



# Programación moderna con aplicaciones

**Omar Iván Trejos Buriticá**  
(Buenaventura, Valle del Cauca, Colombia, 1965).

Ingeniero de Sistemas, especialista en Instrumentación Física, MSc en Comunicación Educativa, PhD en Ciencias de la Educación - RudeColombia Cade UTP.

Docente Titular de planta, Facultad de Ingenierías de la Universidad Tecnológica de Pereira.

Autor de los libros: *Programación Imperativa con Lenguaje C Estadística y Probabilidades para Ingenieros. Lógica de Programación.*  
Ha publicado artículos en revistas especializadas.

Pertenece al Grupo de Investigación: Informática Ingeniería de Sistemas y Computación

[omartrejos@utp.edu.co](mailto:omartrejos@utp.edu.co)

**Luis Eduardo Muñoz Guerrero**  
(San Juan de Pasto, Nariño, Colombia, 1975)

Ingeniero de Sistemas , MSc en Ingeniería de Sistemas y Computación  
PhD (E) en Ciencias de la Educación RudeColombia Cade UTP.

Docente Asociado de planta en el Programa de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira .

Autor del libro: *Programación Funcional con Racket* .

Pertenece el grupo de investigación Informática, Ingeniería de Sistemas y Computación

[lemunozg@utp.edu.co](mailto:lemunozg@utp.edu.co)

**Guillermo Roberto Solarte Martínez**  
(Popayán, Cauca, Colombia, 1974).

Ingeniero de Sistemas, MSc en Investigación Operativa y Estadística  
PhD en Informática de la Universidad Pontificia de Salamanca.

Docente Transitorio Titular en el Programa de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira.

Autor del libro: *Guía Didáctica de Estructuras de Datos*

Pertenece al Grupo de Investigación:

Grupo de Avanzada en Desarrollo de Software y Grupo en Inteligencia Artificial Ingeniería de Sistemas y Computación

[roberto@utp.edu.co](mailto:roberto@utp.edu.co)

# **PROGRAMACIÓN MODERNA CON APLICACIONES**

Omar Iván Trejos Buriticá  
Guillermo Roberto Solarte Martínez  
Luis Eduardo Muñoz Guerrero



Textos Académicos  
Facultad de Ingenierías  
2019

Trejos Buriticá, Omar Iván  
Programación moderna con aplicaciones / Omar Iván Trejos Buriticá,  
Guillermo Roberto Solarte Martínez y Luis Eduardo Muñoz Guerrero. – Pereira :  
Universidad Tecnológica de Pereira,  
2019.  
388 páginas. – (Colección Textos académicos).  
ISBN: 978-958-722-397-2  
1. Lenguajes de programación (Computadores) 2. Métodos  
orientados a objetos (Computadores) 3. Java (Lenguaje de  
programación de computadores) 4. Procesamiento electrónico  
de datos.  
CDD 005.133

© Omar Iván Trejos Buriticá  
© Guillermo Roberto Solarte Martínez  
© Luis Eduardo Muñoz Guerrero  
©Universidad Tecnológica de Pereira  
Primera edición

Universidad Tecnológica de Pereira  
Vicerrectoría de Investigaciones, Innovación  
y Extensión  
Editorial Universidad Tecnológica de Pereira  
Pereira, Colombia

**Coordinador editorial:**  
Luis Miguel Vargas Valencia  
[luismvargas@utp.edu.co](mailto:luismvargas@utp.edu.co)  
Teléfono 313 7381  
Edificio 9, Biblioteca Central “Jorge Roa  
Martínez”  
Cra. 27 No. 10-02 Los Álamos, Pereira,  
Colombia  
[www.utp.edu.co](http://www.utp.edu.co)

**Montaje y producción:**  
Recursos Informáticos y Educativos  
CRIE, [diseno@utp.edu.co](mailto:diseno@utp.edu.co)  
Universidad Tecnológica de Pereira

*A mi Natica y mi Juanjo  
Por siempre,  
los dos oasis del desierto  
de mi existencia  
O.I.T.B.*

*A Dios por darme la sabiduría, el conocimiento, la esperanza y  
la confianza de seguir adelante paso a paso en mi vida. A mis  
padres y hermanos por sus consejos, apoyo incondicional, por  
su tiempo, sus preocupaciones y compañía durante esta etapa  
de mi vida. A mi hijos Jhoan Nicolás Solarte y Nicol Vanessa  
Solarte, por su amor y su compañía, por el tiempo que no les he  
podido dedicar para alcanzar esta meta.*

*G.R.S.M.*

*A mi madre Ana Guerrero  
Quien ha sido un pilar invaluable para el  
fomento y evolución de mi vida  
L.E.M.G.*



## INTRODUCCIÓN

Después de muchos intentos por desarrollar un libro que enseñara la programación orientada a objetos y se soportara en el lenguaje de programación Java, después de muchas solicitudes de estudiantes para que se hiciera realidad y, sobre todo, después de leer y leer tantos libros de programación en Java como pudimos, decidimos escribir este libro por considerar que tanto el paradigma POO (Programación Orientado a Objetos) como el lenguaje de programación Java se han convertido en la gran alternativa de programación del mundo moderno.

