4 = 16  
6 1 * 5 = 5 4 * 5 = 20  
7 1 * 6 = 6 4 * 6 = 24
```

|    |                      |             |
|----|----------------------|-------------|
| 8  | 1 * 7 = 7            | 4 * 7 = 28  |
| 9  | 1 * 8 = 8            | 4 * 8 = 32  |
| 10 | 1 * 9 = 9            | 4 * 9 = 36  |
| 11 | 1 * 10 = 10          | 4 * 10 = 40 |
| 12 | 1 * 11 = 11          | 4 * 11 = 44 |
| 13 | 1 * 12 = 12          | 4 * 12 = 48 |
| 14 | Salimos del bucle... |             |

## 5.4 EJERCICIOS

**Ejercicio 1:** Realizar un programa que permita ingresar dos operandos, seleccionar la operación a realizar (suma, resta, multiplicación y división) y presentar su resultado.

*Ejemplo 55 Calculadora básica*

```

1 import java.util.Scanner;
2
3 public class Calculadora {
4
5     public static void main(String[] args) {
6         double numero1 = 0;
7         double numero2 = 0;
8         double resultado = 0;
9         int opcion = 0;
10        Scanner entrada = new Scanner(System.in);
11        System.out.println("*****Calculadora Básica*****");
12        System.out.println("");
13        System.out.println("Ingrese el primer número:");
14        numero1 = entrada.nextInt();
15        System.out.println("Ingrese el segundo número:");
16        numero2 = entrada.nextInt();
17        System.out.println("*****");
18        System.out.println("1. Sumar");
19        System.out.println("2. Restar");
20        System.out.println("3. Multiplicar");
21        System.out.println("4. Dividir");
22        System.out.println("*****");
23        System.out.println("Elija una opción:");
24        opcion = entrada.nextInt();
25        switch (opcion) {
26            case 1:
27                resultado = numero1 + numero2;
28                System.out.println("La suma es: " + resultado);
29                break;
30            case 2:
31                resultado = numero1 - numero2;
32                System.out.println("La resta es: " + resultado);
33                break;
34            case 3:
35                resultado = numero1 * numero2;
36                System.out.println("La multiplicación es: " +
37                                resultado);
38                break;

```

```

39     case 4:
40         resultado = numero1 / numero2;
41         System.out.println("La división es: " +
42             resultado);
43         break;
44     default:
45         System.out.println("Opción no válida");
46         break;
47     }
48 }
49 }
```

Su salida es:

```

1 \programas_libro> java ConvertidorMonedas
2 *****Calculadora *****
3
4 Ingrese el primer número:
5 60
6 Ingrese el segundo número:
7 8
8 ****
9 1. Sumar
10 2. Restar
11 3. Multiplicar
12 4. Dividir
13 ****
14 Elija una opción:
15 4
16 La división es: 7.5
```

**Ejemplo 2:** Implementar un programa que calcule cuántos billetes de 20 y 10, monedas de 2 y 1 euros y céntimos de 50, 20 y 5 corresponde con una cantidad leída por el programa.

*Ejemplo 56 Convertidor de billetes*

```

1 import java.util.Scanner;
2
3 public class ConvertidorMonedas {
4
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7         System.out.println("Introduzca un cantidad:");
8         double cantidad = sc.nextDouble();
9
10        int unidades;
11        double valormoneda;
12
13        valormoneda = 20;
14        if(cantidad >= valormoneda) {
15            unidades = (int) (cantidad / valormoneda);
16            cantidad = cantidad % valormoneda;
17            System.out.println("Nº billetes " + valormoneda + "
```

```

18         : " + unidades);
19     }
20     valormoneda = 10;
21     if (cantidad >= valormoneda) {
22         unidades = (int) (cantidad / valormoneda);
23         cantidad = cantidad % valormoneda;
24         System.out.println("Nº billetes " + valormoneda + "
25                           : " + unidades);
26     }
27     valormoneda = 2;
28     if (cantidad >= valormoneda) {
29         unidades = (int) (cantidad / valormoneda);
30         cantidad = cantidad % valormoneda;
31         System.out.println("Nº billetes " + valormoneda + "
32                           : " + unidades);
33     }
34     valormoneda = 1;
35     if (cantidad >= valormoneda) {
36         unidades = (int) (cantidad / valormoneda);
37         cantidad = cantidad % valormoneda;
38         System.out.println("Nº billetes " + valormoneda + "
39                           : " + unidades);
40     }
41     valormoneda = 0.5;
42     if (cantidad >= valormoneda) {
43         unidades = (int) (cantidad / valormoneda);
44         cantidad = cantidad % valormoneda;
45         System.out.println("Nº billetes " + valormoneda + "
46                           : " + unidades);
47     }
48     valormoneda = 0.2;
49     if (cantidad >= valormoneda) {
50         unidades = (int) (cantidad / valormoneda);
51         cantidad = cantidad % valormoneda;
52         System.out.println("Nº billetes " + valormoneda + "
53                           : " + unidades);
54     }
55     valormoneda = 0.05;
56     if (cantidad >= valormoneda) {
57         unidades = (int) (cantidad / valormoneda);
58         cantidad = cantidad % valormoneda;
59         System.out.println("Nº billetes " + valormoneda + "
60                           : " + unidades);
61     }
62 }
63 }
```

Su salida es:

|   |                                           |
|---|-------------------------------------------|
| 1 | \programas_libro> java ConvertidorMonedas |
| 2 | Introduzca un cantidad:                   |
| 3 | 50                                        |
| 4 | Nº billetes 20.0 : 2                      |
| 5 | Nº billetes 10.0 : 1                      |

## 5.5 CUESTIONARIO

**1. ¿Cuál es la instrucción que utiliza de forma correcta el operador ternario?**

- String mensaje = (numero > 0) ? "positivo" , "negativo"; ( )
- String mensaje = (numero > 0) "positivo" : "negativo"; ( )
- String mensaje = (numero = 0) ? "positivo" : "negativo"; ( )
- String mensaje = (numero > 0) ? "positivo" : "negativo"; ( )
- String mensaje = (numero > 0) ?: "positivo" : "negativo"; ( )

**2. Analizar el siguiente ejemplo y seleccionar el valor de la variable mayor:**

```
1 public class OperadorTernario{  
2     public static void main(String [] args){  
3         int x = 8, y = 4, mayor = 0;  
4         /*  
5             if (x > y)  
6                 mayor = x;  
7             else  
8                 mayor = y;  
9             */  
10    }  
11 }
```

- 0
- 8
- 4
- null

**3. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 int x=0;  
2 boolean flag = false;  
3  
4 while ((x<10) && !flag) {  
5     System.out.println(x);  
6     x++;  
7 }
```

- Muestra los números del 0 al 9.
- Muestra los números del 1 al 10.
- Muestra un 10.
- Se queda en un bucle infinito.

**4. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 for (int x=0;x<10;x++)  
2     System.out.println(x);  
3  
4     - Los números del 1 al 9.  
5     - Los números del 0 al 9.  
6     - Los números del 1 al 10.  
7     - El programa no compila.
```

**5. Analizar el siguiente código y seleccionar la respuesta correcta el valor de "y":**

```
1 int x = 0;
2 int y = 0;
3
4 while (x<10) {
5     y += x;
6     x++;
7 }
8 System.out.println(y);
- 0
- 10
- 11
- 45
```

**6. Analizar el siguiente ejemplo y seleccionar el valor de la variable "dia":**

```
1 public class EstructuraSwitch{
2     public static void main(String [] args){
3         int opcion = 2;
4         String dia = "";
5         switch(opcion){
6             case 1:
7                 dia = "Lunes";
8             case 2:
9                 dia = "Martes";
10            case 3:
11                dia = "Miércoles";
12            default:
13                dia = "No es un día de la semana";
14        }
15        System.out.println(dia);
16    }
17 }
```

- El valor de día es "Lunes".
- El valor de día es "Martes".
- El valor de día es "Miércoles".
- El valor de día es "No es un día de la semana".

**7. En Java, después de definir un if (expresion\_booleana) se debe continuar con:**

- sentencia else.
- sentencia do.
- sentencia then.
- paréntesis de apertura {.
- sentencias for.

**8. Indicar cuáles de las siguientes palabras reservadas permite interrumpir los ciclos de en una estructura repetitiva:**

- switch
- default
- break
- case
- end

**9. ¿Cuál es el mecanismo para terminar separar una sentencia de la otra?**

- A través del carácter enter.

- Separándolas con comentarios.
- Por medio de la llave de cierre }.
- Utilizando un carácter de punto y coma.
- Escribiéndolas en distintas líneas.
- Separándolas con guiones.

#### **10.Un if es:**

- Una estructura condicional donde se pueden ejecutar tres caminos diferentes.
- Una estructura condicional donde a través de una expresión booleana se ejecutan dos caminos.
- Una estructura condicional que permite crear contadores.
- Todas las anteriores.

#### **11.Un switch es:**

- Una estructura condicional de dos caminos.
- Una estructura condicional de selección múltiple.
- Una estructura condicional de un solo camino.
- Una estructura condicional anidada.

#### **12.La sentencia break es:**

- Permite saltar una iteración en las estructuras repetitivas.
- Permite saltar una línea y continuar la siguiente línea de código.
- Permite saltarse varias iteraciones de una estructura repetitiva.
- Ninguna de las anteriores.

#### **13.La sentencia continue:**

- Forza la finalización de una estructura repetitiva.
- Permite continuar la siguiente iteración.
- Forza a cumplir la condición de un bucle repetitivo.
- Ninguna de las anteriores.

#### **14.Analizar el siguiente código y seleccionar la respuesta correcta:**

```

1 public class Algoritmo {
2     public static void main(String[] args) {
3         int x = 7;
4         System.out.println(x < 1 ? x > 5 ? 4 : 5 : 6);
5     }
6 }
```

- |                                                                                                                                                                                  |                                                                                                                                                  |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- Imprime 2.</li> <li>- Imprime 4.</li> <li>- Imprime 6.</li> <li>- Imprime 8.</li> <li>- Imprime 10.</li> <li>- Imprime true.</li> </ul> | <ul style="list-style-type: none"> <li>- Imprime false.</li> <li>- Error de compilación en la línea 4.</li> <li>- Error de ejecución.</li> </ul> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|

**15. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public class Algoritmo {  
2     public static void main(String[] args) {  
3         int x = 7;  
4         System.out.println(x > 1 ? x < 5 ? 4 : 5 : 6);  
5     }  
6 }
```

- Imprime 4
- Imprime 5
- Imprime 6
- Imprime 8
- Imprime 10
- Imprime true
- Imprime false
- Error de compilación en la línea 4
- Error de ejecución

**16. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public class Algoritmo {  
2     public static void main(String[] args) {  
3         int x = 7;  
4         System.out.println(x > 1 ? x > 5 ? 4 : 5 : 6);  
5     }  
6 }
```

- Imprime 2
- Imprime 4
- Imprime 6
- Imprime 8
- Imprime 10
- Imprime true
- Imprime false
- Error de compilación en la línea 4
- Error de ejecución

**17. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public class ProyectoLibroJava {  
2     public static void main(String[] args) {  
3         int contador = 6;  
4         if((contador > 6 ? contador++: --contador)< 10)  
5             System.out.print("Cerveza" + " ");  
6         if(contador < 6)  
7             System.out.print("Vino" + " ");  
8     }  
9 }
```

- Imprime Cerveza Vino
- Error de compilación en la línea 4
- Error de ejecución
- Imprime Cerveza
- Imprime Vino
- Imprime Vino Cerveza

**18. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public class Algoritmo {  
2     public static void main(String [] args){  
3         int x = 1, y = 15;  
4         while (x < 10)  
5             y--;  
6         x++;  
7         System.out.println(x + "," + y);  
8     }  
9 }
```

- Imprime 10, 5
- Imprime 10, 6
- Imprime 1, 15
- Imprime 11, 5
- Error de compilación en la línea 4
- Error de compilación en la línea 5
- Error de compilación en la línea 3
- Error de ejecución

**19. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public class Algoritmo{  
2     public static void main (String [] args){  
3         int contador = 0;  
4         int x = 3;  
5         while(contador++ < 3) {  
6             int y = (1 + 2 * contador) % 3;  
7             switch(y) {  
8                 case 0: x -= 1; break;  
9                 case 1: x += 5; break;  
10            }  
11        }  
12        System.out.println(x);  
13    }  
14 }
```

- Imprime 3
- Imprime 4
- Imprime 5
- Imprime 6
- Imprime 7
- Imprime 11
- Imprime 13
- Error de compilación en la línea 5
- Error de compilación en la línea 6

**20. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 int contador = 10;  
2 boolean bandera = ((contador++ > 10 ? contador++: --contador) < 10);  
3 System.out.println(bandera + " " + contador);  
4
```

- Imprime false 8
- Imprime false 9
- Imprime false 10
- Imprime false 11
- Imprime true 8
- Imprime true 9
- Imprime true 10
- Imprime true 11
- Error de compilación en la línea 2

## 6 MÉTODOS

---

En este capítulo se realiza una definición de los métodos, su sintaxis y como se debe modelar en UML. Además, se explica a través de ejemplos cómo se comportan los métodos de tipo de dato primitivo, de instancia o de clase; y se establece la diferencia entre métodos que devuelven un dato o método **void**. Para finalizar se explica el rol que cumplen los parámetros.

### Objetivos

- Describir la sintaxis y funcionamiento de los métodos.
- Establecer la diferencia entre métodos que devuelven o no un dato.
- Definir métodos utilizando parámetros en su cabecera.

### 6.1 DEFINICIÓN DE MÉTODOS

El método es una operación que la clase puede realizar, contiene una serie de sentencias para cumplir una determinada tarea. En los ejemplos anteriores se ha trabajado con un método especial llamado *main* para ejecutar las aplicaciones desde la consola de comando. En otros lenguajes de programación un método es equivalente a una función, procedimiento o subrutina.

Al crear métodos en una clase se aplica la frase “divide y vencerás” porque al dividir algoritmos extensos y complejos en tareas pequeñas de forma, hace que el código sea más legible y cualquier error que se genera se pueda depurar fácilmente.

Los métodos son definidos por:

- **Un modificador de acceso:** **public**, **private** o **protected**.
- **El tipo de dato de retorno:** puede ser **void** o algún tipo de datos.

- **El identificador:** nombre del método que el programador le asigna.
- **Lista de parámetros:** datos recibidos al invocarse el método y que son utilizados por el propio método.

Los métodos pueden o no devolver un valor, eso depende del tipo de dato de retorno, cuando el método se lo define con **void**, no devuelve ningún valor pero ejecuta todas las instrucciones del método, cuando se define un tipo de dato, al finalizar sus operaciones, devuelve habitualmente un valor a través de la sentencia **return**, el valor retornado debe coincidir con el tipo de dato declarado en la cabecera del método.

Están estructurados por dos partes:

- **Cabecera del método:** es la declaración del método y establece el comportamiento que tendrá.
- **Cuerpo del método:** es el bloque de código que ejecuta un conjunto de instrucciones y está definido por las llaves abierta y cerrada “{ }”.

Su sintaxis es:

```
modificador_acceso tipoDatos nombreMétodo (lista_argumentos) {
    //sentencias del método
}
```

En la siguiente figura se ilustra las partes que conforman un método:



Figura 23 Estructura de un método

Para utilizar los métodos desde otras clases es imprescindible crear objetos. Creado el objeto el programador puede hacer uso de los atributos y métodos

mediante el carácter punto “.”, la siguiente figura se ilustra la sintaxis para llamar a un método:



Figura 24 Invocación a un método

Si el programador utiliza al método en la misma clase donde se ha definido, no es necesario crear un objeto, se lo llama directamente con el nombre del método.

En UML también se define una sintaxis para definir los métodos, en una clase normal se los define en la tercera caja rectangular, en la siguiente figura se ilustra la sintaxis para definir un método:

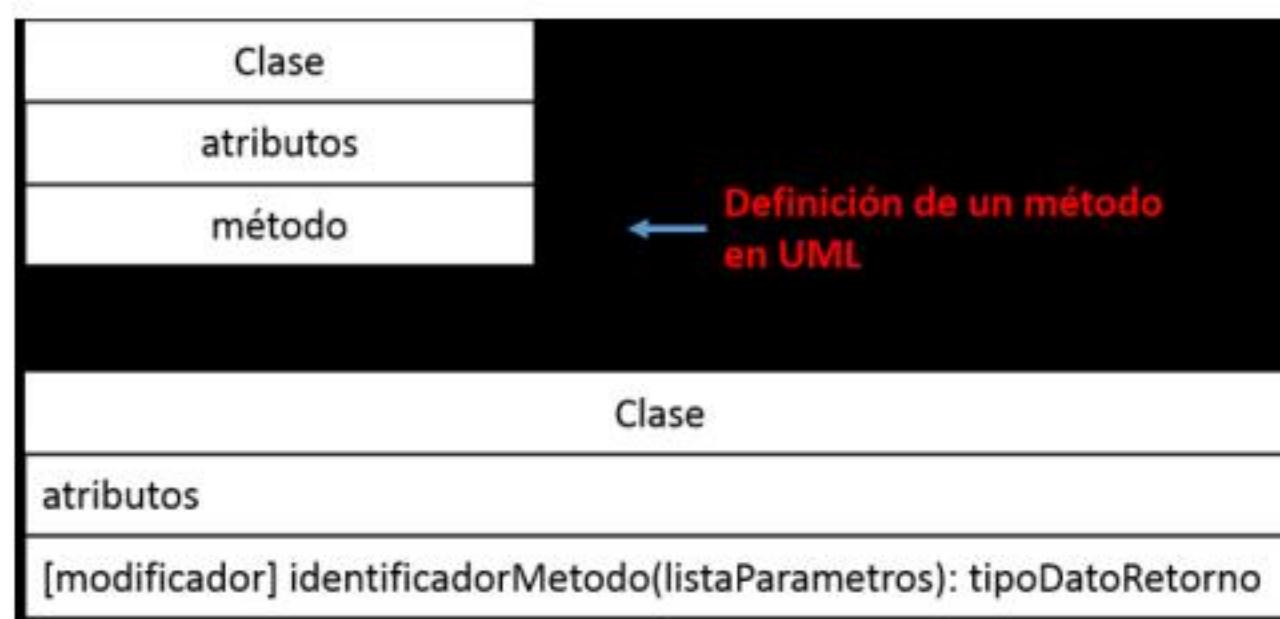


Figura 25 Métodos en UML

## 6.2 MÉTODOS QUE NO DEVUELVE NINGÚN VALOR Y SIN PARÁMETROS

En esta clasificación de métodos se utiliza la palabra reservada **void** para definir que no devolverá ningún tipo de datos, y ejecuta todas las instrucciones. En este contexto, se definirá un método llamado `mostrarMensaje` en UML, en la siguiente figura se ilustra la forma de definir un método:

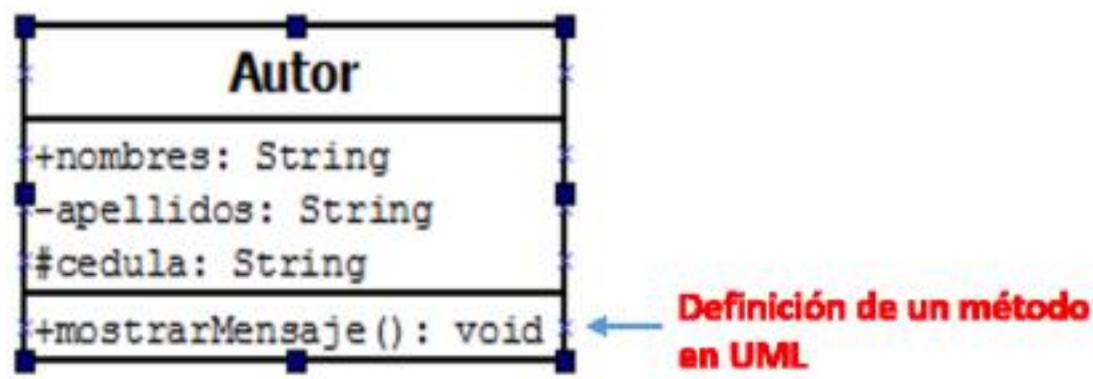


Figura 26 Métodos de la clase en UML

Al traducir de un diagrama UML al lenguaje de programación Java se genera el siguiente código:

Ejemplo 57 Método de una clase

```

1 public class Autor {
2     //declaración de métodos
3     public String nombres;
4     private String apellidos;
5     protected String cedula;
6
7     public void mostrarMensaje(){
8         System.out.println("Datos del autor");
9     }
10 }
```

En el siguiente ejemplo se codifica la clase Ejecutor y crea un objeto “autor” de la clase Autor que permite llamar al método mostrarMensaje a través del invocador “.”

Ejemplo 58 Clase Ejecutara, método - void

```

1 public class Ejecutor {
2
3     public static void main(String[] args) {
4         Autor autor = new Autor();
5         autor.mostrarMensaje();
6     }
7 }
```

La salida del ejemplo es:

```

1 \programas_libro> java Ejecutor
2 Datos del autor
```

### 6.3 MÉTODOS QUE DEVUELVE UN VALOR

Esta clasificación de métodos se define un tipo de datos que puede ser primitivo, de clase o enumerado en la cabecera del método con el objetivo de retornar un valor, la sentencia para devolver el valor es `return`. En este contexto, se definirá un método llamado `devolverMensaje` que devolverá un dato de tipo cadena. En la

siguiente figura se ilustra la forma de definir un método que retorna un dato en UML:

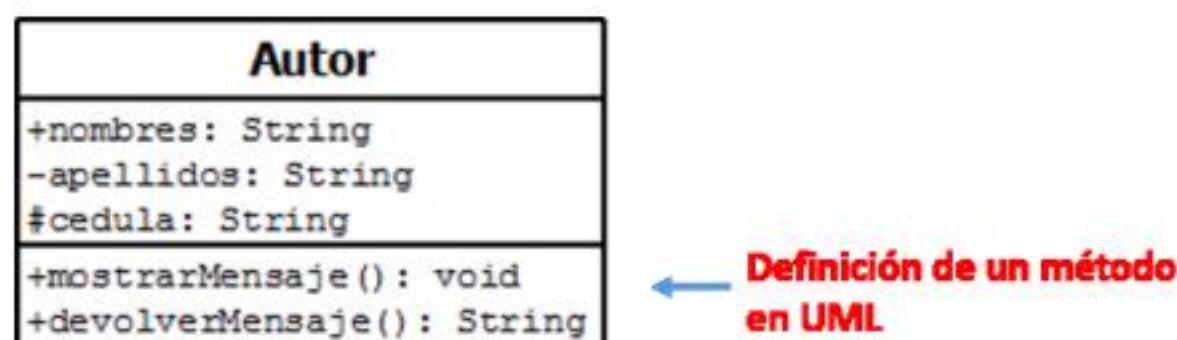


Figura 27 Métodos de la clase en UML sentencia return

Al traducir el diagrama UML al lenguaje de programación Java se genera el siguiente código:

Ejemplo 59 Setencia return

```
1 public class Autor {  
2     public String nombres;  
3     private String apellidos;  
4     protected String cedula;  
5  
6     public void mostrarMensaje(){  
7         System.out.println("Datos del autor");  
8     }  
9  
10    public String devolverMensaje(){  
11        return "Datos del autor devuelto";  
12    }  
13}
```

En este ejemplo, en la línea 10 se define el método como acceso público y su tipo de dato de retorno es *String* por lo que el método está obligado a utilizar la sentencia **return** para devolver un datos de tipo cadena. Este valor puede ser almacenado en alguna variable o atributo siempre que sea del mismo tipo.

En el siguiente ejemplo, en la clase Ejecutor se crea un objeto “autor” de la clase Autor que permite llamar al método devolverMensaje a través del invocador “.”. El método invocado devuelve una cadena y se concatena con la cadena de la sentencias System.out.println porque son del mismo tipo.

Ejemplo 60 Clase ejecuta Ejemplo sentencia return

```
1 public class Ejecutor {  
2  
3     public static void main(String[] args) {  
4         Autor autor = new Autor();  
5         System.out.println("Mensaje: " +  
6                             autor.devolverMensaje());  
7     }  
8 }
```

La salida del ejemplo es:

```
1 \programas_libro> java Ejecutor  
2 Datos del autor devuelto
```

## 6.4 PARÁMETROS DE UN MÉTODO

Los parámetros son datos de entrada que recibe el método para operar con ellos.

Los métodos pueden o no definir un conjunto finito de parámetros.

La definición de un parámetro es el mismo al de una variable y se separan con coma “,”, su sintaxis es:

```
modificador tipo_datos nombreMétodo (tipo_dato parametro1, tipo_dato parametro2)
```

En la clase Autor se definirá un método llamado “imprimir” con un solo parámetro de tipo *String* como se ilustra en la siguiente figura:

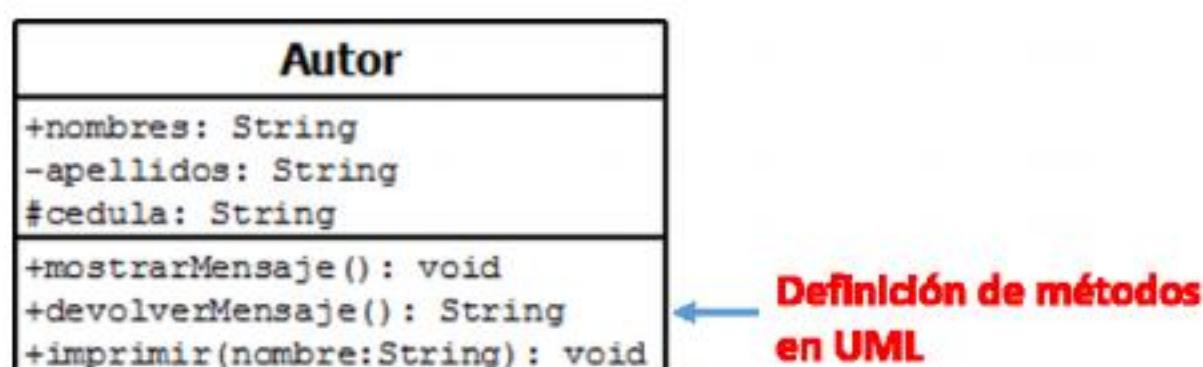


Figura 28 Métodos con parámetros

Al traducir de un diagrama UML al lenguaje de programación Java se genera el siguiente código:

Ejemplo 61 Método con parámetros

```
1 public class Autor {  
2     public String nombres;  
3     private String apellidos;  
4     protected String cedula;  
5  
6     public void mostrarMensaje(){  
7         System.out.println("Datos del autor");  
8     }  
9  
10    public String devolverMensaje(){  
11        return "Datos del autor devuelto";  
12    }  
13  
14    public void imprimir(String nombre){  
15        System.out.println("Nombre del Autor: " + nombre);  
16    }  
17}
```

En el siguiente ejemplo, en la clase Ejecutor se crea un objeto “autor” de la clase Autor que permite llamar al método imprimir a través del invocador “.”. El método invocado necesita un dato que debe ser enviado llamado argumento de tipo *String*.

*Ejemplo 62 Ejemplo método con parámetros*

```
1 public class Ejecutor {  
2  
3     public static void main(String[] args) {  
4         Autor autor = new Autor();  
5         autor.imprimir("María José");  
6     }  
7 }
```

La salida del ejemplo es:

```
1 \programas_libro> java Ejecutor  
2 Nombre del Autor: María José
```

## 6.5 EJERCICIOS

**Ejemplo 1:** Definir una clase que contenga tres métodos: sumar, potencia y división, cada uno recibirá dos parámetros.

*Ejemplo 63 métodos potencia y división*

```
1 public class Matematicas {  
2  
3     public double sumar(double a, double b){  
4         return a+b;  
5     }  
6  
7     public int potencia (int base, int exponente){  
8         int resultado = 1;  
9         for (int i = 1; i < exponente; i++) {  
10             resultado *= base;  
11         }  
12         return resultado;  
13     }  
14  
15     public void division (int dividendo, int divisor){  
16         int contador = 0;  
17         while(dividendo >= divisor){  
18             dividendo -= divisor;  
19             contador++;  
20         }  
21         System.out.println("Cociente es: " + contador);  
22         System.out.println("Residuo es: " + dividendo);  
23     }  
24 }
```

El comportamiento de los métodos lo define el programador. Por ejemplo, en este ejemplo se declaran tres métodos “sumar”, “potencia” y “división” con dos parámetros pero cada uno devuelve un tipo de dato diferente. El método “sumar” devuelve un valor decimal, la “potencia” un valor entero y “división” no devuelve ningún valor. Dentro de “sumar” no se declaran localmente variables pero en “potencia” si se declara una variable local “resultado” e incluye una sentencia **for** para sumar en cada iteración la multiplicación entre el resultado y la base.

El código de la clase ejecutora es:

Ejemplo 64 Ejecutor de la clase Matemáticas

```
1 public class Ejecutor {  
2  
3     public static void main(String[] args) {  
4         Matematicas m = new Matematicas();  
5         double s = m.sumar(78, 65.89);  
6         int p = m.potencia(2, 5);  
7         System.out.println("La suma es: " + s);  
8         System.out.println("La potencia es: " + p);  
9         m.division(32, 5);  
10    }  
11 }
```

Su salida es:

```
1 \programas_libro>java Ejecutor  
2 La suma es: 143.89  
3 La potencia es: 9
```

**Ejemplo 2:** crear una clase que contenga dos métodos para calcular el factorial y Fibonacci de un determinado número.

Ejemplo 65 Métodos factorial y fibonacci

```
1 public class Algoritmos {  
2  
3     public int factorial(int n){  
4         int factorial = 1;  
5         for (int i = 2; i < n; i++) {  
6             factorial *= i;  
7         }  
8         return factorial;  
9     }  
10  
11     public int fibonacci(int n){  
12         int a = 0,b = 1,c;  
13         for(int i = 1;i <= n;i++){  
14             c = a + b;  
15             b = a;  
16             a = c;
```

```

17     }
18     return a;
19 }
20 }
```

El siguiente ejemplo se codifica una clase llamada Algoritmo que contendrá dos métodos para el cálculo de las series del fibonacci y factorial, los métodos definidos devolverán un dato de tipo entero y se define un solo parámetro.

En la clase ejecutora se crea un objeto de la clase Algoritmos permitirá invocar a los métodos declarados en la clase, para invocar cada método se utiliza “.”. El código es:

*Ejemplo 66 Ejecutor de la clase Algoritmos*

```

1 public class ProyectoLibroJava {
2
3     public static void main(String[] args) {
4         Algoritmos a = new Algoritmos();
5         System.out.println("factorial es: " + a.factorial(6));
6         System.out.println("fibonacci es: " + a.fibonacci(6));
7     }
8 }
```

Su salida es:

```

1 \programas_libro> java Ejecutor
2 factorial es: 120
3 fibonacci es: 8
```

## 6.6 CUESTIONARIO

2. **Para invocar un método público de cualquier clase se debe seguir la siguiente sintaxis:**
  - nombre\_objeto.nombre\_metodo(lista\_de\_parámetros)
  - nombre\_clase.nombre\_metodo
  - nombre\_clase.nombre\_metodo(lista\_de\_parámetros)
  - nombre\_objeto.nombre\_metodo
  
3. **Seleccionar la sentencia correcta para definir parámetros en el método cambiarLlanta de la clase Vehículo.**
  - public void cambiarLlanta(Llanta: llanta, int: posicion)
  - public void cambiarLlanta(Llanta llanta; int posicion)
  - public void cambiarLlanta(Llanta llanta, int posicion)
  - public void cambiarLlanta(Llanta: llanta; int: posicion)
  
4. **Indicar cuáles palabras reservadas corresponde a los tipos de dato que se pueden devolver al utilizar la instrucción return:**
  - int
  - double

- char
- boolean
- enum
- class
- void

**5. Evaluar el siguiente el método y seleccionar la opción correcta:**

```

1 public void evaluarMetodo(){
2     int x = 0;
3     while(x++ < 60) {}
4     String message = x > 60 ? "El mensaje es" : false;
5     System.out.println(message + " - " + x);
6 }
```

- No compila por la línea 3.
- Imprime 60 veces el mensaje de la línea 5.
- No compila por la línea 4.
- El método devuelve un valor de tipo String.
- No compila por la línea 5.

**6. Evaluar el siguiente el método y seleccionar la opción correcta:**

```

1 public int evaluarMetodo(){
2     int x = 7;
3     return (x > 5 ? x < 7 ? 80 : 75 : 99);
4 }
```

- Devuelve el valor de 8'
- Devuelve el valor 5
- No compila por la línea 3
- Devuelve el valor 75
- Devuelve el valor 99

**7. Evaluar el siguiente el método y seleccionar la opción correcta:**

```

1 public boolean evaluarMetodo(){
2     boolean a = true, b = true;
3     int x = 20;
4     b = !(x != 8) || (a == true));
5     System.out.println(b);
6     return b;
7 }
```

- Devuelve el valor de true.
- No compila por la línea 4.
- Devuelve el valor 8.
- No compila por la línea 6.
- Devuelve el valor de 20.
- Devuelve el valor de falso.

## **8. Evaluar el siguiente el método y seleccionar la opción correcta:**

```
1 public void evaluarMetodo0{  
2     for(int i=0; i < 5; ){  
3         i = i++;  
4         System.out.println(i);  
5     }  
6 }
```

- Imprime 6 veces i.
- Se vuelve un ciclo infinito.
- No compila por la línea 2.
- Imprime 5 veces i.
- La variable i toma el valor de i + 1.

## **9. Evaluar el siguiente el método y seleccionar la opción correcta:**

```
1 public void evaluarMetodo0{  
2     int x = 2 + 3 * 4 % 5;  
3     System.out.println(x);  
4 }
```

- Imprime 7.
- No compila por la línea 2.
- Imprime 6.
- No compila porque el método no retorna un valor.
- Imprime 4.

## **10. Evaluar el siguiente el método y seleccionar la opción correcta:**

```
1 public void evaluarMetodo0{  
2     int x = 7, y = 5;  
3     boolean resultado = x <= y;  
4     if (resultado = true)  
5         System.out.println("Verdadero");  
6     else  
7         System.out.println("Falso");  
8 }
```

- No compila por la línea 3.
- Imprime Verdadero.
- No compila por la línea 4.
- Imprime Falso.
- Error de compilación.

## **11. Evaluar el siguiente el método y seleccionar la opción correcta:**

```
1 public void evaluarMetodo0{  
2     int x = 0;  
3     while(x <= 5){  
4         System.out.print(x++ + " ");  
5         x = 1;
```

6      }  
7

- Imprime 0 1 2 3 4 5.
- Imprime 0 1 2 3 4 5 6.
- Imprime 1 2 3 4 5.
- Imprime 1 2 3 4 5 6.
- No compila por la línea 3.
- Ciclo infinito.

## 12. Los métodos son:

- Las operaciones de cada clase.
- Existen métodos que no devuelven nada (void).
- Un constructor es un método normal.
- Todas las anteriores.

## 13. La sentencia return sirve para:

- Retornar un valor de una variable.
- Retorna un valor como resultado de una operación.
- Retorna el valor producido de un método.
- Todas las anteriores.

## **7 CLASES Y OBJETOS**

---

En este capítulo se realiza una descripción del concepto, estructura y sintaxis de la clase y objeto. A través de los diagramas UML se representa gráficamente a las clases y su implementación en código Java. Uno de los elementos que la conforman son los atributos, se establece su definición, su comportamiento y los valores por defecto que toman cuando no se inicializan. Además, las clases, métodos y atributos utilizan los modificadores de acceso para encapsular su estructura e implementación, por ellos, mediante ejemplos se explica el funcionamiento de los modificadores de accesos (públicos, privados, protegidos y por defecto) y la manera correcta de invocar a los elementos de la clase cuando se crea un objeto. Por último, se indica los tipos de constructores y el papel que juega la sentencias `this` dentro de los objetos.

### **Objetivos**

- Definir la estructura de una clase.
- Establecer las diferencias entre cada modificador de acceso.
- Implementar los constructores por defecto y con parámetros.
- Analizar el funcionamiento de la sentencia `this` dentro de los objetos.
- Diferenciar los conceptos entre un atributo o método normal con respecto a los estáticos.

### **7.1 CLASE**

Es la parte principal del paradigma orientado a objetos porque permite definir la plantilla donde los objetos serán instanciados. Se componen en atributos y métodos. En UML se representa con cajas rectangulares como se ilustra en la siguiente figura:

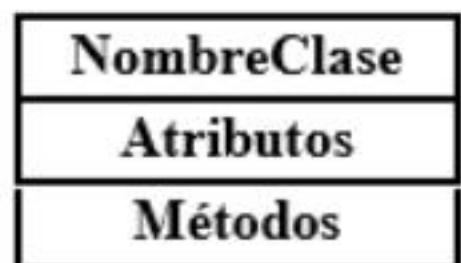


Figura 29 Clase en UML

Su sintaxis es:

```
modificador_acceso class NombreClase {
    // cuerpo de la clase
}
```

La cabecera de una clase tiene tres partes:

- El modificador de acceso es opcional pero las clase por lo general siempre son públicas.
- La palabra reservada **class**, define que el archivo .java contendrá una clase.
- Un identificador legal para nombrar la clase y es definido por el programador.

Una vez descrita la cabecera de la clase es necesario especificar el cuerpo del método, las llaves de apertura y de cierre { } definen el inicio y fin de la clase.

#### **Recomendación**

Por convención el nombre de la clase debe iniciar con mayúscula la primera letra, si el nombre consta de dos palabras, las primeras letras de cada palabra deben ser en mayúscula EstudianteUniversitario. También se deben omitir los caracteres de acentuados y la eñe, porque los lenguajes de programación son diseñados en el idioma inglés y no están contemplados en el lenguaje. Es conveniente utilizar las palabras completas y evitar los acrónimos.

Para explicar los conceptos de clases, objetos, atributos, métodos se graficará tres clases en UML como se ilustra en la siguiente figura.



Figura 30 Diagrama de clases básico

El primer paso es traducir una clase en UML en código java:

Ejemplo 67 Clase Libro

```
1 | public class Libro {  
2 |     //codigo  
3 | }
```

Ejemplo 68 Clase Autor

```
1 | public class Autor {  
2 |     //codigo  
3 | }
```

Ejemplo 69 Clase Copia

```
1 | public class Copia {  
2 |     //codigo  
3 | }
```

En resumen, para declarar una clase es necesario de un modificador de acceso, la palabra reservada **class**, el identificador y el cuerpo de la clase. La siguiente figura ilustra las partes que componen una clase:

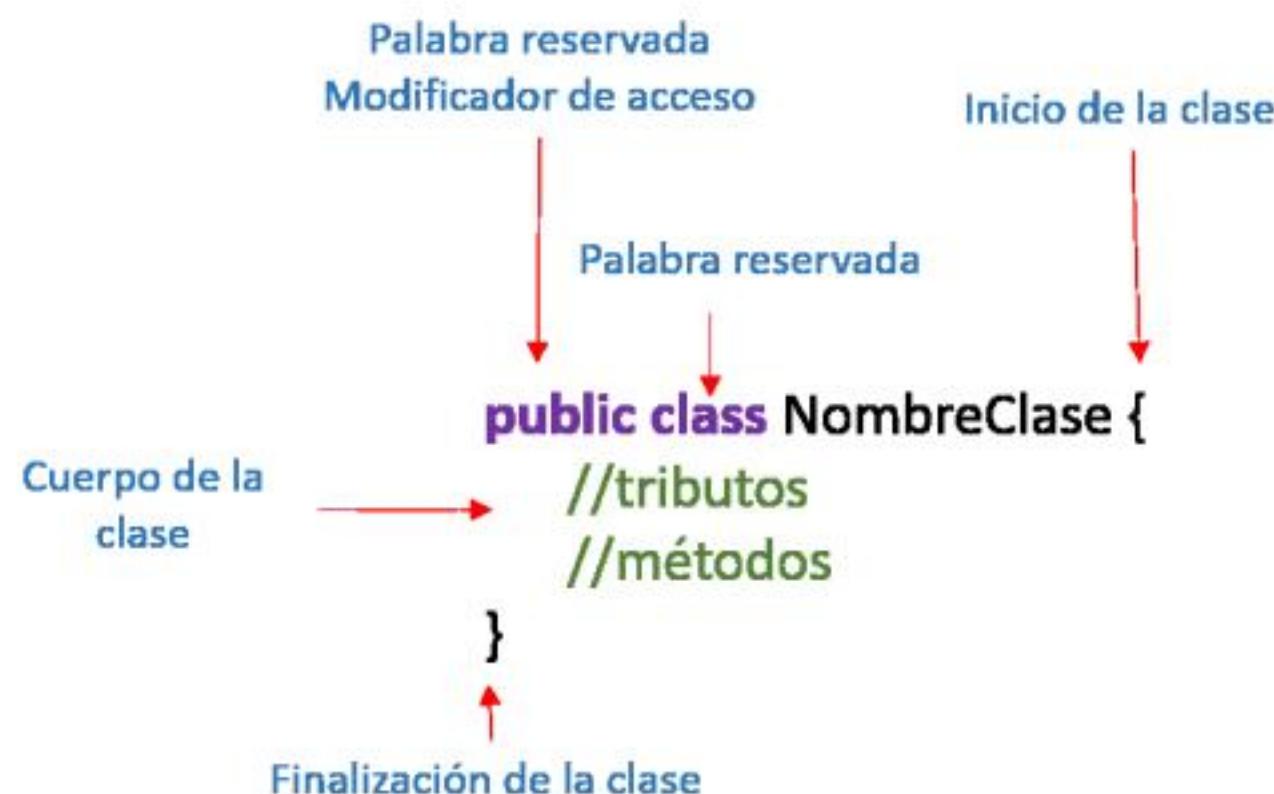


Figura 31 Estructura de una clase

### 7.1.1 ATRIBUTOS

Son llamados propiedades o características de una clase y describen el rango de valores que la propiedad podrá contener en los objetos de la clase. En una clase se pueden definir cualquier número de atributos o simplemente no tener ninguno.

Para definir un atributo se especifica un modificador de acceso que define su accesibilidad, el tipo de datos que y un identificador, su sintaxis es:

```
modificadorAcceso tipoDatos identificador;
```

En la siguiente figura se ilustra la definición de atributos en UML en la segunda caja:

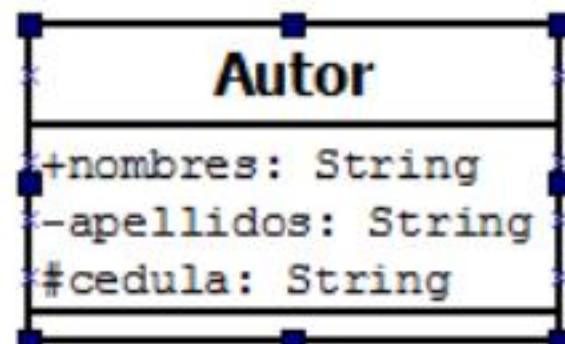


Figura 32 Atributos en UML

Al traducir un diagrama UML a código Java, la codificación de los atributos de la clase Autor es de la siguiente manera:

Ejemplo 70 Clase Autor con atributos

```
1 public class Autor {  
2     //declaración de métodos  
3     public String nombres; //valor por defecto null  
4     private String apellidos;  
5     protected String cedula;  
6 }
```

En la clase Autor se han declarado tres atributos pero no han sido inicializado, Java le asigna un valor por defecto cada uno, si los atributos son definidos con datos primitivos sus valores serán 0 si es numérico o false si es booleano, el resto de atributos tendrán un valor **null**.

La palabra reservada **null**, permite liberar de espacio no utilizado de memoria y solo se asigna a los atributos o variables con tipo de datos diferente a los primitivos.

En la clase, los atributos nombres, apellidos y cédula están declarados con un tipo de dato *String* por lo tanto su valor por defecto es **null**. En el siguiente ejemplo, la clase Ejecutora creará un objeto de la clase Autor utilizando el constructor por defecto, los atributos no se han inicializado e imprimiremos el atributo nombres cuyo resultado será **null**.