Por su concepción y por la manera como se explica el paradigma como camino para interpretar el mundo que nos rodea, consideramos que las aplicaciones que se han desarrollado hasta el momento son apenas un inicio de la gran cantidad de programas, sistemas y software que llegarán un día a facilitar nuestra vida en medio de esa impresionante penetración comercial de la tecnología y que, enhorabuena, nos abre posibilidades a los programadores para que diseñemos soluciones a tantas necesidades que aún no se han resuelto.

Debemos advertir que este primer libro de Java está escrito para personas que no han tenido contacto con la programación orientada a objetos ni menos con el lenguaje de programación Java. Algunas cosas podrán parecer muy sencillas pero tenga en cuenta que lo básico, siendo aparentemente simple, se requiere aprender muy bien para poder asimilar lo profundo. Con este libro, esperamos responder a las inquietudes que tantos alumnos han presentado y, de la misma forma, esperamos que sea la base para que usted ingrese a ese maravilloso mundo que nos rodea, visto desde la perspectiva de la POO.

Los Autores



# TABLA DE CONTENIDO

## LECCIÓN 1 PRIMERA APROXIMACIÓN A JAVA

1.1	Introducción .....	17
1.2	Un poquito de historia.....	18
1.3	¿Qué hace diferente a Java? .....	19
1.4	El primer programa.....	20
	1.4.1 Ejercicios propuestos .....	22

## LECCIÓN 2 VARIABLES Y SALIDA DE DATOS

2.1	Definiciones .....	25
2.2	Tipos de Datos .....	26
2.3	Comentarios .....	28
2.4	Reglas de las Variables.....	29
2.5	Salida de información .....	31
2.6	Ejercicios Propuestos .....	33

## LECCIÓN 3 CONSTANTES, EXPRESIONES Y OPERADORES

3.1	Constantes .....	39
3.2	Expresiones .....	40
3.3	Operadores .....	42
3.4	Ejercicios Propuestos.....	44

## LECCIÓN 4 CLASE MATH Y CAMBIOS DE TIPO

4.1	Algunos métodos de la Clase Math.....	47
4.2	Uso de la Clase Math.....	48
4.3	Cambios de tipo .....	50
4.4	Ejercicios Propuestos.....	52

## LECCIÓN 5 CADENAS

5.1	Definición .....	57
5.2	Declaración de Cadenas.....	57
5.3	La clase String .....	58
5.4	Uso de la Clase String.....	59
5.5	Ejercicios Propuestos.....	61

## LECCIÓN 6 ENTRADA DE DATOS

6.1	Paquetes, Clases y Métodos.....	65
6.2	La Clase Scanner.....	66
6.3	La Clase JOptionPane .....	69
	6.3.1 Leyendo una cadena.....	69
	6.3.2 Leyendo un entero.....	71
	6.3.3 Leyendo un real .....	74

6.4	Formato de números con decimales .....	75
6.5	Ejercicios Propuestos.....	76

## LECCIÓN 7 CONDICIONALES

7.1	El flujo de la ejecución .....	80
7.2	Condicional Simple .....	80
7.2.1	Ejercicios Propuestos .....	85
7.3	Condicional Múltiple .....	85
7.4	Ejercicios Propuestos.....	90

## LECCIÓN 8 CICLOS

8.1	Definición .....	93
8.2	Ciclo while .....	96
8.2.1	Ejemplo resuelto .....	96
8.2.2	Ejercicios Propuestos .....	99
8.3	Ciclo do-while.....	99
8.3.1	Ejemplo resuelto .....	100
8.3.2	Ejercicios Propuestos .....	103
8.4	Ciclo for .....	103
8.4.1	Ejemplo resuelto .....	104
8.4.2	Ejercicios Propuestos .....	107

## LECCIÓN 9 VECTORES

9.1	Definición .....	111
9.2	Utilidad .....	111
9.3	Tipos de arreglos.....	112
9.4	Declaración de un arreglo .....	113
9.5	Carga de datos en un arreglo.....	114
9.5.1	Carga de datos en conjunto.....	114
9.5.2	Carga de datos individual.....	114
9.5.3	Carga de datos por digitación .....	115
9.5.4	Carga de datos por operación .....	115
9.6	Despliegue de datos de un arreglo.....	116
9.7	Ejercicios Propuestos.....	117
9.8	Ciclo for – each .....	117
9.9	Ejercicios Propuestos.....	120

## LECCIÓN 10 MATRICES

10.1	Definición .....	125
10.3	Declaración de una matriz .....	126
10.4	Carga de datos en una matriz .....	127
10.4.1	Carga de datos en conjunto .....	127
10.4.2	Carga de datos individual.....	127
10.4.3	Carga de datos por digitación .....	128

10.5	Despliegue de datos en una matriz .....	128
10.6	Ejercicios Propuestos.....	129

## LECCIÓN 11 APROXIMACIÓN A LA POO

11.1	Paradigmas de programación .....	133
11.2	Objetos en el mundo real .....	134
11.3	Características propias de la POO .....	134
11.3	¿Qué es la POO? .....	135
11.4	Creación de una Clase .....	136
11.5	Creación de un Objeto .....	138
11.6	Ejercicios Propuestos.....	143

## LECCIÓN 12 MÉTODO CONSTRUCTOR CON PARÁMETROS

12.1	Reflexión .....	147
12.2	Constructor de Clase con parámetros .....	147
12.3	Construcción ClaseTriangulo .....	150
12.3	Constructor de Clase parámetros digit.....	155
12.4	Ejercicios Propuestos.....	158

## LECCIÓN 13 MODULARIZACIÓN Y ENCAPSULACIÓN

13.1	¿Qué es la Modularización? .....	161
13.2	Ejemplo Resuelto .....	163
13.4	Ejercicios Propuestos .....	167
13.3	¿Qué es la Encapsulación? .....	167
13.4	Métodos tipo Get o Métodos Getter .....	171
13.5	Métodos tipo Set o Métodos Setter .....	172
13.6	Ejercicios Propuestos.....	175

## LECCIÓN 14 MODIFICADORES DE ACCESO

14.1	Constantes .....	179
14.2	El modificador final .....	179
14.3	Ejercicio Resuelto con final .....	181
14.4	Ejercicios Propuestos.....	186
14.5	Variables Estáticas .....	187
14.6	El modificador static .....	188
14.7	Ejercicio Resuelto con static .....	188
14.8	Ejercicios Propuestos .....	196
14.9	Acceso a variables static .....	196
14.10	Ejercicios Propuestos.....	201

## **LECCIÓN 15 SOBRECARGA DE MÉTODOS**

15.1	Definición .....	205
15.2	Sobrecarga de Constructores .....	205
15.3	Ejercicios Propuestos.....	211

## **LECCIÓN 16 HERENCIA**

16.1	Concepto .....	215
16.2	Ejemplo Resuelto .....	217
16.3	Ejercicios Propuestos.....	227

## **LECCIÓN 17 POLIMORFISMO**

17.1	Polimorfismo .....	231
17.2	Principio de Sustitución .....	233
17.3	Polimorfismo POO .....	233
17.4	Anotaciones .....	234
17.5	Ejemplo Resuelto .....	235
17.6	Ejercicios Propuestos.....	243

## **LECCIÓN 19 CLASES ABSTRACTAS**

19.1	¿Qué es una Clase Abstracta? .....	247
19.2	Ejemplo Resuelto .....	249
19.3	Ejercicios Propuestos .....	261
19.4	Acerca de los modificadores de acceso 262	
19.5	Acerca de la clase Object.....	264

## **LECCIÓN 20 DATOS ENUMERADOS**

20.1	Concepto General .....	269
20.2	Ejemplo Resuelto .....	269
20.3	Ejercicios Propuestos.....	274

## **LECCIÓN 21 INTERFACES**

21.1	Concepto General .....	277
21.2	Ejemplo Resuelto .....	279
21.3	Ejercicios Propuestos .....	286
21.4	Herencia entre Interfaces .....	286
21.5	Ejercicios Propuestos.....	292

## **LECCIÓN 22 CLASES INTERNAS**

22.1	Concepto General .....	295
22.2	Ejemplo Resuelto .....	296
22.3	Ejercicios Propuestos .....	301

## **LECCIÓN 23 INTERFACES DE USUARIO**

23.1	Concepto General .....	305
23.2	Creación de una ventana .....	307
23.3	Cambiando atributos de una ventana .....	310
23.4	Escribiendo texto sobre una ventana .....	316
23.5	Dibujando figuras y líneas .....	323
23.6	Trabajando con Colores .....	327
23.7	Modificando tipos de letras .....	332
23.8	Imágenes .....	336
23.9	Ejercicios Propuestos.....	340

## **LECCIÓN 24 EVENTOS DE MOUSE**

24.1	Concepto General .....	343
24.2	Definición .....	343
24.3	Ejemplo .....	345
24.4	Ejecución del programa .....	365
24.5	Ejercicios Propuestos.....	368

## **LECCIÓN 25 EVENTOS DE VENTANA**

25.1	Concepto general .....	371
25.2	Implementación en Java.....	372
25.3	Código de ejemplo.....	373
25.4	Ejecución del Programa Ejemplo .....	384
25.5	Comentario adicional.....	386
25.6	Ejercicios .....	387



# 1 Lección

## Primera aproximación a Java



## 1.1 Introducción

El lenguaje de programación Java tiene unas características muy propias que lo diferencian de cualquier otro lenguaje de programación, pero primero, vamos a definir lo que es un Lenguaje de Programación. Sencillamente es un conjunto de instrucciones sistemáticamente organizadas que, con la ayuda de un Compilador o de un Intérprete, puede realizar determinadas tareas aprovechando la capacidad de procesamiento, almacenamiento y despliegue de datos de un computador o de cualquier dispositivo electrónico que tenga estas capacidades.

¿Qué es un *Compilador*? Es otro programa que revisa los programas que nosotros escribimos y verificamos si está bien escrito. Veámoslo de la siguiente forma. Lea la siguiente frase: “*Oi no fui al colejio*”. Estoy seguro que usted pensará que el corrector de estilo de este libro no realizó bien su trabajo y no es así. Esta frase, así como está escrita, tiene tres errores ortográficos que, de una u otra forma, saltan a la vista. La palabra “*Oi...*” tal como aparece en la frase, se refiere al día presente, se escribe “*Hoy...*” y la palabra “*...colejio*”, que se refiere al lugar donde estudiamos, se escribe “*...colegio*”.