```
1 public class Ejecutor {  
2     //declaración de métodos  
3     public static void main(String[] args) {  
4         Autor autor = new Autor();  
5         System.out.println(autor.nombres);  
6     }  
7 }
```

Su salida:

```
1 \programas_libro> java Ejecutor  
2 null
```

### Recomendación

Por convención el nombre de un atributo consta de una sola palabra se escribe con minúscula. Si el nombre consta de dos palabras, únalas con una palabra Mayúscula telefonoCelular a excepción de la primera letra como ejemplo: color, colorPiel, etc.

Las reglas para seleccionar al identificador de los atributos son:

- Debe comenzar con una letra.
- No puede contener espacio en blanco.
- No se podrá utilizar palabras reservadas del lenguaje como identificadores.
- Java distinguirá entre MAYÚSCULAS y MINÚSCULAS.
- No incluir caracteres especiales.

En resumen, para declarar un atributo es necesario: un modificador de acceso, un tipo de datos y un identificador. La siguiente figura ilustra su estructura:

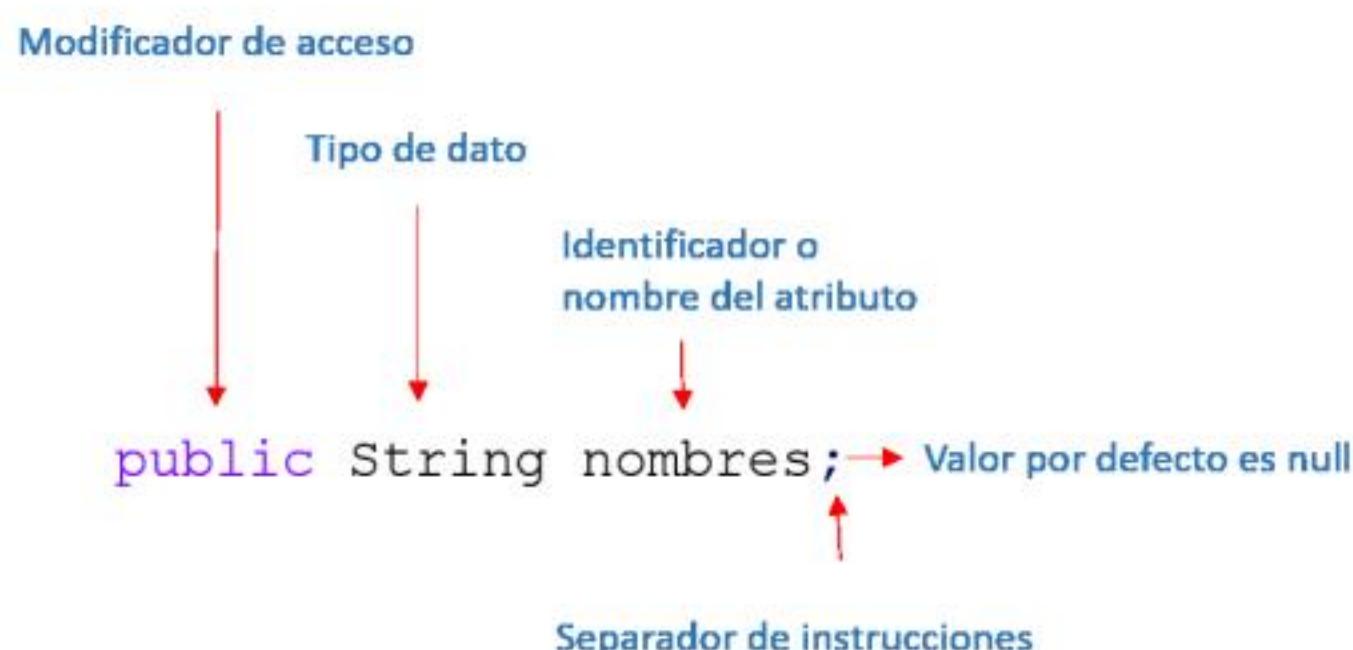


Figura 33 Estructura de un atributo

## 7.2 CONSTRUCTOR

Es el encargado de crear los objetos de la clase al instante de ser invocado. Tiene dos características fundamentales, el primero es que su identificador está determinado por el nombre de clase, el segundo no define ningún tipo de dato de retorno (tampoco no define **void**).

Su sintaxis es:

```
modificadorAcceso NombreClase  
(listaParámetros) {  
    // código  
}
```

En resumen, un constructor tiene las siguientes características:

- Se define con el mismo nombre de la clase.
- No puede ser heredado.
- No retorna ningún valor ni se define con **void** así que carece de tipo de retorno.
- Siempre debe declararse con un modificador de acceso **public**.

### 7.2.1 CONSTRUCTOR POR DEFECTO

Cuando se define una clase, automáticamente se crea un constructor sin parámetros que se denomina constructor por defecto, aunque no se lo declare, inicializa a los atributos con datos primitivos con 0 o falso y a los atributos de referencia con **null**.

Definir una clase llamada Autor con dos atributos y sin constructores. Al crear un objeto de la clase Autor (`Autor autor = new Autor();`) y compilarlo, Java detecta que no se ha definido ningún constructor por lo que se asigna uno por defecto de forma implícita como se ilustra en la siguiente figura:

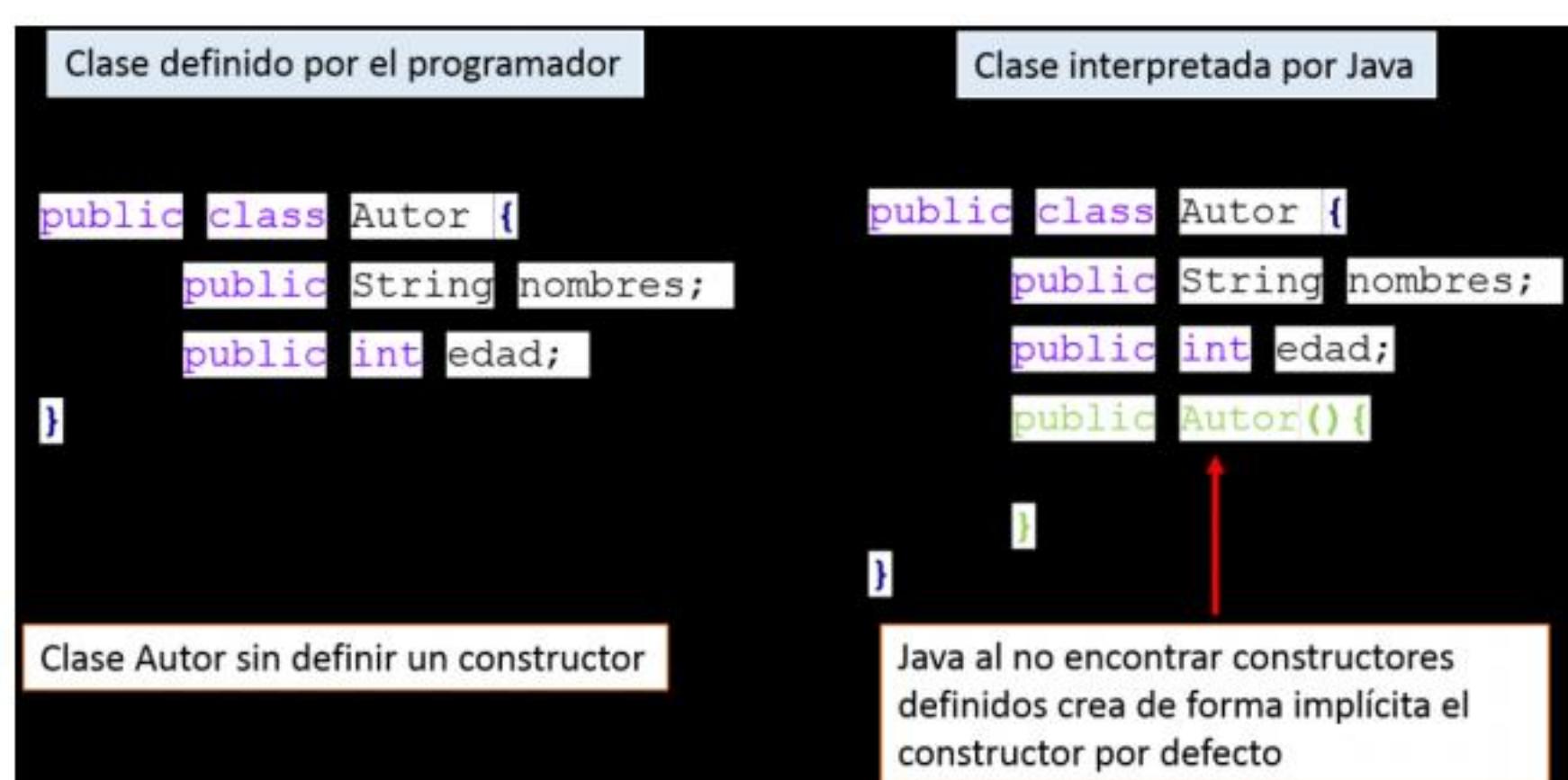


Figura 34 Constructor por defecto

El programador también puede codificar un constructor por defecto, la única regla es que no se debe definir parámetros. Analizar el siguiente ejemplo:

Ejemplo 71 Constructor de una clase

```
1 public class Autor {  
2     public String nombres;  
3     public String apellidos;  
4     public String cedula;  
5  
6     public Autor(){  
7         nombres = "Edwin René"; //inicialización de atributos  
8         apellidos = "Edwin René";  
9         cedula = "1104097553";  
10    }  
11 }
```

Al definir un constructor hay que tener en cuenta que debe tener exactamente el nombre de la clase y no retornará ningún tipo de datos. En este ejemplo el constructor por defecto inicializa los atributos de la clase con valores fijos de tipo *String*.

### 7.2.2 CONSTRUCTOR CON PARÁMETROS

Hasta ahora solo se ha definido constructores por defecto, ahora codificar constructores con parámetros en la misma clase. Cuando se definen más de un constructor se denomina sobrecarga. La sobrecarga es una propiedad del polimorfismo que consiste en crear distintas variantes del método que tenga diferentes parámetros.

Si un programador declara un constructor con parámetro, los atributos serán inicializados con los valores que se almacenan en los parámetros.

#### Recomendación

Al declarar constructores con parámetros se recomienda que se le asigne el mismo identificador de los atributos de la clase a los parámetros.

En el siguiente ejercicio se definen variantes del constructor de la clase Autor:

Ejemplo 72 Sobrecarga de constructores

```
1 public class Autor {  
2  
3     public Autor(){  
4         // constructor por defecto
```

```

5   }
6
7   public Autor(String nombres){
8       // constructor sobrecargado 1
9   }
10
11  public Autor(String nombres, String apellidos){
12      // constructor sobrecargado 2
13  }
14 }
```

En este ejemplo, existen tres constructores de la clase Autor, uno por defecto y dos sobrecargados. Para identificar la sobrecarga se debe observar que en cada constructor se han definido diferentes parámetros.

### 7.3 OBJETOS

Definida las clases Libro, Autor y Copia se tiene que pensar en los elementos reales que derivan de ellas, llamados objetos. Los objetos son elementos de la clase que derivan a través de la instancia y se crea utilizando la palabra reservada **new** e invocando al constructor de la clase.

Son elementos reales derivados de las clases que pueden ser concretos y virtuales, están a alrededor. Además poseen las siguientes características:

- **Estado:** establece las condiciones de existencia del objeto, su estado puede cambiar en transcurso de la ejecución de los programas, por ejemplo, una persona inicia con un estado civil soltero, luego se divorcia o se queda viudo, etc.
- **Comportamiento:** establece como responde el objeto ante peticiones de otros objetos. Por ejemplo los mensajes se envían a través de los métodos definidos en las clases y definirá el comportamiento de los objetos.
- **Identidad:** establece que cada objeto es único aunque tengan los mismos valores. No existen dos objetos iguales.

Para crear un objeto se debe seguir la siguiente sintaxis:

**Clase *identificadorObjeto* = new ConstructorClase();**

Todo objeto para ser creado necesita de la palabra reservada **new** e invocar al constructor de la Clase, como se ilustra en la siguiente figura:



Figura 35 Estructura para crear un objeto

Para definir los objetos, se declara en primera instancia los atributos y métodos de la clase Autor.

Ejemplo 73 Clase Autor con atributos y métodos completos

```

1 public class Autor {
2
3     public String nombres;
4     public String apellidos;
5     public String cedula;
6
7     public Autor(){           // constructor por defecto
8    }
9
10
11    public Autor(String nombres){   // constructor sobrecargado 1
12        }
13
14
15    public Autor(String nombres, String apellidos){   // constructor sobrecargado 2
16        }
17
18
19    public void imprimirDatos(){
20        System.out.println("Nombre: " + nombres + "Apellidos: "
21                      + apellidos + "Cédula: " + cedula );
22    }
23 }
```

Implementada la clase, se puede crear los objetos que el programador establezca, en este caso se creará tres objetos:

```

1 public class Ejecutor {
2     public static void main(String[] args) {
3         Autor autor1 = new Autor();
4         Autor autor2 = new Autor("Jorge");
5         Autor autor3 = new Autor("Jorge","Agilar");
6     }
7 }
```

Las tres instancias provienen de la clase Autor, los nombres de cada objeto son “autor1”, “autor2” y “autor3” pero cada una invoca a un constructor diferente. El primer objeto utiliza un constructor por defecto, el segundo un constructor sobrecargado con un parámetro y el tercero hace referencia a otro constructor sobrecargado con dos parámetros. Cada objeto tiene acceso a sus atributos y métodos de manera independiente, este acceso se realiza a través del invocador “.”.

Para acceder a los atributos de un objeto se sigue la siguiente sintaxis:

nombreObjeto.atributo;

Para acceder a los métodos de un objeto se sigue la siguiente sintaxis:

nombreObjeto.método(  
);

El siguiente ejemplo muestra la instanciación de un objeto de la clase Autor y hace un llamado a sus tres atributos (nombres, apellidos y nombres) para inicializar sus valores y al método imprimir.

*Ejemplo 74 Ejecutor de la clase Autor*

```
1 public class Ejecutor {  
2     public static void main(String[] args) {  
3         Autor autor = new Autor();  
4         autor.nombres = "Luis Javier"  
5         autor.apellidos = "González Jaramillo"  
6         autor.cedula = "1104074815";  
7         autor.imprimirDatos();  
8     }  
9 }
```

Su salida es:

```
1 \programas_libro> java Ejecutor  
2 Nombre: Luis JavierApellidos: González JaramilloCédula: 1104074815  
3
```

## 7.4 MODIFICADORES DE ACCESO

La encapsulación es un mecanismo que consiste en declarar los atributos y métodos de una clase con niveles de acceso. Por lo tanto, la encapsulación garantiza la integridad de los datos que contiene un objeto. Los modificadores de acceso son parte del encapsulamiento de datos que consiste en hacer visible los datos de los atributos y procesos internos que ocurren en las operaciones que

realmente sean necesarios, controlando el acceso a los datos que conforman un objeto.

Para definir un modificador de acceso en los atributos y métodos se utilizan las siguientes palabras reservadas:

- **public**: se puede acceder desde cualquier clase.
- **private**: solo se puede acceder dentro de la clase.
- **protected**: se puede acceder desde una relación jerárquica, de la herencia entre padres a hijos.

Los modificadores se los ubica antes de la declaración de una clase, atributo, constructor o método.

#### 7.4.1 MODIFICADOR PÚBLICO

Al definir los atributos y los métodos con este modificador se puede acceder desde cualquier clase. Se utiliza la palabra reservada **public** para representarlo. En el siguiente ejemplo se declara la clase Libro y se han definido dos atributos, el constructor y el método con el modificador público, con ello se puede acceder sin restricción cuando se cree un objeto de la clase Libro desde cualquier otra clase inclusive si se la clase pertenece a otro paquete.

En UML el modificador **public** se representa con +:

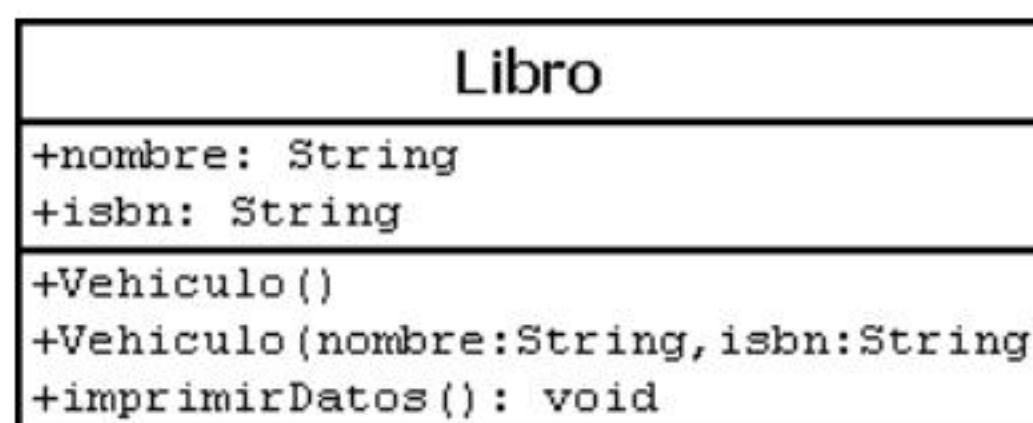


Figura 36 Modificador acceso público

```
1 | public class Libro {
2 |
3 |     public String nombre;
4 |     public String isbn;
5 |
6 |     public Libro(){}
7 |
8 |     public Libro(String nombre, String isbn){
9 |         this.nombre = nombre;
10 |        this.isbn = isbn;
11 |    }
12 |}
```

```
13 |     public void imprimirDatos(){  
14 |         System.out.println("Nombre: " + nombre + " isbn: " +  
15 |             isbn);  
16 |     }  
17 | }
```

Para acceder a los atributos o métodos público se sigue la siguiente sintaxis:

```
nombre_objeto.atributo;  
nombre_objeto.método();
```

Basados en su concepto y su sintaxis, se accederá a los atributos y método de la clase Libro desde la clase ejecutora:

```
1 |     public class Ejecutor {  
2 |         public static void main(String[] args) {  
3 |             Libro libro = new Libro();  
4 |             libro.nombre = "Jorge";  
5 |             libro.isbn = "7969097908";  
6 |             libro.imprimirDatos();  
7 |         }  
8 |     }
```

La clase Ejecutor tiene permiso para acceder a los atributos y métodos de la clase Libro porque se han declarado como públicos.

Su resultado es:

```
1 | \programas_libro>java Ejecutor  
2 | Nombre: Jorge isbn: 7969097908
```

#### 7.4.2 MODIFICADOR PRIVADO

Al definir los atributos y métodos con este modificador solo se puede acceder a ellos en la misma clase. Es el modificador que tiene un nivel de restricción alta. Ninguna clase podrá acceder a ellos aunque pertenezcan al mismo paquete o estén vinculado por alguna relación. En UML se representa con el carácter menos “-”.

Para acceder a los atributos privado es necesario definir un mecanismo para acceder a sus datos o manipularlos a través de los métodos getter y setter, su definición es la siguiente:

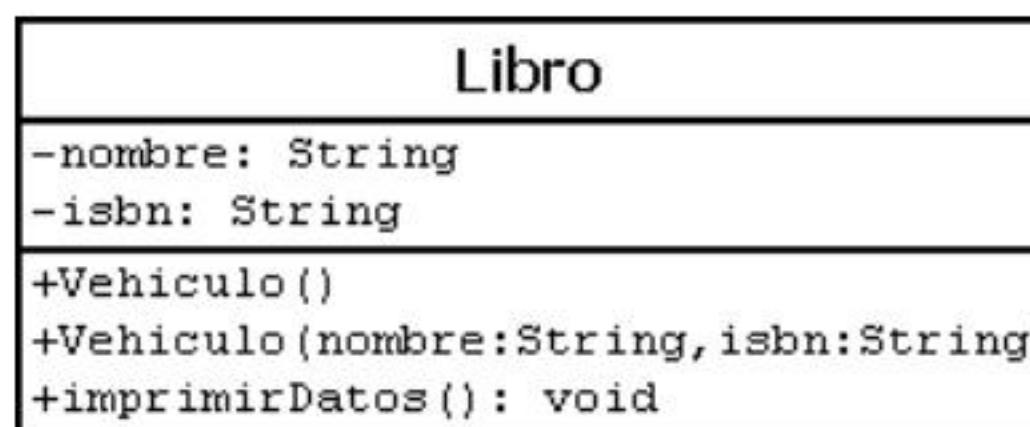
```
public void setAtributo(tipoDatos parámetro){  
    this.atributoPrivado = parámetro;  
}
```

```

public tipoDatos getAtributo(){
    return atributoPrivado;
}

```

En el siguiente ejemplo se declara la clase Libro y se han definido dos atributos y un método con el modificador privado, con ello no se podrá acceder directamente con el invocador “.” desde otra clase.



*Figura 37 Modificador acceso privado*

```

1 public class Libro {
2
3     private String nombre;
4     private String isbn;
5
6     public Libro(){} 
7
8     public Libro(String nombre, String isbn){
9         this.nombre = nombre;
10        this.isbn = isbn;
11    }
12
13     private void imprimirDatos(){
14         System.out.println("Nombre: " + nombre + " isbn: " +
15             isbn);
16    }
17 }

```

Para tener acceso a los atributos privados solo se puede hacer mediante los métodos getter y setter, estos métodos son públicos por lo que se pueden acceder a ellos desde cualquier clase.

*Ejemplo 75 Métodos getter y setter*

```

1 public class Libro {
2
3     private String nombre;
4     private String isbn;
5
6     public Libro(){} 
7
8     public Libro(String nombre, String isbn){
9         this.nombre = nombre;

```

```

10     this.isbn = isbn;
11 }
12
13 public void imprimirDatos(){
14     System.out.println("Nombre: " + getNombre() + " isbn: " +
15         getIsbn());
16 }
17
18 public String getNombre(){
19     return nombre;
20 }
21
22 public void setNombre(String nombre){
23     this.nombre = nombre;
24 }
25
26 public String getIsbn(){
27     return isbn;
28 }
29
30 public void setIsbn(String isbn){
31     this.isbn = isbn;
32 }
33
34 }
```

En este ejemplo por cada atributo se ha generado un método get y set. El siguiente paso es acceder a estos atributos mediante la clase ejecutora.

```

1 public class Ejecutor {
2     public static void main(String[] args) {
3         Libro libro = new Libro();
4         libro.nombre = "Jorge"; // error
5         libro.isbn = "7969097908"; // error
6     }
7 }
8 }
```

En la clase Ejecutor se crea un objeto de la clase Libro y se accede a los atributos de la forma convencional o directa, al compilar se genera errores de compilación “error: nombre has private access in Libro” debido a que el atributo está oculto para las todas las clases.

Su resultado es:

```

1 \Desktop> javac Ejecutor.java
2 Ejecutor.java:5: error: nombre has private access in Libro
3     libro.nombre = "Jorge";
4         ^
5 Ejecutor.java:6: error: isbn has private access in Libro
6     libro.isbn = "7969097908";
7         ^
8 2 errors
```

Modificará el código de la clase Ejecutor y utilizará los métodos getter y setter para fijar y acceder atributos privados.

```
1 public class Ejecutor {  
2     public static void main(String[] args) {  
3         public static void main(String[] args) {  
4             Libro libro = new Libro();  
5             libro.setNombre("Metodología de la Programación");  
6             libro.setIsbn("90876590899");  
7             System.out.println("Nombre: " + libro.getNombre());  
8             System.out.println("Isbn: " + libro.getIsbn());  
9         }  
10    }
```

Su salida es:

```
1 \Desktop> javac Ejecutor.java  
2 Nombre: Metodología de la Programación  
3 Isbn: 90876590899
```

#### **Recomendación**

Recordar que los atributos de una clase se declaren privados para dar un nivel de seguridad, restringiendo el acceso a los atributos y método evitan el mal uso de sus datos.

#### **Recomendación**

Al declarar atributos privado por defecto el programador debe generar los métodos getter y setter de cada atributo privado.

Se cita algunas reglas de convención para generar los métodos getter y setter:

Tabla 15 Reglas para generar los métodos get y set

| Concepto                                                                                            | Codificación                                                                                       |
|-----------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Los atributos deben ser privados                                                                    | <code>private String nombre;</code><br><code>private boolean contratado;</code>                    |
| El nombre de los métodos getter inician con "get" si el tipo de dato no es un dato de tipo booleano | <code>public int getNombre() {</code><br><code>    return nombre;</code><br>}                      |
| El nombre de los métodos getter inician con "is" si el tipo de dato es booleano                     | <code>public boolean isContratado() {</code><br><code>    return contratado;</code><br>}           |
| El nombre de los métodos setter inician con "set"                                                   | <code>public void setNombre(String nombre) {</code><br><code>    this.nombre = nombre;</code><br>} |

### 7.4.3 MODIFICADOR PROTEGIDO

Permite el acceso a los atributos, constructores y métodos desde la misma clase, clases del mismo paquete y clases que hereden de ella. En UML se representa con el carácter "#". Ahora definir un diagrama UML para ver el comportamiento de los atributos protegidos:

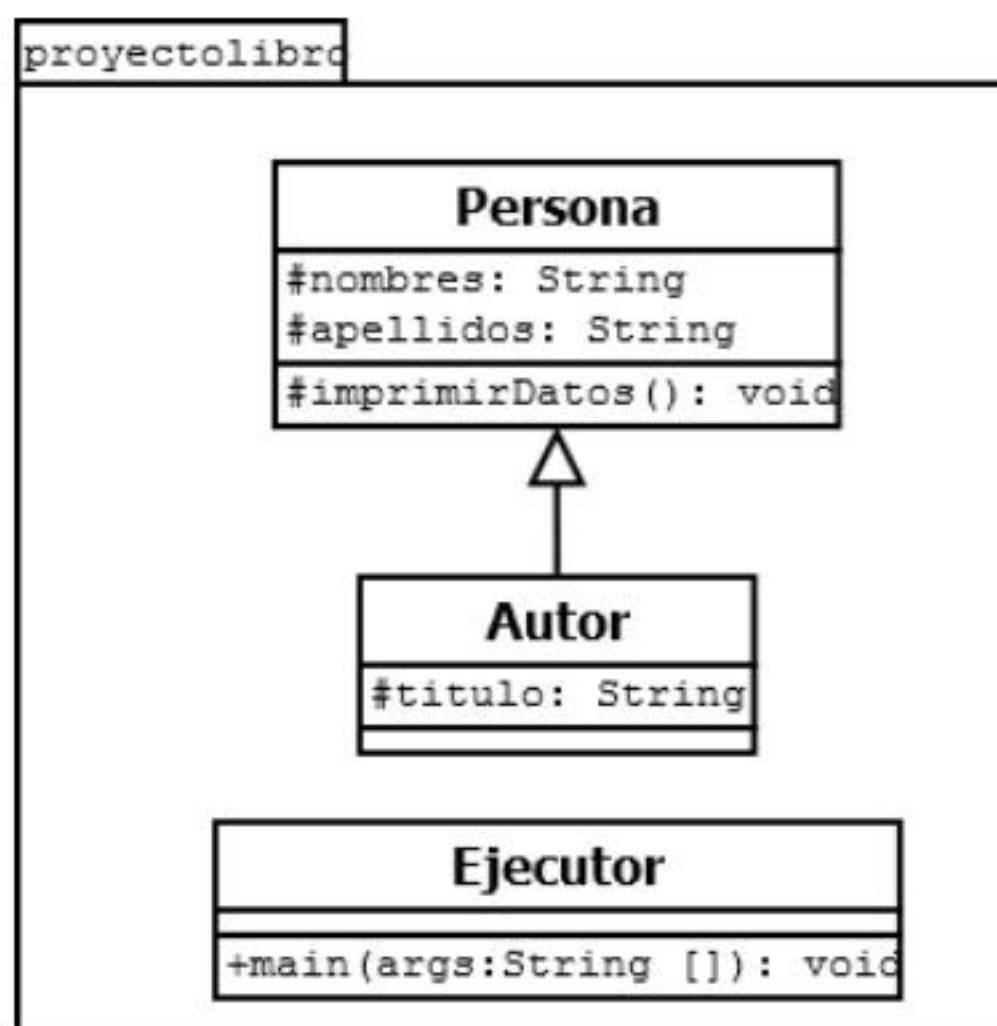


Figura 38 Diagrama paquete proyecto libro

En este diagrama, las tres clases se encuentran en el mismo paquete (proyectolibro) y existe una relación de herencias entre las clases Persona y Autor. Persona es padre de Autor. El código de las clases es la siguiente:

Ejemplo 76 Clase Persona con atributos y método protegidos

```
1 package proyectolibro;
2
3 public class Persona {
4     protected String nombres;
5     protected String apellidos;
6
7     protected void imprimirDatos(){
8         System.out.println("Nombre: " + nombres + "Apellidos: " +
9             apellidos );
10    }
11 }
```

Ejemplo 77 Clase Autor que hereda de Persona

```
1 package proyectolibro;
2
3 public class Autor extends Persona{
4     private String titulo;
5
6     public String getTitulo(){
7 }
```

```
7     return titulo;
8 }
9
10    public void setTitulo(String titulo) {
11        this.titulo = titulo;
12    }
13 }
```

Ejemplo 78 Clase Ejecutor

```
1 package proyectoLibro;
2
3 public class ProyectoLibroJava {
4
5     public static void main(String[] args) {
6         Autor autor = new Autor();
7         autor.nombres = "María";
8         autor.apellidos = "Jaramillo";
9         autor.imprimirDatos();
10    }
11 }
```

La clase Autor establece una relación de herencia con la clase Persona, por lo que hereda sus atributos y métodos mientras que la clase ProyectoLibroJava es la clase ejecutora y crea un objeto de la clase Autor y en las líneas 5, 6 y 7 accede de manera directa a los atributos y método de Autor que han sido heredados de Persona, las tres clases al pertenecer al mismo paquete (proyectoLibro) no genera ningún error en su acceso porque todo elemento que se ha definido como protegido se puede acceder sin restricciones en el mismo paquete.

Ahora realizar una variante para analizar el comportamiento de los elementos protegidos. Se crea un nuevo paquete “editorial” y se mueve la clase Persona al paquete como lo ilustra la siguiente figura:

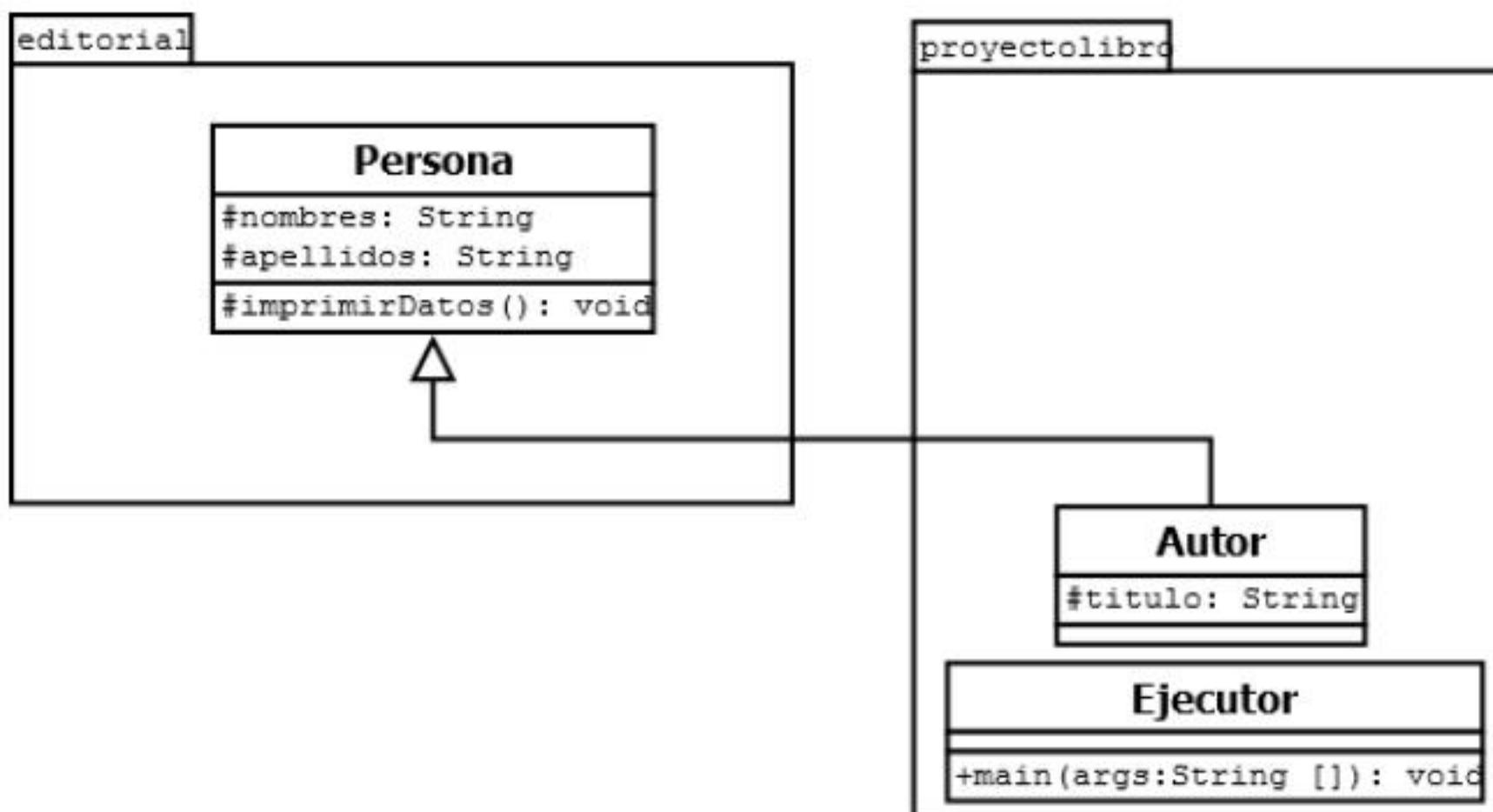


Figura 39 Diagrama de paquetes

Ejecutar la clase Ejecutor sin modificar el código el resultado es el siguiente:

```

1  \Desktop> javac Ejecutor.java
2  ProyectoLibroJava.java:9: error: nombres has protected access in Persona
3      autor.nombres = "María";
4          ^
5  ProyectoLibroJava.java:10: error: apellidos has protected access in Persona
6      autor.apellidos = "Jaramillo";
7          ^
8  ProyectoLibroJava.java:11: error: imprimirDatos() has protected access in Persona
9      autor.imprimirDatos();
10     ^
11  3 errors

```

Se establece que solo se puede acceder a los atributos protegido de forma directa desde clases que estén en el mismo paquete o cuando se ha establecido una herencia. En el caso de la clase Ejecutor trata de acceder a los atributos de la clase Autor.

**¿Pero ambas clase pertenecen al paquete “proyectolibro”?**

¡Se supone que no debería generar errores de compilación!

En realidad sí se generará errores porque Autor hereda atributos protegidos de la clase Persona que está ubicado en el paquete “editorial”, al ser protegidos solo Autor podrá acceder a esos atributos al establecer una herencia como se muestra en el siguiente código:

```

1 package proyectoLibro;
2
3 import editorial.Persona;
4
5 public class Autor extends Persona{
6     private String titulo;
7
8     public String getTitulo() {
9         return titulo;
10    }
11
12    public void setTitulo(String titulo) {
13        this.titulo = titulo;
14    }
15
16    public void presentarInformación(String apellidos, String
17                                    nombres){
18        this.apellidos = apellidos;
19        this.nombres = nombres;
20        imprimirDatos();
21    }
22}

```

En las líneas 18, 19 y 20 se utiliza los elementos protegidos de la clase Persona desde la clase Autor, cada una pertenece a diferentes paquetes, en este caso es un código correcto porque los elementos protegidos solo se da en herencias y clases del mismo paquete.

#### 7.4.4 MODIFICADOR POR DEFECTO

Este modificador se establece cuando el programador no define ningún modificador a los atributos o métodos de la clase, permite usar los elementos en la propia solo en la misma clase y las clases del mismo paquete. Según algunos autores de libros en java se lo denomina como modificador friendly o de paquete. Su símbolo en UML es a través del carácter “~”.

Si se utiliza el diagrama UML anterior con la diferencia que se emite el modificador de acceso de los atributos y métodos de la clase Persona se genera el siguiente código:

```

1 package proyectoLibro;
2
3 public class Persona {
4     String nombres;
5     String apellidos;
6
7     void imprimirDatos(){

```

```
8     System.out.println("Nombre: " + nombres + "Apellidos: " +
9         apellidos );
10    }
11 }
```

Ejecutar la clase Ejecutora:

Su salida es:

```
1 \Desktop> javac Ejecutor.java
2 Autor.java:17: error: apellidos is not public in Persona; cannot be accessed fro
3 m outside package
4     this.apellidos = apellidos;
5     ^
6 Autor.java:18: error: nombres is not public in Persona; cannot be accessed from
7 outside package
8     this.nombres = nombres;
9     ^
10 Autor.java:19: error: cannot find symbol
11     imprimirDatos();
12     ^
13     symbol: method imprimirDatos()
14     location: class Autor
15 ProyectoLibroJava.java:9: error: nombres is not public in Persona; cannot be acc
16 essed from outside package
17     autor.nombres = "María";
18     ^
19 ProyectoLibroJava.java:10: error: apellidos is not public in Persona; cannot be
20 accessed from outside package
21     autor.apellidos = "Jaramillo";
22     ^
23 ProyectoLibroJava.java:11: error: cannot find symbol
24     autor.imprimirDatos();
25     ^
26     symbol: method imprimirDatos()
27     location: variable autor of type Autor
28 6 errors
29
30
31
32
```

Se genera errores porque la herencia se da en clases de diferentes paquetes, un modificador por defecto solo comparte sus atributos o métodos estén estrictamente en el mismo paquete. Para corregir estos errores usando el modificador por defecto se debe mover las clases a un mismo paquete.

En la siguiente tabla se resume los niveles de accesibilidad que tienen los modificadores de acceso:

Tabla 16 Modificadores de acceso

|                    | En la misma clase | Subclase en el mismo paquete - herencia | Otra clase del mismo paquete | Subclase de otro paquete - herencia | Otra clase de otro paquete |
|--------------------|-------------------|-----------------------------------------|------------------------------|-------------------------------------|----------------------------|
| <b>Público</b>     | X                 | X                                       | X                            | X                                   | X                          |
| <b>Privado</b>     | X                 |                                         |                              |                                     |                            |
| <b>Protegido</b>   | X                 | X                                       |                              |                                     | X                          |
| <b>Por defecto</b> | X                 | X                                       | X                            |                                     |                            |

## 7.5 SENTENCIA this

La palabra reservada **this** es una referencia al propio objeto en el que se está trabajando, puede referirse a los atributos, métodos y constructores del objeto actual.

Se utiliza cuando existe ambigüedad entre atributos y parámetros en los constructores o métodos, es decir, cuando ambos tienen el mismo identificador. Al aplicar **this**, hace la diferencia entre ambos, el que tenga la referencia **this** es el atributo y el que no tenga la referencia **this** es el parámetro. Siempre el parámetro inicializa o le asigna un valor al atributo de la clase. En la siguiente figura se ilustra el comportamiento de la sentencia **this**.

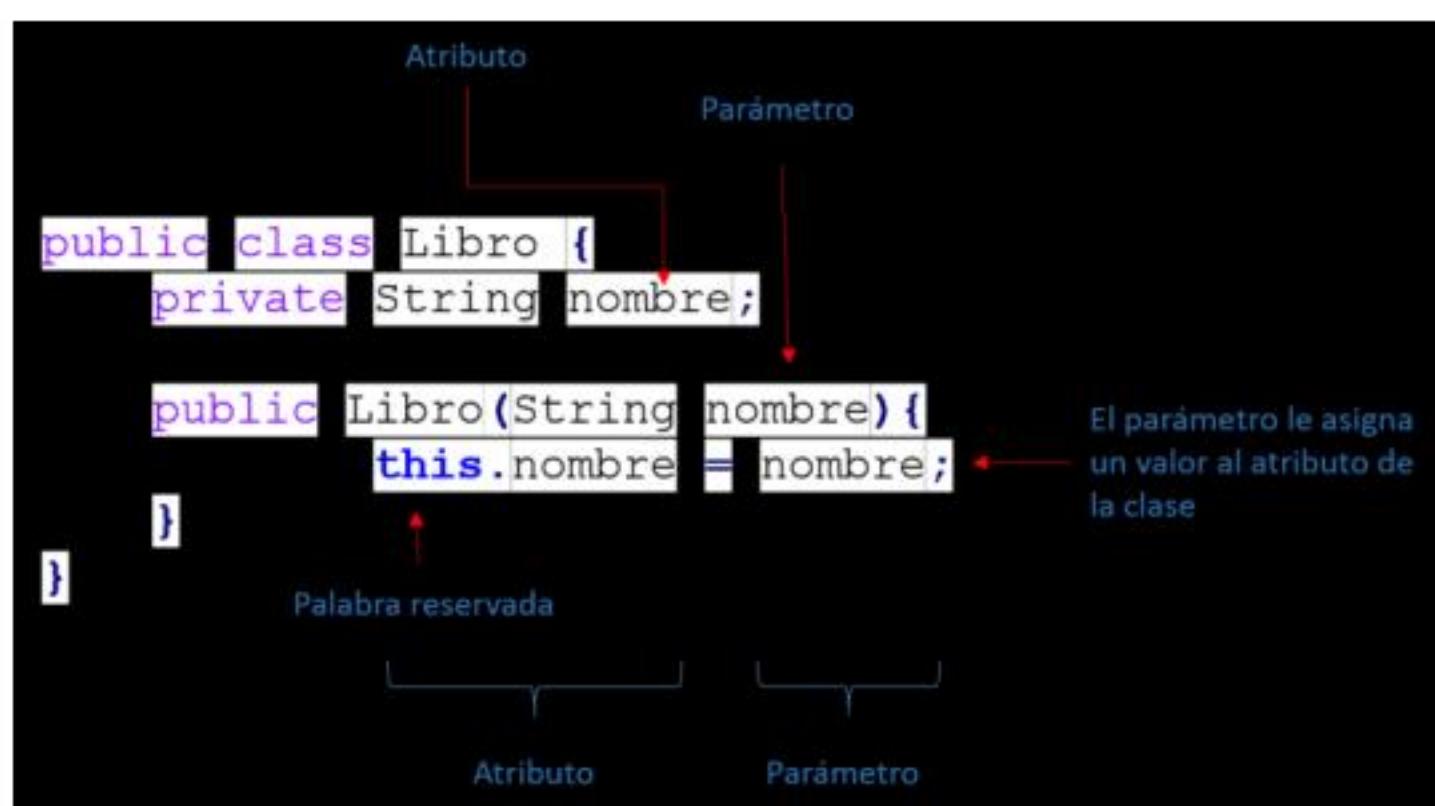


Figura 40 this, diferencia entre atributo y parámetro

Para usar **this**, el atributo y el parámetro deben tener el mismo identificador. Pero si el programador define con otro identificador al parámetro se puede omitir **this** ya que no existirán ambigüedades en los identificadores como se ilustra en el siguiente ejemplo:

*Ejemplo 79 Atributo y parámetro no ambiguos*

```
1 public class Libro {  
2     public String nombre;  
3  
4     public Libro(String no){  
5         nombre = no; // se omite this  
6     }  
7 }
```

Otra característica que tiene **this**, es que se puede invocar a los constructores de la misma clase, el llamado siempre se debe hacer en la primera línea del método de invocación.

En siguiente ejemplo, en la línea 6 se realiza el llamado del constructor con dos parámetros declarado en la línea 10 a través de **this** mediante la sentencia “**this (nombre, "9867097798");**”. La sentencia **this** se convierte en un constructor implícito.