Pues bien, esos son errores ortográficos y los reconocemos inmediatamente porque nuestro cerebro tiene unas reglas para escribir bien el español. ¿De dónde salen esas reglas? De un montón de razones culturales, sociales, históricas, políticas, etc., etc., lo cierto es que HOY y COLEGIO se escriben como aparecen aquí en mayúsculas. Un programa, independiente del lenguaje en el cual se escriba, va a tener también unas reglas para escribirse y precisamente el *Compilador* es el encargado de verificar que así sea, es decir, que se han cumplido las reglas de escritura, dentro del contexto del lenguaje de programación, para que el programa sea entendido plenamente. Le doy un ejemplo, en Java, la instrucción *System.out.println* se escribe tal como aparece aquí. Si escribiéramos *Sistem.out.println*, el compilador de Java detectará un error de sintaxis (que en lenguajes de programación, aproximadamente, equivale a lo que en el idioma español es la ortografía) y lo reportará indicando que debe corregirse.

¿Qué es un *Interpretador*? Realiza la misma tarea de un compilador, solo que el *Interpretador* revisa el programa línea por línea y va ejecutando cada línea que esté bien escrita; el *Compilador* revisa el programa línea por línea pero lo ejecuta sólo cuando el programa completo está bien escrito. Ahora sí podemos decir que Java es un lenguaje de programación que requiere se compilado e interpretado para que sus programas se ejecuten.

El lenguaje de programación Java también tiene la característica de ser *Multiplataforma*, lo cual significa que puede funcionar en cualquier sistema operativo. Veamos qué significa esto. Cuando usted enciende un computador, usted nota que el computador se demora un pequeño tiempo mientras aparece el entorno que le indica que ya está listo para que usted lo utilice. Ese pequeño tiempo (que algunas veces es un poquito más demorado de lo normal) es el momento en el cual el computador “carga” su sistema operativo. Como cuando nos despertamos, después de dormir toda una noche, es ese primer momento en el cual abrimos los ojos y comenzamos a estirar los brazos, las piernas y nuestros órganos se van “despertando” también para que, un momento después, estemos listos para levantarnos, irnos a trotar y comenzar el día con mucha energía.

El sistema operativo es la plataforma que permite que un computador esté en condiciones de ser utilizado. En el mercado encontramos computadores con diferentes plataformas para funcionar. Las más comerciales son Macintosh, Linux y Windows. El mismo programa que usted construye en Java en una plataforma sirve para otra plataforma. ¿Por qué se hace hincapié en este detalle? Porque, de no ser así, es posible que tuviéramos que hacer un mismo programa en tres versiones: una para Macintosh, una para Linux y otra para Windows, como sucede con otros lenguajes de programación. Con Java no pasa eso.

## 1.2 Un poquito de historia

Tenga en cuenta que el lenguaje de programación Java fue el aporte de los ingenieros James Gosling y Patrick Naughton quienes hacia 1991 quisieron cumplir la misión que les habían encomendado en Sun MicroSystems, empresa para la cual trabajaban, y que consistía en desarrollar un lenguaje de programación que permitiera la interacción con electrodomésticos que, a su vez, fueran diseñados para tal fin.

El objetivo era que los electrodomésticos que aparezcan nuevos en el mercado puedan funcionar con el software de los que ya existen. La condición principal para este desarrollo era que el lenguaje de programación que se diseñara no estuviera atado a la arquitectura de los dispositivos. Debe aclararse que en la actualidad el lenguaje de programación Java es propiedad de la empresa Oracle, famosa por su sistema de gestión de bases de datos, la cual es posiblemente la más popular en el mundo actual.

Con la aparición de la Internet hacia 1995 se potenció la inmensa utilidad del lenguaje de programación Java dado que sus características de diseño parecían ajustarse perfectamente a las necesidades de la telaraña mundial. A partir de esta articulación entre el lenguaje y la nueva tendencia mundial en informática como era la Internet se empezaron a desarrollar aplicaciones que podían ser independientes de las máquinas que estuvieran conectadas pero que las interconectaba de manera efectiva y permitía que visualizaran la misma información, precisamente, por la independencia entre el lenguaje y las diferentes plataformas.

### 1.3 ¿Qué hace diferente a Java?

Java es un lenguaje que ha marcado un hito en la historia de la programación fundamentalmente por lo siguiente:

- Es un lenguaje de programación que prescinde, de manera explícita, de las características más complejas de otros lenguajes de programación. Eso lo hace más sencillo en lo que se refiere a su diseño sintáctico y semántico
- Es el lenguaje de programación orientado a objetos por excelencia. ¿Qué significa que sea orientado a objetos? Muy sencillo, en el mundo de la programación de computadores existen diferentes enfoques para resolver un problema computable. Cada uno de estos enfoques se conoce como un Paradigma de Programación. Uno de estos enfoques, que más se ha popularizado por su alta proximidad a la realidad que nos rodea, es la programación orientada a objetos (POO) que no es más que una aproximación a la realidad. Un objeto es algo que tiene algunos usos específicos y que se distingue por unas características, por ejemplo, un lápiz sirve para rayar, para señalar, para rascarse, para apoyarse, etc. De otra parte un lápiz tiene un tamaño, un peso, un color,

etc. Ahí están reflejados tanto los usos como las características (que en POO se conocen como métodos y atributos respectivamente). Pues bien, Java está diseñado para que el mundo de soluciones se construya basándose en métodos y atributos. Que usted aprenda cómo hacerlo es el objetivo de este libro. No se preocupe.

- Tiende a aprovechar al máximo las características de las diferentes plataformas de los computadores y, por esta razón, es un lenguaje que intenta ser de alto rendimiento, intenta brindar alta seguridad en sus aplicaciones (recuerde, intentar ser un lenguaje seguro, no significa que efectivamente lo sea) y trata de ser altamente adaptable
- Es un lenguaje completamente independiente de la plataforma en donde quieran ejecutar las aplicaciones
- También es un lenguaje de programación multihilo, es decir, se pueden programar varios procesos para que se ejecuten al tiempo (en paralelo) dentro de un mismo computador

#### 1.4 El primer programa

Lo primero que debe recordar es que las instrucciones propias de Java se deben escribir tal como se van aprendiendo. Si se nos va una letra mayúscula en donde debería ir una minúscula o viceversa, el compilador encontrará un error y no permitirá que el programa funcione como esperamos. Esto se conoce técnicamente como un lenguaje Case Sensitive.

En segundo lugar debe tener en cuenta que, por la misma naturaleza de que un lenguaje de programación no es nuestra lengua materna, conviene ser muy estético al momento de escribir un programa no solamente en Java sino en cualquier otro lenguaje. Sea muy estético en Java, le favorecerá en más de una ocasión para entender lo que ha escrito. Sin más rodeos, aquí le presento el primer programa en Java.

```
public class Clase1 {                                ← línea 1
    public static void main ( String args [] ) {      ← línea 2
        System.out.println("Mi primer programa");   ← línea 3
    }                                              ← línea 4
}
```

← línea 5

No se preocupe si lo que observa le parece inentendible pues ya verá como, después de leer lo que sigue, podrá asimilar lo que allí aparece. La línea 1 está indicando que se trata de un programa en Java que comienza con la

definición de una clase. ¿...y qué es una clase? Más adelante nos centraremos en este concepto. Por ahora, tenga en cuenta que cada uno de los programas que hagamos (en estos primeros capítulos) deberán comenzar por *public class* y a continuación el nombre que queremos que tenga nuestra clase (o lo que, por ahora, podríamos mencionar como nuestro programa). En este caso hemos llamado a nuestro programa *Clase1*. Observe que después del nombre de la clase se abre una llave.

El objetivo de las llaves que abren es iniciar un bloque de instrucciones que se cierra con la respectiva llave que se cierra (o sea con su equivalente). Por ejemplo, la llave que se abre en la línea 1 corresponde a la que se cierra en la línea 5, pues la 1<sup>a</sup> llave que se abre corresponde a la última llave que se cierra. En la línea 2 aparecen las palabras *public static void main*. Esas palabras por ahora siempre debe escribirlas. Más adelante profundizaremos en ellas y veremos las otras opciones que tiene esta línea. Después de *main* aparece un paréntesis que abre dentro del cual están las palabras *String args[ ]* y al final de la línea 2 se cierra con el equivalente paréntesis.

En la línea 3 aparece la instrucción *System.out.println*(“*Mi primer programa*”); con la cual se le está indicando al lenguaje Java que ponga en pantalla la frase “*Mi primer programa*”. Esto quiere decir que para poner un título en pantalla simplemente escriba *System.out.println* y a continuación escriba, entre comillas dobles, lo que usted quiere que aparezca en pantalla. No se le vayan a olvidar las comillas dobles (por ahora). La parte final de esta instrucción (o sea *println*) le indica al computador que debe poner el título en pantalla y pasar a la siguiente línea. Al finalizar la línea aparece un signo punto y coma ( ; ) que es un signo obligatorio para indicar el final de cada sentencia. Sobre esto también profundizaremos más adelante, no se preocupe.

En la línea 4 aparece la llave de cierre equivalente a la llave que abre en la línea 2 y en la línea 5 aparece la llave de cierre equivalente a la llave que abre en la línea 1, es decir, la 1<sup>a</sup> llave se cierra de última y la penúltima llave se cierra de penúltima. La utilización de estas llaves es obligatoria pues cada una indica algo: la llave que aparece en la línea 1 indica que allí comienza la definición de la clase y la llave que aparece en la línea 2 indica que allí comienza la definición del método principal. Solo tenga estos conceptos pro ahora. Más adelante profundizaremos en ellos.

Para resumir, por ahora, sepa que esta es la estructura más simple de un programa en Java pues como ejemplo tenemos el más sencillo de todos. Tenga en cuenta que todo programa en Java debe estar enmarcado, al menos, dentro de una clase y que el modificador *public* indica que este programa puede ser ejecutado por cualquiera.