```
1 public class Libro {  
2  
3     private String nombre;  
4  
5     public Libro(String nombre){  
6         this(nombre, "9867097798");  
7         this.nombre = nombre;  
8     }  
9  
10    public Libro(String nombre, String isbn ){  
11        this.imprimirDatos();  
12    }  
13  
14    public void imprimirDatos(){  
15        System.out.println("Nombre: " + nombre + " isbn: " +  
16                    isbn);  
17    }  
18}  
19}
```

La última característica de **this** es que puede llamar a los métodos de la misma clase. En la línea 11, se invoca al método **imprimirDatos( )** a través de **this** en el constructor con parámetros.

A través de la siguiente tabla se resume las funcionalidades del **this**:

| Sentencia            | Descripción                              |
|----------------------|------------------------------------------|
| <b>this</b>          | Hace referencia al objeto actual.        |
| <b>this.atributo</b> | Hace referencia al atributo de la clase. |

---

|                                      |                                                                                                       |
|--------------------------------------|-------------------------------------------------------------------------------------------------------|
| <code>this(parámetros)</code>        | Llama a un constructor del objeto actual. Sólo se puede llamar en la primera línea de un constructor. |
| <code>this.método(parámetros)</code> | Llamar a un método del objeto actual.                                                                 |

---

## 7.6 ATRIBUTOS Y MÉTODOS ESTÁTICOS

Los elementos estáticos permiten llamar a los atributos o métodos de la clase sin necesidad de crear objetos. Hasta esta sección se ha trabajado con objetos con el objetivo de invocar a los atributos y métodos, pero estas llamadas eran en base a cada objeto y los cambios efectuados solo afectaban al objeto.

Ahora realizar las llamadas para que afecten a todos los objetos a través de elementos estáticos. Como su palabra lo definen los elementos estáticos afectan a todos los objetos por lo que se lo denomina atributos o métodos de clase. La palabra reservada que establece que un elemento es estático es `static`.

Se definen métodos estáticos cuando las tareas son comunes en todos los objetos.

Para definir un atributo estático se sigue la siguiente sintaxis:

modificador\_acceso **static** nombreAtributo

Para definir un método estático se sigue la siguiente sintaxis:

modificador\_acceso **static** tipo\_dato nombreMétodo (parámetros)

Definir la clase Libro con un atributo y método estático:

```
1 | public class Libro {  
2 |  
3 |     public static String nombre;  
4 |  
5 |     public static void imprimirDatos(){  
6 |         System.out.println("Nombre: " + nombre);  
7 |     }  
8 | }
```

En este ejemplo no hace falta crear un objeto para acceder a sus componentes porque son estáticos. Para llamar a cualquier método estático, se especifica el nombre de la clase, seguido de un punto (.) y del nombre del método como se indica en el siguiente código:

```
1 | public class Ejecutor {  
2 |     public static void main(String[] args) {  
3 |         Libro fisica = new Libro();
```

```

4 Libro programacion = new Libro();
5 Libro redes = new Libro();
6
7 Libro.nombre = "Libro Universal";
8
9 fisica.imprimirDatos();
10 programacion.imprimirDatos();
11 redes.imprimirDatos();
12 }
13 }
```

Se crea tres objetos de la clase Libro, en la línea 7 se realiza una asignación a un atributo estático, afectando a todos los objetos, cada uno se fija el mismo valor al atributo nombre porque ha sido definido como estático. Su salida es:

```

1 \Desktop> javac Ejecutor.java
2 Nombre: Libro Universal
3 Nombre: Libro Universal
4 Nombre: Libro Universal
```

## 7.7 CUESTIONARIO

### 1. La definición de clase es:

- Una instancia para acceder a los atributos y métodos.
- Una plantilla para definir atributos y métodos que tendrán los objetos.
- Un tipo de dato enumerado, o de referencia.
- Uno de los principales fundamentos de la POO junto con el encapsulamiento, herencia y polimorfismo.

### 2. Un atributo es:

- Un elemento que establece las acciones las clases.
- Una propiedad o característica de una clase.
- Un tipo de dato de tipo primitivo.
- Un elemento que define en los elementos llamados variable local.

### 3. Un constructor es:

- Un método especial que no define ningún parámetro.
- Un método especial que permite inicializar las variables de la clase.
- Un método especial que permite instanciar los objetos.
- Un método especial cuyo identificador lo define el programador.

### 4. Evaluar el siguiente ejercicio y seleccionar la opción correcta:

```

1 public class Cancion {
2     public String nombre;
3     public String genero;
4 }
```

- Se genera un error de compilación al no definir el constructor.

- No se podrán crear objetos de la clase Canción al omitirse los constructores.
- El programador está en la obligación de implementar constructores de la clase Canción.
- Al no definir ningún constructor en la Clase Cancion, java le asigna un constructor por defecto.

### **5. Evaluar el siguiente ejercicio y seleccionar la opción correcta:**

```

1 public class Cancion {
2     public String nombre;
3     public String genero;
4
5     public Cancion(String nombre, String genero) {
6         this.nombre = nombre;
7         this.genero = genero;
8     }
9 }
```

- Se genera un error de compilación al no definir el constructor por defecto.
- El constructor con parámetros es un método sobrecargado.
- Existe redundancia de términos entre los parámetros y los atributos, en el constructor se deben redefinir los identificadores de los parámetros.
- El Constructor con parámetros inicializa dos atributos a través de los parámetros.

### **6. Se establece la sobrecarga de constructores cuando:**

- Existen más de dos constructores con el mismo número parámetros.
- Existen más de dos constructores con diferentes parámetros, pero devuelven tipos de datos diferentes.
- Existen más de dos constructores con diferentes parámetros.
- Se define el constructor con parámetros.

### **7. Evaluar el siguiente código y seleccionar la opción correcta que permita crear un objeto de la clase Canción:**

```

1 public class Cancion {
2     public String nombre;
3     public String genero;
4
5     public Cancion(String nombre, String genero) {
6         this.nombre = nombre;
7         this.genero = genero;
8     }
9 }
```

- Cancion canción = new Cancion( );
- Canción canción = Cancion("Mi primer día sin ti", "balada") ;
- Canción canción = new Cancion("Mi primer día sin ti", "balada") ;
- Canción canción = Cancion( ) ;

**8. El elemento que permite invocar métodos y atributos es:**

- Método get.
- ":"
- ";"
- ""
- Método set.

**9. Un modificador de acceso público permite:**

- Acceder a los atributos y métodos desde cualquier clase, independientemente si se encuentra en otro paquete.
- Acceder a los atributos y métodos solo en la misma clase ocultando los parámetros de implementación de la clase.
- Acceder a los atributos y métodos siempre que se encuentren en el mismo paquete.
- Acceder a los atributos y métodos cuando existe una relación de herencia, independientemente si se la clase padre se encuentra en otra paquete.

**10. Un modificador de acceso privado permite:**

- Acceder a los atributos y métodos desde cualquier clase, independientemente si se encuentra en otro paquete.
- Acceder a los atributos y métodos solo en la misma clase ocultando los parámetros de implementación de la clase.
- Acceder a los atributos y métodos siempre que se encuentren en el mismo paquete.
- Acceder a los atributos y métodos cuando existe una relación de herencia, independientemente si se la clase padre se encuentra en otra paquete.

**11. Un modificador de acceso protegido permite:**

- Acceder a los atributos y métodos desde cualquier clase, independientemente si se encuentra en otro paquete.
- Acceder a los atributos y métodos solo en la misma clase ocultando los parámetros de implementación de la clase.
- Acceder a los atributos y métodos siempre que se encuentren en el mismo paquete.
- Acceder a los atributos y métodos cuando existe una relación de herencia, independientemente si se la clase padre se encuentra en otra paquete.

**12. Un modificador de acceso por defecto permite:**

- Acceder a los atributos y métodos desde cualquier clase, independientemente si se encuentra en otro paquete.
- Acceder a los atributos y métodos solo en la misma clase ocultando los parámetros de implementación de la clase.
- Acceder a los atributos y métodos siempre que se encuentren en el mismo paquete.
- Acceder a los atributos y métodos cuando existe una relación de herencia, independientemente si se la clase padre se encuentra en otra paquete.

### **13. La palabra reservada this permite:**

- Hacer referencia al propio objeto en el que se está trabajando, puede referirse a los atributos, métodos y constructores del objeto actual.
- Hace referencia a la clase Padre, invocando a sus atributos y métodos.
- Hacer referencia al propio objeto pero limitando el acceso a los constructores de la clase.
- Hacer referencia al objeto actual, y para acceder a un constructor se utiliza super siempre que se defina en la primera línea de método invocador.

### **14. Analizar el código y seleccionar la respuesta correcta:**

```
1 public class Cancion {  
2     public String nombre;  
3     public String genero;  
4  
5     public Cancion0 {  
6         System.out.println("Se llama al constructor con  
7             parámetro");  
8         this("Mi primer día sin ti", "balada");  
9     }  
10  
11    public Cancion(String nombre, String genero) {  
12        this.nombre = nombre;  
13        this.genero = genero;  
14    }  
15 }
```

- En línea 8 this puede inicializar atributos pero no puede llamar a constructores de la misma clase.
- En línea 8 se genera un error de Ejecución.
- En línea 8 se genera un error de Compilación.
- En línea 8 this puede corregir las ambigüedades que se generan cuando el atributo y el parámetro tienen el mismo identificador, pero siempre deben ser declarados en la primera línea del método que los utiliza.

### **15. Para acceder a un método estático se debe seguir la siguiente sintaxis:**

- identificadorObjeto.método.
- identificadorObjeto.método(parámetros).
- identificadorClase.método.
- identificadorClase.método(parámetros).

### **16. Analizar el código y seleccionar la respuesta correcta:**

```
1 public class Algoritmo {  
2     private static int $;  
3     public static void main(String[] args) {  
4         String a_b;  
5         System.out.print($);  
6         System.out.print(a_b);  
7     }
```

8 | }

- Error de compilación en la línea 1.
- Error de compilación en la línea 2.
- Error de compilación en la línea 4.
- Error de compilación en la línea 5.
- Error de compilación en la línea 6.
- El programa imprime null.

**17. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public class Albacora {  
2     int cantidad;  
3     public void getPicudo0 {  
4         cantidad = 3;  
5     }  
6     public static void main(String [] args){  
7         Albacora al = new Albacora();  
8         System.out.println(al.cantidad);  
9     }  
10 }
```

- Error de compilación en la línea 8.
- Imprime 4.
- Imprime 6.
- Imprime null.
- Error de ejecución.
- Imprime.
- Imprime 0.
- Imprime 2.

**18. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public class BotellaAgua {  
2     private String descripcion;  
3     private boolean vacia;  
4     public static void main(String[] args) {  
5         BotellaAgua wb = new BotellaAgua();  
6         System.out.print("Vacía = " + wb.vacia);  
7         System.out.print(", Descripción = " + wb.descripcion);  
8     }  
9 }
```

- Vacía = , Descripción =
- Vacía = false, Descripción = null.
- Vacía = false, Descripción =
- Vacía = true, Descripción = null.
- Vacía = true, Descripción =
- Error de compilación en la línea 6.
- Error de compilación en la línea 7.
- Error de ejecución.

## **8 DATOS DE TIPO ENUMERADO**

---

En este capítulo se realiza una descripción de los datos de tipo enumerados, su concepto, estructura y sintaxis. A través ejemplos se explicará su funcionamiento y la ventaja de implementarlo en las clases. Además se establece la diferencia entre los tipos de datos primitivos y los de referencia. Por último se implementa constructores, atributos y métodos en los enum.

### **Objetivos**

- Describir la sintaxis y funcionamiento de los tipos datos enumerado.
- Establecer la diferencia entre tipos enumerados, primitivos y de referencia.
- Representa un tipo de datos enumerado en UML.

### **8.1 INTRODUCCIÓN**

En la vida cotidiana se puede encontrar atributos o variables que contengan datos predeterminados, un ejemplo es el estado civil de cada persona, al nacer, su estado civil es soltero, luego al contraer matrimonio, cambia a “casado”, Finalmente al envejecer la pareja fallece, cambia a “viudo”. En este caso, el estado civil ha cambiado en tres ocasiones. Estos valores son predeterminados, y solo contendrá un valor de los cuatro posibles (soltero, casado, divorciado y viudo). No existe un estado civil con valores como “si relación”, “solitario”, “en espera”, etc. Por lo tanto, un dato enumerado es un conjunto de datos predeterminados o estáticos.

Algunos otros datos predeterminados que se puede tomar como ejemplo: los días de la semana, los meses del año, los colores. etc. Por ello, los datos de tipo enumerado sirven para restringir el contenido de una variable a una serie de valores predefinidos.

## 8.2 DEFINICIÓN

Son datos de tipo especial que permiten que un atributo o variable tenga un conjunto finito de constante predeterminada. Los atributos o variables deben contener un valor que haya sido definido en el tipo de dato enumerado.

Para definir un dato de tipo enumerada se utilizan la palabra reservada **enum**. Su sintaxis es:

```
public enum NombreEnum{  
    ELEMENTE1, ELEMENTO2, .....,  
}
```

La forma de nombrar a los **enum** es igual al de las clases, por ello se debe iniciar con mayúscula y el resto en minúscula. Dentro de las llaves se declarar la lista de valores y cada elemento enumerado debe ser separado por "," y tendrán un comportamiento igual a los objetos.

El punto y coma al final de la declaración de los tipos enumerados es opcional.

Los elementos del **enum** de forma implícita son datos declarados como públicos estáticos (**public final static**).

### Recomendación

Por convención, los valores del **enum**, llamados enums (valores o elementos), al ser constantes lo recomendable es escribirlos en mayúsculas para recordar que son valores fijos.

Tomar el ejemplo del estado civil, sus valores predeterminados: SOLTERO, CASADO, DIVORCIADO Y VIUDO. En Java, los tipos de datos enumerados se pueden definir dentro o fuera de la clase. Empezar declarando el estado civil dentro de la clase Persona:

Ejemplo 80 Dato de tipo enumerado definido dentro de la clase

```
1 public class Persona {  
2  
3     enum EstadoCivil{  
4         SOLTERO, CASADO, DIVERCIADO, VIUDO  
5     };  
6  
7     public String nombres;  
8     public String apellidos;  
9     public EstadoCivil estadoCivil;  
10 }
```

En la clase Ejecutor llamaremos al tipo de dato enumerado declarada en la clase Persona:

*Ejemplo 81 Clase Ejecutora, llamada de un dato enumerado declarado dentro de una clase*

```
1 public class Ejecutor {  
2  
3     public static void main(String[] args) {  
4         Persona persona = new Persona();  
5         persona.nombres = "Manuel Vicenye";  
6         persona.apellidos = "Jaramillo González";  
7         persona.estadoCivil = Persona.EstadoCivil.DIVERCIADO;  
8  
9         Persona p = new Persona();  
10        p.nombres = "Diana Patricia";  
11        p.apellidos = "Valarezo Álvarez";  
12        p.estadoCivil = Persona.EstadoCivil.CASADO;  
13    }  
14 }
```

Los tipos de datos enumerados son valores estáticos y para acceder a ellos se realiza directamente con el nombre de la clase, su sintaxis es:

NombreClase.nombreEnum.elemento

En la línea 7 y 12 se acceden a dos tipos enumerados diferentes para cada objeto Persona. A través de estos tipos de datos, se limita a que se le asigne otros valores al estado civil que no estén definidos en el enum.

Al utilizar este tipo de datos, se restringe al usuario el ingreso erróneo de información, facilitan la legibilidad de los programas y permiten que el compilador chequee errores.

### Recomendación

No es recomendable que se defina un dato enumerado en la misma clase ya que se mezcla código de diferentes estructuras, haciendo que las aplicaciones no sean legibles.

La segunda forma de declarar un tipo de dato enumerado es fuera de la clase, en un archivo aparte. Para ello, es necesario realizar su representación en UML, para diferenciar entre una clase y un enum, se utiliza una etiqueta <<enum>>. Su representación es:

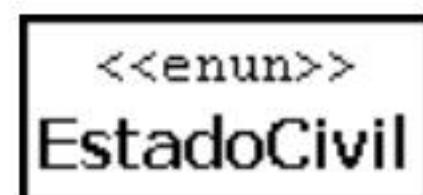


Figura 41 Representación de un tipo de dato enumerado un UML

En el ejercicio se establecerá una relación entre la clase Persona y el dato de tipo enumerado, en UML se representa:



Figura 42 Relación entre un dato de tipo enumerado y una clase

Los **enum** tienen casi las mismas características que una clase y su extensión es .java, al traducir del modelo UML a lenguaje Java se genera el siguiente código:

Ejemplo 82 Dato de tipo enumerada declarado fuera de la clase

```

1 public enum EstadoCivil{
2     SOLTERO,
3     CASADO,
4     DIVERCIADO,
5     VIUDO
6 }
```

El **enum** EstadoCivil se convierte en un tipo de dato, el mismo que puede ser referenciado.

Ejemplo 83 Relación entre una clase u un tipo de dato enumerado

```

1 public class Persona {
2     public String nombres;
3     public String apellidos;
4     public String cedula;
5     public EstadoCivil estadoCivil;
6 }
```

Ejemplo 84 Clase Ejecutora - enum codificado fuera de la clase

```

1 public class Ejecutor {
2
3     public static void main(String[] args) {
4         Persona persona = new Persona();
5         persona.nombres = "Manuel Vicenye";
6         persona.apellidos = "Jaramillo González";
7         persona.estadoCivil = EstadoCivil.DIVERCIADO;
8
9         Persona p = new Persona();
10        p.nombres = "Diana Patricia";
11        p.apellidos = "Valarezo Álvarez";
```

```

12     p.estadoCivil = EstadoCivil.CASADO;
13 }
14 }
```

Es recomendable que los datos de tipo enumerado se separen de las clases y se declaren en archivos separados, Para acceder a este tipo de dato su sintaxis es:

NombreEnum.elemento

Todos los **enum** heredan de la la clase *Enum* que está en el paquete `java.lang.Enum` por lo que tiene un conjunto de métodos heredados:

*Tabla 17 Métodos de la clase Enum*

| Método                            | Descripción                                                                                        |
|-----------------------------------|----------------------------------------------------------------------------------------------------|
| <code>name()</code>               | Devuelve un String con el nombre de la constante.                                                  |
| <code>toString()</code>           | Devuelve un String con el nombre de la constante.                                                  |
| <code>ordinal()</code>            | Devuelve un entero con la posición del <b>enum</b> según está declarada.                           |
| <code>compareTo(Enum otro)</code> | Compara el <b>enum</b> con el parámetro según el orden en el que están declarados lo <b>enum</b> . |
| <code>Demarcacion.values()</code> | Devuelve un array que contiene todos los <b>enum</b> .                                             |

En la clase ejecutora, se llamará a cada método del **enum** en el tipo `EstadoCivil` e imprimiremos su resultado.

*Ejemplo 85 Clase Ejecutora - métodos de la clase Enum*

```

1 public class Ejecutor {
2
3     public static void main(String[] args) {
4         Persona persona = new Persona();
5         persona.nombres = "Manuel Vicente";
6         persona.apellidos = "Jaramillo González";
7         persona.estadoCivil = EstadoCivil.DIVERCIADO;
8
9         Persona p = new Persona();
10        p.nombres = "Diana Patricia";
11        p.apellidos = "Valarezo Álvarez";
12        p.estadoCivil = EstadoCivil.CASADO;
13
14        System.out.println("Nombre: " + p.estadoCivil.name());
15        System.out.println("toString: " +
16                           p.estadoCivil.toString());
17        System.out.println("Posición: " +
18                           p.estadoCivil.ordinal());
19        System.out.println("Comparar: " +
20                           p.estadoCivil.compareTo(persona.estadoCivil));
21        for(EstadoCivil d: p.estadoCivil.values()){
22            System.out.println(d.toString() + " - ");
23        }
24    }
```

Su salida es:

```
1 \programas_libro>java Ejecutor
2 Nombre: CASADO
3 toString: CASADO
4 Posición: 1
5 Comparar: -1
6 SOLTERO -
7 CASADO -
8 DIVERCIADO -
9 VIUDO -
```

### 8.3 DATOS DE TIPO ENUMERADO AVANZADOS

En un **enum**, sus elementos se comportan como objetos y se pueden definir campos, métodos e incluso constructores. El compilador añade automáticamente algunos métodos especiales cuando crea un **enum** descritos en la tabla anterior.

Definir un **enum** País con atributos, un constructor y métodos, ejemplo: a país tiene como atributos el nombre y un dominio (Colombia con su dominio co). Para definir el **enum**, existen algunos aspectos de ser considerados para declararlos:

1. **Definir el nombre del enum:** su estructura es similar al de la clase, y estará sujeta a las reglas de convención.
2. **Declarar los elementos del enum:** cada elemento estará separado por una coma, el identificador debe ser escrito en mayúscula como “ALEMANIA” y “ECUADOR”, etc.
3. **Definir los atributos:** Cada elemento definirá uno o más atributos dentro de los paréntesis y estarán separados por comas. “nombre” y “dominio” son los atributos de “País”.
4. **Definir el constructor:** debe ser privado o de paquete; y se define los parámetros dependiendo del número de atributos del **enum**. Es obligatorio codificar un constructor cuando se declaran atributos pero no se lo puede invocar de forma directa.
5. **Generar métodos:** se pueden definir cualquier método privado, público, etc.

En la figura se ilustra las principales partes de un **enum** avanzado.

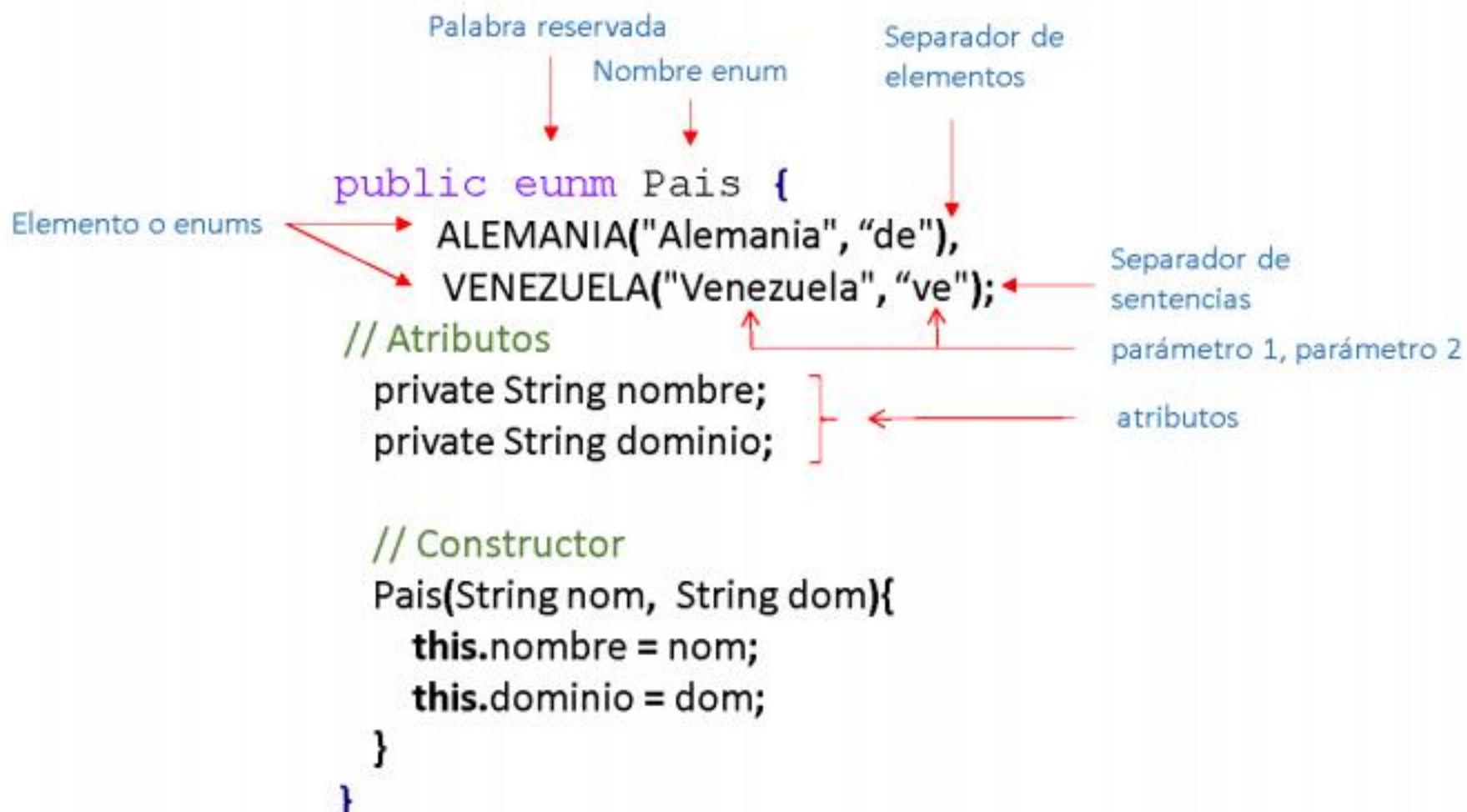


Figura 43 Dato de tipo enumerado avanzado

Este ejemplo, incluye la declaración de un **enum** llamado País con las características de una clase, con atributos, su constructor y los métodos getter y setter.

Ejemplo 86 Declaración de un enum como clase

```

1  public enum Pais {
2      ALEMANIA("Alemania", "de"),
3      ECUADOR("Ecuador", "ec"),
4      ESPAÑA("España", "es"),
5      VENEZUELA("Venezuela", "ve");
6
7      // Atributos
8      private String nombre;
9      private String dominio;
10
11     // Constructor
12     Pais(String nom, String dom){
13         this.nombre = nom;
14         this.dominio = dom;
15     }
16
17     public String toString(){
18         return this.getNombre() + " - " + this.getDominio();
19     }
20
21     public String getNombre(){
22         return nombre;
23     }
24
25     public void setNombre(String nombre){
26         this.nombre = nombre;
27     }
28
29     public String getDominio(){
30         return dominio;
31     }
32 }
```

```

31 }
32
33 public void setDominio(String dominio) {
34     this.dominio = dominio;
35 }
36 }
```

En un dato de tipo enumerado se genera una instancia cuando se hace un llamado al **enum** correspondiente País, en este instante, todos los elementos se inicializan y ejecutan los constructores según el orden de cada instancia definida. Por ejemplo para crear instancias y utilizar sus elementos se definirá algunas instancias:

*Ejemplo 87 Clase Ejecutora - enum como clase*

```

1 public class ProyectoLibroJava {
2
3     public static void main(String[] args) {
4         System.out.println(Pais.ESPAÑA);
5         System.out.println(Pais.VENEZUELA);
6         Pais ecuador = Pais.ECUADOR;
7         System.out.println("Nombre del país es: " +
8                           ecuador.getNombre());
9         System.out.println("Su dominio es: " +
10                        ecuador.getDominio());
11        System.out.println("Nombre del elemento: " +
12                           ecuador.name());
13    }
14 }
```

Al definir la instancia “ecuador” e inicializar con el elemento ECUADOR, se puede acceder a todos los atributos y métodos, los valores en los atributos son “Ecuador” y “ec”.

No es necesario almacenar la instancia en una variable, en la línea 4 se imprime el **enum** “Pais.ESPANA” de forma directa porque es una constante de tipo **enum**.

Este código produce la siguiente salida:

```

1 \programas_libro> java Ejecutor
2 España - es
3 Venezuela - ve
4 Nombre del país es: Ecuador
5 Su dominio es: ec
6 Nombre del elemento: ECUADOR
```

## 8.4 EJEMPLOS

**Ejemplo 1:** Diseñar un programa para crear jugadores y asignarle a un equipo de futbol.

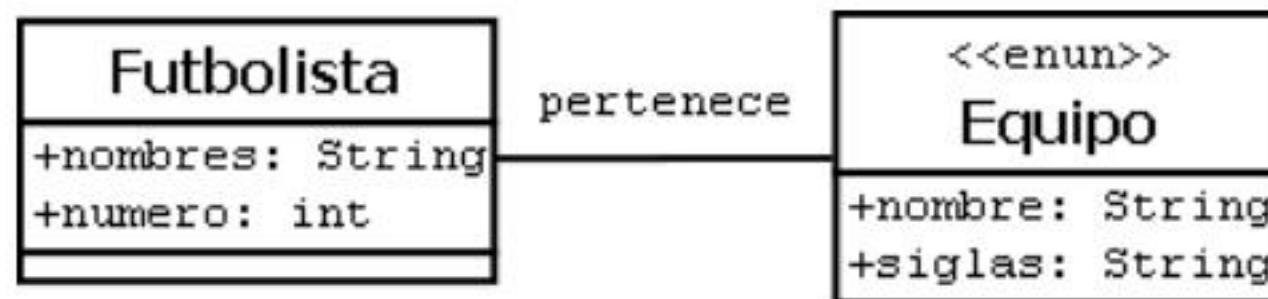


Figura 44 Relación entre una clase y un enum

En este diagrama se puede observar que el tipo de datos enumerado Equipo contiene dos atributos de tipo cadena, es decir cada elemento tendrá definido dos parámetros: “BAYERN(“*BAYERN DE MUNICH*”, “*BM*” )”. Al definir atributos es imprescindible codificar un constructor que inicialice a los atributos, como se ilustra en el siguiente ejemplo:

Ejemplo 88 Enum: Equipo de Futbol

```

1 public enum Equipo {
2
3     REAL_MADRID("REAL MADRID", "RMFC"),
4     BAYERN("BAYERN DE MUNICH", "BM"),
5     MANCHESTER_UNIDED("MANCHESTER UNITED", "MU");
6
7     public String nombre;
8     public String siglas;
9
10    Equipo(String nombre, String siglas){
11        this.nombre = nombre;
12        this.siglas = siglas;
13    }
14}

```

Es importante recordar que los constructores definidos en el **enum** deben ser definidos con modificadores privados o por defecto, porque el constructor permite crear un objeto de una clase, y **Futbolista** no es una clase. En un **enum** la instancia se la establece al acceder a un elemento: Equipo.BAYERN.

```

1 public class Futbolista {
2     public String nombres;
3     public int numero;
4     public Equipo equipo;
5
6     public Futbolista() {
7
8
9     public Futbolista(String nombres, int numero, Equipo equipo)
10    {
11        this.nombres = nombres;
12        this.numero = numero;
13        this.equipo = equipo;
14    }
15}

```

En la clase Ejecutor, se crea dos objetos de la clase Futbolista y se inicializa el atributo equipo con una instancia de Equipo. Al ser Equipo un **enum**, la instancia se genera al invocar un elemento como: Equipo.MANCHESTER\_UNITED.

```
1 public class Ejecutor {  
2  
3     public static void main(String[] args) {  
4         Futbolista cannavaro = new Futbolista();  
5         cannavaro.nombres = "Fabio Cannavaro";  
6         cannavaro.numero = 5;  
7         cannavaro.equipo = Equipo.MANCHESTER_UNITED;  
8         System.out.println(cannavaro.nombres + " " +  
9             cannavaro.numero + " " + cannavaro.equipo);  
10    }  
11    Futbolista puyol = new Futbolista("Carles Puyol", 8,  
12        Equipo.REAL_MADRID);  
13    System.out.println(puyol.nombres + " " + puyol.numero + "  
14        " + puyol.equipo);  
15    }  
16 }
```

Su salida es:

```
1 \programas_libro> java Ejecutor  
2 Fabio Cannavaro 5 MANCHESTER_UNITED  
3 Carles Puyol 8 REAL_MADRID
```

**Ejemplo 2:** Crear un programa que imprima el día de la semana, seleccionando un número que corresponda a cada una, por ejemplo:

1 = Lunes; 2 = Martes; 3 = Miércoles; ..... 7 = Domingo

**Solución:**

```
1 public enum DiasSemana {  
2     LUNES(1, "Lunes"),  
3     MARTES(2, "Martes"),  
4     MIERCOLES(3, "Miércoles"),  
5     JUEVES(4, "Jueves"),  
6     VIERNES(5, "Viernes"),  
7     SABADO(6, "Sábado"),  
8     DOMINGO(7, "Domingo");  
9  
10    public int numero;  
11    public String denominacion;  
12  
13    private DiasSemana(int numero, String denominacion) {  
14        this.numero = numero;  
15        this.denominacion = denominacion;  
16    }  
17 }  
  
1 import java.util.Scanner;
```

```

2
3 public class Ejecutor {
4
5     public static void main(String[] args) {
6         Scanner lector = new Scanner(System.in);
7         System.out.println("Selecciones un número del 1 al 7");
8         int opcion = lector.nextInt();
9         DiasSemana dia = null;
10        switch(opcion){
11            case 1:
12                dia = DiasSemana.LUNES;
13                break;
14            case 2:
15                dia = DiasSemana.MARTES;
16                break;
17            case 3:
18                dia = DiasSemana.MIERCOLES;
19                break;
20            case 4:
21                dia = DiasSemana.JUEVES;
22                break;
23            case 5:
24                dia = DiasSemana.VIERNES;
25                break;
26            case 6:
27                dia = DiasSemana.SABADO;
28                break;
29            case 7:
30                dia = DiasSemana.DOMINGO;
31                break;
32            default:
33                System.out.println("la opción esta fuera del
34                    rango de 1 a 7");
35        }
36        if(dia != null)
37            System.out.println("Día selecconado es:" +
38                dia.denominacion + "\n");
39    }
40 }
```

Su salida es:

```

1 \programas_libro> java Ejecutor
2 Selecciones un número del 1 al 7
3 5
4 Día selecconado es:Viernes
```

## 8.5 CUESTIONARIO

- 1. ¿Cuál es la palabra reservada que define un dato de tipo enumerado?**
  - case
  - switch
  - enum
  - class
  - interface
  - abstract
  - implements
  - extends

**2. Los datos de tipo enumerados son:**

- constantes que se definen en cada clase para que todos los objetos tengan el mismo comportamiento.
- tipos de dato por referencia que contiene un conjunto finito de constantes predeterminadas.
- tipos de datos que permite asignarles un constante determinado por el usuario.
- datos de tipo especial contiene un conjunto finito de constante predeterminada.

**3. Los elementos de cada tipo de dato enumerado se separan por:**

- |                    |                     |
|--------------------|---------------------|
| - Coma ","         | - punto ":"         |
| - Guion "-"        | - Paréntesis "(" )" |
| - punto y coma ";" | - Llaves "{}"       |

**4. Los elementos del enum de forma implícita son datos declarados como.**

- |                       |             |
|-----------------------|-------------|
| - private             | - protected |
| - public final static | - public    |

**5. Para acceder a un dato de tipo enumerado se sigue la siguiente sintaxis:**

- NombreClase.nombreEnum.elemento()
- NombreObjeto.nombreEnum.elemento
- NombreClase.nombreEnum.elemento
- NombreObjeto.nombreEnum.elemento()

**6. Analizar el siguiente código y seleccionar la opción correcta:**

```
1 public enum Meses{  
2     ENERO;  
3     FEBRERO;  
4     MARZO;  
5     ABRIL;  
6     MAYO;  
7     JUNIO;  
8     JULIO;  
9     AGOSTO;  
10    SEPTIEMBRE;  
11    OCTUBRE  
12    NOVIEMBRE;  
13    DICIEMBRE;  
14 }
```

- Error de compilación
- Error de ejecución
- Se define un enum con 12 constantes
- Cada enun consta de un solo atributo

**7. Los constructores en un enum deben ser:**

- Privados
- Públicos
- Protegidos
- Abstractos

**8. Analice el siguiente ejercicio y seleccione la respuesta correcta:**

```
1 public enum Sexo {  
2     FEMENINO('f'), MASCULINO('m');  
3  
4     public char letra;  
5  
6     publico Sexo(char letra) {  
7         this.letra = letra;  
8     }  
9 }
```

- Es un enum los constructor deben ser público.
- Es un enum los constructor deben ser protegido.
- Genera un error porque los constructores deben ser privados.
- Se ha definido correctamente al enum.

## **9 HERENCIA**

---

En este capítulo se introduce el concepto de herencia y los elementos que la componen. Además se representa en UML la relación que se establecen entre clases hijas a padres. Se explica cómo se utiliza la sentencia “super” en las clases hijas y Padre. Por último se explica en qué consiste la sobre escritura de métodos y como es su implementación.

### **Objetivos**

- Definir el concepto de Herencia y de relaciones jerárquicas entre clases padres e hijas.
- Representa la relación de herencia en UML y código java.
- Establecer la redefinición de métodos en clase jerárquicas.
- Aplicar la instrucción super para invocar los constructores de la clase Padre.

### **9.1 DEFINICIÓN**

Es uno de los pilares fundamentales del paradigma orientado a objetos permitiendo la utilización de atributos y métodos entre clase padres a hijas. Además es un mecanismo que ayuda a reutilizar código entre clases que tienen un grado de afinidad. La herencia es una propiedad de la POO que permite declarar clases padres y a partir de ellas crear otras llamadas clases hija o subclases. Esta relación entre padres e hijos se denomina relación jerárquica.

En UML relación de herencia se representa a través de una flecha como se ilustra en la siguiente figura:



Figura 45 Representación de herencia en UML

La flecha en UML representa la ubicación de las clases padre e hija, donde nace la flecha se ubica la clase hija, descendiente, derivada o subclase, mientras en la cabecera de la flecha se ubica la clase Padre, base o superclase:

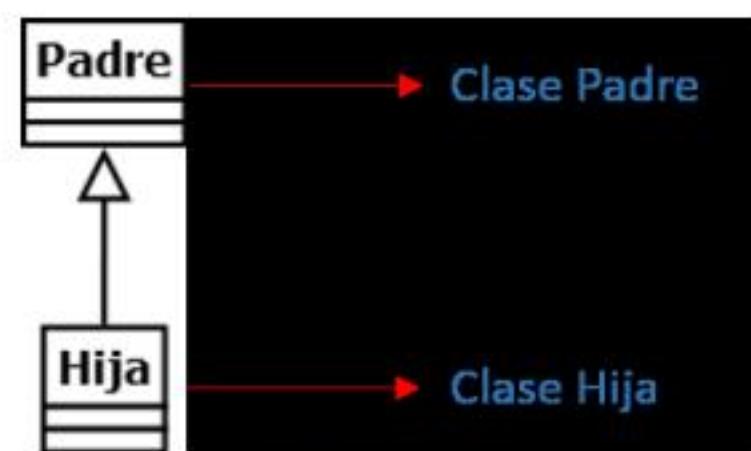


Figura 46 Relación de herencia

Cuando una clase (hija) se construye a partir de otra (padre) mediante la relación de herencia, la clase hija hereda todos los atributos, métodos y clases internas de la clase padre. Una característica de la herencia, la clase hija puede redefinir los componentes heredados, en especial los métodos, esta operación se define como sobre escritura.

En Java, para establecer la relación de herencia entre dos clases se utiliza la palabra reservada **extends**. La clase hija emplea esta palabra reservada en la cabecera de la clase, su sintaxis es:

```
class NombreClase extends ClasePadre {
    // cuerpo de la clase
}
```

Java soporta la herencia simple, otros lenguajes soportan la múltiple herencia como python, c++. En la siguiente figura se ilustra un ejemplo de herencia en Java:

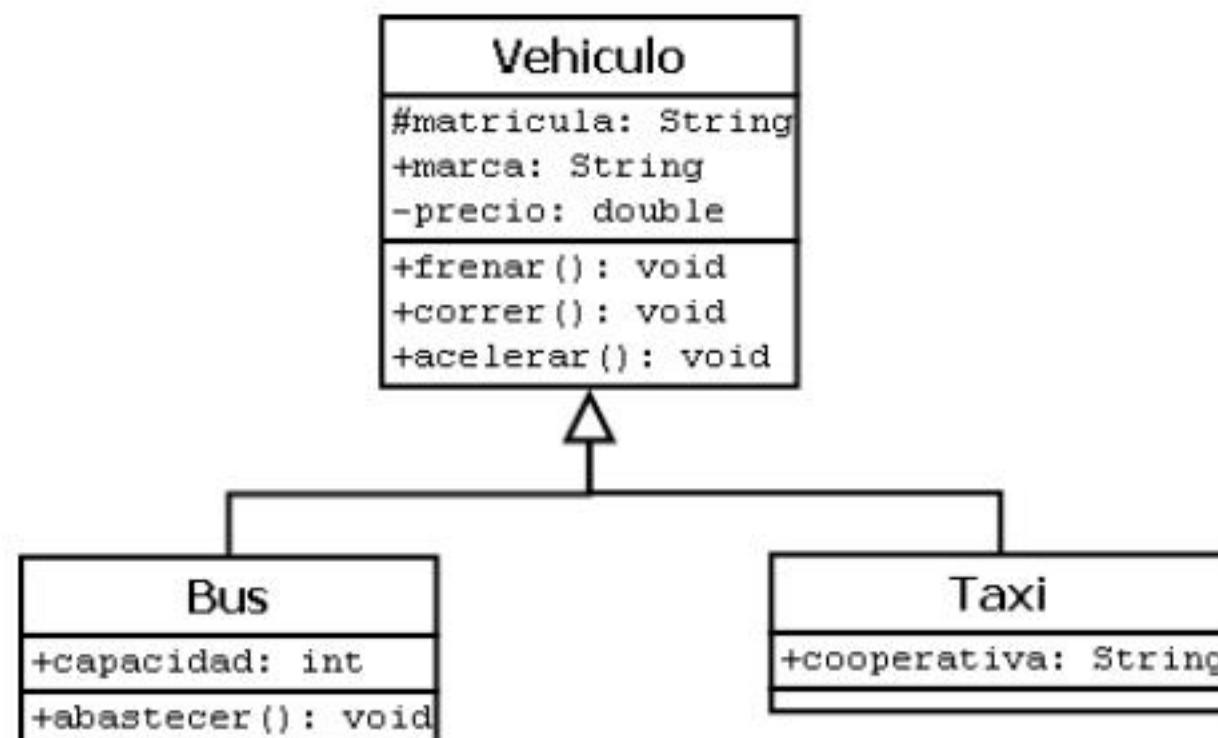


Figura 47 Herencia

La clase Vehículo se convierte en la super clase, su codificación es:

Ejemplo 89 Herencia, clase Padre

```

1 public class Vehiculo {
2     protected String matricula;
3     public String marca;
4     private double precio;
5
6     public void frenar(){
7         System.out.println("Se aplica el mecanismo de freno");
8     }
9
10    public void correr(){
11        System.out.println("Vehículo en movimiento");
12    }
13
14    public void acelerar(){
15        System.out.println("Vehículo incrementa su velocidad");
16    }
17 }
```

La clase Bus es una subclase de Vehículo heredando sus atributos y métodos. El método abastecer no es heredado porque se lo declarado en la clase hija, de forma implícita tiene tres métodos y tres atributos heredados.

Los elementos que no se pueden heredar son los que la clase Padre ha definido con un modificador de acceso es privado. En el caso de Bus no puede heredar el atributo precio porque en la superclase se ha definido como privado.

Su codificación es:

Ejemplo 90 Herencia - clase Hija Bus

```

1 public class Bus extends Vehiculo{
2     public int capacidad;
3
4     //método no heredado
```

```

5 |     public void abastecer() {
6 |         System.out.println("El bus se abastece por las noches");
7 |     }
8 |

```

La clase Taxi es una subclase de Vehículo heredando sus atributos y métodos. No contiene métodos propios pero a través de la herencia posee dos métodos heredados, su codificación es:

*Ejemplo 91 Herencia - clase Hija Taxi*

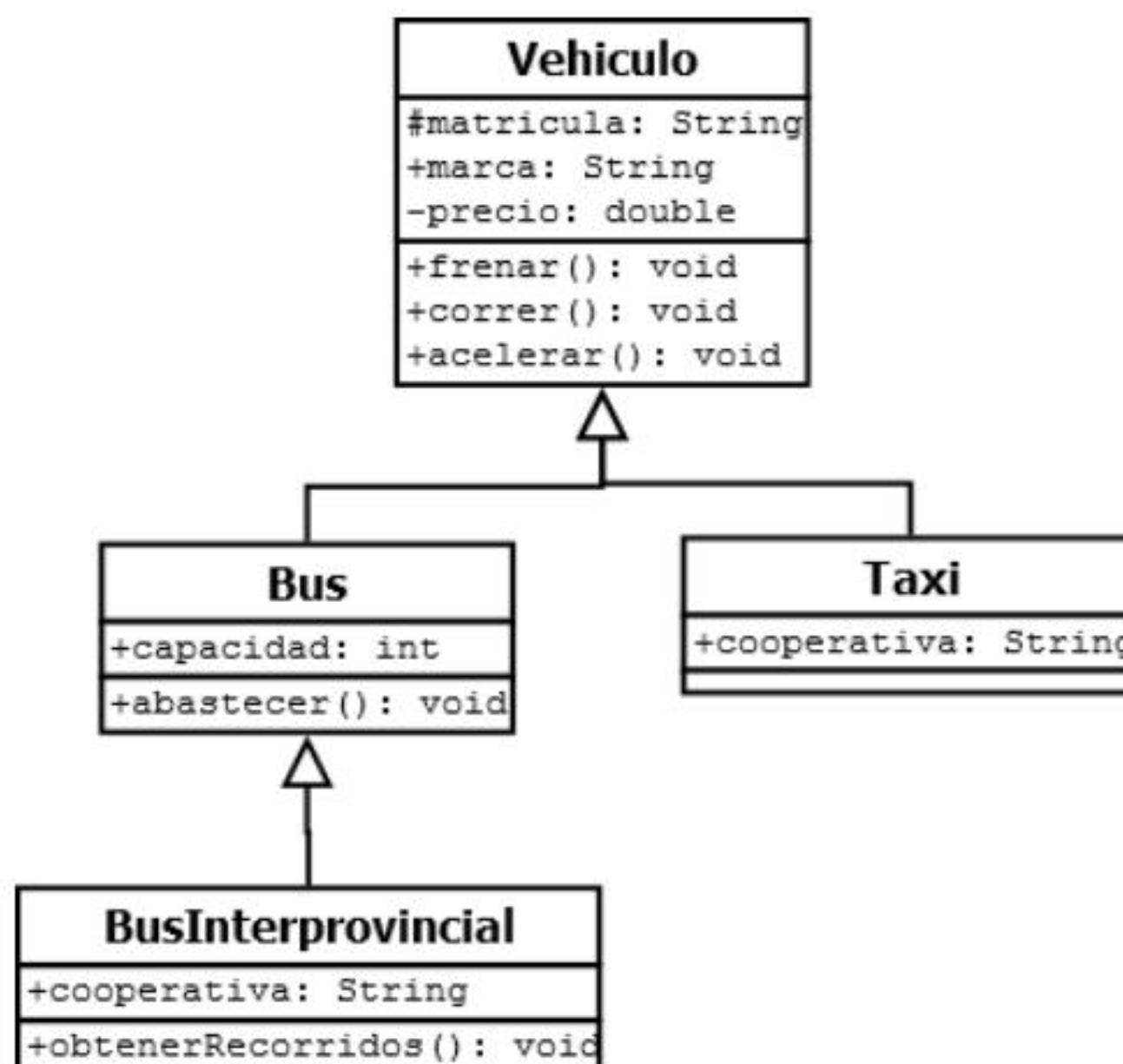
```

1 | public class Taxi extends Vehiculo{
2 |     public String cooperativa;
3 |

```

## 9.2 JERARQUÍA DE CLASES

Como se ha mencionado antes, Java solo permite establecer la herencia simple, pero se puede definir múltiples niveles a través de jerarquías. De forma directa, una clase solo tendrá un solo parent pero un parent podrá tener múltiples hijos. Se ilustra un ejemplo de jerarquías entre clases:



*Figura 48 Jerarquía de clases*

En esta figura se establece una jerarquía de clase, BusInterprovincial de forma directa define un método propio (no heredado) y al establecer una relación de

herencia con la clase Bus, hereda de forma implícita el método "abastecer ()" y el atributo "capacidad".

Hasta este punto BusInterprovincial tiene a su disposición dos métodos y dos atributos.

La clase Bus establece una relación de herencia con Vehículo y de forma explícita hereda dos métodos y tres atributos.

Las jerarquía hace que la clase BusInterprovincial tenga una múltiple herencia implícita, porque Bus al heredar de Vehículo y BusInterprovincial de Bus, hace que BusInterprovincial herede también los atributos y métodos de Vehículo. BusInterprovincial ahora posee un método y un atributo propio, además tres métodos y cuatro atributos heredados.

```
1 public class BusInterprovincial extends Bus{  
2     private String cooperativa;  
3  
4     public void obtenerRecorrido(){  
5         System.out.println("Recorridos establecido");  
6     }  
7 }
```

El código ejecutor:

```
1 public class Ejecutor {  
2  
3     public static void main(String[] args) {  
4         Vehiculo vehiculo = new Vehiculo();  
5  
6         Bus bus = new Bus();  
7         bus.correr(); // método heredado de vehículo  
8  
9         BusInterprovincial bi = new BusInterprovincial();  
10        bi.acelerar(); // método heredado de vehículo  
11        bi.abastecer(); // método heredado de bus  
12        bi.obtenerRecorrido(); // método propio  
13    }  
14 }
```

Su salida es:

```
1 \programas_libro>java Ejecutor  
2 Vehículo en movimiento  
3 Vehículo incrementa su velocidad  
4 Vehículo incrementa su velocidad  
5 El bus se abastece por las noches  
6 Recorridos establecido
```

### Recomendación

No se puede heredar los atributos y métodos definidos con el modificador de acceso privado en las clases Padres.

## 9.3 REDEFINICIÓN O SOBRE ESCRITURA DE ELEMENTOS

La herencia permite utilizar los métodos y atributos de las superclases pero también se pueden modificar los métodos heredados. Esta operación se la denomina sobre escritura y es un mecanismo para redefinir a un atributo o métodos.

Al heredar los métodos, estamos sujetos a la implementación que realizó la clase padre, pero a veces el programador debe volverlos a codificar. Cuando se vuelve a escribir un método heredado se dice que se está sobre escribiendo el método.

La sobre escritura de métodos en UML se representa en la siguiente figura:

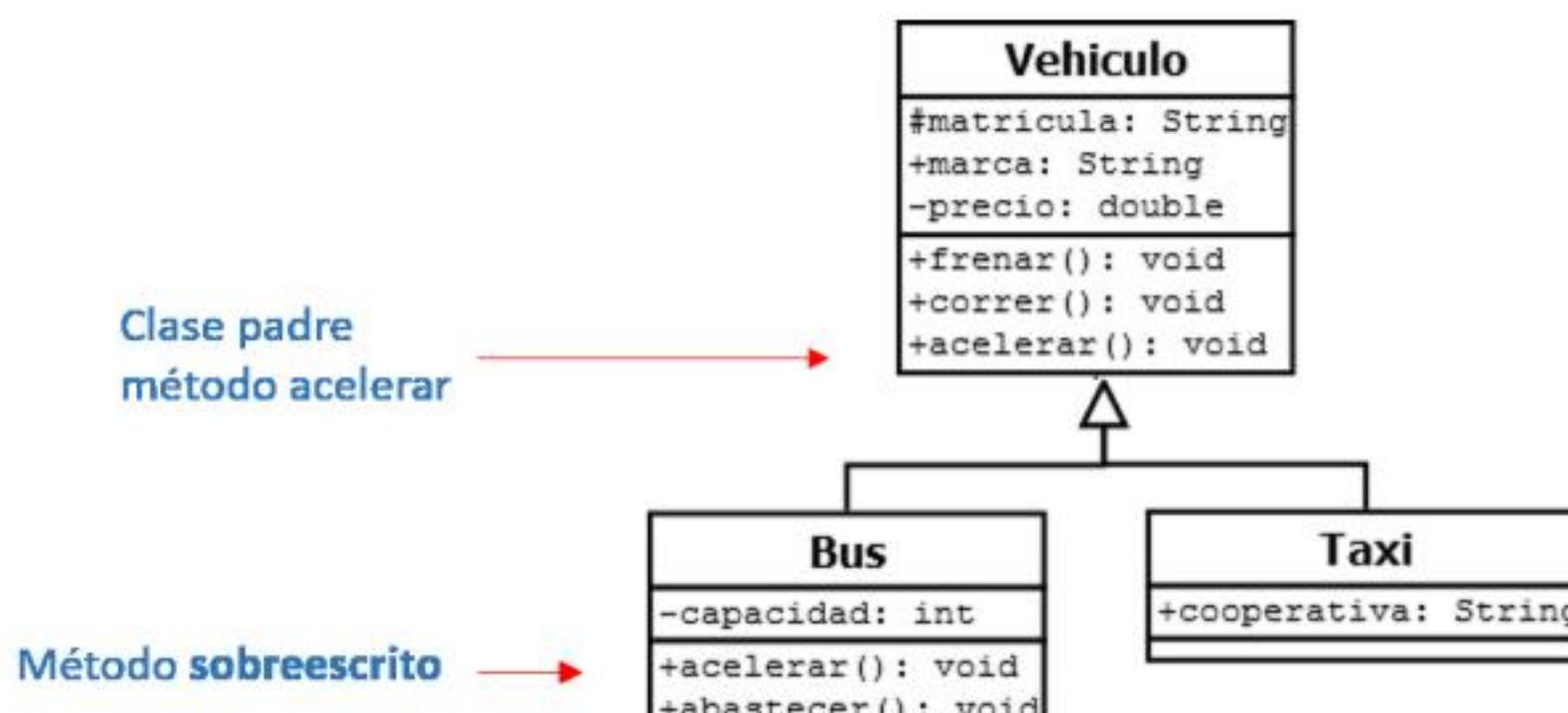


Figura 49 Sobre escritura de métodos

```
1 public class Bus extends Vehiculo{
2     public int capacidad;
3
4     @Override
5     public void acelerar() {
6         System.out.println("Los buses están prohibidos fuera de
7             los límites permitidos");
8     }
9
10    public void abastecer() {
11        System.out.println("El bus se abastece por las noches");
12    }
13}
```

Por defecto se heredan todos los métodos y propiedades **protected** y **public**. Los elementos privados no se pueden heredar porque solo pueden ser accedidos desde la propia clase. La sobrecarga en java es reemplazar el método heredado por un nuevo. En este caso se ha sobre escrito el método “acelerar”, para identificar una sobre escritura se utiliza una anotación “@Override”.

Al crear un objeto de la clase Bus e invocar al método “acelerar” se notará el método utiliza el código sobre escrito como se indica en el siguiente ejemplo:

```
1 public class Ejecutor {  
2     public static void main(String[] args) {  
3         Bus bus = new Bus();  
4         bus.correr(); // método heredado  
5         bus.acelerar(); // método sobreescrito  
6     }  
7     Taxi taxi = new Taxi();  
8     taxi.frenar(); //método heredado  
9 }  
10 }  
11 }
```

La salida del ejemplo es:

```
1 \programas_libro>java Ejecutor  
2 Vehículo en movimiento  
3 Los buses están prohibidos fuera de los límites permitidos  
4 Se aplica el mecanismo de freno
```

## 9.4 SENTENCIA súper

Cuando se establece una relación de herencia, a veces las clases hijas necesitan invocar a los constructores o métodos de la clase Padre, esto se lo realiza a través de la palabra reservada **súper**. Además, al aplicar la sentencia **súper** se puede acceder a métodos anulados por herencia. Para llamar al constructor padre se debe codificarlo siempre en la primera línea del constructor hijo, caso contrario genera un error de compilación.

La instrucción **súper** hace referencia a los elementos de la clase padre por medio de la herencia, la sentencia **this** hace referencia a la clase actual y no podrá acceder a los métodos de la clase padre.

Ahora a la clase “Vehículo” se le agrega un constructor con parámetros y desde la clase “Bus” se invoca al constructor de la clase Padre:

Ejemplo 92 Clase Vehículo con constructor con parámetros

```
1 public class Vehiculo {  
2     private String matricula;  
3     private String marca;  
4     private double precio;  
5  
6     public Vehiculo(String matricula, String marca, double  
7                     precio) {  
8         this.matricula = matricula;  
9         this.marca = marca;  
10        this.precio = precio;  
11    }  
12  
13    public void frenar(){  
14        System.out.println("Se aplica el mecanismo de freno");  
15    }  
16  
17    public void correr(){  
18        System.out.println("Vehículo en movimiento");  
19    }  
20  
21    public void acelerar(){  
22        System.out.println("Vehículo incrementa su velocidad");  
23    }  
24 }
```

Ejemplo 93 sentencias súper

```
1 public class Bus extends Vehiculo{  
2     private int capacidad;  
3  
4     public Bus(int capacidad, String matricula, String marca,  
5                 double precio) {  
6         super(matricula, marca, precio);  
7         this.capacidad = capacidad;  
8     }  
9  
10    @Override  
11    public void acelerar() {  
12        super.acelerar();  
13    }  
14  
15    public void abastecer(){  
16        System.out.println("El bus se abastece por las noches");  
17    }  
18 }
```

En este ejemplo, en la línea 6 se define la sentencia `super()` y permite invocar al constructor de la superclase, un requisito imprescindible para utilizar `súper` como invocador de constructores es que se defina en la primera línea del método

invocador caso contrario generará un error de compilación. En la línea 12, la llamada `super.acelerar()` invoca al método acelerar de la clase `Vehículo`.

## 9.5 CLASE OBJECT

Todas las clases que se definen en las aplicaciones y las que se encuentran en el API de Java provienen de una súper clase llamada *Object*, es la raíz de todo el árbol jerárquico en Java. En la siguiente tabla se ilustra algunos métodos que define la clase *Object*.

Tabla 18 Métodos de la clase Object

| Método   | Significador                                                     |
|----------|------------------------------------------------------------------|
| Clone    | Clona una instancia u objeto a partir de otra de la misma clase. |
| equals   | Compara contenidos de un objeto devolviendo un valor booleano.   |
| toString | Permite visualizar un objeto.                                    |
| finalize | Destructor del objeto.                                           |
| hashCode | Devuelve una clave hash del objeto.                              |
| getClass | Devuelve la clase a la que pertenece el objeto.                  |

## 9.6 EJERCICIO

**Ejercicio 1:** Traducir en código Java el siguiente diagrama UML propuesto, se debe establecer la relación de herencia, sobre escritura del método “área” e invocar al constructor de la clase padre mediante `súper`.

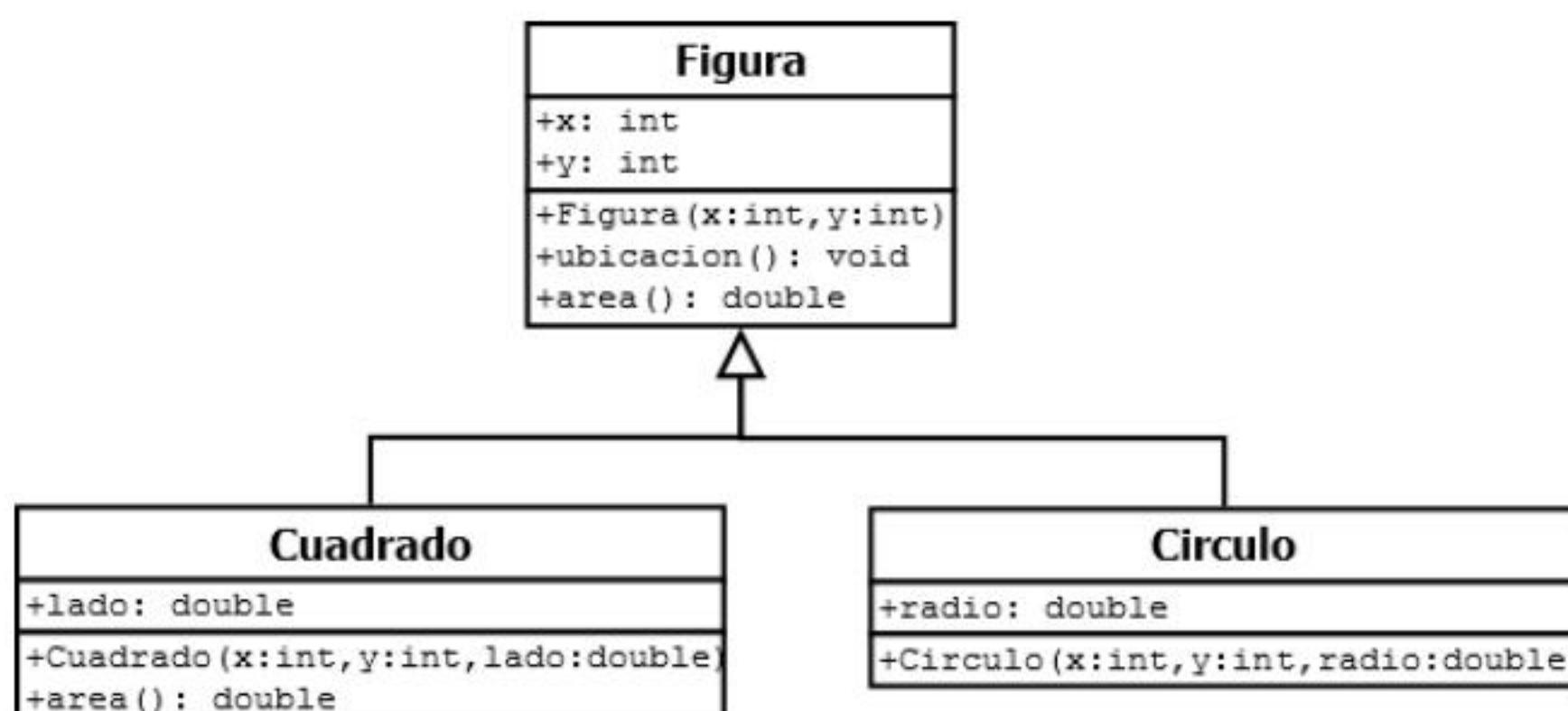


Figura 50 Ejemplo de herencia

Para codificar el diagrama se recomienda que se inicie con las clases que tienen mayor jerarquía, en este caso se codificará la clase `Figura`:

*Ejemplo 94 Herencia - clase padre Figura*

```
1 public class Figura {  
2     public int x;  
3     public int y;  
4  
5     public Figura(int x, int y) {  
6         this.x = x;  
7         this.y = y;  
8     }  
9  
10    public void ubicacion(){  
11        System.out.println("La ubicación en es x: " + x + " y: "  
12                           + y);  
13    }  
14  
15    public double area(){  
16        return 0;  
17    }  
18 }
```

La subclase Cuadrado establece herencia con Figura:

*Ejemplo 95 Herencia - clase hija Cuadrado*

```
1 public class Cuadrado extends Figura{  
2     public double lado;  
3  
4     public Cuadrado(double lado, int x, int y) {  
5         super(x, y);  
6         this.lado = lado;  
7     }  
8  
9     @Override  
10    public double area(){  
11        return super.area();  
12    }  
13 }
```

En la línea 5 la clase Cuadrado hace un llamado al constructor de la clase padre Figura a través de `super()`. La sobre escritura es reescribir el método que ha definido la clase Padre, a este procedimiento se lo denomina anulación de métodos y solo se realiza en herencias. Java para identificar que un método ha sido sobre escrito utiliza la anotación `@Override`.

*Ejemplo 96 Herencia - clase hija Círculo*

```
1 public class Circulo extends Figura{  
2     public double radio;  
3  
4     public Circulo(double radio, int x, int y) {  
5         super(x, y);  
6         this.radio = radio;  
7     }  
8 }
```

En la línea 5 la clase Circulo hace un llamado al constructor de la clase padre Figura a través de `super()`. En esta clase no se realiza ninguna sobre escritura de métodos.

## 9.7 CUESTIONARIO

**1. Para establecer una relación de herencia se utiliza la palabra reservada:**

- `super`
- `implements`
- `interface`
- `abstract`
- `extends`
- `enum`

**2. Java no soporta la herencia:**

- Simple
- Jerárquica
- Múltiple
- Compuesta

**3. Evaluar el siguiente código y seleccionar la respuesta correcta**

```
1 | public class Tigre extends Animal, Mamifero {  
2 |  
3 | }
```

- Error de ejecución.
- Error de compilación.
- La clase Tigre establece relación de herencia con Animal y Mamífero.
- Se establece una relación jerárquica entre las clases Tigre, Animal y Mamífero.

**4. La sobre escritura de métodos permite:**

- La sobre carga de métodos.
- Utilizar el método de la clase padre a través de la sentencia super.
- Recodificar un método heredado.
- Implementar un método abstracto.

**5. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 | public class Tigre{  
2 |  
3 |     public Tigre(){  
4 |         super();  
5 |     }  
6 |  
7 |     @Override  
8 |     public void comer(){  
9 |         super.comer();  
10|     }  
11| }
```

- El constructor de la clase Tigre invoca al constructor de la superclase a través de la sentencia super.
- Se establece una sobre escritura en el método comer ( ).
- La clase Tigre no establece ninguna relación de herencia.
- El método comer de la clase Tigre llama al método comer de la clase Padre.

#### **6. En una relación de herencia, las clases hijas no podrán acceder:**

- Elementos privados.
- Elementos públicos.
- Elementos protegidos.
- Elementos con modificador de acceso por defecto.

#### **7. Analizar el siguiente método y seleccionar la respuesta correcta**

```

1 public class Casa extends Infraestructura{
2
3     public Casa(){
4         System.out.println("Constructor");
5         super();
6     }
7 }
```

- La sentencia “super” solo se puede utilizar en la primera línea del constructor Casa.
- Se llama al constructor sin parámetros de la clase Infraestructura.
- Imprime el mensaje “Constructor” y luego se llama al constructor de la clase Infraestructura.
- Se genera un error de ejecución.

#### **8. Analizar el siguiente código y seleccionar la respuesta correcta:**

```

1 public class Animal{
2
3 }
4
5 public class Pez extends Animal{
6
7 }
8
9 public class Albacora extends Pez, Animal{
10}
11}
```

- Error de compilación en la línea 5.
- Error de compilación en la línea 9.
- Se establece la herencia múltiple entre la clase Albacora y Pez y Animal.
- La clase Animal se convierte en la súper clase de Pez y Albacora.

## **10 INTERFACES Y CLASES ABSTRACTAS**

---

El presente capítulo se introduce a los conceptos de interfaces y clases abstractas. Se explica el significado de los métodos abstractos, su funcionalidad y cuál es el papel que juega en la programación orientada a objetos. Además, se codifica las clases implementadoras y se establecen las implementaciones de los métodos abstractos. Por último se explica a través de ejemplos el funcionamiento y comportamiento de cada una.

### **Objetivos**

- Definir el concepto de interfaz, clases abstractas y métodos abstractos.
- Representar a las interfaces y clases abstractas en diagramas UML.
- Traducir un diagrama UML en código Java para representar interfaces y clases abstractas.
- Implementar métodos abstractos en clases implementadoras.

### **10.1 INTERFACES**

Son plantillas que permiten definir métodos abstractos, en otras palabras en una colección de métodos sin implementar. Las interfaces establecen el comportamiento que tendrán las clases que las implementen y resuelven el problema de herencia simple porque una clase puede implementar más de una interfaz.

Las interfaces establecen el término “able” que significa modifiable, configurable porque las clases que implementan a las interfaces pueden codificar el cuerpo de los métodos abstractos de acuerdo a sus requerimientos. Además las interfaces pueden declarar constantes pero siempre que no sean variables de instancia.

La estructura de una interfaz es similar al de un `enum` o clase, pero para diferenciarlas se utiliza la palabra reservada `interface`, su sintaxis es:

```
public interface NombreInterfaz{  
    // métodos abstractos  
}
```

Todas las interfaces son abstractas por tal razón deben definir o declarar métodos abstractos.

### ¿Qué es un método abstracto?

Es un método declarado pero no implementado, en otras palabras, es un método del que solo se define su cabecera y se omite su cuerpo, su sintaxis es:

```
modificador tipo_dato nombreMétodo(parámetros);
```

Al agregar el separador de instrucciones ";" al final de la cabecera del método, se establece que carece de implementación y de forma implícita es un método abstracto. En las interfaces NO se puede declarar métodos con cuerpo o implementados.

En UML las interfaces se definen a través del estereotipo <<interfaz>>. En la siguiente figura se ilustra una interfaz:

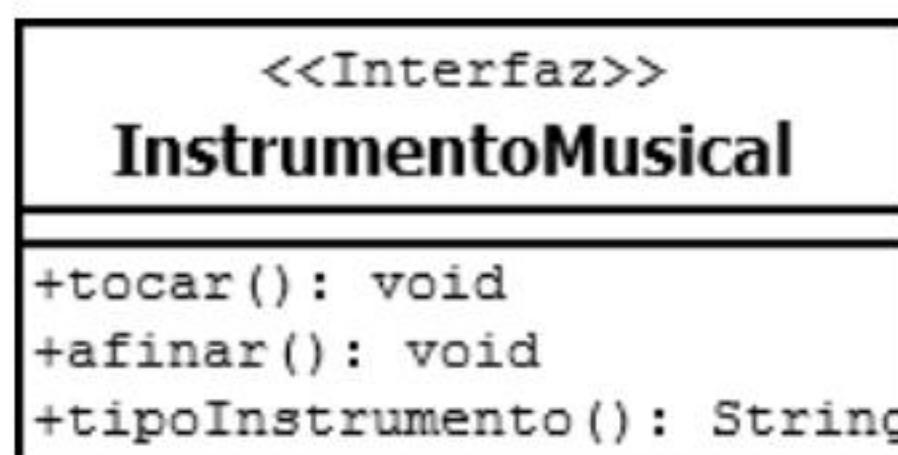


Figura 51 Interfaz en UML

Al traducir el diagrama UML al lenguaje Java se genera el siguiente código:

```
1 public interface InstrumentoMusical {  
2     public void tocar();  
3     public void afinar();  
4     public String tipoInstrumento();  
5 }
```

El siguiente paso es implementar la interfaz mediante las clases implementadora. En UML la flecha entre cortada especifica que una clase implementará los métodos abstractos de una interfaz. Donde nace la flecha se ubica la clase implementadora y

en la cabecera la interfaz, en la siguiente figura se representa la relación de implementación:



Figura 52 Representación de la implementación de interfaces

Ahora representaremos en UML la relación de implementación entre la interfaz "InstrumentoMusical" y la clase "InstrumentoViento".

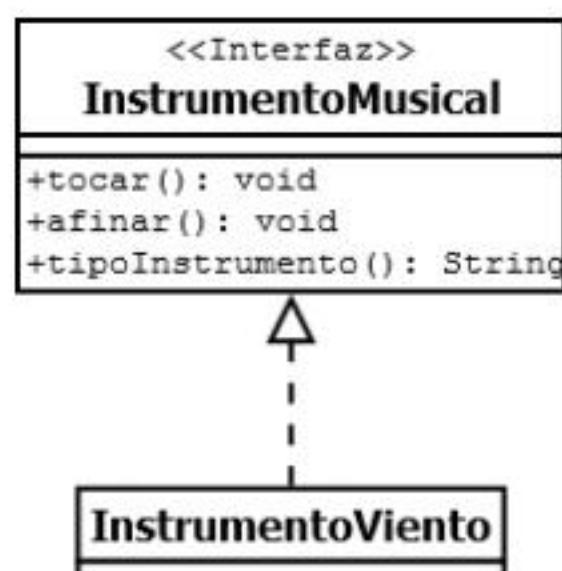


Figura 53 Implementación de una interfaz

En Java, la palabra reservada **implements** establece que una clase implementará los métodos abstractos de una interfaz, su sintaxis es la siguiente:

```
public class NombreClase implements Interfaz{  
}
```

La implementación de los métodos abstractos de la interfaz InstrumentoMusical es:

Ejemplo 97 Implementación de la Interfaz Instrumento Musical

```
1 public class InstrumentoViento implements InstrumentoMusical {  
2  
3     @Override  
4     public void tocar() {  
5         System.out.println("entonar");  
6     }  
7  
8     @Override  
9     public void afinar() {  
10        System.out.println("Afinar");  
11    }  
12  
13    @Override
```

```

14 |     public String tipoInstrumento() {
15 |         return "";
16 |     }
17 |

```

En este ejemplo, la implementación se establece en la cabecera de la clase agregando la palabra reservada **implements** y el nombre de la interfaz. La clase InstrumentoViento está en la obligación de implementar los métodos abstractos de la interfaz, en otras palabras, sobre escribirlos y codificar su cuerpo. A cada método implementado se suele utilizar la anotación @Override sobre la cabecera del método, esto permite aclarar ha sido sobre escrito.

En las interfaces no es necesario utilizar la palabra reservada **abstract** en los métodos porque de manera implícita los métodos son declarados como abstractos.

### 10.1.1 ATRIBUTOS EN LAS INTERFACES

Otra característica de las interfaces es que puede definir constantes pero deben ser públicas o abstractas, pero no se pueden declarar variables de instancia. Analizar el siguiente ejercicio:

*Ejemplo 98 Atributos en interfaces*

```

1 | public interface InstrumentoMusical {
2 |     public int notasMusicales = 7; // bien definida
3 |     public int sonidosMusicales; // error
4 |     public Piano p = new Piano(); // error
5 |
6 |     public void tocar();
7 |     public void afinar();
8 |     public String tipoInstrumento();
9 |

```

En este ejercicio se han definido tres atributos, la primera es pública de tipo entera, se le asigna un valor de 7 y está correctamente definida porque en una interfaz solo se pueden declarar constantes; la segunda es pública entera pero no se le inicializa valores generando un error de compilación debido a que los atributos declarados en las interfaces son constantes y es imprescindible asignarles valores; la tercera es pública de tipo de dato Piano y se le asigna un objeto generando un error porque los atributos declarados en una interfaz no deben ser variables de instancias.

Depurando el ejemplo anterior de los errores su codificación es:

Ejemplo 99 Interfaz - InstrumentoMusical

```
1 public interface InstrumentoMusical {  
2     public int notasMusicales = 7;  
3  
4     public void tocar();  
5     public void afinar();  
6     public String tipoInstrumento();  
7 }
```

### 10.1.2 IMPLEMENTACIÓN DE VARIAS INTERFACES

Una clase implementadora puede implementar varias interfaces, en cambio en la herencia es diferente, una clase sólo podrá heredar de una sola clase padre.

Analizar el siguiente diagrama:

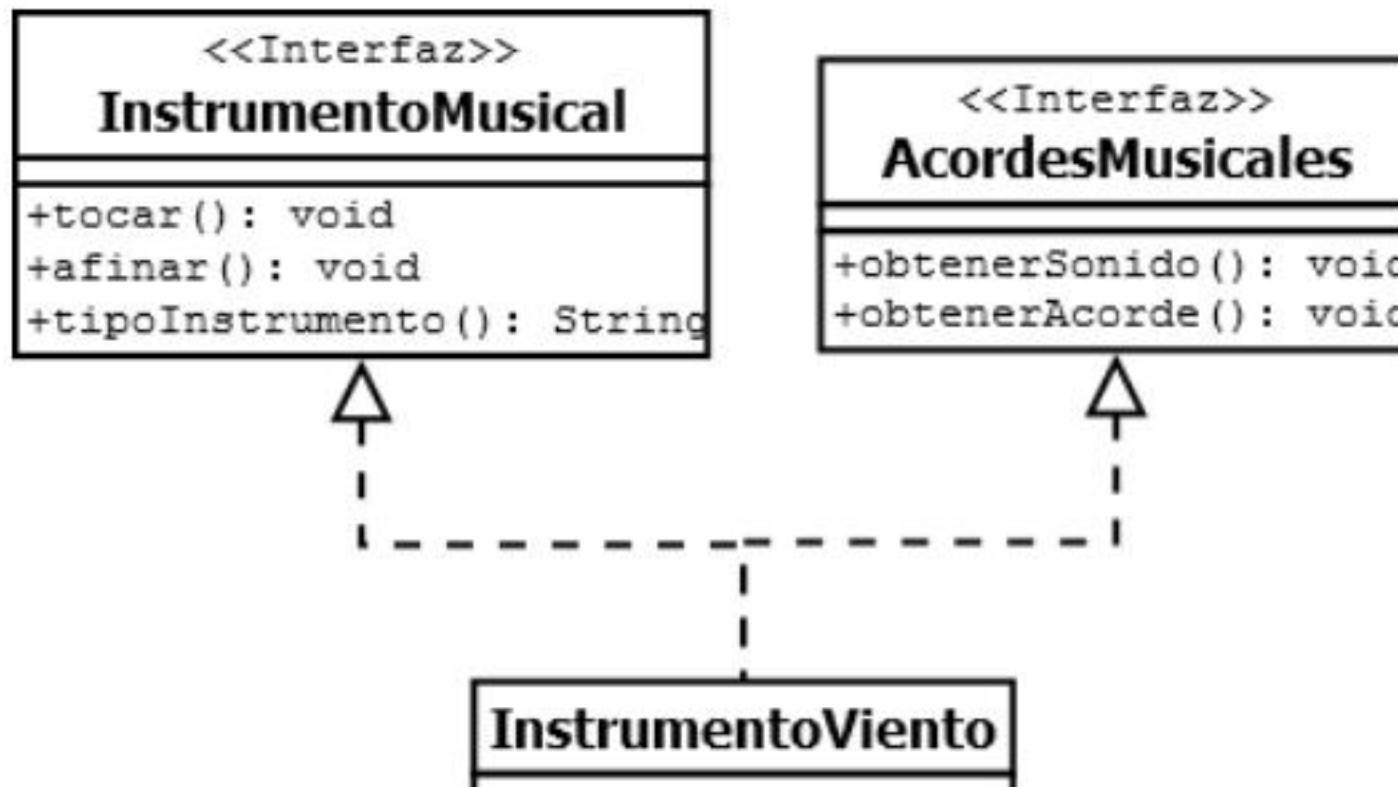


Figura 54 Implementación de varias interfaces

En este diagrama existen dos interfaces cada una ha definido métodos abstractos, la clase implementadora **InstrumentoViento** puede implementar una o más interfaces. Java permite la implementación de varias interfaces y las separa a través de “,”. Su sintaxis es:

```
public class NombreClase implements InterfazUna, InterfazDos, .....{  
}
```

La codificación de la interfaz AcordesMusicales es:

*Ejemplo 100 Interfaz AcordesMusicales*

```
1 public interface AcordesMusicales {  
2     public void obtenerSonido();  
3     public void obtenerAcorde();  
4 }
```

La codificación de la clase implementadora InstrumentoViento es:

*Ejemplo 101 Clase Implementadora implementa dos interfaces*

```
1 public class InstrumentoViento implements InstrumentoMusical, AcordesMusicales {  
2  
3     @Override  
4     public void tocar() {  
5         System.out.println("entonar");  
6     }  
7  
8     @Override  
9     public void afinar() {  
10        System.out.println("Afinar");  
11    }  
12  
13    @Override  
14    public String tipoInstrumento() {  
15        return "";  
16    }  
17  
18    @Override  
19    public void obtenerSonido() {  
20        System.out.println("sonido devuelto");  
21    }  
22  
23    @Override  
24    public void obtenerAcorde() {  
25        System.out.println("acorde devuelto");  
26    }  
27}  
28}
```

La clase InstrumentoViento al implementar las interfaces InstrumentoMusical y AcordesMusicales está en la obligación de implementar todos los métodos abstractos declarados en las interfaces y para identificarlos se utiliza la anotación @Override.

## 10.2 CLASES ABSTRACTAS

Es un tipo de clase de la que no se puede instanciar, es decir no se puede crear objetos a partir de ella. Para establecer que una clase sea abstracta al menos alguno de sus métodos debe ser abstracto.

Otras clases pueden derivar de las clases abstractas a través de la relación de herencia proporcionando un modelo de referencia para que las clases hijas implementen ese comportamiento por medio de la sobre escritura de los métodos abstractos.

Las clases abstractas es una colección de métodos normales y al menos un método abstracto. En java para definir que una clase es abstracta se utiliza la palabra reservada **abstract** en la cabecera de la clase.

```
public      abstract      class  
NombreClase{  
}
```

Los métodos abstractos declarados en las clases abstractas también deben utilizar la palabra reservada **abstract**

```
abstract tipo_dato nombreMétodo(parámetros);
```

Las clases abstractas al no ser instanciados no se crean constructores y sus métodos abstractos deben ser redefinidos por lo que no pueden ser estáticos.