#### 1.4.1 Ejercicios propuestos

- a. Escribir un programa en Java que muestre en pantalla su nombre completo
- b. Escribir un programa en Java que muestre en pantalla su edad
- c. Escribir un programa en Java que muestre en pantalla su nombre completo seguido de su edad
- d. Escribir un programa en Java que muestre en pantalla su nombre completo y la edad pero que los muestre, dentro del mismo programa, en renglones diferentes
- e. Escribir un programa en Java que muestre en pantalla su nombre completo y la edad pero separados de tres renglones en blanco
- f. Escribir un programa en Java que muestre en pantalla su 1º nombre, su 2º nombre, su 1º apellido y su 2º apellido separados de una línea en blanco entre cada uno de ellos
- g. Escribir su 1º nombre de forma que cada letra quede en un renglón diferente
- h. Escribir su edad en letras (Ej. 25 = “Veinticinco”)
- i. Escribir el resultado de sumar  $5+4$
- j. Escribir el resultado de sumar mal  $5+4$

# 2 Lección

## Variables y salidas de datos



## 2.1 Definiciones

Una de los primeros conceptos que se deben conocer en programación, en los paradigmas pertinentes, es el concepto de lo que es una Variable. Como *variable* se define un espacio de memoria en donde se pueden almacenar datos (en la memoria principal) y cuyo contenido podría cambiar durante una misma ejecución de un programa. Las variables podríamos asociarlas como las cajitas en donde podemos guardar los datos con los cuales necesitamos que los programas trabajen.

Tanto en la programación imperativa (que es uno de los paradigmas de programación) como en la programación orientada a objetos (que es el paradigma que trataremos en este libro), el concepto de *variable* constituye uno de los factores más importantes y vale la pena que usted se tome el tiempo para capitalizar las ventajas de almacenar la información.

Para utilizar una variable, debe definirse qué tipo de dato se va a almacenar. Un tipo de dato es la forma como se reconocen las características de la información que nos rodea. No es lo mismo la cantidad de hijos de una persona que su estatura puesto que en el primer dato no hay posibilidad de decimales (no existen 2.3 hijos); en cambio, la estatura de una persona normalmente viene acompañado de decimales (una persona puede medir 1.71m).

Este detalle, que pareciera ser tan insignificante, tiene mucha incidencia en la forma como se almacena la información y por eso, para utilizar una variable, ésta debe declararse lo cual significa que se le debe especificar primero el tipo de dato a almacenar. Haga de cuenta que es como si alguien le pide a usted que consiga una caja, posiblemente usted preguntará ¿qué va a empacar en ella? para poder saber de qué tamaño debe ser.

Para declarar una variable primero se escribe el tipo de datos y después el nombre que queremos ponerle a la variable. De los detalles correspondientes al tipo de datos nos ocuparemos en el siguiente numeral. El nombre de una variable puede ser cualquiera siempre que tengamos en cuenta que a) debe ser un nombre único en todo el programa (al menos en esta primera fase de aprendizaje); b) no debe contener, en su nombre, caracteres especiales como \$, %, &, #, etc. y c) conviene que sea lo más

mnemónico posible, es decir, que el nombre de la variable permita recordar fácilmente lo que almacena. De esta forma, cuando en un programa usted tenga muchas variables, le será fácil saber cuál almacena qué. Por ejemplo, si una variable va a almacenar un salario mensual entonces podría llamarse *SalMens* y de esta forma usted podrá acordarse fácilmente qué contiene dicha variable.

Un detalle obligatorio en el lenguaje de programación Java es que toda variable debe ser inicializada. Esto quiere decir que cuando se realiza la declaración de una variable, allí mismo o a renglón seguido debe asignársele un valor para poder utilizarla. La idea es que se inicialice la variable antes de ser utilizada.

Antes de pasar al siguiente ítem, en donde explicaremos los tipos de datos que se utilizan en Java, debe tenerse en cuenta que una constante (en programación) es un tipo de variable cuyo contenido no cambia durante una misma ejecución de un programa bien por razones conceptuales (como el caso de la constante  $\pi$ ) o bien por razones algorítmicas como las constantes que pueden utilizarse en un determinado programa para definir ciertos parámetros preestablecidos (como el caso del valor del IVA en ciertos programas).

## 2.2 Tipos de Datos

¿Por qué existen los tipos de datos? La respuesta es muy sencilla, para poder definir apropiadamente las variables pues los tipos de datos son los que dicen qué puede almacenar cada variable. Debemos de recordar que sin variables será muy difícil programar bajo la perspectiva de la programación orientada a objetos. En Java existen cuatro categorías de tipos de datos: los datos que no tienen punto decimal, los datos que tienen punto decimal, los datos que involucran caracteres de todo tipo y los datos que permiten almacenar un valor *Verdadero* o *Falso*.

Los datos que no involucran punto decimal se conocen como datos de tipo *Entero*. Por su parte, los datos que involucran punto decimal (y algunas veces decimales) se conocen como datos de tipo *Real*. Los datos que involucran caracteres se conocen como datos de tipo *Carácter* y los datos que pueden almacenar una respuesta lógica se conocen como datos de tipo *Booleano*.

Ahora bien, debido a que los datos enteros que describen el mundo que nos rodea son diferentes. Tenga en cuenta que la cantidad de hijos de una persona, que es un dato entero, podría ser (máximo) 20 en un caso muy prolífico, sin embargo, la distancia de la tierra al sol (que también es otro dato entero) es del orden de miles de millones. En virtud de esto se han creado cuatro tipos de datos enteros:

<b>Entero</b>	<b>Memoria</b>	<b>Capacidad</b>	<b>Rango de valores posibles</b>
byte	1 bytes	$-2^8 \dots (2^8 - 1)$	-256 ... 256
short	2 bytes	$-2^{16} \dots (2^{16} - 1)$	-65536 ... 65535
int	4 bytes	$-2^{31} \dots (2^{31} - 1)$	-2.147.483.648 ... 2.147.483.647
long	8 bytes	$-2^{64} \dots (2^{64} - 1)$	-18446744073709551616 ... 18446744073709551615

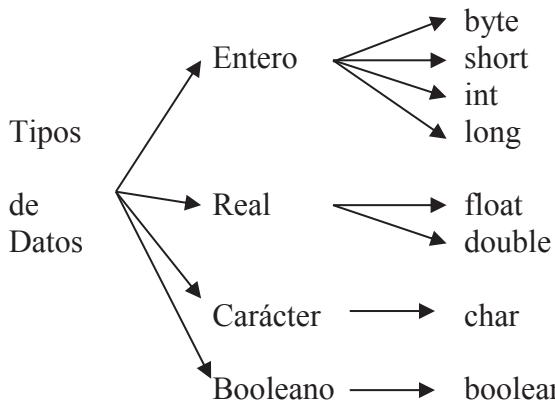
En esta tabla se pueden observar las palabras reservadas que identifican cada una de las vertientes del tipo de datos Entero. Así mismo se observa la cantidad de bytes que se requieren para almacenar cada tipo de datos (un byte equivale a 8 bits y 1 bit es igual a un 1 ó un 0). En la columna rotulada como Capacidad se observa el rango posible de almacenamiento y en la última columna se observan los valores posibles a almacenar.

Como se puede observar, la diferencia entre los diferentes tipos de datos enteros radica en su capacidad de almacenamiento. Debe tenerse en cuenta que un dato como el número 200 puede almacenarse en una variable tipo *byte*, *short*, *int* o *long*. Sin embargo, un dato como 1.000.000 podrá almacenarse en una variable tipo *int* o *long*. La idea con los tipos de datos es que se utilice lo más eficientemente posible la memoria principal del computador para que no se desperdicie.

Por su parte los datos Reales, es decir, los que involucran decimales (que también se conocen como datos de punto flotante) pueden ser de dos tipos según la cantidad de decimales que se quieren obtener. Si se quieren almacenar datos de tipo real con hasta 8 decimales de precisión entonces la variable debe declararse de tipo *float*. Si, en cambio, lo que se quiere es almacenar un dato que tenga hasta 16 decimales de precisión, entonces debe declararse la variable tipo *double*.

Finalmente las variables de tipo carácter ocupan un byte y se declaran con la palabra reservada *char* y las variables booleanas se declaran con la

palabra reservada *boolean*. Para hacer un poco más entendible esta clasificación, el siguiente esquema será de gran utilidad.



### 2.3 Comentarios

Cuando se escribe un programa en un lenguaje de programación, debido a que éste no es nuestro lenguaje natural, es posible que lo que queda escrito no sea tan fácilmente entendible. Por esta razón, el código de los programas debe ser comentado para que, en cualquier momento, podamos entender la lógica con la cual se construyó. Los comentarios permiten que el código se pueda documentar de manera que sea muy fácil entenderlo.

En Java, cuando se requiere escribir un comentario de una sola línea, debe comenzarse con los signos // y cuando este comentario incluye varias líneas entonces debe iniciarse con /\* y cerrarse con \*/. Debe tenerse en cuenta que los comentarios, sea los que fueren, son completamente ignorados por el intérprete de Java. De esta forma un código como:

```
public class Clase1 {  
    public static void main ( String args [] ) {  
        System.out.println("Mi primer programa");  
    }  
}
```

Podría ser más entendible si se escribiera de la siguiente forma:

*/\* Este es el primer programa que se construye en Java para explicar el buen uso de los comentarios de forma que sea más entendible \*/*

```
public class Clase1 {                                // clase ppal
    public static void main ( String args [ ] ) {      // método ppal
        // Se muestra el título Mi primer programa
        System.out.println("Mi primer programa");
    }                                              // Fin método ppal
}
```

// Fin clase ppal

Como usted puede ver, la utilización de los comentarios facilita la comprensión del programa. Es posible que en este caso, debido a la altísima simplicidad de este programa, aparentemente no sean tan necesarios los comentarios pero tenga en cuenta que apenas estamos comenzando nuestro contacto con el lenguaje de programación Java y, a medida en que avancemos, veremos la gran utilidad de ir documentándolo para que sea más entendible y para que, en cualquier momento, podamos recordar cómo fue que lo construimos.