En UML las clases abstractas se definen como el esteriotipo <>abstract<> y sus métodos se subrayan como ilustra la siguiente figura:

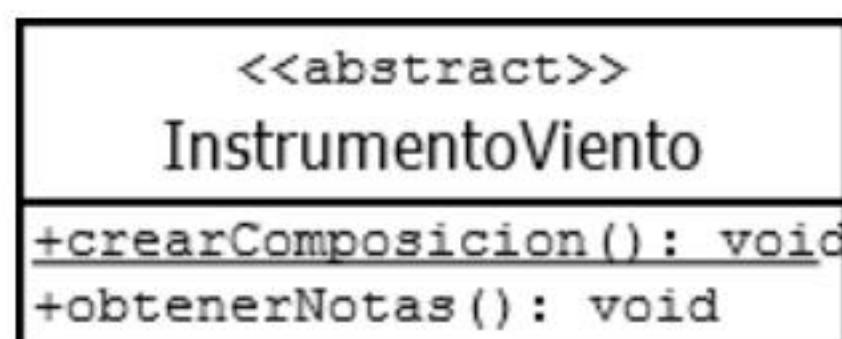


Figura 55 Clase Abstracta

Al traducir la clase en UML a código java se genera:

Ejemplo 102 Clase abstracta

```
1 public abstract class InstrumentoViento {  
2  
3     abstract void crearComposicion();  
4  
5     public void obtenerNotas(){  
6         System.out.println("DO-RE-MI-FA-SOL-LA-SI");  
7     }  
8 }
```

Para implementar una clase abstracta se debe establecer una relación de herencia como se ilustra en la siguiente figura:

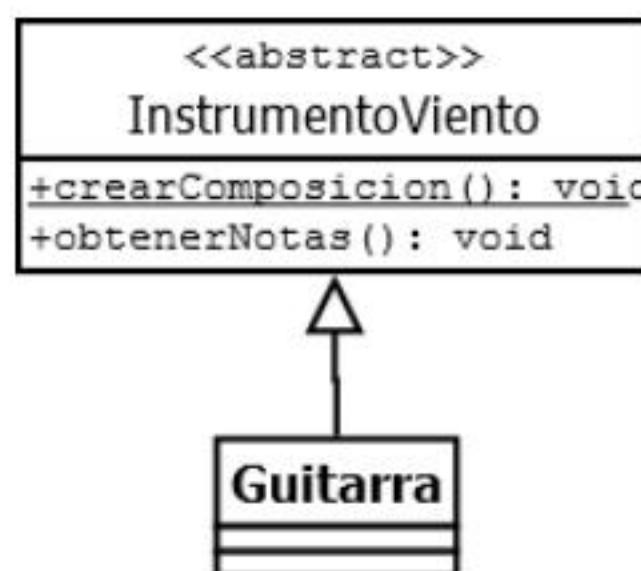


Figura 56 Implementación de una clase abstracta

En el diagrama UML se establece la clase abstracta `InstrumentoViento` que componen de un método normal y un abstracto. Al establecer una relación de herencias entre `InstrumentoViento` y `Guitarra`, la clase `Guitarra` hereda el método `obtenerNotas` y está en la obligación de implementar el método abstracto `crearComposición` como se indica en el siguiente ejemplo:

Ejemplo 103 Implementación de clases abstractas

```
1 public class Guitarra extends InstrumentoViento{  
2  
3     @Override  
4     void crearComposicion(){  
5         System.out.println("composición creada");  
6     }  
7 }
```

## 10.3 EJEMPLO

**Ejercicio 1:** Realice la codificación en lenguaje Java del siguiente diagrama UML.

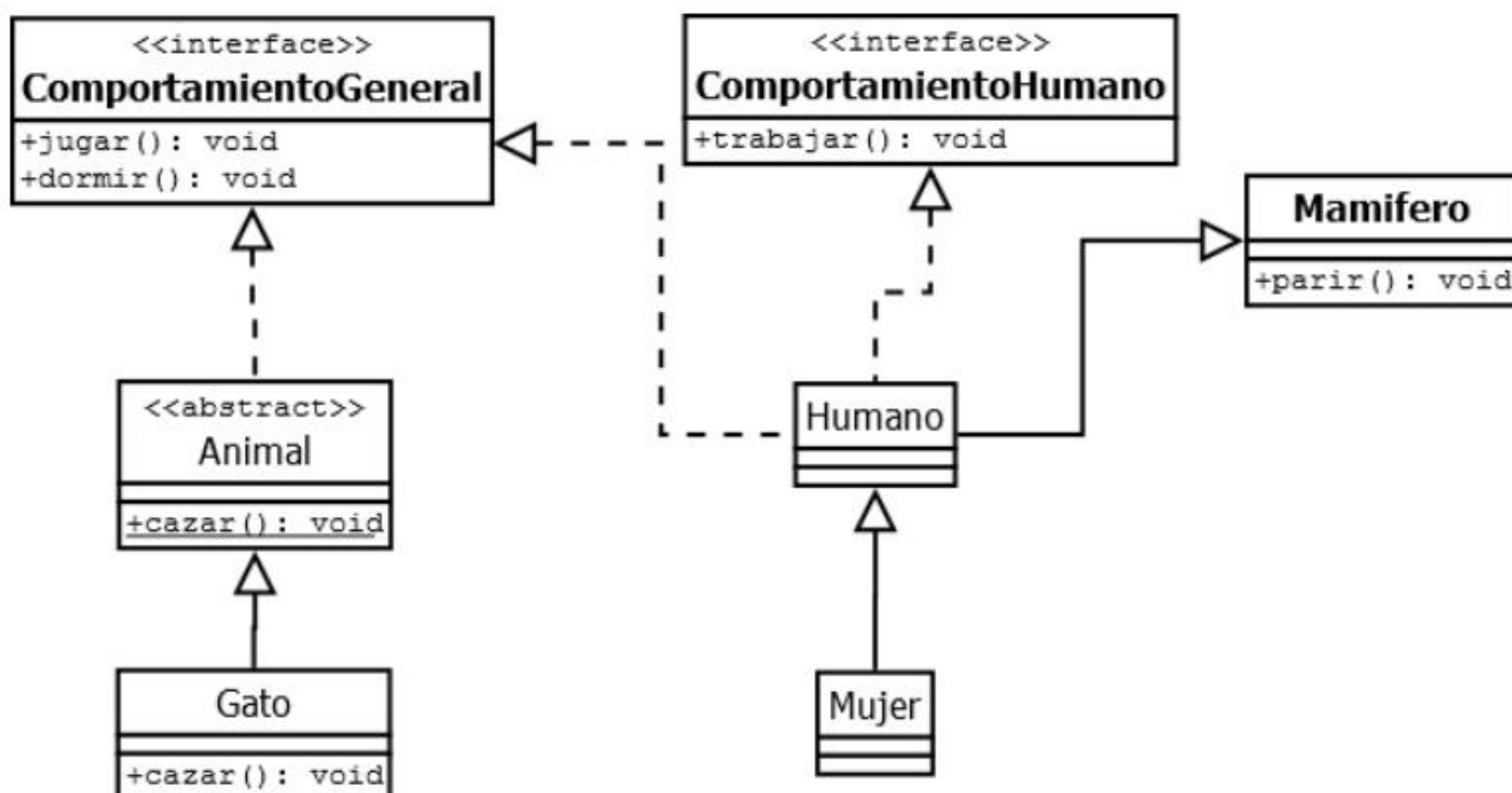


Figura 57 Diagrama interfaces y clases abstractas

### Solución:

El primer paso para codificar diagramas UML cuando existen jerarquías de herencia e implementación de interfaces es codificar las clases con mayor jerarquía (padres) y las interfaces.

Empezar con la interfaz ComportamientoGeneral que define dos métodos abstractos:

Ejemplo 104 Interfaz ComportamientoGeneral

```

1 | public interface ComportamientoGeneral {
2 | 
3 |     public void jugar();
4 |     public void dormir();
5 |

```

La interfaz ComportamientoHumano define un método abstracto:

Ejemplo 105 Interfaz ComportamientoHumano

```

1 | public interface ComportamientoHumano {
2 | 
3 |     public void trabajar();
4 |

```

La clase Mamífero es la súper clase de este diagrama, define un método normal parir, su codificación es:

Ejemplo 106 Clase Mamífero

```

1 | public class Mamifero {
2 | 
3 |     public void parir(){
4 |         System.out.println("Los mamíferos paren");
5 |

```

6      }

La clase abstracta Animal define un método abstracto cazar e implementa la interfaz ComportamientoGeneral, por lo que está en la obligación de implementar los métodos abstractos jugar y dormir, su código es:

*Ejemplo 107 Clase abstracta Animal*

```
1 public abstract class Animal implements ComportamientoGeneral{  
2  
3     public abstract void cazar();  
4  
5     @Override  
6     public void jugar() {  
7  
8         }  
9  
10    @Override  
11    public void dormir() {  
12  
13        }  
14    }
```

En resumen, la clase abstracta Animal está compuesta de dos métodos implementados y un método abstracto.

La clase Gato establece una relación de herencia con la clase abstracta Animal y está en la obligación de implementar el método abstracto cazar. Gato hereda tres métodos, dos implementados jugar y dormir y uno abstracto cazar.

*Ejemplo 108 Clase Gato*

```
1 public class Gato extends Animal{  
2  
3     @Override  
4     public void cazar() {  
5         System.out.println("caza ratones");  
6     }  
7 }
```

La clase Humano establece una relación de herencia con la clase Mamifero heredando su método parir. Además implementa dos interfaces ComportamientoGeneral y ComportamientoHumano, declarando los tres métodos y escribiendo su código respectivamente. Es decir, hereda un método directo e implementa tres métodos abstractos.

Ejemplo 109 Clase Humano

```
1 public class Humano extends Mamifero implements  
2     ComportamientoGeneral, ComportamientoHumano{  
3  
4     @Override  
5     public void jugar() {  
6         System.out.println("humano jugando");  
7     }  
8  
9     @Override  
10    public void dormir() {  
11        System.out.println("humano durmiendo");  
12    }  
13  
14    @Override  
15    public void trabajar() {  
16        System.out.println("humano trabajando");  
17    }  
18 }
```

La clase Mujer al establecer una relación de herencia con Humano, se beneficia de sus métodos, tres implementados por Humano y uno por herencia jerárquica con Mamífero.

Ejercicio 2: Realizar la codificación en lenguaje Java del siguiente diagrama UML.

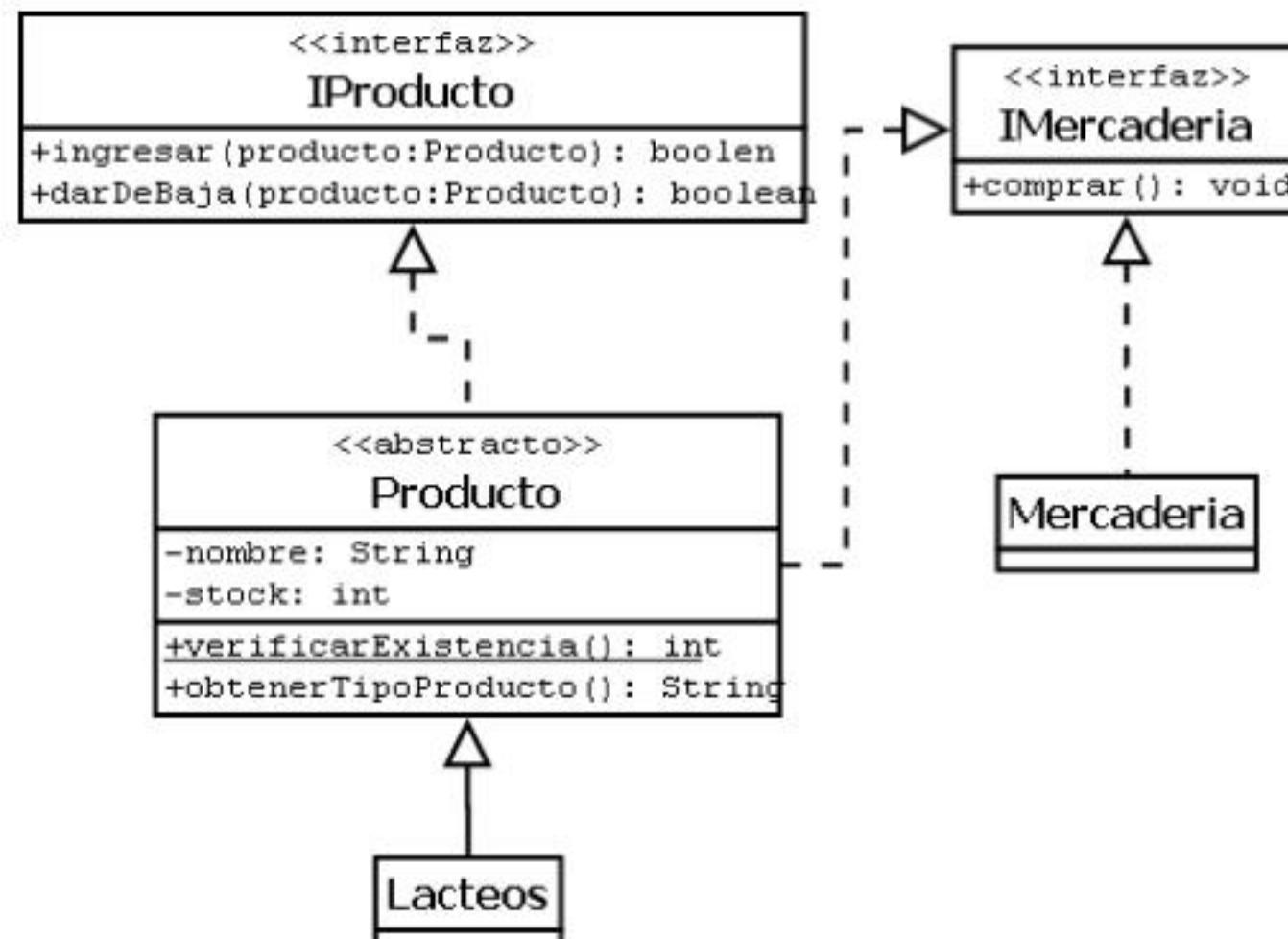


Figura 58 Diagrama Clase Abstracta e Interfaces

**Solución:**

El primer paso para codificar diagramas UML cuando existen jerarquías de herencia e implementación de interfaces es codificar las clases con mayor jerarquía (padres) y las interfaces.

Empezar con la interfaz “IProducto” que define dos métodos abstractos:

```
1 public interface IProducto {  
2     public boolean ingresar(Producto producto);  
3     public boolean darDeBaja(Producto producto);  
4 }
```

La interfaz “IMercaderia” define un método abstracto:

```
1 public interface IMercaderia {  
2     public void comprar();  
3 }
```

La clase “Mercadería” implementa a la interfaz IMercadería. Por lo tanto, está obligada a implementar un método abstracto “comprar”, su codificación es:

```
1 public class Mercaderia implements IMercaderia {  
2  
3     @Override  
4     public void comprar() {  
5         System.out.println("se compra mercadería");  
6     }  
7 }
```

La clase abstracta implementa las interfaces IMercado e IProducto, pero no está obligada a implementar los métodos de las interfaces al ser abstractos. La clase Producto define un método abstracto verificarExistencia y un método normal obtenerTipoProducto pero de forma implícita tiene tres métodos abstractos por implementar o no. En el siguiente ejemplo no se implementa ningún método de las interfaces, dejando a las clases que hereden de la clase Producto implementen todos los métodos abstractos.

```
1 public abstract class Producto implements IMercaderia, IProducto {  
2     private String nombres;  
3     private int stock;  
4  
5     //método abstracto  
6     public abstract int verificarExistencia();  
7  
8     //método normal  
9     public String obtenerTipoProducto(){  
10        return "";  
11    }  
12  
13    public String getNombres(){  
14        return nombres;  
15    }  
16  
17    public void setNombres(String nombres) {
```

```

18     this.nombres = nombres;
19 }
20
21 public int getStock() {
22     return stock;
23 }
24
25 public void setStock(int stock) {
26     this.stock = stock;
27 }
28
29 }
```

La clase Lácteos establece una relación de herencia con la clase abstracta Producto. Por lo tanto, Lácteos se debe implementar 4 métodos abstractos definidos en Producto de forma implícita o explícita. Además hereda el método obtenerTipoProducto

```

1 public class Lacteos extends Producto {
2
3     @Override
4     public int verificarExistencia() {
5         return getStock();
6     }
7
8     @Override
9     public void comprar() {
10        System.out.println("se compró un producto");
11    }
12
13     @Override
14     public boolean ingresar(Producto producto) {
15         return true;
16     }
17
18     @Override
19     public boolean darDeBaja(Producto producto) {
20         return false;
21     }
22 }
```

## 10.4 CUESTIONARIO

1. ¿Cuál es la palabra reservada que se emplea siempre en la cabecera de declaración de una interfaz?

- |             |              |
|-------------|--------------|
| - enum      | - implements |
| - interface | - super      |
| - extends   | - class      |
| - abstract  | - enum       |

**2. ¿Cuál es la palabra reservada que se emplea siempre en la cabecera de declaración de una clase abstracta?**

- |             |              |
|-------------|--------------|
| - enum      | - extends    |
| - super     | - implements |
| - interface | - class      |
| - abstract  | - eunm       |

**3. ¿Cuál es la palabra reservada que se emplea en la cabecera de una clase para implementar una interfaz**

- |              |             |
|--------------|-------------|
| - extends    | - interface |
| - implements | - super     |
| - class      | - enum      |
| - interface  | - abstract  |

**4. ¿Cuál es la palabra reservada que se emplea en la cabecera de una clase para heredar una clase abstracta**

- |              |             |
|--------------|-------------|
| - extends    | - interface |
| - implements | - super     |
| - class      | - enum      |
| - interface  | - abstract  |

**5. Seleccionar la que considere verdadera**

- Solo las clases abstractas pueden implementar las interfaces.
- Una clase solo puede implementar una interfaz.
- Una clase puede implementar dos interfaces.
- Una clase puede implementar un número indefinida de interfaces.

**6. Un método abstracto es:**

- Un método que no se implementa, es decir, no tiene un cuerpo.
- Un método sobre escrito por una clase implementadora.
- Un método que se implementa, es decir, tiene un cuerpo.
- Es un método que es definido solo por las interfaces.

**7. Una clase abstracta...**

- no se puede declarar atributos.
- no puede tener descendiente.
- no declara métodos.
- no se pueden crear instancias.

**8. Analizar el siguiente código y seleccionar la respuesta correcta:**

```
1 public interface Operaciones {  
2     public void guardar{  
3         System.out.println("se ha registrado");  
4     }  
5 }
```

- La interfaz define un método abstracto.

- Es una clase que define métodos implementados.
- Una interfaz solo puede definir métodos abstractos, no se puede definir métodos implementados.
- Al no definir un constructor en la interfaz, Java le asigna uno por defecto.

### 9. Analizar el siguiente código y seleccionar la respuesta correcta:

```

1 public interface IDedos {
2     public int obtenerNumeroDedos();
3 }
4
5 public abstract class Puma implements IDedos{
6     public int obtenerNumeroDedos(){
7         return 4;
8     }
9 }
10
11 public class GatoMontes extends Puma{
12     public static void main (String [] args){
13         Puma puma = new Puma();
14         System.out.println(puma.obtenerNumeroDedos());
15     }
16 }
```

- Imprime 4.
- Error de compilación en la línea 2.
- Error de compilación en la línea 6.
- Error de compilación en la línea 11.
- Error de compilación en la línea 13.
- Error de ejecución.

## 11 RELACIONES ENTRE CLASE

---

En este capítulo se explica los conceptos de relaciones entre clases, los roles y cardinalidades. Se indica la representación de las relaciones de asociación, agregación y composición en UML y el mecanismo que se implementa en código Java. Para finalizar se explica en ejemplos cómo se vinculan las clase.

### Objetivos:

- Representar en UML las relaciones de asociación, agregación y composición.
- Codificar los mecanismos para establecer las relaciones entre clases.

### 11.1 DEFINICIÓN

Hasta este capítulo se ha revisado la herencia y la implementación de interfaces definidas en UML; se podría decir que se ha trabajado con relaciones horizontales. Las relaciones verticales entre clases están aisladas unas de otras, pero de alguna manera deben estar vinculas.

De acuerdo con la Real Académica de la Lengua Española, la definición de relación es “Conexión, correspondencia de algo con otra cosa”. Partiendo de este concepto se puede definir desde la programación orientada a objetos como conectar dos objetos entre sí permitiendo que los objetos colaboren entre sí, intercambiándose información a través de sus atributos y métodos.

Las relaciones en Java pueden ser de dos tipos:

- **Persistentes:** si la comunicación entre los objetos se registra de algún modo y por tanto puede ser utilizada en cualquier momento.
- **No persistentes:** entonces el vínculo entre objetos desaparece tras ser empleado.

Las relaciones se pueden establecer entre clases y objetos:

- **Entre clases:** herencia o generalización.
- **Entre objetos o instancias:** asociación, agregación y composición.

## 11.2 ASOCIACIÓN

Es una relación estructural que describe una conexión entre dos instancias A y B. Una asociación es una relación débil debido a que cada una existe de forma independiente, inclusive si se rompe la relación no afecta el desarrollo normal de los dos objetos.

Para establecer la asociación al menos, un atributo de la clase B es una referencia a un objeto de la clase A. Es decir, la creación de B no implica la creación de A. En UML este tipo de relación se representa con una línea recta uniendo dos clases.

### 11.2.1 NAVEGACIÓN DE LAS ASOCIACIONES

Es importante establecer el sentido o la dirección para interpretar la relación, estas pueden ser unidireccionales o bidireccionales. Cuando la relación es unidireccional, se representa con una flecha el sentido de la asociación. En UML se representa así:

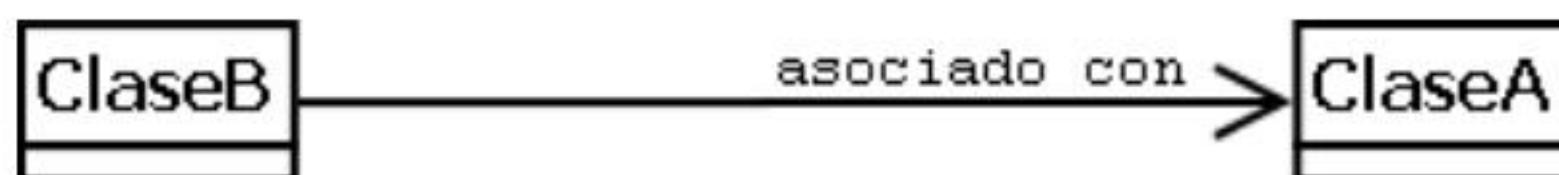


Figura 59 Asociación unidireccional

En la figura se establece que la Clase B está asociada a la clase A.

Las asociaciones comparten características importantes para interpretar su comportamiento:

- **Rol:** Es el papel que juega en dicha relación el objeto situado en cada extremo.
- **Cardinalidad o multiplicidad:** Es el número mínimo y máximo de instancias que pueden relacionarse con la otra instancia del extremo opuesto de la relación.

Tabla 19 Cardinalidades

| Cardinalidad | Descripción         |
|--------------|---------------------|
| 0.. 1        | De cero a uno (0,1) |
| 1.. 1        | De uno a uno (1)    |
| 0..*         | De cero a muchos    |
| 1..*         | De uno a muchos     |
| *..*         | De muchos a muchos  |

En UML la relación de asociación con su rol y cardinalidad se representan:



Figura 60 Asociación unidireccional

Si se toma el concepto de “al menos, un atributo de la clase B es una referencia a un objeto de la clase A”, se puede deducir que “un Estudiante está asociado con una Dirección”, su rol establece el papel que juega en la relación y la cardinalidad fija el número de instancias entre ambas clases: “un Estudiante tiene una Dirección”.

La codificación en Java es:

Ejemplo 110 Clase Dirección

```
1 public class Direccion {
2     private String barrio;
3     private String callePrincipal;
4     private String calleSecundaria;
5 }
```

Ejemplo 111 Clase Estudiante

```
1 public class Estudiante {
2     private String nombres;
3     private String apellidos;
4     // se establece la relación de asociación
5     private Direccion direccion;
6 }
```

Al diseñar un diagrama de clases y no especificar la flecha en una relación, de forma implícita la relación es bidireccional.

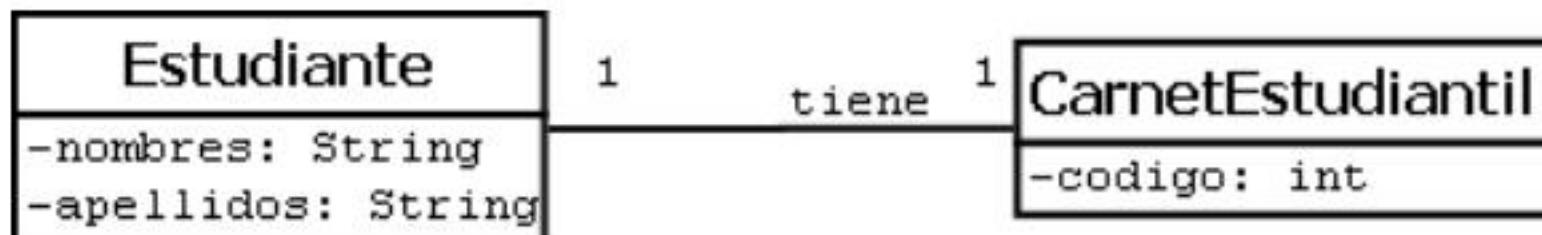


Figura 61 Relación de asociación bidireccional

Para establecer la asociación al menos, un atributo de la clase Estudiante es una referencia a un objeto de la clase CarnetEstudiantil y al menos, un atributo de la clase CarnetEstudiantil es una referencia a un objeto de la clase Estudiante. Los atributos de referencia se aplican a ambas clases. Su codificación es:

Ejemplo 112 Relación Bidireccional - Clase Matrícula

```

1 import java.util.Date;
2
3 public class CarnetEstudiantil {
4     private int codigo;
5     // se establece la relación de asociación
6     private Estudiante estudiante;
7 }
  
```

Ejemplo 113 Relación Bidireccional - Clase Estudiante

```

1 public class Estudiante {
2     private String nombres;
3     private String apellidos;
4     // se establece la relación de asociación
5     private CarnetEstudiantil carnet;
6 }
  
```

Hasta este punto, se ha codificado relaciones de uno a uno, pero existen casos o diseños en UML que la relación tiene más de una instancia. Las relaciones de uno a muchos (1..\*) se las representa por medio de un arreglo, lista, colección o cualquier tipo estructura de lineal estática o dinámica. En este caso, solo se utilizará los arreglos [ ].

Tabla 20 Representación de la cardinalidad

| Cardinalidad | Representación          |
|--------------|-------------------------|
| 1            | Atributo por referencia |
| *            | Arreglo []              |

En UML este tipo de relaciones se representa por el carácter \*, como se ilustra en la siguiente figura:

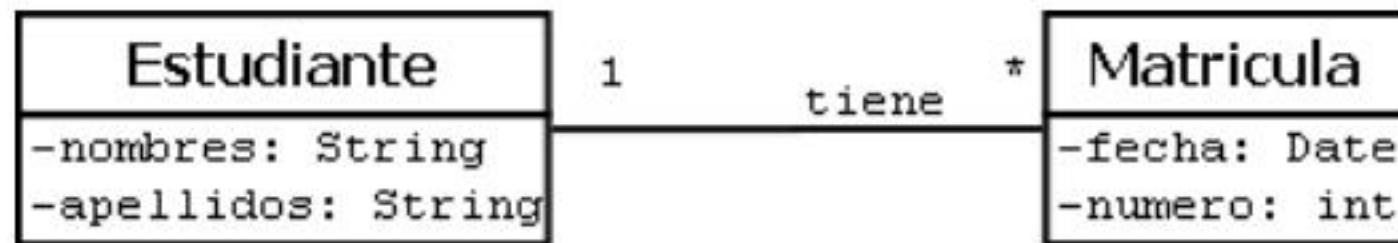


Figura 62 Asociación de 1..\*

Su código en Java es:

Ejemplo 114 Relación de uno a muchos - Clase Matrícula

```

1 import java.util.Date;
2
3 public class Matricula {
4     private int codigo;
5     // se establece la relación de asociación
6     private Estudiante estudiante;
7 }

```

Ejemplo 115 Relación Bidireccional - Clase Estudiante

```

1 public class Estudiante {
2     private String nombres;
3     private String apellidos;
4     // se establece la relación de asociación
5     private Matricula [] matricula;
6 }

```

El último escenario es que la relación de muchos a muchos(\*..\*). En UML se representa de la siguiente forma:

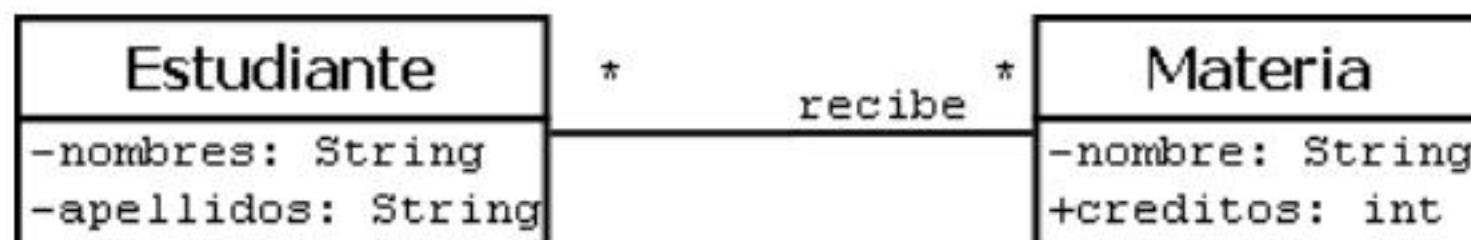


Figura 63 Relación \*..\*

Ejemplo 116 Relación de uno a muchos - Clase Matrícula

```

1 import java.util.Date;
2
3 public class Materia {
4     private String nombre;
5     private int creditos;
6     // se establece la relación de asociación
7     private Estudiante [] estudiante;
8 }

```

Ejemplo 117 Relación Bidireccional - Clase Estudiante

```

1 public class Estudiante {
2     private String nombres;
3     private String apellidos;
4     // se establece la relación de asociación
5     private Materia [] materia;
6 }

```

## 11.3 AGREGACIÓN Y COMPOSICIÓN

Son relaciones que tienen un acoplamiento débil y fuerte respecto a las clases que se vinculan. En Java estas relaciones no tienen una palabra reservada específica, sino que se debe establecer un mecanismo a nivel de código.

Este tipo de relaciones son definidas por “todo-parte”, “tiene-un”, “parte-de”, describiendo que existe dos clases, la primera es un contenedor de elementos y la segunda es una clase que es parte de otra.

### 11.3.1 AGREGACIÓN

Es una relación similar a la asociación unidireccional, que representa “todo - partes”. El “todo” representa a la clase que agrega las clases y las “partes” son las clases que serán agregadas en la clase agregadora (todo). En UML se representa por una línea y un diamante de color blanco en su extremo:

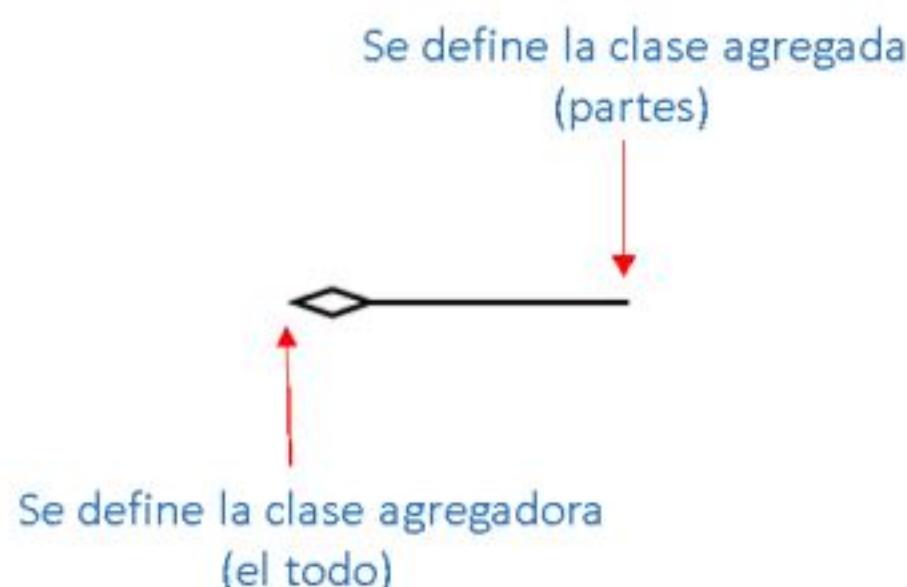


Figura 64 Representación de agregación

Donde nace la flecha se ubica la clase “partes” y en la cabecera de la flaca el “todo”.

La agregación es un tipo de composición débil debido a que los componentes pueden ser compartidos por varios contenedores. La destrucción de los elementos agregados no conlleva la destrucción de la clase contenedora.

Analicemos el siguiente diagrama:



Figura 65 Agregación entre clases

En este ejemplo, existe una relación de agregación siendo el todo la clase “Carrera” y la parte “Postulante”. La cardinalidad es un punto a tener en consideración, cuando existe una relación de uno a muchos se debe crear métodos para agregar y quitar el objeto en la clase todo. Su codificación es:

Ejemplo 118 Clase parte - Agregación - Carrera

```
1 public class Postulante {  
2     private String nombres;  
3  
4     public Postulante(String nombres) {  
5         this.nombres = nombres;  
6     }  
7  
8     public String getNombres() {  
9         return nombres;  
10    }  
11  
12    public void setNombres(String nombres) {  
13        this.nombres = nombres;  
14    }  
15}
```

En la clase “parte” no es necesario debe existir una relación con “Carrera”.

Ejemplo 119 Clase todo - agregación - Carrera

```
1 public class Carrera {  
2     private String denominacion;  
3     private Postulante[] postulantes;  
4  
5     public Carrera(int limite){  
6         postulantes = new Postulante[limite];  
7     }  
8  
9     public void agregarPostulante(int posicion, Postulante  
10                                postulante){  
11         postulantes[posicion] = postulante;  
12     }  
13  
14     public void quitarPostulante(int posicion){  
15         postulantes[posicion] = null;  
16     }  
17  
18     public void imprimirPostulantes(){  
19         for (int i = 0; i < postulantes.length; i++) {  
20             if (postulantes[i] != null)  
21                 System.out.println("p[" + i + "]=" +  
22                               postulantes[i].getNombres());  
23             else  
24                 System.out.println("p[" + i + "]=" +  
25                               postulantes[i]);  
26         }  
27     }  
28 }
```

La clase Carrera es el todo o el contenedor de elementos y Postulante se convierte en su atributo, pero por la cardinalidad de uno a muchos, se convierte en un arreglo de objetos de la clase Postulante.

En la agregación los objetos se añaden o se quitan si comprometer el comportamiento de ambos objetos.

Ejemplo 120 Agregación - Clase Ejecutora

```
1 public class Ejecutor {  
2  
3     public static void main(String[] args) {  
4         Postulante p1 = new Postulante("Fabián Valarezo");  
5         Postulante p2 = new Postulante("María Delgado");  
6         Postulante p3 = new Postulante("Karina Zumba");  
7  
8         //agregan las partes al todo  
9         Carrera carrera = new Carrera(3);  
10        carrera.agregarPostulante(0, p1);  
11        carrera.agregarPostulante(1, p2);  
12        carrera.agregarPostulante(2, p3);  
13  
14        // se quitan algunas partes  
15        carrera.quitarPostulante(2);  
16  
17        carrera.imprimirPostulantes();  
18    }  
19 }
```

### 11.3.2 COMPOSICIÓN

Es una relación fuerte que establece que el todo controle la existencia de las partes. Es decir, la vida del objeto parte debe coincidir con la del todo. El vínculo que se establece es tan fuerte que las partes no pueden pertenecer a otras clases y al eliminar el objeto “todo”, también es eliminado los objetos “partes”

En UML se representa por una línea y un diamante de color negro en su extremo:

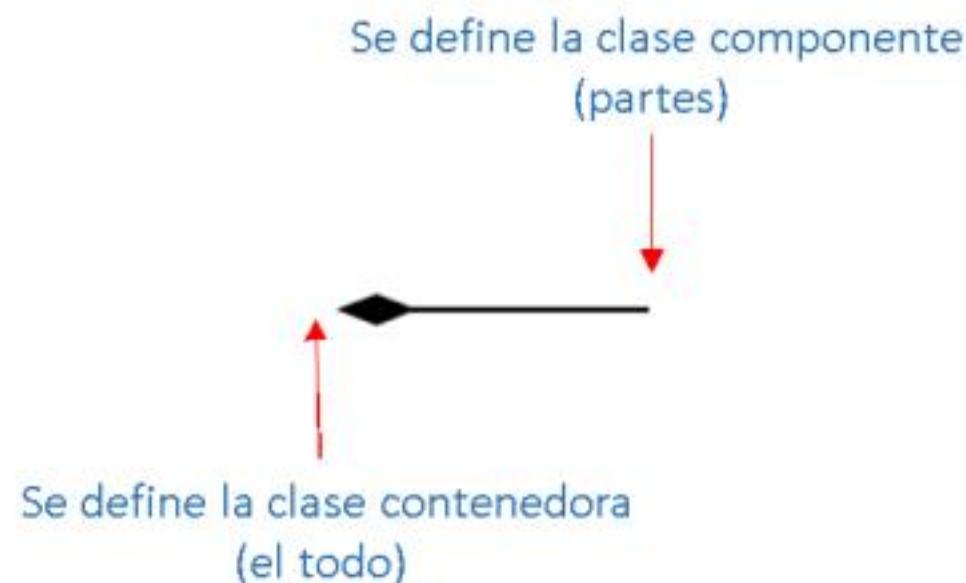


Figura 66 Relación de composición

Donde nace la flecha se ubica la clase “partes” y en la cabecera de la flaca el “todo”.

La composición es un tipo de composición fuerte debido a que los componentes pueden solo pertenecer a la clase contenedora. La destrucción de los elementos agregados conlleva a la destrucción de la clase contenedora.

Analicemos el siguiente diagrama:

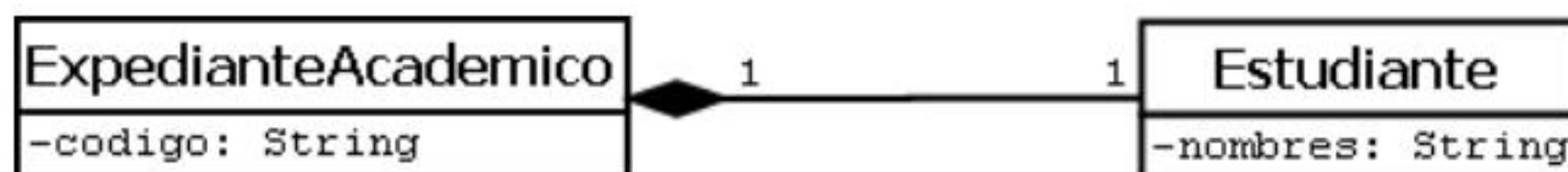


Figura 67 Composición entre clases

En este diagrama, existe una relación de composición siendo el todo la clase ExpedienteAcademico y la parte Estudiante. Su codificación es:

```

1 public class Estudiante {
2     private String nombres;
3     private ExpedienteAcademico expediente;
4
5     public String getNombres() {
6         return nombres;
7     }
8
9     public void setNombres(String nombres) {
10        this.nombres = nombres;
11    }
12
13    public ExpedienteAcademico getExpediente() {
14        return expediente;
15    }
16
17    public void setExpediente(ExpedienteAcademico expediente) {
18        this.expediente = expediente;
19    }
20 }
  
```

En la clase “parte” es imprescindible la relación con el todo ExpedienteAcademico. La cardinalidad siempre se establece de 0..1 o de 1..1. En la línea 3 se establece la relación.

```

1  public class ExpedienteAcademico {
2      private String codigo;
3      private Estudiante estudiante;
4
5      public ExpedienteAcademico(String codigo, String nombres) {
6          this.codigo = codigo;
7          // se establece la composición
8          this.estudiante = new Estudiante(nombres, this);
9      }
10
11     public String getCodigo() {
12         return codigo;
13     }
14
15     public void setCodigo(String codigo) {
16         this.codigo = codigo;
17     }
18
19     public Estudiante getEstudiante() {
20         return estudiante;
21     }
22
23     public void setEstudiante(Estudiante estudiante) {
24         this.estudiante = estudiante;
25     }
26 }
```

La composición es una relación fuerte que se establece al momento de crear la instancia del todo ExpedienteAcademico, es decir, en su constructor. En la línea 8 se crea una instancia de “Estudiante” obligando que la clase parte exista al crear la clase contenedora.

Para finalizar las relaciones de agregación y composición tienen algunas diferencias que es necesario tomar a consideración:

*Tabla 21 Diferencias entre las relaciones de agregación y composición*

| Componentes                                                    | Agregación | Composición |
|----------------------------------------------------------------|------------|-------------|
| Los objetos partes pueden ser compartidos por otras relaciones | SI         | NO          |
|                                                                | 0..1       | 0..1        |
|                                                                | 1..1       | 1..1        |
| Cardinalidad                                                   | 0..1       | 1..*        |

\* .. \*

|                                                                      |                              |                             |
|----------------------------------------------------------------------|------------------------------|-----------------------------|
| Destrucción de los objetos partes al destruir el objeto todo         | NO                           | SI                          |
| Relación se establece en un atributo por referencia en ambos objetos | NO                           | SI                          |
| Representación gráfica en UML                                        | Flecha<br>diamante<br>blanco | Flecha<br>diamante<br>negra |

---

## **12 EXCEPCIONES**

---

El presente capítulo introduce los conceptos de excepciones, se explica cómo funciona cada bloque para el tratamiento de los errores de ejecución. Para finalizar se crea clases propias para el manejo de excepciones.

### **Objetivos**

- Explicar el funcionamiento de cada bloque que compone el manejo de excepciones.
- Lanzar y propagar excepciones.
- Crear una clase personalizada para el manejo de excepciones.

### **12.1 DEFINICIÓN**

Las excepciones son errores que se dan al momento de ejecutar la aplicación (tiempo de ejecución), es decir son aquellos errores que la máquina virtual no puede manejar por sí sola, estos errores distorsionan la lógica de una aplicación, un ejemplo que se generan este tipo errores son:

- Transformar de un *String* a un número, si el formato del número está incorrecto.
- Acceder algún archivo que no existe.
- Establecer conexión a una base de datos que está en una ubicación incorrecta.
- Programar aplicaciones distribuidas.

Los errores normalmente se los detecta en el proceso de compilación, pero las excepciones no se las puede detectar en ese proceso, son errores que en el proceso de ejecución aparecen, el programador no las puede corregir. En Java se las puede tratar por medio de las excepciones.

En el desarrollo de sistemas (programación) siempre hay errores de ejecución y hay que saber gestionarlos de una manera óptima, es por ello, que Java posee una clase que permite gestionar los errores y las excepciones, y es la clase *Throwable*.

La clase *Throwable* contiene la instancia en la pila del objeto que se acaba de crear (*stack trace*), almacena el mensaje de error y las causas que produjeron dicho error.

Java permite diferenciar dos tipos de errores:

**Error:** son subclases de *Throwable*, e indican que existe un error grave en la aplicación, estos errores no se los debe tratar de solucionar dentro de la aplicación.

Ejemplo: agotamiento de memoria, llamado a librerías que no existen, error interno de la máquina virtual de java, etc.

**Excepción:** igual que los Error son subclases de *Throwable*, tanto esta clase como sus subclases indican que se ha producido errores y que se los puede tratar adecuadamente, se describirá algunas de las excepciones que se pueden suscitar:

- **IOException:** son excepciones de entrada/salida, un ejemplo clásico sería el acceso a un repositorio de datos que no exista (archivos).
- **RunTimeException:** son excepciones del desarrollador de sistemas y pueden ser lanzadas durante el funcionamiento normal de la máquina virtual, ejemplo al momento de hacer un casting de objetos que son de diferente naturaleza.
- **SQLException:** son excepciones enviadas por el proveedor de base de datos y se lanzan al momento de enviar un comando erróneo.
- **TimeoutException:** esta excepción se ejecuta cuando una operación se demora demasiado.

En la siguiente figura se ilustra el árbol jerárquico las excepciones en Java.

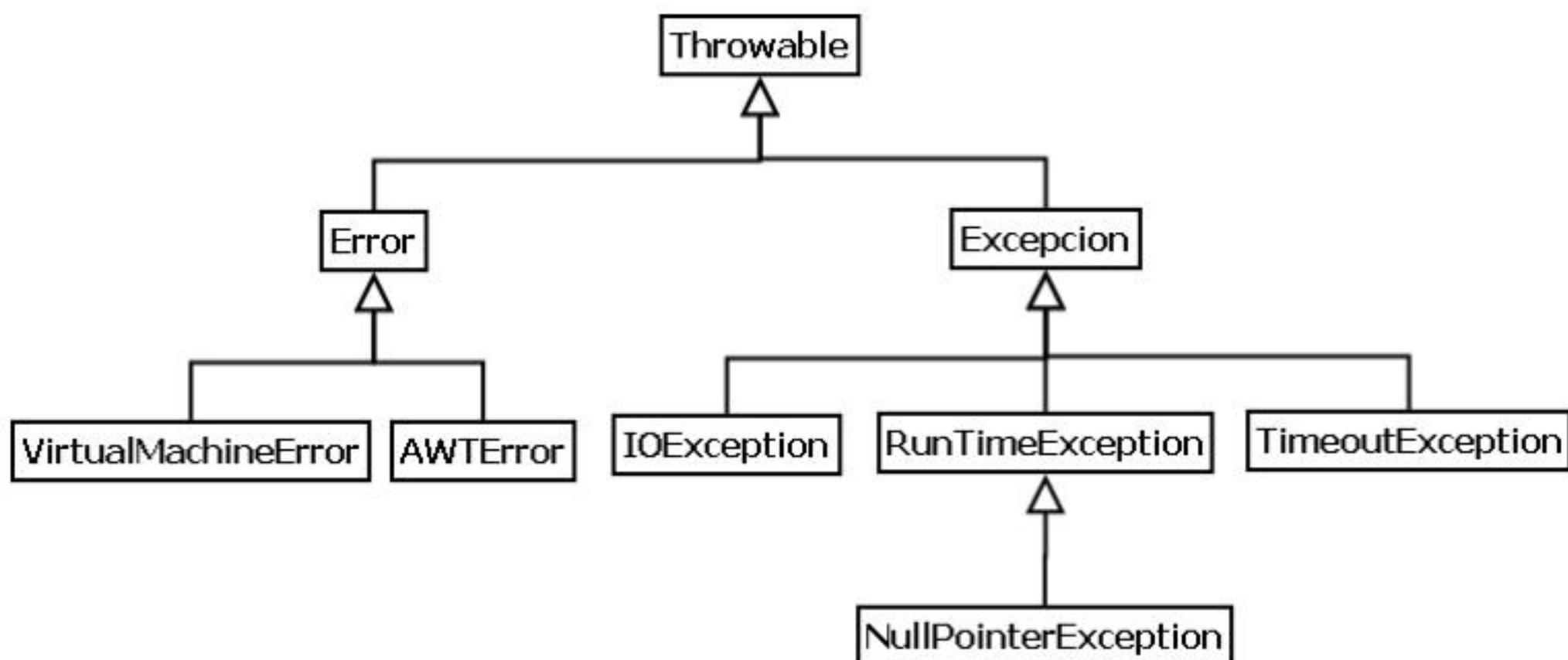


Figura 68: Diagrama de clase de Excepciones

Java proporciona tres mecanismos para atrapar excepciones las cuales se demuestran a continuación.

## 12.2 CAPTURA DE EXCEPCIONES

Para la captura de excepciones se utiliza los bloques **try** y **catch** y si se desea ejecutar código aunque se produzca una excepción se utiliza el bloque **finally**, estos bloques se los utiliza dentro de los métodos.

### BLOQUE try

En este bloque va el código que en donde posiblemente se producirá una excepción y así poder atraparlo y tratarlo correctamente, su sintaxis sería la siguiente:

```

....método () {
    try {
        // código donde posiblemente ocurra una excepción
    }.....
}
    
```

### BLOQUE catch

Es el bloque en el cual permite tratar la excepción, recibe un objeto *Throwable* como argumento, entra en acción cuando atrapa una excepción en el bloque try, su sintaxis es la siguiente:

```

....método () {
}
    
```

```

try {
    // código donde posiblemente ocurra una excepción
} catch (Exception ex) {
    // código donde trataremos la excepción
}
}

```

Se pueden anidar más de un **catch** después de la sentencia **try**.

Analizar el siguiente ejemplo: Se quiere transformar un número en tipo *String* a un **double**, se crea el siguiente método:

*Ejemplo 121 Clase EjemploExcepcion*

```

1 public class EjemploExcepcion {
2
3     public double transformarString(String number) {
4         return Double.parseDouble(number);
5     }
6
7     public static void main(String[] args) {
8         EjemploExcepcion ejemplo = new EjemploExcepcion();
9         ejemplo.transformarString("2N");
10    }
11 }

```

En el siguiente código se define un método *transformaString* que recibe como parámetro una cadena de texto con el objetivo de transformar un tipo de dato *String* a un **double** con el método *parseDouble* de la clase *Double*.

En el método *main* se crea un objeto de la clase *EjemploException*, se invoca al método *transformarString* y se envía una cadena “2N”. Al enviar este dato se genera una excepción porque el dato enviado es incorrecto y al transformarlo en un double no coincide en un elemento válido, “2N” no se podrá transformar en un dato decimal, la salida generada es:

La salida del ejemplo es:

```

1 \programas_libro>java EjemploExcepcion
2 Exception in thread "main" java.lang.NumberFormatException: For input string: "2N"
3         at sun.misc.FloatingDecimal.readJavaFormatString( FloatingDecimal.java:2043)
4         at sun.misc.FloatingDecimal.parseDouble( FloatingDecimal.java:110)
5         at java.lang.Double.parseDouble( Double.java:538)
6         at controlador.EjemploExcepcion.transformarString( EjemploExcepcion.java:14)
7         at controlador.EjemploExcepcion.main( EjemploExcepcion.java:18)
8 Java Result: 1
9
10
11

```

12  
13

La excepción genera es un error de ejecución, y la única manera de manejarlas o tratarles es a través del bloque de **try / catch**. Agregando este bloque en el ejemplo anterior, se genera el siguiente código:

Ejemplo 122 Método atrapando excepciones

```
1 public class EjemploExcepcion {  
2  
3     public double transformarString(String number) {  
4         double numeroTransformado = 0;  
5         try {  
6             numeroTransformado =  
7                 Double.parseDouble(number);  
8         } catch (NumberFormatException e) {  
9             System.out.println("Se produjo un error al  
10                momento de transformar");  
11             numeroTransformado = -1;  
12         }  
13         return numeroTransformado;  
14     }  
15 }  
16  
17 public static void main(String[] args) {  
18     EjemploExcepcion ejemplo = new EjemploExcepcion();  
19     ejemplo.transformarString("2N");  
20 }  
21 }
```

A través del **try / catch** se atrapa la excepción, si en el bloque **try** se genera un error de ejecución, el bloque **catch** atrapa la excepción e imprime un mensaje indicando el “error generado”. Al ejecutar el ejemplo, imprime el siguiente resultado:

1 \programas\_libro> java EjemploExcepcion  
2 Se produjo un error al momento de transformar

## 12.3 BLOQUE finally

En bloque **finally** se puede ejecutar código independientemente si se haya producido un error o no, incluso se ejecuta después de una sentencia **return** dentro del bloque **try**, su sintaxis es:

```
.....método () {  
    try{  
        .....  
    } finally {  
        // Código a ejecutarse no importa si se produjo una excepción
```

```
 }.....
```

En el ejemplo de trasformar de un *String* a un *double* se puede añadir el bloque *finally* para imprimir un mensaje, el código generado es el siguiente:

Ejemplo 123 Método transformarString con bloque finally

```
1 public class EjemploExcepcion {  
2     public Double transformarString(String number) {  
3         double numeroTransformado = 0;  
4         try {  
5             numeroTransformado =  
6             Double.parseDouble(number);  
7         } catch (NumberFormatException e) {  
8             System.out.println("Se produjo un error al  
9             momento de transformar");  
10            numeroTransformado = -1;  
11        } finally {  
12            System.out.println("bloque finally ");  
13        }  
14        return numeroTransformado;  
15    }  
16  
17    public static void main(String[] args) {  
18        EjemploExcepcion ejemplo = new EjemploExcepcion();  
19        ejemplo.transformarString("2N");  
20    }  
21 }
```

Al ejecutar la clase, se genera la excepción, se la atrapa mediante *catch* e imprime y ejecuta el bloque *finally*, su salida es la siguiente:

La salida del ejemplo es:

```
1 \programas_libro> java EjemploExcepcion  
2 Se produjo un error al momento de transformar  
3 Bloque finally
```

## 12.4 LANZAR UNA EXCEPCIÓN

Se puede hacer que el programa lance una excepción de manera explícita utilizando la sentencia *throw* junto a una instancia de la clase *Excepción* ya sea propia de Java o construida por el programador. Su sintaxis es la siguiente:

```
.....método () {  
    throw new Exception("mensaje");  
}.....
```

Para comprobar su utilidad, se definirá un método que permita validar la edad para hacer compras con tarjeta de crédito, si la edad es mayor a 18 años, se podrá

hacer compras caso contrario lanzará una excepción con el mensaje “La edad para hacer compra es de 18 años o más”. En la clase EjemploExcepcion se agrega el siguiente método ValidarEdadCompra:

Ejemplo 124 Lanzar una excepción

```
1 public class EjemploExcepcion {  
2  
3     public boolean validarEdadCompra(int edad) {  
4         boolean verificar = false;  
5         if(edad < 0)  
6             throw new RuntimeException("La edad no puede ser  
7                 negativa");  
8         if(edad < 18)  
9             throw new RuntimeException("La edad para hacer  
10                compra es de 18 años o más");  
11         return verificar;  
12     }  
13  
14     public static void main(String[] args) {  
15         EjemploExcepcion ejemplo = new EjemploExcepcion();  
16         ejemplo.validarEdadCompra(1);  
17         System.out.println("****");  
18     }  
19 }
```

En este ejemplo, en la línea 6 y 9 se lanza una Excepción con la sentencia `throw` a través de una instancia de la clase `RuntimeException`. Al ejecutar el método `main` genera la siguiente salida:

```
1 \programas_libro> java EjemploExcepcion  
2 Exception in thread "main" java.lang.RuntimeException: La edad para hacer compra es de  
3 18 años o más  
4     at controlador.EjemploExcepcion.validarEdadCompra(EjemploExcepcion.java:31)  
5     at controlador.EjemploExcepcion.main(EjemploExcepcion.java:37)  
6 Java Result: 1  
7  
8  
9
```

En el instante de lanzar una excepción en un método, habrá que atraparla donde se lo está invocando. En el método ejecutor se agrega el bloque `try / catch` para atrapar la excepción como se ilustra en el siguiente ejemplo:

Ejemplo 125 Lanzar una excepción - atrapar try / catch

```
1 public class EjemploExcepcion {  
2  
3     public boolean validarEdadCompra(int edad) {  
4         boolean verificar = false;  
5         if(edad < 0)  
6             throw new RuntimeException("La edad no puede ser  
7                 negativa");  
8     }  
9 }
```

```

8   if(edad < 18)
9     throw new RuntimeException("La edad para hacer
10       compra es de 18 años o más");
11   return verificar;
12 }
13
14 public static void main(String[] args) {
15   EjemploExcepcion ejemplo = new EjemploExcepcion();
16   try { //atrapa la excepción
17     ejemplo.validarEdadCompra(1);
18   } catch (RuntimeException e) {
19     System.out.println("Error en validar edad");
20   }
21   System.out.println("****");
22 }
23 }
```

Al capturar la excepción, automáticamente se puede continuar con la ejecución del programa. A ejecutar la aplicación se imprime el siguiente resultado:

```

1 \programas_libro> java EjemploExcepcion
2 error al validar edad
3 ****
```

## 12.5 PROPAGACIÓN DE EXCEPCIONES

La propagación de excepciones se da cuando los métodos lanzan una excepción, esto se puede llegar hacer al momento de incluir en la cabecera del método la palabra reservada **throws**, cuando este método es llamado desde otra clase le informa que necesariamente debe atrapar una excepción, un ejemplo clásico son cuando se trabaja con archivos. Su sintaxis es la siguiente:

```

.... método (parámetros) throws excepcion1, excepcion2 {
    // código del método
}
```

En este método “validarEdadCompra”, se efectuará un cambio para que la excepción se propague:

*Ejemplo 126 Propagación de excepciones*

```

1 public class EjemploExcepcion {
2
3   public boolean validarEdadCompra(int edad) throws
4     Exception {
5     boolean verificar = false;
6     if(edad < 0)
7       throw new RuntimeException("La edad no puede ser
8         negativa");
9     if(edad < 18)
```

```

10     throw new RuntimeException("La edad para hacer
11         compra es de 18 años o más");
12     return verificar;
13 }
14
15 public static void main(String[] args) {
16     EjemploExcepcion ejemplo = new EjemploExcepcion();
17     ejemplo.validarEdadCompra(1);
18     System.out.println("****");
19 }
20 }
```

Al añadir la cláusula `throws` en la cabecera del método, se obliga a atrapar la excepción al invocar al método “validarEdadCompra”, las excepciones que se hace referencia es la clase `Exception` o `IOException`.

## 12.6 CREAR NUESTRAS PROPIAS EXCEPCIONES

En java se puede crear propias excepciones de forma sencilla, solo se necesita crear una subclase de un tipo de excepción ya existente. En este ejemplo, se crea una clase “ValidarEdadCompraExcepcion” que hereda de la clase `Exception`. Además, se implementa el método “validarEdadCompra” que llama al constructor de la clase Padre enviándole una cadena de texto que será el mensaje que se imprima utilizar esta clase. Su código es:

*Ejemplo 127 Código de creación de una Excepción*

```

1 public class ValidarEdadCompraExcepcion extends RuntimeException {
2     public ValidarEdadCompraExcepcion() {
3         super("La edad para hacer compra es de 18 años o más");
4     }
5 }
```

En la clase `EjemploExcepcion` se agrega una instancia de la clase “ValidarEdadCompraException” para generar una excepción personalizada.

*Ejemplo 128 Llamada a la clase ValidarEdadCompraExcepcion en el método validarEdadCompra*

```

1 public class EjemploExcepcion {
2
3     public boolean validarEdadCompra(int edad) throws
4         RuntimeException{
5         boolean verificar = false;
6         if(edad < 0)
7             throw new RuntimeException("La edad no puede ser
8                 negativa");
9         if(edad < 18){
10             // se llama a la excepción personalizada
```

```

11     throw new ValidarEdadCompraExpcion();
12 }
13 return verificar;
14 }
15
16 public static void main(String[] args) {
17     EjemploExpcion ejemplo = new EjemploExpcion();
18     ejemplo.validarEdadCompra(1);
19     System.out.println("****");
20 }
21 }
```

A ejecutar la clase, se genera la siguiente salida:

```

1 \programas_libro> java EjemploExpcion
2 Exception in thread "main" controlador.ValidarEdadCompraExpcion: La edad para hacer
3 compra es de 18 años o mas
4     at controlador.EjemploExpcion.validarEdadCompra(EjemploExpcion.java:33)
5     at controlador.EjemploExpcion.main(EjemploExpcion.java:41)
6 Java Result: 1
7
8
9
```

## 12.7 PREGUNTAS

### 1. ¿Qué son las excepciones?

- Son errores que se dan al momento de ejecutar la aplicación, es decir son aquellos errores que la máquina virtual no puede manejar por sí sola.
- Son errores de compilación que deben ser corregidas en el código por el programador.
- Es un mecanismo que permite atrapar los errores en los bloques try / catch.
- Son clase que permiten manejar errores en tiempo de ejecución.

### 2. El bloque dónde se coloca el código y posiblemente se producirá una excepción, así poder atraparlo y tratarlo correctamente es:

- |                                                                          |                                                                               |
|--------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- try</li> <li>- catch</li> </ul> | <ul style="list-style-type: none"> <li>- finally</li> <li>- throws</li> </ul> |
|--------------------------------------------------------------------------|-------------------------------------------------------------------------------|

### 3. El bloque en el cual permite tratar la excepción, recibe un objeto Throwable como argumento, entra en acción cuando detecta una excepción es:

- |                                                                          |                                                                               |
|--------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- try</li> <li>- catch</li> </ul> | <ul style="list-style-type: none"> <li>- finally</li> <li>- throws</li> </ul> |
|--------------------------------------------------------------------------|-------------------------------------------------------------------------------|

**4. En bloque dónde se puede ejecutar código independientemente si se haya producido un error o no es:**

- try
- catch
- finally
- throws

**5. Para lanzar una excepción de manera explícita se utiliza la palabra reservada:**

- try
- catch
- throw
- finally
- throws

**6. La propagación de excepciones se da cuando los métodos lanzan una excepción, esto se puede llegar hacer al momento de incluir en la cabecera del método la palabra reservada:**

- throw
- throws
- try
- catch
- finally

## 13 CLASES FUNDAMENTALES

---

En este capítulo se describe las clases más utilizadas y comunes en Java. Se detalla en tablas sus métodos principales y su descripción. Se explica el rol que cumplen las clases envolventes con respecto a los datos primitivos y la funcionalidad de cada una. Para finalizar, a través de ejemplos se explica el comportamiento de las clases *Math*, *Scanner* y *Date*.

### Objetivos

- Establecer la representación de una clase por cada tipo de dato primitivo.
- Realizar operaciones matemáticas con la clase *Math*.
- Generar entradas de datos con diferentes tipos de datos.
- Manejar fecha con los métodos de la clase *Date*.

En Java existen una gran cantidad de clases que se puede utilizar para realizar las aplicaciones, pero existen clases que son fundamentales para crear las aplicaciones, las cuales se expone a continuación:

### 13.1 CLASE Object

Es la superclase de todas las demás clases de Java, y contiene métodos que se puede utilizar directamente o modificarlos, los cuales se menciona a continuación:

#### 13.1.1 MÉTODO equals

Este método permite hacer comparaciones entre objetos, indica si los objetos son del mismo tipo y tienen los mismos valores. Devuelva *true* si los dos objetos son

iguales y **false** si no lo son. Esta comparación es diferente a la utilizada con el operador “==” porque las comparaciones se las realiza entre datos primitivos.

En el siguiente ejemplo se realiza la comparación de dos cadenas de caracteres (*String*).

Ejemplo 129 método equals

```
1 public class CompararCadenas {  
2  
3     public static void main(String [] args){  
4         String a = "java";  
5         String b = "java";  
6         if(a.equals(b))  
7             System.out.print("Las cadenas son iguales ");  
8         else  
9             System.out.print("Las cadenas NO son iguales ");  
10    }  
11 }
```

En este ejemplo en la línea 6, se compara dos cadenas de texto a través del método “equals”, al ejecutar evalúa que ambos objetos sean iguales devolviendo un valor verdadero al ser iguales y falso cuando no coinciden. Su salida es:

```
1 \programas_libro> java CompararCadenas  
2 Las cadenas son iguales
```

Hay que recordar que Java es sensible con las cadenas mayúsculas y minúsculas, si a la variable “a” le modifica su valor de “java” a “Java”, al realizar la comparación, a y b siguen teniendo los mismos valores desde un punto de vista semántico, pero el valor de “a” es ligeramente diferente porque su primera letra está con mayúscula y “b” tiene toda la cadena con minúscula. Su código es el siguiente:

```
1 public class CompararCadenas {  
2  
3     public static void main(String [] args){  
4         String a = "Java";  
5         String b = "java";  
6         if(a.equals(b))  
7             System.out.print("Las cadenas son iguales ");  
8         else  
9             System.out.print("Las cadenas NO son iguales ");  
10    }  
11 }
```

Su salida es:

```
1 \programas_libro> java CompararCadenas  
2 Las cadenas NO son iguales
```

### **Recomendación**

En Java, el método “`equalsIgnoreCase`” permite comparar cadenas sin tener si la cadena está en mayúsculas o minúsculas, solo compara el valor semántico de las cadenas.

#### **13.1.2 MÉTODO `toString`**

Permite mostrar la representación de un objeto, permitiendo así comparar dos objetos con esa salida o mostrar información relevante al usuario, cualquier clase puede utilizar este método porque viene heredado por defecto de la clase `Object`. Para utilizar el “`toString`” hay que sobre escribirlo.

Implementar una clase llamada “`Celular`” para sobre escribir el método “`toString`” e imprimir toda la información que contienen sus atributos. Su código es el siguiente:

*Ejemplo 130 Método `toString`*

```
1 public class Celular {  
2     public String marca;  
3     public String modelo;  
4     public int anio;  
5     public double precio;  
6  
7     @Override  
8     public String toString(){  
9         return "Marca: " + marca + "\n" +  
10            "Modelo: " + modelo + "\n" +  
11            "Año: " + anio + "\n" +  
12            "Precio: " + precio;  
13     }  
14  
15    public static void main(String [] args){  
16        Celular celular = new Celular();  
17        celular.marca = "Samsung";  
18        celular.modelo = "J2";  
19        celular.anio = 2017;  
20        celular.precio = 150.8;  
21        System.out.println(celular.toString());  
22    }  
23 }
```

Su salida es:

```
1 \programas_libro> java Celular  
2 Marca: Samsung  
3 Modelo: J2  
4 Año: 2017  
5 Precio: 150.8
```

## 13.2 CLASES ENVOLVENTE

En Java se puede trabajar con datos primitivos o con sus clases envolventes. Una clase envolvente es la representación un dato primitivo a nivel de objetos, es decir, estas clases proporcionan métodos que permiten manipular al dato primitivo como si fuera objeto, también se lo conoce como *clases contenedores* o *wrappers*.

Cada dato primitivo en Java tiene su propia clase envolvente como se describe en la siguiente tabla:

Tabla 22 Datos primitivos y Clases envolventes

| Dato primitivo | Clase envolvente |
|----------------|------------------|
| int            | Integer          |
| short          | Short            |
| long           | Long             |
| double         | Double           |
| byte           | Byte             |
| boolean        | Boolean          |
| float          | Float            |
| char           | Character        |

### 13.2.1 CLASE Integer

La clase Integer representa al dato primitivo entero (`int`) en un objeto y a su vez proporcionan métodos que permiten la transformación de un *String* a `int`. Los atributos y métodos más importantes son:

Tabla 23 Atributos de la clase Integer

| Atributo               | Descripción                                                                                  |
|------------------------|----------------------------------------------------------------------------------------------|
| <code>MAX_VALUE</code> | El valor máximo que puede tener un entero (2147483647)                                       |
| <code>MIN_VALUE</code> | El valor mínimo que puede tener un entero (-2147483648)                                      |
| <code>SIZE</code>      | Numero de bits utilizados para representar un entero en forma binaria (32)                   |
| <code>TYPE</code>      | Devuelve el tipo de dato primitivo ( <code>int</code> )                                      |
| <code>BYTES</code>     | Numero de bits utilizados para representar un entero en forma binaria del complemento A2 (4) |

Tabla 24 Métodos principales de la clase Integer

| Métodos                             | Descripción                                                               |
|-------------------------------------|---------------------------------------------------------------------------|
| <code>Integer (int value)</code>    | Constructor de la clase.                                                  |
| <code>Integer (String value)</code> | Constructor de la clase que recibe un String.                             |
| <code>parseInt(String)</code>       | Método estático que permite transformas un String a un <code>int</code> . |

---

|                                |                                                                                   |
|--------------------------------|-----------------------------------------------------------------------------------|
| <code>value)</code>            |                                                                                   |
| <code>shortValue()</code>      | Método que devuelve el valor del objeto Integer como <code>short</code> .         |
| <code>valueOf(int i)</code>    | Método estático que transforma un entero a un objeto Integer.                     |
| <code>valueOf(String i)</code> | Método estático que transforma un String a un objeto Integer.                     |
| <code>intValue()</code>        | Método que permite obtener el valor del objeto Integer a <code>int</code> .       |
| <code>doubleValue()</code>     | Método que permite obtener el valor del objeto Integer como <code>double</code> . |
| <code>floatValue()</code>      | Método que permite obtener el valor del objeto Integer como <code>float</code> .  |
| <code>longValue()</code>       | Método que permite obtener el valor del objeto Integer como <code>long</code> .   |

---

### 13.2.2 CLASE Short

La clase Short representa al dato primitivo `short` en un objeto y a su vez proporciona métodos que permiten la transformación de un *String* a `short` así como obtener su valor. Los atributos y métodos más importantes son:

Tabla 25 Atributos de la clase Short

---

| Atributo               | Descripción                                                                                   |
|------------------------|-----------------------------------------------------------------------------------------------|
| <code>MAX_VALUE</code> | El valor máximo que puede tener un <code>short</code> (32767).                                |
| <code>MIN_VALUE</code> | El valor mínimo que puede tener un <code>short</code> (-32768).                               |
| <code>SIZE</code>      | Numero de bits utilizados para representar un entero en forma binaria (32).                   |
| <code>TYPE</code>      | Devuelve el tipo de dato primitivo ( <code>short</code> ).                                    |
| <code>BYTES</code>     | Numero de bits utilizados para representar un entero en forma binaria del complemento A2 (2). |

---

Tabla 26 Métodos de la clase Short

---

| Métodos                               | Descripción                                                                            |
|---------------------------------------|----------------------------------------------------------------------------------------|
| <code>Short (int value)</code>        | Constructor de la clase.                                                               |
| <code>Short (String value)</code>     | Constructor de la clase que recibe un String.                                          |
| <code>parseShort(String value)</code> | Método estático que permite transformar un String a un <code>short</code> .            |
| <code>shortValue()</code>             | Método que devuelve el valor del objeto Short como dato primitivo <code>short</code> . |
| <code>valueOf(short i)</code>         | Método estático que transforma un entero a un objeto Short.                            |
| <code>valueOf(String i)</code>        | Método estático que transforma un String a un objeto Short.                            |
| <code>intValue()</code>               | Método que permite obtener un dato primitivo entero.                                   |
| <code>doubleValue()</code>            | Método que permite obtener el valor del objeto Short como <code>double</code> .        |
| <code>floatValue()</code>             | Método que permite obtener el valor del objeto Short como <code>float</code> .         |
| <code>longValue()</code>              | Método que permite obtener el valor del objeto Short como <code>long</code> .          |

---

### 13.2.3 CLASE Long

La clase Long representa al dato primitivo **long** en un objeto y a su vez proporciona métodos que permiten la transformación de un *String* a **long** así como obtener su valor. Los atributos y métodos más importantes son:

Tabla 27 Atributos de la clase Long

| Atributo  | Descripción                                                                                   |
|-----------|-----------------------------------------------------------------------------------------------|
| MAX_VALUE | El valor máximo que puede tener un <b>long</b> (9223372036854775807).                         |
| MIN_VALUE | El valor mínimo que puede tener un <b>long</b> (-9223372036854775808).                        |
| SIZE      | Número de bits utilizados para representar un entero en forma binaria (64).                   |
| TYPE      | Devuelve el tipo de dato primitivo ( <b>short</b> ).                                          |
| BYTES     | Número de bits utilizados para representar un entero en forma binaria del complemento A2 (8). |

Tabla 28 Métodos de la clase Long

| Métodos                         | Descripción                                                                      |
|---------------------------------|----------------------------------------------------------------------------------|
| Long ( <b>int</b> value)        | Constructor de la clase.                                                         |
| Long ( <i>String</i> value)     | Constructor de la clase que recibe un <i>String</i> .                            |
| parseLong( <i>String</i> value) | Método estático que permite transformar un <i>String</i> a un dato <b>long</b> . |
| shortValue()                    | Método que devuelve el valor del objeto Long a un <b>short</b> .                 |
| valueOf( <b>long</b> i)         | Método estático que transforma un entero a un objeto Long.                       |
| valueOf( <i>String</i> i)       | Método estático que transforma un <i>String</i> a un objeto Long.                |
| intValue()                      | Método que permite obtener el valor del objeto Long a <b>int</b> .               |
| doubleValue()                   | Método que permite obtener el valor del objeto Long como <b>double</b> .         |
| floatValue()                    | Método que permite obtener el valor del objeto Long como <b>float</b> .          |
| longValue()                     | Método que permite obtener el valor del objeto Long como <b>long</b> .           |

### 13.2.4 CLASE Byte

La clase Byte representa al dato primitivo **byte** en un objeto y a su vez proporciona métodos que permiten la transformación de un *String* a byte así como obtener su valor. Los atributos y métodos más importantes son:

Tabla 29 Atributos clase Byte

| Atributo  | Descripción                                                   |
|-----------|---------------------------------------------------------------|
| MAX_VALUE | El valor máximo que puede tener un <b>byte</b> (127).         |
| MIN_VALUE | El valor mínimo que puede tener un <b>byte</b> (-128).        |
| SIZE      | Número de bits utilizados para representar un entero en forma |

---

|       |                                                                                               |
|-------|-----------------------------------------------------------------------------------------------|
|       | binaria (8).                                                                                  |
| TYPE  | Devuelve el tipo de dato primitivo ( <b>short</b> ).                                          |
| BYTES | Numero de bits utilizados para representar un entero en forma binaria del complemento A2 (1). |

---

Tabla 30 Métodos de la clase Byte

| Métodos                 | Descripción                                                               |
|-------------------------|---------------------------------------------------------------------------|
| Byte (int value)        | Constructor de la clase.                                                  |
| Byte (String value)     | Constructor de la clase que recibe un String.                             |
| parseByte(String value) | Método estático que permite transformar un String a un dato <b>byte</b> . |
| shortValue()            | Método que devuelve el valor del objeto Byte a un <b>short</b> .          |
| valueOf( <b>byte</b> i) | Método estático que transforma un byte a un objeto Byte.                  |
| valueOf(String i)       | Método estático que transforma un String a un objeto Byte.                |
| intValue()              | Método que permite obtener el valor del objeto Byte a <b>int</b> .        |
| doubleValue()           | Método que permite obtener el valor del objeto Byte como <b>double</b> .  |
| floatValue()            | Método que permite obtener el valor del objeto byte como <b>float</b> .   |
| longValue()             | Método que permite obtener el del objeto byte como <b>long</b> .          |

### 13.2.5 CLASE Double

La clase Double representa al dato primitivo **double** en un objeto y a su vez proporciona métodos que permiten la transformación de un *String* a **double** así como obtener su valor. Los atributos y métodos más importantes son:

Tabla 31 Atributos de la clase Double

| Atributo     | Descripción                                                                                   |
|--------------|-----------------------------------------------------------------------------------------------|
| MAX_VALUE    | El valor máximo que puede tener un <b>double</b> (1.7976931348623157E308).                    |
| MIN_VALUE    | El valor mínimo que puede tener un <b>double</b> (4.9E-324).                                  |
| SIZE         | Numero de bits utilizados para representar un entero en forma binaria (64).                   |
| TYPE         | Devuelve el tipo de dato primitivo ( <b>double</b> ).                                         |
| BYTES        | Numero de bits utilizados para representar un double en forma binaria del complemento A2 (8). |
| MIN_EXPONENT | Mínimo exponente que puede tener un <b>double</b> (-1022).                                    |
| MAX_EXPONENT | Máximo exponente que puede tener un <b>double</b> (1023).                                     |
| NAN          | Representa un valor que no es numérico.                                                       |

Tabla 32 Métodos de la clase Double

| Métodos                    | Descripción              |
|----------------------------|--------------------------|
| Double ( <b>int</b> value) | Constructor de la clase. |

|                                        |                                                                                   |
|----------------------------------------|-----------------------------------------------------------------------------------|
| <code>Double (String value)</code>     | Constructor de la clase que recibe un String.                                     |
| <code>parseDouble(String value)</code> | Método estático que permite transformar un String a un dato <code>double</code> . |
| <code>shortValue()</code>              | Método que devuelve el valor del objeto Double a un <code>short</code> .          |
| <code>valueOf(double i)</code>         | Método estático que transforma un byte a un objeto Double.                        |
| <code>valueOf(String i)</code>         | Método estático que transforma un String a un objeto Double.                      |
| <code>intValue()</code>                | Método que permite obtener el valor del objeto Double a <code>int</code> .        |
| <code>doubleValue()</code>             | Método que permite obtener el valor del objeto Double como <code>double</code> .  |
| <code>floatValue()</code>              | Método que permite obtener el valor del objeto Double como <code>float</code>     |
| <code>longValue()</code>               | Método que permite obtener el valor del objeto Double como <code>long</code>      |

### 13.2.6 CLASE Float

La clase `Float` representa al dato primitivo `float` en un objeto y a su vez proporciona métodos que permiten la transformación de un *String* a `float` así como obtener su valor. Los atributos y métodos más importantes son:

Tabla 33 Atributos de la clase `Float`

| Atributo                  | Descripción                                                                                               |
|---------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>MAX_VALUE</code>    | El valor máximo que puede tener un <code>float</code> (3.4028235E38).                                     |
| <code>MIN_VALUE</code>    | El valor mínimo que puede tener un <code>float</code> (1.4E-45).                                          |
| <code>SIZE</code>         | Número de bits utilizados para representar un entero en forma binaria (32).                               |
| <code>TYPE</code>         | Devuelve el tipo de dato primitivo ( <code>float</code> ).                                                |
| <code>BYTES</code>        | Número de bits utilizados para representar un <code>float</code> en forma binaria del complemento A2 (4). |
| <code>MIN_EXPONENT</code> | Mínimo exponente que puede tener un <code>float</code> (-126).                                            |
| <code>MAX_EXPONENT</code> | Máximo exponente que puede tener un <code>float</code> (127).                                             |
| <code>NAN</code>          | Representa un valor que no es numérico.                                                                   |

Tabla 34 Métodos de la clase `Float`

| Métodos                               | Descripción                                                                            |
|---------------------------------------|----------------------------------------------------------------------------------------|
| <code>Float (int value)</code>        | Constructor de la clase.                                                               |
| <code>Float (String value)</code>     | Constructor de la clase que recibe un String.                                          |
| <code>parseFloat(String value)</code> | Método estático que permite transformar un String a un dato <code>float</code> .       |
| <code>shortValue()</code>             | Método que devuelve el valor del objeto <code>Float</code> a un <code>short</code> .   |
| <code>valueOf(double i)</code>        | Método estático que transforma un <code>float</code> a un objeto <code>Float</code> .  |
| <code>valueOf(String i)</code>        | Método estático que transforma un String a un objeto <code>Float</code> .              |
| <code>intValue()</code>               | Método que permite obtener el valor del objeto <code>Float</code> a <code>int</code> . |
| <code>doubleValue()</code>            | Método que permite obtener el valor del objeto <code>Float</code> como                 |

---

|                           |                                                                                             |
|---------------------------|---------------------------------------------------------------------------------------------|
|                           | <code>double</code> .                                                                       |
| <code>floatValue()</code> | Método que permite obtener el valor del objeto <code>Float</code> como <code>float</code> . |
| <code>longValue()</code>  | Método que permite obtener el del objeto <code>Float</code> como <code>long</code> .        |

---

### 13.2.7 CLASE Boolean

La clase Boolean representa al dato primitivo `boolean` en un objeto y a su vez proporciona métodos que permiten la transformación de un *String* a `boolean` así como obtener su valor. Los atributos y métodos más importantes son:

Tabla 35 Atributos de la clase Boolean

---

| Atributo           | Descripción                                                   |
|--------------------|---------------------------------------------------------------|
| <code>TRUE</code>  | Devuelve un Object Boolean con valor <code>true</code> .      |
| <code>FALSE</code> | Devuelve un Object Boolean con valor <code>false</code> .     |
| <code>TYPE</code>  | Devuelve el tipo de dato primitivo ( <code>boolean</code> ) . |

---

Tabla 36 Métodos principales de la clase Boolean

---

| Métodos                                 | Descripción                                                                        |
|-----------------------------------------|------------------------------------------------------------------------------------|
| <code>Byte (boolean value)</code>       | Constructor de la clase.                                                           |
| <code>Byte (String value)</code>        | Constructor de la clase que recibe un String.                                      |
| <code>parseBoolean(String value)</code> | Método estático que permite transformar un String a un dato <code>boolean</code> . |
| <code>shortValue()</code>               | Método que devuelve el valor del objeto Byte a un <code>short</code> .             |
| <code>valueOf(boolean i)</code>         | Método estático que transforma un <code>byte</code> a un objeto Boolean.           |
| <code>valueOf(String i)</code>          | Método estático que transforma un String a un objeto Boolean.                      |

---

### 13.2.8 CLASE Character

La clase Character representa al dato primitivo `char` en un objeto y a su vez proporciona métodos que permiten la transformación de un *String* a `char`, obtener su valor, categoría, valor Unicode, entre otras funciones. Los atributos y métodos más importantes son:

Tabla 37 Atributos principales de la clase Character

---

| Atributo                     | Descripción                                                                              |
|------------------------------|------------------------------------------------------------------------------------------|
| <code>CONTROL</code>         | Devuelve un <code>byte</code> con valor unicode representado al Control (15).            |
| <code>LINE_SEPARATOR</code>  | Devuelve un <code>byte</code> con valor unicode representado al separador de línea (13). |
| <code>SPACE_SEPARATOR</code> | Devuelve un <code>byte</code> con valor unicode representado al espacio                  |

---

---

|           |                                                                             |
|-----------|-----------------------------------------------------------------------------|
|           | (12).                                                                       |
| MAX_VALUE | El valor máximo que puede tener un <code>char</code> (16).                  |
| MIN_VALUE | El valor mínimo que puede tener un <code>char</code> ( ).                   |
| SIZE      | Numero de bits utilizados para representar un entero en forma binaria (16). |
| TYPE      | Devuelve el tipo de dato primitivo ( <code>char</code> ).                   |

---

Tabla 38 Métodos principales de la clase Character

| Métodos                               | Descripción                                                                |
|---------------------------------------|----------------------------------------------------------------------------|
| Character ( <code>char</code> value)  | Constructor de la clase.                                                   |
| isDigit( <code>char</code> ch)        | Método estático que permite saber si un carácter es numérico.              |
| isLetter( <code>char</code> value)    | Método estático que permite saber si un carácter es alfabético.            |
| toUpperCase( <code>char</code> value) | Método que devuelve el valor del carácter en Mayúscula.                    |
| valueOf( <code>char</code> i)         | Método estático que transforma un <code>char</code> a un objeto Character. |
| toLowerCase( <code>char</code> value) | Método que devuelve el valor del carácter en Minúscula.                    |

---

### 13.2.9 CONVERSIONES ENTRE LOS TIPOS PRIMITIVOS Y SUS CLASES ENVOLVENTE

En este tipo de conversiones no se requiere hacer un *casting* ya que esto se lo lleva de forma automática. Para esto se hace uso del *Boxing* (transformar de un dato primitivo a un *Wrapper*) y *Unboxing* (transformar de un *Wrapper* a un dato primitivo). Ejemplo:

```
1 | Integer a = 123; //boxing
2 | int b = a; //unboxing
```

Analicemos el siguiente código:

Ejemplo 131 Demostración de los métodos de las clases envolventes

```
1 | public class ClaseEnvolventes {
2 |     public static void main(String[] args) {
3 |         //para transformar de String a int
4 |         int a = Integer.parseInt("4");
5 |         //para transformar de String a double
6 |         double b = Double.parseDouble("8.9");
7 |         //para transformar de String a boolean
8 |         boolean band = Boolean.parseBoolean("true");
9 |         System.out.println("entero : " + a);
10 |        System.out.println("double : " + b);
11 |        System.out.println("boolean : " + band);
12 |        //unboxing
13 |        Float f = 23f;
14 |        //boxing
15 |        float f1 = f;
16 |    }
```

En este ejemplo, se utilizan los métodos de las clases envolventes. En la línea 4, el método “parseInt(“4”)” transforma una cadena en un número siempre que la cadena a transformar tenga como valor semántico un número. De la misma forma, en la línea 6 se realizar la transformación de una cadena a un decimal a través del método “parseDouble(“8.9”)”. En la línea 8 se transforma una cadena a un dato booleano al aplicar el método “parseBoolean(“true”)” . Para finalizar, en la línea 15 se convierte un objeto envolvente en un tipo de dato primitivo (*boxing*).

Al ejecutar su código, genera la siguiente salida:

```

1 \programas_libro>java ClaseEnvolventes
2 entero : 4
3 double : 8.9
4 boolean : true
5 unboxing: 23.0
6 boxing: 23.0

```

### 13.3 CLASE Math

Es la clase matemática de java, permite hacer operaciones matemáticas tales como operaciones trigonométricas, de redondeo, de exponente, de raíz cuadrada entre otras, además contiene las constantes universales como PI o E.

El constructor de la clase Math es privado pero los métodos son estáticos, para llamarlos o invocarlos seguimos la siguiente sintaxis:

Math.funcion()

Se Describe algunos métodos importantes:

*Tabla 39 Métodos principales de la clase Scanner*

| Funciones                      | Descripción                                                                                                                                     |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| abs (int, double long o float) | Método estático que recibe como parámetro cualquier número int, double, float o long y permite retornar el valor absoluto del número ingresado. |
| acos (double ángulo)           | Método estático que permite calcular el arco coseno en                                                                                          |

---

|                                                                                                            |                                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                            | radianes Devuelve un valor <b>double</b> .                                                                                                                                      |
| asen ( <b>double</b> ángulo)                                                                               | Método estático que permite calcular el arco seno en radianes. Devuelve un valor <b>double</b> .                                                                                |
| atan ( <b>double</b> ángulo)                                                                               | Método estático que permite calcular el arco tangente en radianes. Devuelve un valor <b>double</b> .                                                                            |
| atan ( <b>double</b> ángulo)                                                                               | Método estático que permite calcular el arco tangente en radianes entre PI - PI. Devuelve un valor <b>double</b> .                                                              |
| ceil ( <b>double</b> valor)                                                                                | Método estático que devuelve el valor entero superior. Devuelve un <b>double</b> .                                                                                              |
| floor ( <b>double</b> valor)                                                                               | Método estático que devuelve el valor entero más inferior. Devuelve un <b>double</b> .                                                                                          |
| round ( <b>double</b> valor)                                                                               | Método estático que devuelve el valor entero redondeado Devuelve un <b>double</b> .                                                                                             |
| cos ( <b>double</b> ángulo)                                                                                | Método estático que permite calcular el coseno en radianes Devuelve un valor <b>double</b> .                                                                                    |
| sen ( <b>double</b> ángulo)                                                                                | Método estático que permite calcular el seno en radianes. Devuelve un valor <b>double</b> .                                                                                     |
| tan ( <b>double</b> ángulo)                                                                                | Método estático que permite calcular la tangente en radianes. Devuelve un valor <b>double</b> .                                                                                 |
| expl ( <b>double</b> valor)                                                                                | Método estático que devuelve el valor exponencial de un número. Devuelve un <b>double</b> .                                                                                     |
| log ( <b>double</b> valor)                                                                                 | Método estático que devuelve el valor logarítmico natural de un número. Devuelve un <b>double</b> .                                                                             |
| max ( <b>int, double, long</b> o<br><b>float</b> valor, <b>int, double, long</b><br>o <b>float</b> valor1) | Método estático que recibe dos valores como parámetro cualquier número <b>int, double, float</b> o <b>long</b> y permite retornar el valor máximo entre los números ingresados. |
| min ( <b>int, double, long</b> o<br><b>float</b> valor, <b>int, double, long</b><br>o <b>float</b> valor1) | Método estático que recibe dos valores como parámetro cualquier número <b>int, double, float</b> o <b>long</b> y permite retornar el valor mínimo entre los números ingresados. |
| random ()                                                                                                  | Método estático que devuelve un valor aleatorio entre 0 y 1. Devuelve un <b>double</b> .                                                                                        |
| sqrt ( <b>double</b> valor)                                                                                | Método estático que permite sacar la raíz cuadrada de un número. Devuelve un <b>double</b> .                                                                                    |
| pow ( <b>double</b> base, <b>double</b><br>exponente)                                                      | Método estático que devuelve el valor de un número elevado a una exponente X. Devuelve un <b>double</b> .                                                                       |

---

*Ejemplo 132 Demostración de los métodos de la clase Math*

```

1 public class DemostracionMath {
2     public static void main(String[] args) {
3         System.out.println("E :" + Math.E);
4         System.out.println("Abs: " + Math.abs(-9));
5         System.out.println("Acos: " + Math.acos(0.98));
6         System.out.println("Atan2: " + Math.atan2(0.9, 13));
7         System.out.println("Ceil: " + Math.ceil(8.34));
8         System.out.println("Floor: " + Math.floor(8.74));
9         System.out.println("Round: " + Math.round(8.34));
10        System.out.println("Round con decimales superior a 0.5: "
11                           + Math.round(8.74));

```

```

12 System.out.println("Exp: " + Math.exp(8));
13 System.out.println("Log: " + Math.log(10));
14 System.out.println("Max: " + Math.max(9, 6));
15 System.out.println("Min: " + Math.min(9, 6));
16 System.out.println("Sqrt: " + Math.sqrt(9.6));
17 System.out.println("Pow: " + Math.pow(2, 5));
18 }
19 }
```

Su salida es:

```

1 \programas_libro>java DemostracionMath
2 E:2.718281828459045
3 Abs: 9
4 Acos: 0.20033484232311968
5 Atan2: 0.06912048084718227
6 Ceil: 9.0
7 Floor: 8.0
8 Round: 8
9 Round con decimales superior a 0.5: 9
10 Exp: 2980.9579870417283
11 Log: 2.302585092994046
12 Max: 9
13 Min: 6
14 Sqrt: 3.0983866769659336
15 Pow: 32.0
```

## 13.4 CLASE Scanner

Esta clase permite leer entrada de datos de varias fuentes ya sean desde el teclado o desde datos almacenados en archivos, además esta clase permite leer Byte y convertirlos a valores legibles para su utilización. Para acceder a los datos desde teclado se debe utilizar la clase System.in. Además, *Scanner* puede analizar cadenas para extraer datos primitivos y cadenas utilizando expresiones regulares.

*Tabla 40 Métodos principales de la clase Scanner*

| Métodos                    | Descripción                                                                                                      |
|----------------------------|------------------------------------------------------------------------------------------------------------------|
| Scanner (InputStream data) | Constructor de la clase Scanner, instancia un nuevo objeto especificando en flujo de entrada. Ejemplo System.in. |
| Scanner (File data)        | Constructor de la clase Scanner, instancia un nuevo objeto especificando el flujo de entrada desde un archivo.   |
| Scanner (String cadena)    | Constructor de la clase Scanner, instancia un nuevo objeto con una cadena de caracteres.                         |
| hasNext()                  | Método que permite saber si existe otro token de entrada. Devuelve un valor booleano.                            |
| next()                     | Método que devuelve un byte escaneado del token de entrada.                                                      |
| nextByte()                 | Método que devuelve un byte escaneado del token de entrada.                                                      |
| nextInt()                  | Método que devuelve un int escaneado del token de                                                                |

---

|              |                                                                                  |
|--------------|----------------------------------------------------------------------------------|
| nextDouble() | entrada.<br>Método que devuelve un <b>double</b> escaneado del token de entrada. |
|--------------|----------------------------------------------------------------------------------|

---

Analizar el siguiente código:

Ejemplo 133 Demostración de métodos Scanner

```

1 import java.util.Scanner;
2
3 public class DemoencionScanner {
4
5     public static void main(String[] args) {
6         //para leer datos por teclado
7         Scanner scanner = new Scanner(System.in);
8         System.out.println("Ingrese una cadena");
9         String cadena = scanner.nextLine();
10        System.out.println("La cadena es: " + cadena);
11        System.out.println("Ingrese un numero entero");
12        int num = scanner.nextInt();
13        System.out.println("El numero ingresado es: " + num);
14        System.out.println("Ingrese un numero decimal");
15        double numD = scanner.nextDouble();
16        System.out.println("El numero ingresado es: " + numD);
17        scanner.close();
18        //para leer cadenas
19        String input = "5 bolitas 8 bolitas rojas";
20        Scanner s = new
21            Scanner(input).useDelimiter("\\s*bolitas\\s*");
22        System.out.println(s.nextInt());
23        System.out.println(s.nextInt());
24        System.out.println(s.nextInt());
25        s.close();
26    }
27 }
```

En este ejemplo, en la línea 7 se crea una instancia de la clase Scanner utilizando el teclado (System.in), además en las líneas 9, 12, y 15 se están utilizando los métodos para leer diferentes tipos de datos nextLine( ), nextInt( ), nextDouble( ). Para finalizar, en la línea 20 se crea una instancia de la clase Scanner con un *String*, y se llama al método useDelimiter( ) que recibe una expresión regular y devuelve un objeto de la clase Scanner. Si se lo ejecuta dará la siguiente salida:

```

1 \programas_libro>java DemoencionScanner
2 Ingrese una cadena
3 esta cobardía
4 La cadena es: esta cobardía
5 Ingrese un numero entero
6 9
7 El numero ingresado es: 9
8 Ingrese un numero decimal
```

```

9 9,8
10 El numero ingresado es: 9.8
11 5
12 8
13 Rojas

```

## 13.5CLASE Date

Esta clase se la utiliza para la manipulación de fechas y horas. Desde versiones anteriores muchos métodos han sido depreciados. Un método depreciado es un método obsoleto de uso no recomendado ya que puede desaparecer en futuras versiones. Se lista los métodos más importantes:

Tabla 41 Métodos principales de la clase Date

| Métodos                                                                                                          | Descripción                                                                                                                                                    |
|------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Date ()                                                                                                          | Constructor de la clase Date, instancia un nuevo objeto con la fecha actual por defecto.                                                                       |
| Date ( <i>int</i> year, <i>int</i> month, <i>int</i> day)                                                        | Constructor de la clase Date, instancia un nuevo objeto estableciendo la fecha y la por defecto en 00:00:00.                                                   |
| Date ( <i>int</i> year, <i>int</i> month, <i>int</i> day, <i>int</i> hour, <i>int</i> minute, <i>int</i> second) | Constructor de la clase Date, instancia un nuevo objeto con la fecha y hora exacta.                                                                            |
| Date ( <i>long</i> )                                                                                             | Constructor de la clase Date, instancia un nuevo objeto con la fecha y hora exacta a partir de los milisegundos establecidos desde el 31 de diciembre de 1969. |
| getTime()                                                                                                        | Método que devuelve los milisegundos transcurridos desde el 31 de diciembre de 1969.                                                                           |
| before(Date date)                                                                                                | Método que permite comparar si una fecha es anterior a la fecha especificada. Devuelve un <i>boolean</i> .                                                     |
| after(Date date)                                                                                                 | Método que permite comparar si una fecha es posterior a la fecha especificada. Devuelve un <i>boolean</i> .                                                    |
| getDate()                                                                                                        | Método que permite invertir el orden de los caracteres del StringBuilder, devuelve la misma instancia.                                                         |
| toDate()                                                                                                         | Método depreciado, permite devolver el número de día del mes.                                                                                                  |

### Recomendación

Si se desea manejar fechas para conversiones de fechas y campos se recomienda utilizar la clase *Calendar* y si desea formatear y parsear fechas utilice la clase *DateFormat*.

Analizar el siguiente código:

Ejemplo 134 Manejo de fechas

```
1 import java.util.Date;
2
3 public class Fecha {
4
5     public static void main(String[] args) {
6         Date fechaActual = new Date();
7         Date fechaString = new Date(0);
8         System.out.println("Fecha actual: " +
9             fechaActual.toString()); //mostrar la fecha
10        System.out.println("Fecha anterior: " +
11            fechaString.toString()); //mostrar la fecha
12        //demostracion del método after
13        System.out.println("Metodo after: " +
14            fechaActual.after(fechaString));
15        //demostracion del método before
16        System.out.println("Metodo before: " +
17            fechaActual.before(fechaString));
18        //demostracion del metodo getTime
19        System.out.println("Metodo getTime: " +
20            fechaActual.getTime());
21    }
22 }
```

Se puede apreciar en la línea 6 y 7 se crean instancias de la clase Date, el primero contiene la fecha actual, la segunda una fecha con valor 0 que significa que contendrá la fecha: 31 de diciembre de 1969. Se llama a los métodos after y before los mismos que permiten comparar fechas. Al ejecutar el código no imprime lo siguiente:

```
1 \programas_libro>java Fecha
2 Fecha actual: Sun Sep 10 18:03:21 COT 2017
3 Wed Dec 31 19:00:00 COT 1969
4 Metodo after: true
5 Metodo before: false
6 Metodo getTime: 1505084601780
```

## 13.6 EJEMPLOS

**Ejercicio 1:** Calcular la potencia de un número y su raíz cuadrada.

**Solución:**

Se diseña una clase llamada Matemáticas para implementar dos métodos: potencia y raizCuadrada:

```

1 public class Matematicas {
2
3     public double potencia (int base, int exponente){
4         return Math.pow(base, exponente);
5     }
6
7     public double raizCuadrada(int numero){
8         return Math.sqrt(numero);
9     }
10 }
```

El método potencia hace una llamada al método estático pow de la clase Math para realizar la operación de potencia. De la misma forma, el método raizCuadrada llama al método estático sqrt para obtener la raíz cuadrada.

*Ejemplo 135 Clase Ejecutara de la Clase Matemáticas*

```

1 import java.util.Scanner;
2
3 public class Ejecutor {
4
5     public static void main(String[] args) {
6         Scanner lector = new Scanner(System.in);
7         System.out.println("Ingrese la base");
8         int base = lector.nextInt();
9         System.out.println("Ingrese el exponente");
10        int exponente = lector.nextInt();
11        Matematicas m = new Matematicas();
12        System.out.println("La potencia es: " +
13                         m.potencia(base, exponente));
14        System.out.println("La raíz cuadrada es: " +
15                         m.raizCuadrada(base));
16    }
17 }
```

En la clase Ejecutor se realiza la lectura de datos desde teclado a través de la clase Scanner y se crea un objeto de la clase Matemáticas para utilizar sus métodos y realizar las operaciones de potencia y raíz cuadrada. Su salida es:

|   |                                |
|---|--------------------------------|
| 1 | \programas_libro>java Ejecutor |
| 2 | Ingrese la base:               |
| 3 | 4                              |
| 4 | Ingrese el exponente:          |
| 5 | 2                              |
| 6 | La potencia es: 16.0           |
| 7 | La raíz cuadrada es: 2.0       |

**Ejercicio 2:** Calcular el área de un círculo.

## Solución

Ejemplo 136 Clase Círculo

```
1 public class Circulo{  
2  
3     public double calcularArea(double radio){  
4         return Math.PI * Math.pow(radio, 2);  
5     }  
6 }
```

Ejemplo 137 Clase Ejecutora de Circunferencia

```
1 import java.util.Scanner;  
2  
3 public class Ejecutor {  
4  
5     public static void main(String[] args) {  
6         Scanner lector = new Scanner(System.in);  
7         System.out.println("Ingrese el radio de la  
8             circunferencia:");  
9         double radio = lector.nextDouble();  
10        Circulo c = new Circulo();  
11        System.out.println("El área de la circunferencia es: " +  
12                         c.calcularArea(radio));  
13    }  
14 }
```

Su salida es:

```
1 \programas_libro>java Ejecutor  
2 Ingrese el radio de la circunferencia:  
3 6,8  
4 El área de la circunferencia es: 145.267244301992
```

## 13.7 CUESTIONARIO

- 1. Para comparar cadenas, por ejemplo “cadena corta” y “cadena grande” se recomienda utilizar:**
  - Operador ==
  - Método equals
  - Método compareTo
  - Método instanceof
- 2. El método que compara cadenas sin importar que estén escritas en mayúsculas o minúsculas es:**
  - equals
  - equalsIgnoreCase
  - compareTo
  - instanceof
- 3. Para sobre escribir el método `toString` se utiliza la siguiente sintaxis:**
  - @Override
  - public void `toString()` {
  - // cadena

```

    }
    - @Override
    public void toString(String cadena) {
        // cadena
    }
    - @Override
    public String toString(String cadena) {
        return "";
    }
    - @Override
    public String toString() {
        return "";
    }
}

```

**4. El método para transformar cadenas a un entero (`int`) es:**

- `Integer.parseInt("45")`
- `Integer.parseInt('45')`
- `int.parseInt("45")`
- `int.parseInt('45')`
- `Integer.change("45")`
- `Integer.valueOf()`

**5. El método para transformar cadenas a un decimal (`double`) es:**

- `Double.valueOf("8.0")`
- `Double.doubleValue()`
- `Double.change("8.0")`
- `Double.parseDouble("8.0")`

**6. Es la representación un dato primitivo a nivel de objetos, es decir, estas clases proporcionan métodos que permiten manipular al dato primitivo como si fuera objeto se la denomina clase:**

- Envoltorio
- Abstracta
- Interfaz
- Object
- Math
- Scanner

**7. Analizar el siguiente código y seleccionar la respuesta correcta:**

```

1 | double numero = Double.parseDouble("2,9");
2 | System.out.println(numero);

```

- Se imprime el número 2,9.
- Error de ejecución.
- Error de compilación.
- El método `parseDouble` no puede transformar una cadena a un número.

**8. Para utilizar el método `pow` de la clase `Math` se sigue la sintaxis:**

- `Math.pow(6, 2);`
- `Math.pow("6", "2");`
- `objetoMath.pow(6, 2);`
- `objetoMath.pow("6", "2");`

## 14 MANEJO DE CADENAS

---

El presente capítulo se hace una introducción al manejo y manipulación de cadenas. Se crean objetos de tipo *String* de forma explícita e implícita. Además, se divide en grupos a los métodos de acuerdo a su funcionalidad. Para finalizar, se realiza un ejemplo por cada método de la clase *String* explicando su comportamiento.

### Objetivos:

- Establecer las diferencias al crear objeto *String* de forma explícita e implícita.
- Explicar el comportamiento de los métodos de la clase *String* en las cadenas.
- Realizar ejemplos de comparación, conversión de cadenas.

### 14.1 DEFINICIÓN

- Para el manejo y manipulación de cadenas, Java utiliza una clase especial llamada *String*. La clase *String* es la representación de una secuencia de caracteres (arreglos de caracteres), sus objetos se pueden crear de forma explícita o implícita.
- **Implícitamente:** solo basta con asignarle a una variable de tipo *String* una cadena de caracteres. Esta forma es la más utilizada.

```
1 | String cadena1 = "Casa";  
2 | String cadena2 = "789";  
3 | String cadena3 = "new";
```

- **Explícitamente:** cuando se crea un objeto directamente.

```
1 | String cadena1 = new String("Casa");  
2 | String cadena2 = new String("789");  
3 | String cadena3 = new String("new");
```

Las cadenas pueden unirse o concatenarse con el operador + ocupando una o varias líneas de código:

```

1 String cadena1 = "MENSAJE";
2 String cadena2 = "La información es relevante";
3 String cadena3 = "para todas las personas en el país";
4 String cadena4 = cadena1 + cadena2 + cadena3 + cadena4 +
5     "pero ha sido utilizada de forma incorrecta."

```

Los métodos más importantes de la clase String son:

Tabla 42 Métodos principales de la clase String

| Métodos                                   | Descripción                                                                                                                       |
|-------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <code>String ()</code>                    | Constructor de la clase String, instancia un nuevo objeto.                                                                        |
| <code>String (byte[])</code>              | Constructor de la clase String, instancia un nuevo objeto mediante un arreglo de bytes.                                           |
| <code>String (char[])</code>              | Constructor de la clase String, instancia un nuevo objeto mediante un arreglo de caracteres.                                      |
| <code>String (StringBuilder s)</code>     | Constructor de la clase String, instancia un nuevo objeto mediante un objeto de la clase StringBuilder.                           |
| <code>charAt(int pos)</code>              | Método que permite devolver un carácter de una cadena en una posición específica.                                                 |
| <code>compareTo(String value)</code>      | Método que permite comparar dos cadenas lexicográficamente iguales. Retorna 0 si es lexicográficamente iguales y -1 si no lo son. |
| <code>equals(char value)</code>           | Método que permite comparar dos cadenas, devuelve true si son iguales.                                                            |
| <code>equalsIgnoreCase(char value)</code> | Método que permite comparar dos cadenas sin importar si son mayúsculas o minúsculas, devuelve true si son iguales.                |
| <code>getBytes()</code>                   | Método que devuelve un arreglo de bytes de una cadena de caracteres.                                                              |
| <code>getBytes(Charset ch)</code>         | Método que devuelve un arreglo de bytes de una cadena de caracteres, según el mapeo se secuencias Unicode.                        |
| <code>contains(CharSequence ch)</code>    | Método que permite buscar en una cadena coincidencias con otra cadena. Devuelve true si existen coincidencias.                    |
| <code>isEmpty()</code>                    | Método que permite saber si una cadena está vacía. Devuelve true si la cadena está vacía.                                         |
| <code>length()</code>                     | Método que permite devolver el número total de caracteres.                                                                        |
| <code>split(String regex)</code>          | Método que permite devolver un arreglo de cadenas de caracteres de acuerdo a la expresión regular.                                |
| <code>toUpperCase()</code>                | Método que transforma una cadena de caracteres a mayúscula.                                                                       |
| <code>toLowerCase()</code>                | Método que transforma una cadena de caracteres a minúscula.                                                                       |
| <code>subString(int beginIndex)</code>    | Método que permite devolver una sub cadena de caracteres de acuerdo a la posición de la cadena.                                   |

## 14.2 COMPARACIÓN ENTRE CADENAS

En el capítulo de clases fundamentales se revisó la comparación entre objetos usando el método “equals” que se lo hereda directamente de la clase Object, pero se revisará las cuatro formas de comparar cadenas.

### 14.2.1 MÉTODO equals

Devuelve un valor verdadero si las cadenas son idénticas y falso si no lo son. Es necesario recordar que Java es sensible entre minúsculas y mayúsculas., si la variable cadena1 tiene como valor “Saludo” y cadena2 se le asigna “saludo”, ambas tienen un valor semántico idéntico pero Java la toma como elementos diferentes.

Su sintaxis es:

```
cadena1.equals(cadena2)
```

Analizará el primer caso:

*Ejemplo 138 Clase cadena - método equals, valor verdadero*

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String cadena1 = "uleam";  
5         String cadena2 = "uleam";  
6         String comparacion = (cadena1.equals(cadena2)) ? "cadenas  
7             iguales" : "cadenas diferentes";  
8         System.out.println(comparacion);  
9     }  
10 }
```

Se crean dos variables de tipo cadena, al aplicar el método equals, devuelve un valor booleano al comparar las cadenas, en este ejemplo, ambas cadenas son iguales. Su salida es:

```
1 \programas_libro> java Cadenas  
2 cadenas iguales
```

Ahora cambiar el valor de la variable cadena2 por “Uleam”.

*Ejemplo 139 Clase Cadena - método equals, valor falso*

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String cadena1 = "uleam";  
5         String cadena2 = "Uleam";  
6         String comparacion = (cadena1.equals(cadena2)) ? "cadenas
```

```
7     |         iguales" : "cadenas diferentes");
8     |     System.out.println(comparacion);
9     |
10    }
```

Al parecer ambas cadenas son iguales pero son diferentes porque el valor de la cadena2 empieza con mayúscula. Para los usuarios el valor semántico de ambas variables es el mismo, pero Java la considera distintas porque es susceptible con valores con mayúsculas o minúsculas. Su salida es:

```
1 \programas_libro> java Cadenas
2 cadenas diferentes
```

Otro dato a considerar, es que no se debe utilizar el operador de igualdad == para realizar la comparación entre cadenas.

#### 14.2.2 MÉTODO equalsIgnoreCase

Tiene la misma función y comportamiento que el método equals, la única diferencia es que al momento de comparar no tiene en cuenta si las cadenas están en mayúsculas o minúsculas.

Modificar el anterior ejemplo y aplicar este método en cadenas con valor semántico idéntico pero diferentes por las mayúsculas y minúsculas.

*Ejemplo 140 método equalsIgnoreCase*

```
1 public class Cadenas {
2
3     public static void main(String[] args) {
4         String cadena1 = "uleam";
5         String cadena2 = "UIEaM";
6         String comparacion = (cadena1.equalsIgnoreCase(cadena2))
7             ? "cadenas iguales" : "cadenas diferentes";
8         System.out.println(comparacion);
9     }
10 }
```

Su salida es:

```
1 \programas_libro> java Cadenas
2 cadenas iguales
```

#### 14.2.3 MÉTODO compareTo

Comparar dos cadenas, pero toma en cuenta el orden alfabético tomando el valor de la tabla ASCII. Si las cadenas son iguales devuelve un valor 0, si la primera

cadena es mayor devuelve un valor positivo (1). Si la segunda cadena es mayor devuelve un valor negativo (-1).

Analizar el siguiente código:

Ejemplo 141 Método compareTo

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String cadena1 = "españ";  
5         String cadena2 = "uleam";  
6         int comparacion = cadena1.compareTo(cadena2);  
7         System.out.println("+" + comparacion);  
8         if (comparacion > 0)  
9             System.out.println("cadena 1 mayor");  
10        else if (comparacion < 0)  
11            System.out.println("cadena 2 mayor");  
12        else  
13            System.out.println("cadenas iguales");  
14    }  
15 }
```

La cadena1 empieza con “e” y la cadena2 con “u”, a simple vista “e” es menor “u” y su resultado será negativo. La cadena2 es mayor con respecto a la cadena1. Su salida es:

```
1 \programas_libro> java Cadenas  
2 -16  
3 cadena 2 mayor
```

El método “compareToIgnoreCase” cumple con la misma función que “compareTo” con la diferencia que el primero ignora las mayúsculas y minúsculas.

## 14.3 INFORMACIÓN BÁSICAS DE CADENAS

### 14.3.1 MÉTODO length

Permite devolver el tamaño de una cadena, es decir, el número de caracteres que la componen. En este ejemplo, se obtiene el número de caracteres de la variable cadena:

Ejemplo 142 método length

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String cadena = "uleam";
```

```

5   System.out.println("tamaño de la cadena" +
6       cadena.length());
7   System.out.println("último elemento " +
8       cadena.charAt(cadena.length() - 1));
9 }
10 }
```

Su salida es:

```

1 \programas_libro> java Cadenas
2 tamaño de la cadena5
3 último elemento m
```

#### 14.3.2 MÉTODO charAt

Permite obtener un carácter de una cadena dependiendo de una posición. Por concepto, una cadena es una colección de caracteres y de forma implícita se convierte en un arreglo de caracteres. La primera posición de una cadena es 0 y la última es el número de caracteres menos uno.

Analizar el siguiente código:

*Ejemplo 143 Método charAt*

```

1 public class Cadenas {
2
3     public static void main(String[] args) {
4         String cadena = "uleam";
5         System.out.println("Primer caracter " +
6             cadena.charAt(0));
7         System.out.println("Segundo caracter " +
8             cadena.charAt(1));
9         System.out.println("Tercer caracter " +
10            cadena.charAt(2));
11         System.out.println("Cuarto caracter " +
12            cadena.charAt(3));
13         System.out.println("Quinto caracter " +
14            cadena.charAt(4));
15     }
16 }
```

En este ejemplo se define una cadena con un valor “uleam”, la cadena se compone de 5 caracteres y para acceder a cada uno de ellos es a través de su posición. En la siguiente figura se ilustra las partes que componen una cadena:

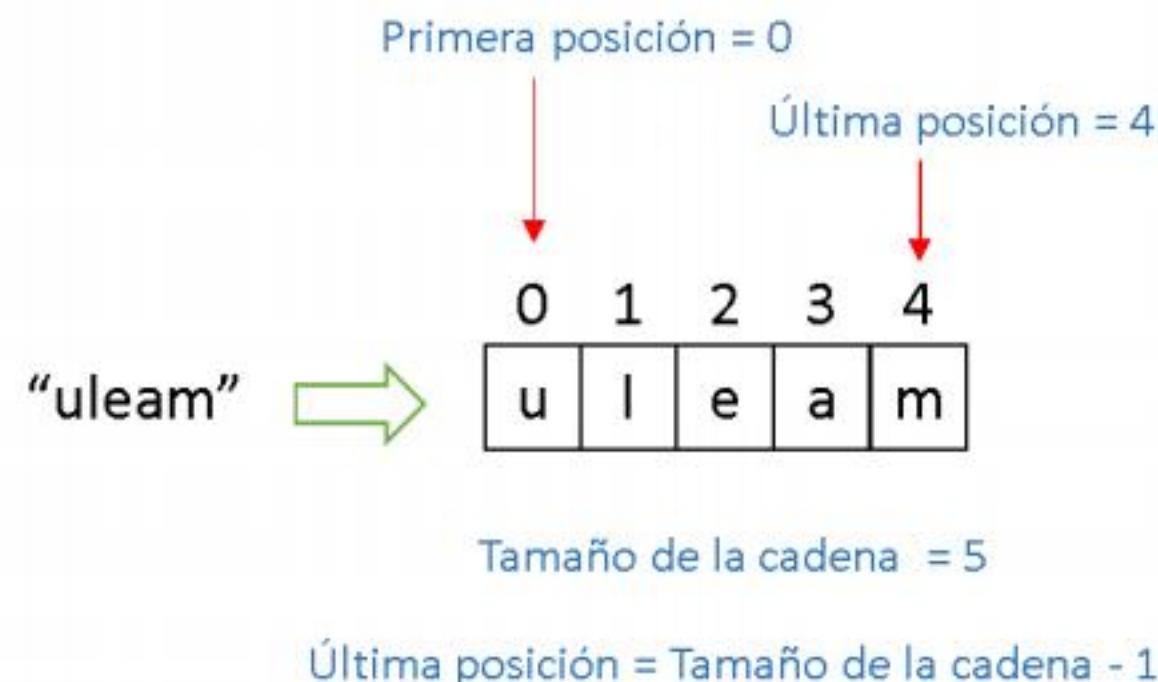


Figura 69 Estructura de una cadena

Su salida es:

```

1 \programas_libro> java Cadenas
2 Primer carácter u
3 Segundo carácter l
4 Tercer carácter e
5 Cuarto carácter a
6 Quinto carácter m

```

Hay que tener en cuenta la última posición para no cometer errores. Suponer que se quiere obtener un carácter que esté fuera del rango de posiciones de la cadena. En el ejemplo se obtiene un carácter en la posición 5, el error que se genera es de ejecución:

Ejemplo 144 método charAt - error común

```

1 public class Cadenas {
2
3     public static void main(String[] args) {
4         String cadena = "uleam";
5         System.out.println("carácter " + cadena.charAt(5));
6     }
7 }

```

Su salida es:

```

1 \programas_libro> java Cadenas
2 Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of
3 range: 5
4         at java.lang.String.charAt(String.java:658)
5         at proyectoLibro.ProyectoLibroJava.main(ProyectoLibroJava.java:7)
6

```

## 14.4 MANEJO DE CADENAS

### 14.4.1 MÉTODO concat

Tiene la misma funcionalidad que el operador +. Permite unir cadenas. En el siguiente ejemplo se realiza esta operación de concatenación entre dos cadenas.

Ejemplo 145 método concat

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String cadena1 = "Universidad Laica";  
5         String cadena2 = "Eloy Alfaro de Manabí";  
6         // cadena1 + cadena2  
7         String cadena3 = cadena1.concat(cadena2);  
8         System.out.println(cadena3);  
9         System.out.println("tamaño de la cadena " +  
10            cadena3.length());  
11         System.out.println("último elemento " +  
12            cadena3.charAt(cadena3.length() - 1));  
13     }  
14 }
```

Su salida es:

```
1 \programas_libro> java Cadenas  
2 Universidad LaicaEloy Alfaro de Manabí  
3 tamaño de la cadena 38  
4 último elemento í
```

### 14.4.2 MÉTODO replace

Permite reemplazar un carácter por otro. Para que la cadena almacene los nuevos cambios, se debe asignar la cadena reemplazada.

```
cadena = cadena.replace(char antiguo, char nuevo)
```

Ejemplo 146 método replace

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String mensaje = "Universidad Laica Eloy Alfaro de  
5             Manabí";  
6         System.out.println("cadena : " + mensaje);  
7         mensaje = mensaje.replace('a', '@');  
8         System.out.println("cadena nueva : " + mensaje);  
9     }  
10 }
```

Su salida es:

```
1 \programas_libro> java Cadenas
2 cadena : Universidad Laica Eloy Alfaro de Manabí
3 cadena nueva : Universid@d L@ic@ Eloy Alf@ro de M@n@bí
```

#### 14.4.3 MÉTODO trim

Elimina los espacios al inicio y final de la cadena. Este método devuelve un *String*.

Analizar el siguiente código:

Ejemplo 147 Método trim

```
1 public class Cadenas {
2
3     public static void main(String[] args) {
4         String mensaje = "    Universidad Laica Eloy Alfaro de
5             Manabí    ";
6         System.out.println("Mensaje original: " + mensaje);
7         // aplica el método trim
8         String mensajeSinEspacios = mensaje.trim();
9         System.out.println("Mensaje sin espacios al inicio y
10            final: " + mensajeSinEspacios);
11     }
12 }
```

En la línea 4, se define una variable “mensaje” y se le asigna la cadena:

“ Universidad Laica Eloy Alfaro de Manabí ”

La cadena tiene espacios al inicio y fin, el método “trim” limpia esos espacios dejando solo los caracteres alfabéticos.

Su salida es:

```
1 \programas_libro> java Cadenas
2 Mensaje original:    Universidad Laica Eloy Alfaro de Manabí
3 Mensaje sin espacios al inicio y final: Universidad Laica Eloy Alfaro de Manabí
4
```

#### 14.4.4 MÉTODO split

Permite convertir una cadena en un arreglo a partir de un carácter. Es decir, se necesita que exista un carácter que sirva como separador entre los elementos del arreglo.

Es necesario recordar que un arreglo es una estructura de datos lineal estática que permite almacenar datos del mismo tipo. Para definirlo se utiliza [ ].

*Ejemplo 148 método split*

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String mensaje = "Universidad Laica Eloy Alfaro de  
5             Manabí";  
6         String [] arregloCaracteres = mensaje.split(" ");  
7         for (int i = 0; i < arregloCaracteres.length; i++) {  
8             System.out.println("arreglo [" + i + "] = " +  
9                     arregloCaracteres [i]);  
10        }  
11    }  
12 }
```

En la línea 6, se declara un arreglo de *String* y su identificador es “arregloCaracteres”:

String [] arregloCaracteres

Se utiliza el método “split” para convertir una cadena en una arreglo, el separador de elementos es el espacio “ ”. Al aplicar “split” se genera 6 elementos en el arreglo.

Su salida es:

```
1 \programas_libro> java Cadenas  
2 arreglo [0] = Universidad  
3 arreglo [1] = Laica  
4 arreglo [2] = Eloy  
5 arreglo [3] = Alfaro  
6 arreglo [4] = de  
7 arreglo [5] = Manabí
```

#### 14.4.5 MÉTODOS **toUpperCase** y **toLowerCase**

Son métodos que convierten una cadena en mayúsculas o minúsculas.

Analizar el siguiente ejemplo:

*Ejemplo 149 método toUpperCase y toLowerCase*

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         String mensaje = "Universidad Laica Eloy Alfaro de  
5             Manabí";  
6         // cadena en mayúsculas  
7         String mayusculas = mensaje.toUpperCase();  
8         System.out.println("cadena mayúsculas: " + mayusculas);  
9         // cadena en minúsculas  
10        String minusculas = mensaje.toLowerCase();  
11        System.out.println("cadena minúscula: " + minusculas);  
12    }  
13 }
```

Su salida es

```
1 \programas_libro> java Cadenas
2 cadena mayúsculas: UNIVERSIDAD LAICA ELOY ALFARO DE MANABÍ
3 cadena minúscula: universidad laica eloy alfaro de manabí
```

## 14.5 MÉTODO CON SUBCADENAS

El método “subString” permite devolver un pedazo de una cadena dependiendo de la posición inicial y final. A partir de este método se puede obtener una porción de una cadena respecto de otra.

Analizar el siguiente método:

Ejemplo 150 método subString

```
1 public class Cadenas {
2
3     public static void main(String[] args) {
4         String mensaje = "Universidad Laica Eloy Alfaro de
5             Manabí";
6         String subcadena1 = mensaje.substring(0,11);
7         System.out.println("cadena 1: " + subcadena1 + ", "
8             "tamaño: " + subcadena1.length());
9         String subcadena2 = mensaje.substring(12, 17);
10        System.out.println("cadena 2: " + subcadena2 + ", "
11            "tamaño: " + subcadena2.length());
12        String subcadena3 = mensaje.substring(18, 22);
13        System.out.println("cadena 3: " + subcadena3 + ", "
14            "tamaño: " + subcadena3.length());
15        String subcadena4 = mensaje.substring(23);
16        System.out.println("cadena 4: " + subcadena4 + ", "
17            "tamaño: " + subcadena4.length());
18    }
19 }
```

El método subString tiene dos variantes, la primera es que se puede obtener una subcadena desde una posición de inicio y una final (línea 6, 9, 12). La segunda solo con la posición de inicio, esto supone que la posición final será hasta el último valor de la cadena (línea 15). Al ejecutar se imprime:

```
1 \programas_libro> java Cadenas
2 cadena 1: Universidad, tamaño: 11
3 cadena 2: Laica, tamaño: 5
4 cadena 3: Eloy, tamaño: 4
5 cadena 4: Alfaro de Manabí, tamaño: 16
```

## 14.6 CONVERSIÓN A CADENAS

El método estático “valueOf” permite convertir valores que no son cadena en cadena. Analizar el siguiente ejemplo:

Ejemplo 151 método valueOf

```
1 public class Cadenas {  
2  
3     public static void main(String[] args) {  
4         Date fecha = new Date();  
5         System.out.println("fecha: " + String.valueOf(fecha));  
6         System.out.println("números: " + String.valueOf(9087));  
7         System.out.println("decimales: "  
8             + String.valueOf(9.987));  
9         System.out.println("decimales: "  
10            + String.valueOf(false));  
11    }  
12 }
```

Con el método “valueOf” se puede transformar un tipo de datos a cadena, por ejemplo las fechas, números enteros, decimal, datos booleanos, etc. Es útil para imprimir el contenido de un objeto.

Su salida es:

```
1 \programas_libro> java Cadenas  
2 fecha: Tue Sep 12 08:50:54 COT 2017  
3 números: 9087  
4 decimales: 9.987  
5 decimales: false
```

Además se pueden utilizar los siguientes métodos para transformar diferente tipos de datos en cadenas:

- String valueOf(boolean b);
- String valueOf(int i);
- String valueOf(long l);
- String valueOf(float f);
- String valueOf(double d);
- String valueOf(Object obj);

## 14.7 CLASE StringBuilder

Los *String* son inmutables esto significa que cualquier operación en un *String* no altera el contenido del objeto sino que es un nuevo valor transformado. Las concatenaciones de *String* se las realiza con el operador “+” y es cuadrática.

Para ello, se creó la clase `StringBuilder` y similar al `String` pero se puede fijar su tamaño, y su contenido puede modificarse, además un `StringBuilder` está indexado, esto ayuda que el rendimiento del manejo de cadenas de caracteres sea más eficiente.

Sus métodos principales son:

Tabla 43 Métodos principales de la clase `StringBuilder`

| Métodos                                   | Descripción                                                                                                                    |
|-------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| <code>StringBuilder()</code>              | Constructor de la clase <code>StringBuilder</code> , instancia un nuevo objeto con una capacidad de 16 caracteres por defecto. |
| <code>StringBuilder(int size)</code>      | Constructor de la clase <code>StringBuilder</code> , instancia un nuevo objeto con una capacidad fijada por defecto.           |
| <code>StringBuilder(String cadena)</code> | Constructor de la clase <code>StringBuilder</code> , instancia un nuevo objeto con una cadena de caracteres.                   |
| <code>append(String cadena)</code>        | Método que Permite añadir cadenas de caracteres y retorna la misma instancia.                                                  |
| <code>capacity()</code>                   | Método que devuelve la capacidad del <code>StringBuilder</code> .                                                              |
| <code>length()</code>                     | Método que devuelve el número de caracteres del <code>StringBuilder</code> .                                                   |
| <code>reverse()</code>                    | Método que permite invertir el orden de los caracteres del <code>StringBuilder</code> , devuelve la misma instancia.           |
| <code>toString()</code>                   | Método que permite convertir la instancia a un <code>String</code> .                                                           |

Analizar el siguiente código:

Ejemplo 152 Demostración de los métodos de la clase `StringBuilder`

```
1 public class DemostracionStringBuilder {  
2     public static void main(String[] args) {  
3         StringBuilder cadena = new StringBuilder();  
4         cadena.append("la ").append("casa ").append("es  
5             ").append("mía");  
6         System.out.println(cadena.toString());  
7     }  
8 }
```

En la línea 4, se añade varias cadenas utilizando el mismo objeto con el método “append”, evitando hacer tantas concatenaciones. Si se lo ejecuta dará la siguiente salida:

Su salida es:

```
1 \programas_libro> java DemostracionStringBuilder  
2 la casa es mía
```

## 14.8 EJEMPLOS

**Ejercicio 1:** Crear un programa que permita obtener cada carácter que componen una cadena de texto.

**Solución:** Para obtener cada carácter de la cadena se utiliza el método charAt.

```
1 public class Ejecutor {  
2     public static void main(String[] args) {  
3         String mensaje = "Lenguaje de programación";  
4         for (int i = 0; i < mensaje.length(); i++) {  
5             System.out.println("posición " + i + " " +  
6                     mensaje.charAt(i));  
7         }  
8     }  
9 }
```

Su salida es:

```
1 \programas_libro> java Ejecutor  
2 posición 0: L                      posición 12: p  
3 posición 1: e                      posición 13: r  
4 posición 2: n                      posición 14: o  
5 posición 3: g                      posición 15: g  
6 posición 4: u                      posición 16: r  
7 posición 5: a                      posición 17: a  
8 posición 6: j                      posición 18: m  
9 posición 7: e                      posición 19: a  
10 posición 8:                      posición 20: c  
11 posición 9: d                      posición 21: i  
12 posición 10: e                     posición 22: ó  
13 posición 11:                      posición 23: n
```

**Ejercicio 2:** Convertir la primera letra de cada palabra en mayúscula de la cadena “ecuador es un país de paz”

```
1 public class Ejecutor {  
2     public static void main(String[] args) {  
3         String cadena = "Ecuador es un país de paz";  
4         String[] arreglo = cadena.split(" ");  
5         for (int i = 0; i < arreglo.length; i++) {  
6             String palabra = arreglo[i];  
7             arreglo[i] =  
8                 palabra.valueOf(palabra.charAt(0)).toUpperCase().concat(palabra.substring(1, palabra.length()));  
9             System.out.print(arreglo[i] + " ");  
10        }  
11    }  
12 }  
13 }
```

Su salida es:

```
1 \programas_libro> java Ejecutor  
2 Ecuador Es Un País De Paz
```

## 14.9 CUESTIONARIO

**1. El método que permite convertir de una cadena a un arreglo:**

- |           |               |
|-----------|---------------|
| - replace | - charAt      |
| - trim    | - toUpperCase |
| - split   | - equals      |

**2. El método que permite eliminar los espacios en al inicio y final de la cadena es:**

- |           |               |
|-----------|---------------|
| - replace | - charAt      |
| - trim    | - toUpperCase |
| - split   | - equals      |

**3. El método que permite comparar cadenas sin considerar si son mayúsculas y minúsculas es:**

- |                    |               |
|--------------------|---------------|
| - equalsIgnoreCase | - charAt      |
| - trim             | - toUpperCase |
| - split            | - equals      |

**4. El método que permite unir cadena es:**

- |           |               |
|-----------|---------------|
| - replace | - charAt      |
| - trim    | - toUpperCase |
| - split   | - concat      |

**5. El método que permite devolver el número de caracteres que componen la cadena es:**

- |           |               |
|-----------|---------------|
| - replace | - charAt      |
| - trim    | - toUpperCase |
| - length  | - equals      |

**6. El método que permite obtener un carácter dependiendo de una posición es:**

- |           |               |
|-----------|---------------|
| - replace | - charAt      |
| - trim    | - toUpperCase |
| - split   | - equals      |

## **15 ARREGLOS**

---

El presente capítulo hace una introducción a las estructuras de datos líneas estáticas a través de los arreglos y matrices. Se define las características de cada uno, su sintaxis y se explica las operaciones básicas de definición, creación, escritura y lectura. Para finalizar se realiza ejemplos para explicar su funcionalidad.

### **Objetivos:**

- Explicar la creación de un arreglo y matriz de forma explícita e implícita.
- Definir el concepto de índice y tamaño de un arreglo y matriz.
- Explicar las operaciones básicas entre arreglos y matrices.

### **15.1 DEFINICIÓN**

Ante de definir un arreglo, es necesario pensar dónde se puede utilizar esta estructura de datos. Por ejemplo: en la Universidad un docente imparte clases a 53 estudiantes, cada uno tiene un registro de notas de sus aportes, lecciones, etc. Si analizamos el contexto, al decir: “un docente imparte clases a 53 estudiantes”, estamos refiriéndonos que existen más de un estudiante y el sistema deberá tener esa cantidad de atributos o variables en alguna clase Java. Qué pasaría si el número de estudiantes fuera mayor, por ejemplo 1000. No sería práctico tener esa cantidad de atributos en la clase. Por ellos hay que agrupar a todos estos objetos estudiantes en una estructura lineal estática llamada arreglo.

Un arreglo es una estructura de datos que contiene una colección finita de datos del mismo tipo y de tamaño fijo. Ejemplo: un arreglo de fechas, de números enteros, de estudiantes, etc. Se clasifican en unidimensionales (vectores) y multidimensionales (matrices).

Los arreglos tienen las siguientes características:

- Se los utiliza como contenedores para datos del mismo tipo.
- Una vez creados su tamaño no cambia, es decir su longitud es estática.
- Se accede a los datos del arreglo por posiciones (índices).
- Un arreglo se crea con la palabra reservada **new** aunque se los puede inicializar con valores por defecto.

## 15.2 ARREGLOS UNIDIMENSIONALES

Los arreglos unidimensionales o vectores tienen una sola dimensión. Es decir, solo se define el número de columnas, también se lo conoce como vector.

Está conformado por dos partes básicas:

- **Índices:** es la posición de un determinado elemento.
- **Elementos:** son los datos almacenados en una posición.

La representación gráfica de un arreglo es:

| índices   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|-----------|---|---|---|---|---|---|---|
| elementos | 4 | 5 | 1 | 3 | 7 | 9 | 0 |

Figura 70 Representación de un arreglo

En Java, los arreglos son considerados como objetos cuyo tamaño se lo considera en tiempo de compilación y no puede ser modificado en tiempo de ejecución.

Pueden ser de dos tipos:

- **Tipos de datos primitivos:** **int**, **char**, **boolean**, **double**, etc.
- **Objetos de clases:** clases del API de Java (*Scanner*, *Date*, etc.), definidos por el programador (Person, Guitarra, etc.).

Para establecer un arreglo, se utiliza el operador [ ].

## 15.2.1 DECLARACIÓN Y CREACIÓN

### DECLARACIÓN

La declaración se define como cualquier atributo o variable, su sintaxis es:

```
tipo_dato [ ] identificador_arreglo;  
o  
tipo_dato identificador_arreglo [ ];
```

Ejemplos:

```
1 String [] díasSemana;  
2 int [] edadesEstudiantes;  
3 double [] notas;  
4 char [] vocales;  
5 boolean [] productosDisponibles;
```

### CREACIÓN

Se requiere de la palabra reservada **new** y fijar el tamaño del arreglo dentro de los corchetes **[ ]**. Su sintaxis es:

```
identificador_arreglo = new tipo_datos [ tamaño]
```

Ejemplos:

```
1 díasSemana = new String [7];  
2 edadesEstudiantes = new int [90];  
3 notas = new double [8];  
4 vocales = new char [5];  
5 productosDisponibles = new boolean [78];
```

La definición y creación de un arreglo se puede realizar en una sola sentencia:

```
1 String [] díasSemana = new String [7];  
2 int [] edadesEstudiantes = new int [90];  
3 double [] notas = new double [8];  
4 char [] vocales = new char [5];  
5 boolean [] productosDisponibles = new boolean [78];
```

La siguiente figura detalla las partes de un arreglo:



Figura 71 Partes de un arreglo

Otra forma para crear un arreglo, es crearlos con sus valores ya definidos o valores iniciales. Su sintaxis es:

```
tipo_dato [] identificador_arreglo = { valor1, valor2, valor3, valor4, ... }
```

Ejemplos:

```

1 | String [] díasSemana = {"lunes", "martes", "miércoles"};
2 | int [] edadesEstudiantes = {98, 77, 5, 76};
3 | double [] notas = {9.8, 7.6, 5.5, 8.0, 7.7};
4 | char [] vocales = {'a', 'b', 'c', 'd', 'e'};
5 | boolean [] productosDisponibles = {true, false};

```

En Java, si un arreglo es creado con datos primitivos siempre se inicializaran con su valor por defecto, ejemplo si un vector es de enteros sus valores por defecto serán cero.

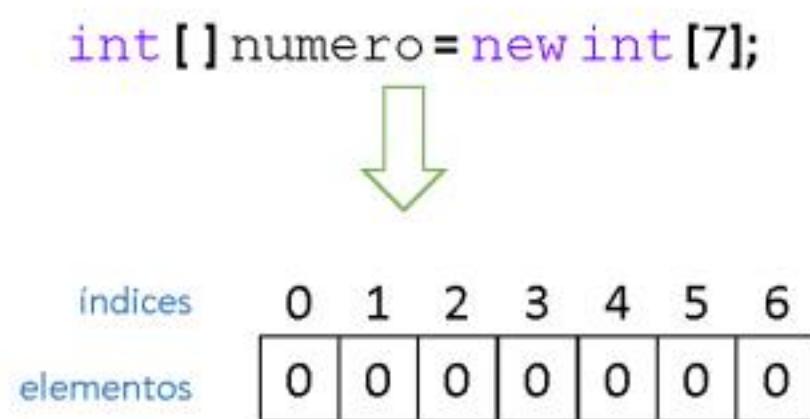


Figura 72 Valores por defecto en un arreglo de enteros

En cambio, sí un arreglo es de objetos se inicializaran con su valor null por defecto, ejemplo un arreglo de String

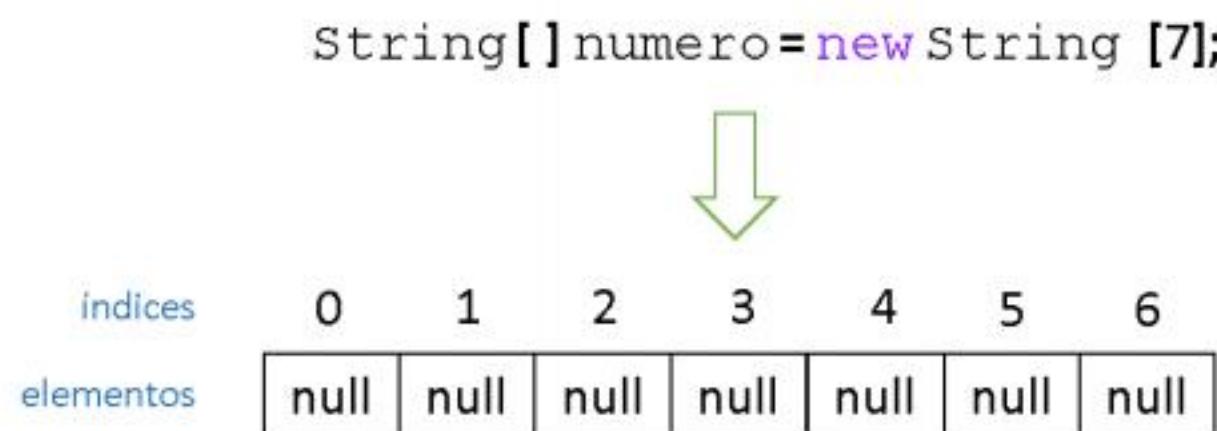


Figura 73 Valores por defecto en un arreglo de String

En las operaciones de definir y crear arreglos se pueden cometer errores de compilación. Es necesario conocerlos para evitar cometerlos:

1. Crear arreglos estáticos en tiempo de compilación. Ejemplo:

1 | int [4]edades;

2. Asignar valor a un arreglo sin haber determinado su tamaño. Ejemplo:

1 | int [] edades;  
2 | edades [0]=15;

### 15.2.2 LECTURA Y ESCRITURA

#### ESCRITURA

Una vez definido y creado el arreglo, se debe escribir o llenar de elementos al arreglo.

Para llenar un arreglo se lo realiza indicando su posición y el dato a ingresar.

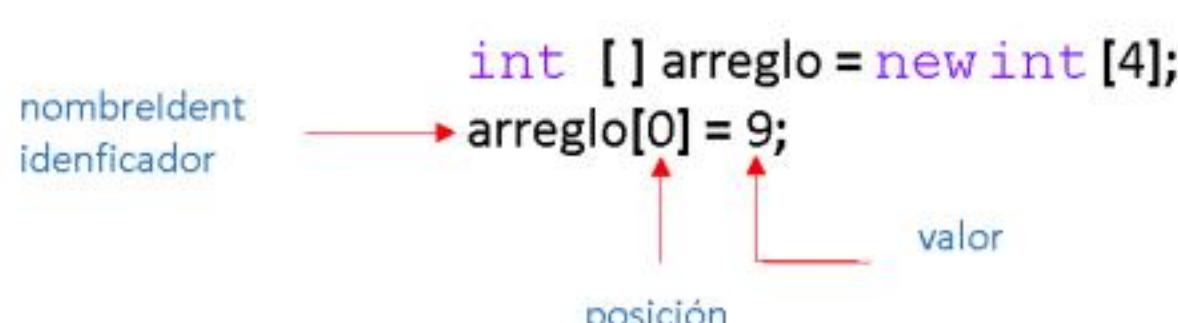


Figura 74 Escritura de un arreglo

Ejemplo:

1 | int arreglo [] = new int [4];
2 | arreglo[0] = 9;
3 | arreglo[1] = 7;
4 | arreglo[2] = 6;
5 | arreglo[3] = 0;

Al realizar esta operación se puede cometer errores de ejecución, por ejemplo, si se accede a un elemento a través de una posición fuera del rango, al ejecutar el programa se genera una excepción de tipo `ArrayIndexOutOfBoundsException`.

## LECTURA O RECORRIDO

Para acceder al contenido o el valor del arreglo se debe especificar la posición entre corchetes. La lectura de datos se lo realiza mediante una estructura repetitiva **for** o cualquier otra estructura repetitiva, accediendo al elemento mediante su índice.

Cabe destacar que para recorrer un arreglo se requiere saber la longitud del arreglo esto se lo hace utilizando con la propiedad `length`.

Analizar el siguiente ejemplo:

*Ejemplo 153 Lectura y recorrido de un arreglo*

```
1 public class Arreglos {  
2  
3     public static void main(String[] args) {  
4         int arreglo [] = {9, 7, 7, 0, 55, 98};  
5         for (int i = 0; i < arreglo.length; i++) {  
6             System.out.println(arreglo[i]);  
7         }  
8     }  
9 }
```

En el ciclo **for** permite recorrer a los elementos de una arreglo a través de los índices, cada iteración hace que incremente en uno el valor de la variable controladora “i” permitiendo que se acceda a los elementos del arreglo.

Su salida es:

```
1 \programas_libro> java Arreglos  
2 9  
3 7  
4 7  
5 0  
6 55  
7 98
```

### 15.2.3 EJEMPLOS CON ARREGLOS

**Ejercicio 1:** Iniciando, llenado y recorrido de un vector.

*Ejemplo 154 llenado y recorrido de un vector*

```
1 public class VectoresInicial {  
2     public static void main(String[] args) {  
3         int a [] = new int[8];//declaramos  
4         //lo llenamos con numeros aleatorios
```

```

5   for(int i = 0; i < a.length; i++) {
6       a[i] = (int)(Math.random() * 100);
7   }
8   //lo recorremos
9   System.out.println("");
10  for(int i = 0; i < a.length; i++) {
11      System.out.print(a[i] + "\t");
12  }
13  System.out.println("");
14 }
15 }
```

En la línea 3 se crea el arreglo de tamaño 8, luego en las líneas 5, 6, 7 se procede a llenar el arreglo y finalmente en la línea 10, 11, 12 se recorre a través de la estructura **for** e imprime los elementos del arreglo.

Al ejecutas, se imprime la siguiente salida:

|   |                                 |
|---|---------------------------------|
| 1 | \programas_libro> java Arreglos |
| 2 | 92 7 13 28 42 10 13 58          |

**Ejercicio 2:** Calcular la media de un conjunto de datos.

Los arreglos se los utiliza para cálculos estadísticos como es el caso de la media aritmética.

*Ejemplo 155 Media aritmética de un vector*

```

1 public class VectorMedia {
2
3     public void imprimir(int[] a) {
4         System.out.println("");
5         for(int i = 0; i < a.length; i++) {
6             System.out.print(a[i] + "\t");
7         }
8         System.out.println("");
9     }
10
11    public static void main(String[] args) {
12        //lo creamos solo para llamar al metodo imprimir
13        VectorMedia vectorMedia = new VectorMedia();
14        int a[] = new int[8];//declaramos
15        //lo llenamos con numeros aleatorios
16        for(int i = 0; i < a.length; i++) {
17            a[i] = (int)(Math.random() * 100);
18        }
19        //lo imprimimos
20        vectorMedia.imprimir(a);
21        double media = 0;
22        for(int i = 0; i < a.length; i++) {
23            media += a[i];
24        }
25        media = media / a.length;
26        System.out.println("La media del vector es: "+media);
27    }
28 }
```

```
27 | }  
28 }
```

En este ejemplo, en las líneas 3 al 8 se codifica un método que permite imprimir los datos del vector. En la línea 14, se crea un arreglo de enteros de tamaño 8. Además, en las líneas 16 a la 18 se procede a llenar el vector. Para finalizar, en las líneas 22 a la 24 se almacena la suma en la variable media y en la línea 25 se calcula la media aritmética.

Al ejecutar, salida es:

```
1 \programas_libro> java VertorMedia  
2 52    15    66    40    62    65    17    22  
3 La media del vector es: 42.375
```

**Ejercicio 3:** Agrega el nombre de  $n$  alumnos a un arreglo.

Se requiere un arreglo que permita almacenar  $n$  estudiantes y luego mostrarlos.

*Ejemplo 156 Arreglo de estudiantes*

```
1 import java.util.Scanner;  
2 public class VectorAlumnos {  
3     public static void main(String[] args) {  
4         Scanner scanner = new Scanner(System.in);  
5         System.out.println("Ingrese el número de estudiantes: ");  
6         int nroEstudiantes = Integer.parseInt(scanner.nextInt());  
7         String alumnos[] = new String[nroEstudiantes];  
8         System.out.println("Ingrese el nombre de los  
9             estudiantes");  
10        for(int i = 0; i < alumnos.length; i++) {  
11            System.out.println("Nombre del alumno nro "+(i+1));  
12            alumnos[i] = scanner.nextLine();  
13        }  
14        System.out.println("Los alumnos son:");  
15        for(int i = 0; i < alumnos.length; i++) {  
16            System.out.println(alumnos[i]);  
17        }  
18    }  
19 }
```

En la línea 4 se instancia un objeto de la clase Scanner para leer el número de estudiante por teclado, luego en la línea 7 se crea un arreglo de *String* con la longitud especificada por el usuario, en la línea 12 el usuario que ingrese los nombres de los estudiantes y de la línea 15 a 17 se imprime la lista de estudiantes.

Al ejecutar el programa su salida es:

```
1 \programas_libro> java VectorAlumnos  
2 Ingrese el número de estudiantes:
```

```

3   5
4   Ingrese el nombre de los estudiantes
5   Nombre del alumno nro 1
6   Maria
7   Nombre del alumno nro 2
8   José
9   Nombre del alumno nro 3
10  Manuel
11  Nombre del alumno nro 4
12  Karina
13  Nombre del alumno nro 5
14  Miguel
15  Los alumnos son:
16  Maria
17  José
18  Manuel
19  Karina
20  Miguel

```

**Ejercicio 4:** agregar el nombre de n alumnos a un arreglo y poder modificar.

Se requiere un arreglo que permita almacenar n estudiantes, mostrarlos y luego modificar algún nombre.

*Ejemplo 157 Arreglo de estudiantes para modificar sus nombres*

```

1  import java.util.Scanner;
2
3  public class VectorAlumnosModificar {
4
5      public void imprimir(String alumnos[]) {
6          System.out.println("Los alumnos son:");
7          for (int i = 0; i < alumnos.length; i++) {
8              System.out.println((i + 1) + ":" + alumnos[i]);
9          }
10     }
11
12     public static void main(String[] args) {
13         Scanner scanner = new Scanner(System.in);
14         VectorAlumnosModificar vam = new
15             VectorAlumnosModificar();
16         System.out.println("Ingrese el número de estudiantes: ");
17         int nroEstudiantes = Integer.parseInt(scanner.next());
18         String alumnos[] = new String[nroEstudiantes];
19         System.out.println("Ingrese el nombre de los
20             estudiantes");
21         for (int i = 0; i < alumnos.length; i++) {
22             System.out.println("Nombre del alumno nro " + (i +
23                 1));
24             alumnos[i] = scanner.nextLine();
25         }
26         vam.imprimir(alumnos);
27         System.out.println("Ingrese la posición del Alumno que se
28             desea modificar");
29         int posicion = Integer.parseInt(scanner.nextLine());
30     }
31 }

```

```

32 if (posicion > 0 && (posicion - 1) <= alumnos.length) {
33     System.out.println("Ingrese el nuevo nombre");
34     alumnos[posicion - 1] = scanner.next();
35     vam.imprimir(alumnos);
36 } else {
37     System.out.println("No ha seleccionado una posición
38         valida");
39 }
40 }
41 }
```

En este ejemplo se crea un método para imprimir la lista de alumnos separado del método principal. Además, en la línea 35, se solicita al usuario que ingrese la posición en la lista de alumnos que desee modificar, si la posición está en el rango de la longitud del arreglo modificará el elemento en esa posición, pero si se ingresa otro valor fuera del rango, mostrará un mensaje “**No ha seleccionado una posición valida**”

Si se hace ejecutar saldría la siguiente salida:

```

1 \programas_libro> java VectorAlumnosModificar
2 Ingrese el número de estudiantes:
3 3
4 Ingrese el nombre de los estudiantes
5 Nombre del alumno nro 1
6 Manuel
7 Nombre del alumno nro 2
8 Diana
9 Nombre del alumno nro 3
10 José
11 Los alumnos son:
12 1 : Manuel
13 2 : Diana
14 3 : José
15 Ingrese la posición del Alumno que se desee modificar
16 2
17 Ingrese el nuevo nombre
18 Karina
19 Los alumnos son:
20 1 : Manuel
21 2 : Karina
22 3 : José
```

## 15.3 ARREGLOS BIDIMENSIONALES

Los arreglos bidimensionales tienen una dos o más dimensiones, también se lo conoce como vector. Está compuesto por filas y columnas.

Ejemplo una matriz de enteros de 3x3

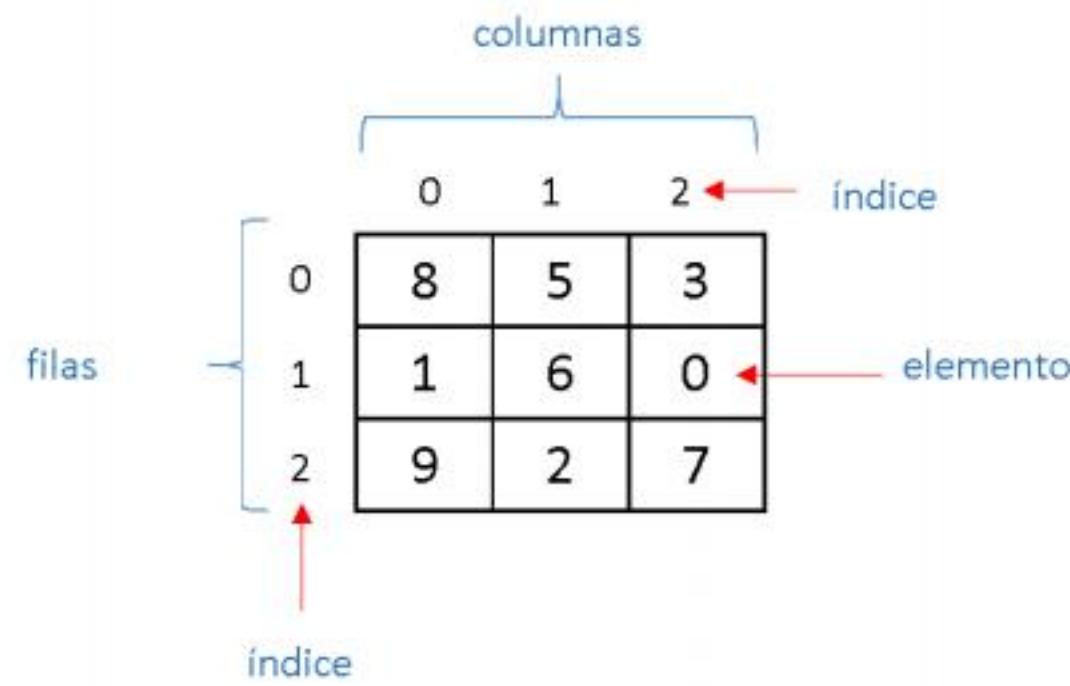


Figura 75 Componentes de una matriz

### 15.3.1 DECLARACIÓN Y CREACIÓN

#### DECLARACIÓN

La declaración se define como cualquier atributo o variable, pero para establecer que es un matriz se utiliza [ ] [ ], su sintaxis es

```
tipo_dato [ ] [ ] identificador_matriz;
o
tipo_dato identificador_matriz [ ] [ ];
```

Ejemplos:

```
1 int [ ] [ ] edadesEstudiantes;
2 double [ ] [ ] notas;
3 char [ ] [ ] alfabeto;
4 boolean [ ] [ ] productosDisponibles;
5
```

#### CREACIÓN

Se requiere de la palabra reservada **new** y fijar el tamaño del arreglo dentro de los corchetes [ ] [ ]. Su sintaxis es:

```
identificador_arreglo = new tipo_datos [ filas ]
[ columnas ]
```

Ejemplos:

```
1 edadesEstudiantes = new int [90][3];
2 notas = new double [8][5];
3 alfabeto = new char [5][2];
4 productosDisponibles = new boolean [78][3];
```

La definición y creación de un arreglo se puede realizar en una sola sentencia:

```
1 int [] edadesEstudiantes = new int [90][3];  
2 double [] notas = new double [8][5];  
3 char [] vocales = new char [5][2];  
4 boolean [] productosDisponibles = new boolean [78][3];
```

La siguiente figura detalla las partes de una matriz:

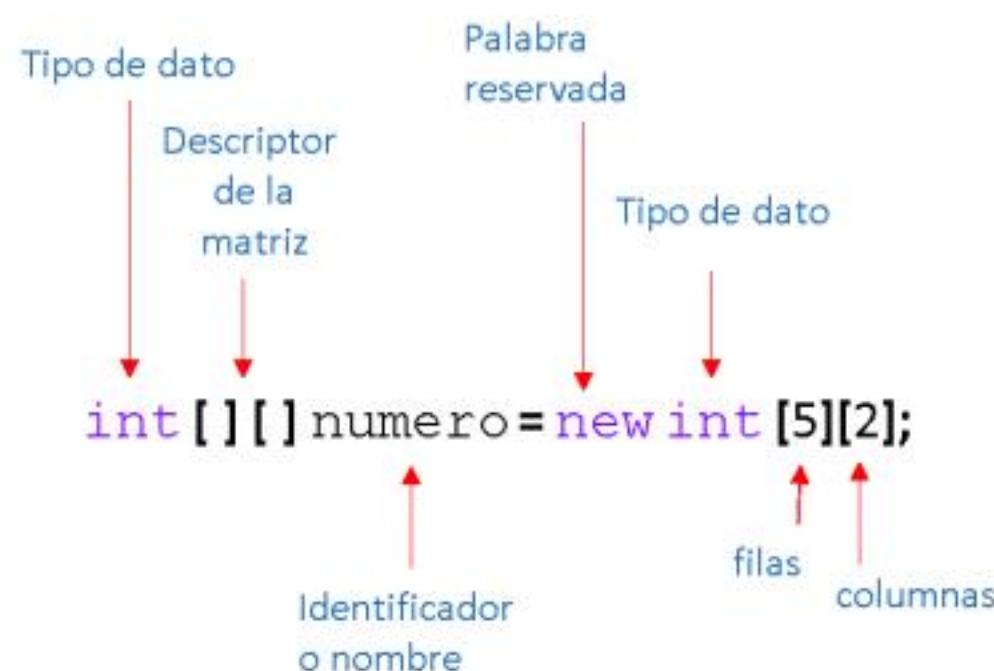


Figura 76 Partes de un arreglo

Otra forma para crear una matriz, es crearlo con sus valores ya definidos o valores iniciales. Su sintaxis es:

```
tipo_dato [][] identificador_matriz = {{ valor1, valor2, valor3}, { valor1,  
valor2, valor3}, { valor1, valor2, valor3}, { valor1, valor2, valor3}...}
```

Ejemplos:

```
1 int [][] edadesEstudiantes = {{98, 77, 5, 76}, {8, 2, 0, 6}, {8, 66, 88, 99}};  
2 double [] [] notas = {{9.8, 7.6, 5.5, 8.0, 7.7}, {8, 8.6, 9.5, 8.0, 2.0}};  
3 char [] [] alfabeto = {{'a', 'b', 'c', 'd', 'e'}, {'f', 'g', 'h', 'i', 'j'}, {'k', 'l', 'ñ', 'm', 'n'}, {'o', 'p', 'q', 'r',  
's'}};  
4 boolean [] [] productosDisponibles = {{true, false}, {true, true}, {false, false}, {false, true},  
{true, false}, {false, false}};
```

Para este tipo de declaración es importante conocer algunos aspectos:

- **Identificador:** nombre de la matriz.
- **Llaves externas:** representa toda la matriz en general.
- **Llaves internas:** representan cada una de las filas que puede contener la matriz.
- **Valores:** representan los valores de cada columna en la fila respectiva.

En Java, si una matriz es creada con datos primitivos siempre se inicializarán con su valor por defecto, ejemplo si una matriz es de enteros sus valores por defecto serán cero.

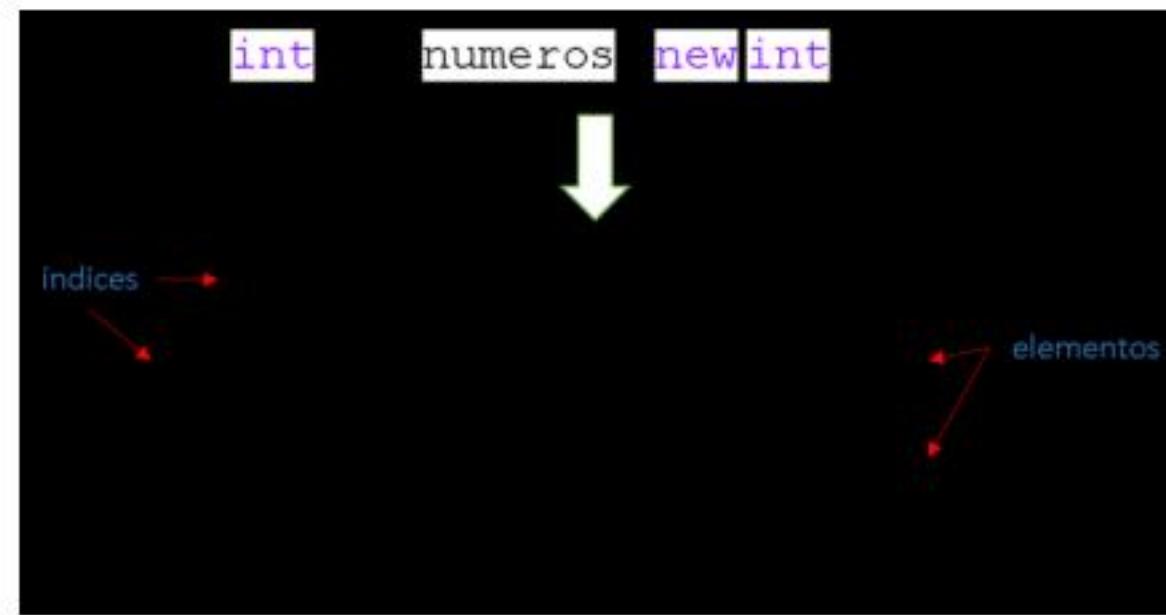


Figura 77 Valores por defecto en una matriz de enteros

En cambio, sí un arreglo es de objetos se inicializarán con su valor `null` por defecto, por ejemplo en un arreglo de `String`.

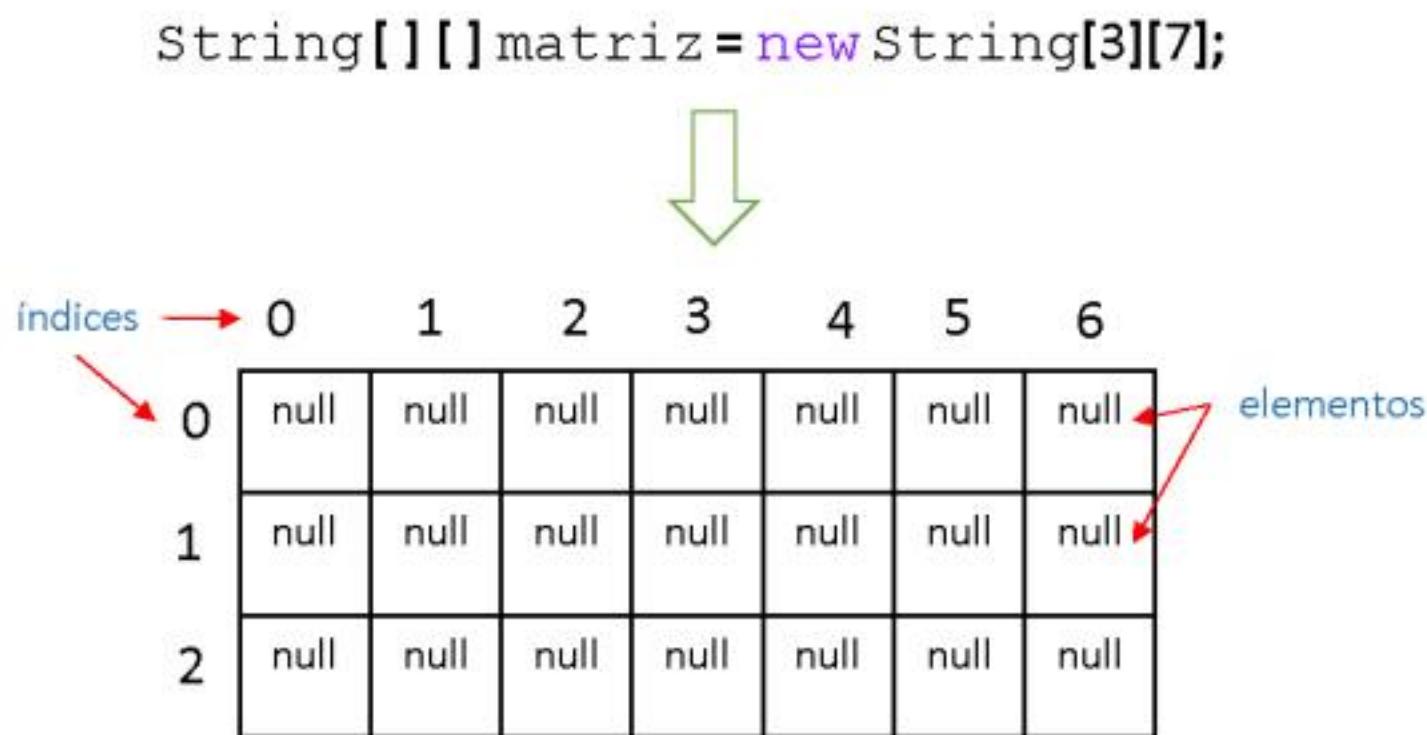


Figura 78 Valores por defecto en una matriz de String

En las operaciones de definir y crear matrices se pueden cometer errores de compilación. Es necesario conocerlos para evitar cometerlos:

1. Crear arreglos estáticos en tiempo de compilación. Ejemplo:

```
1 | int [4][4]edades;
```

2. Asignar valor a una matriz sin haber determinado su tamaño. Ejemplo:

```
1 | int [ ][ ] edades;
2 | edades [0][0] = 15;
```

### 15.3.2 LECTURA Y ESCRITURA

#### ESCRITURA

Una vez definido y creada la matriz, se debe escribir o llenar de elementos.

Para llenar una matriz, se lo realiza indicando la posición indicando la fila y la columna.



Figura 79 Escritura de una matriz

Ejemplo:

```
1 | int matriz [][] = new int [4][3];
2 | matriz [0][1] = 9;
3 | matriz [1][2] = 7;
4 | matriz [2][0] = 6;
5 | matriz [3][3] = 0;
```

Si se accede a una determinada posición en sus filas y columnas que no estén en el rango se generará un error de ejecución de tipo `ArrayIndexOutOfBoundsException`.

#### LECTURA O RECORRIDO

Para acceder al contenido o el valor de una matriz se debe especificar la posición de la fila y columna entre corchetes. La lectura de datos se lo realiza mediante dos estructuras repetitivas anidadas `for` o cualquier otra estructura repetitiva, accediendo al elemento mediante sus índices.

Cabe destacar que para recorrer una matriz se requiere saber la longitud del arreglo tanto de las filas y de las columnas mediante la propiedad `length`:

- **Filas:** `matriz.length`
- **Columnas:** `matriz[0].length`

Analicemos el siguiente ejemplo:

*Ejemplo 158 Lectura y recorrido de un arreglo*

```
1 public class Matriz {  
2  
3     public static void main(String[] args) {  
4         int matriz [][] = {{7,6,8,9},{55,66,77,88},{98,6,0,98}};  
5         for (int i = 0; i < matriz.length; i++)  
6             for (int j = 0; j < matriz[0].length; j++)  
7                 System.out.println("matriz [" + i + "][" + j + "]  
8                     = " + matriz[i][j]);  
9     }  
}
```

Los ciclos **for** anidados, permiten recorrer a los elementos de una matriz a través de los índices, cada iteración hace que incremente en uno el valor de la variable controladora “i” para las filas y “j” para las columnas, permitiendo que se acceda a los elementos del matriz.

Su salida es:

```
1 \programas_libro> java Matriz  
2 matriz [0][0] = 7  
3 matriz [0][1] = 6  
4 matriz [0][2] = 8  
5 matriz [0][3] = 9  
6 matriz [1][0] = 55  
7 matriz [1][1] = 66  
8 matriz [1][2] = 77  
9 matriz [1][3] = 88  
10 matriz [2][0] = 98  
11 matriz [2][1] = 6  
12 matriz [2][2] = 0  
13 matriz [2][3] = 98
```

### 15.3.3 EJEMPLOS

**Ejercicio 1:** Crear una matriz cuadrática de números decimales, agregar elementos con dos cifras decimales menores a 100 y presentarla.

*Ejemplo 159 Matriz de números decimales*

```
1 public class MatrizInicial {  
2     public static void main(String[] args) {  
3         double[][] a = new double[3][3];  
4         //llenado  
5         for(int i = 0; i < a.length; i++) {  
6             for(int j = 0; j < a[0].length; j++) {  
7                 a[i][j] = (double) Math.round(Math.random() *  
8                     10000) / 100;  
9             }  
10        }  
11        //presentar  
12        System.out.println("La matriz es");
```

```

13   for(int i = 0; i < a.length; i++) {
14     for(int j = 0; j < a[0].length; j++) {
15       System.out.print(a[i][j] + "\t");
16     }
17     System.out.println("");
18   }
19   System.out.println("");
20 }
21 }
```

En este ejemplo, en la línea 3 se define y crea una matriz cuadrática de 3 filas \* 3 columnas. En la línea 5 al 10 se llena la matriz con números aleatorios con los dos ciclos repetitivos. Además, en la línea 7 se registra números menores a 100 mediante (Math.random()\*10000) y se redondea el numero decimal a 2 cifras utilizando el método Math.round, dividiendo para 100. Para finalizar, en las líneas 13 y 18 se presenta los datos de la matriz.

Al ejecutar el algoritmo se presenta los siguientes resultados:

```

1 \programas_libro> java MatrizInicial
2 La matriz es
3 87.81 87.06 44.42
4 83.69 32.02 41.08
5 25.31 56.05 16.53
```

**Ejercicio 2:** Crear una matriz que se ingrese n filas y n columnas de números decimales, agregar elementos con dos cifras decimales menores a 100 y presentarla, esto se lo hará con llamadas a métodos.

*Ejemplo 160 arreglo de números decimales haciendo llamados de métodos*

```

1 import java.util.Scanner;
2
3 public class UtilidadesMatriz {
4
5   public double[][] cargarMatrizAleatorio(double[][] a) {
6     for(int i = 0; i < a.length; i++) {
7       for(int j = 0; j < a[0].length; j++) {
8         a[i][j] = (double) Math.round(Math.random() *
9           10000) / 100;
10      }
11    }
12    return a;
13  }
14
15  public void presentarMatrizDouble(double[][] a) {
16    System.out.println("La matriz es");
17    for(int i = 0; i < a.length; i++) {
18      for(int j = 0; j < a[0].length; j++) {
19        System.out.print(a[i][j] + "\t");
20      }
21    System.out.println("");
```

```

22 }
23 System.out.println("");
24 }
25
26 public static void main(String[] args) {
27     Scanner sc = new Scanner(System.in);
28     System.out.println("Ingrese el # de filas:");
29     int fila = Integer.parseInt(sc.nextInt());
30     System.out.println("Ingrese el # de columnas:");
31     int columnas = Integer.parseInt(sc.nextInt());
32     double[][] a = new double[fila][columnas];
33     UtilidadesMatriz um = new UtilidadesMatriz();
34     a = um.cargarMatrizAleatorio(a);
35     um.presentarMatrizDouble(a);
36 }
37

```

En este ejemplo, se separa la entrada y presentación de los datos en la matriz que serán invocados en el método *main*. En la línea 28 se solicita al usuario que ingrese las filas y columnas de la matriz a través de la clase Scanner, esto significa que el usuario establece sus dimensiones. Para finalizar se crea un objeto de la clase “UtilidadesMatrizAleatoria” para utilizar los métodos “cargarMatrizAleatoria” y “presentarMatrizDouble”.

El resultado es:

```

1 \programas_libro> java UtilidadesMatriz
2 Ingrese el # de filas:
3 3
4 Ingrese el # de columnas:
5 5
6 La matriz es
7 8.23 9.43 6.03 23.56 5.1
8 46.94 40.71 78.3 96.33 35.95
9 60.26 31.36 31.35 79.39 56.18

```

## BIBLIOGRAFÍA

- Arnow, D., & Weiss, G. (2001). *Introducción a la programación con Java*. Pearson Publications Company.
- Barnes, D. J., & Kölking, M. (2007). *Programación orientada a objetos con Java. Manejo eficiente del tiempo*.
- Boyarsky, J., y Selikoff, S. (2014). *OCA: Oracle Certified Associate Java SE 8 Programador I Study Guide*, Examen 1Z0-808.
- John Wiley & Sons. Dean, J. S., & Dean, R. H. (2000). *Introducción a la programación con Java*. McGraw-Hill Interamericana.
- Deitel, H. M., & Deitel, P. J. (2004). *Cómo programar en Java*. Pearson Educación.
- Durán, F., Gutiérrez, F., & Pimentel, E. (2007). *Programación orientada a objetos con Java*. Editorial Paraninfo.
- Froufe Quintas, A., & Cárdenes, J. (2004). *J2ME, Java 2 Micro Edition: manual de usuario y tutorial*.
- Joyanes Aguilar, L., & Zahonero Martínez, I. G. N. A. C. I. O. (2002). *Programación en Java 2*. Madrid: McGraw-Hill.
- Osorio, F. L. (2007). *Introducción a la Programación en Java*. ITM.
- Sánchez, J. (2004). *Java2-Incluye Swing, Threads, programación en red, JavaBeans, JDBC y JSP/Servlets*. España:(www. jorgesanchez.net).
- Torres, S. A. C., Valbuena, S. J., & Ramirez, M. L. V. (2008). *Introducción a la Programación en Java*. ELIZCOM SAS.
- Wu, C. T., & Wu, C. T. (2001). *Introducción a la programación orientada a objetos con Java*.

## SOBRE LOS AUTORES

### **EDWIN RENÉ GUAMÁN QUINCHE**

Nació en Ecuador en 1982; es Ingeniero en Sistemas, a nivel de postgrado tiene un máster en Sistemas Informáticos Avanzados en la Universidad del País Vasco - España. Desde el año 2017 se desempeña como docente de la Facultad de Ciencias Informáticas en la Universidad Laica Eloy Alfaro de Manabí - Ecuador, tutorando asignaturas relacionadas a la Programación Orientada a Objetos, Web y Móvil. Ha investigado sobre tecnologías web, sistemas distribuidos y lenguajes de programación, habiendo publicado varios artículos y realizado ponencias al respecto.



### **JOSÉ OSWALDO GUAMÁN QUINCHE**

Nació en Ecuador en 1984; es Ingeniero en Sistemas, a nivel de postgrados tiene un máster en Ingeniería Informática en la Universidad de Girona - España. Desde el 2012 se desempeña como docente en el Instituto Superior Tecnológico Daniel Álvarez Burneo y en el Instituto Tecnológico Superior Sudamericano, tutorando las materias de Análisis y Diseño de Sistemas, Programación Orienta a Objetos y Sistemas Operativos. Además, es desarrollador consultor de la empresa IOET y en proyectos de automatización.



### **DAYSI MIREYA ERREYES PINZÓN**

Nació en Ecuador en 1978, es Ingeniera en Informática, a nivel de posgrado tiene una maestría Gestión Estratégica de Tecnologías de la Información, se ha desempeñado como docente de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja UNL - Ecuador (2005-2012), tutorando las asignaturas de Metodología de la programación, Estructuras de Datos Orientada a Objetos, Proyectos Informáticos e Ingeniería del Software; como técnico se ha desempeñado en la Unidad de Telecomunicaciones e Información como Responsable de Bases de Datos y Estadística. Ha investigado sobre seguridad en entornos móviles, y ha publicado una Metodología para validación de herramientas para la seguridad en dispositivos móviles.



### **HERNÁN LEONARDO TORRES CARRIÓN**

Nació en Ecuador en 1981; recibió el título de Ingeniero en Sistemas Informáticos y Computación de la Universidad Técnica Particular de Loja en el 2005, también el Diplomado en Auditoría en Gestión de la Calidad en la misma universidad en el 2006 y Magister en Telemática de la Universidad de Cuenca en 2011. Actualmente es docente Titular y Director de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja y su interés en investigación se relacionan con las nuevas Tecnologías de Información y Comunicación específicamente en Internet de las Cosas (IoT), habiendo publicado varios artículos y realizado ponencias al respecto.



ISBN: 978-9942-775-05-4



9789942775054

[www.uleam.edu.ec](http://www.uleam.edu.ec)