Tenga en cuenta que muchas aplicaciones desarrolladas en Java o en cualquier otro lenguaje de programación deben ajustarse con el tiempo bien porque el usuario así lo requiere, bien porque la tecnología así lo exige o bien porque aparecen normas o leyes que obligan a cambiar ciertos procesos. En cualquier caso la documentación será vital para que, después de mucho tiempo, podamos retomar un programa, entenderlo y ajustarlo de acuerdo a las necesidades del momento.

No es conveniente excederse en la ubicación de los comentarios y en la documentación pero siempre acostúmbrese, como lo haremos en este libro, a escribir programas documentados apropiadamente de manera que tenga, en un mismo documento, una versión tecnológica de su solución (que corresponde al código escrito en Java) y una versión informal de la lógica que utilizó (los comentarios sobre el programa). De esa manera, cuando tenga que reescribir su programa en otro lenguaje de programación se le hará más sencillo de lo que usted se imagina.

## 2.4 Reglas de las Variables

Para utilizar eficientemente las variables, en sus programas, tenga en cuenta las siguientes reglas y todo se le hará más fácil.

No.	Regla	Ejemplo	Explicación
1	En una variable un dato se guarda usando el signo =	byte num; num = 8	Se declara la variable num de tipo byte. Se almacena 8 en la variable num
2	En una variable solo puede almacenarse UNO y solo UN dato a la vez	short n1;	Se declara la variable con nombre n1 de tipo short
		n1 = 5;	Se almacena en n1 el valor 5 y es el único valor que queda almacenado
		n1 = 7;	Se almacena el n1 el valor 7, por lo tanto el valor 5 se borra
3	Cuando se hace referencia al nombre de una variable, en realidad se está haciendo referencia al contenido de la variable	n1 = 9;	Se almacena 9 en n1, el 7 se borra
		byte num;	Se declara una variable llamada num de tipo byte
		num = 4	Se almacena el valor 4 en la variable num
4	El último dato que se almacenó en la variable, es el dato que queda	System.out.println(num)	Se muestra en pantalla num que, por ser una variable, muestra el contenido de num o sea 4
		short n3;	Se declara una variable llamada n3 de tipo short
		n3 = 4	Se almacena en la variable n3 el número 4
		n3 = 6	Se almacena en la variable n3 el número 6 y por lo tanto se borra el número 4 almacenado previamente
		System.out.println(n3)	Se muestra en pantalla n3 que, por ser una variable, muestra el contenido de n3 o sea 6

## 2.5 Salida de información

Una de las principales características de un programa consiste en la posibilidad de mostrar información que sea *entendible* y *útil*. “Entendible” significa que podamos entender esa información; por ejemplo, si se nos entrega información escrita en Mandarín (el idioma que usan en China) seguramente no lo entenderíamos. Que sea “Útil” significa que nos podamos servir de dicha información para darle un uso específico.

No es obligatorio que toda información que se despliegue en el computador sea útil (pues yo podría darle a usted mi número de celular y para usted eso podría ser insignificante) pero cuando se habla de programas por computador lo que se busca es que la información que se muestre realmente lo sea y le podamos dar un uso específico. Aunque ya la hemos utilizado ahora vamos a ver cómo le podemos sacar mayor provecho a la “instrucción” `System.out.println`. Lea detenidamente el siguiente cuadro y verá de qué le hablo.

Si usted quiere...	Entonces hágalo así
...mostrar un título	<code>System.out.println("este es mi título");</code>
...mostrar el contenido de una variable	<code>System.out.println(num);</code>
...mostrar un título y después el contenido de una variable	<code>System.out.println("Mi edad es "+edad);</code>
...mostrar dos títulos en renglones diferentes	<code>System.out.println("Hola amigos"); System.out.println("¿Cómo están?");</code>
...mostrar el contenido de dos variables en renglones diferentes	<code>System.out.println(num1); System.out.println(num2);</code>

Tenga en cuenta que los títulos y las variables usados en la tabla anterior son solamente ejemplos para que usted pueda entender su uso. Sobra decir que, en su lugar, podrían ir cualquier otro título o cualquier otra variable. Cuando queremos que los datos aparezcan presentados de una manera, todavía, más estética, entonces recurrimos a dos caracteres muy especiales:

- \t Que tabula los datos escribiéndolos separados a la derecha
- \n Que pasa a la siguiente línea en la pantalla

De esta manera la siguiente tabla ilustra otros tipos de presentaciones que, en más de una ocasión podrán ser de gran utilidad. Recuerde que el objetivo de un programa es que, al final, presente información que sea entendible y útil, por lo tanto, estos dos caracteres le ayudarán notablemente a cumplir con la primera condición, o sea, que sean entendibles.

```
public class Clase1 { // clase ppal
    public static void main ( String args [ ] ) { // método ppal
        byte Edad; // Se declara un variable tipo byte
        Edad = 36; // Se almacena 36 en la variable Edad
        System.out.println("Dato = " + Edad );
        // Se muestra en pantalla
    }
}
```

De acuerdo a los comentarios, según los cuales además de declarar una variable de tipo byte (a la que se le ha puesto por nombre Edad), también se almacena el valor 36, en pantalla se mostrará lo siguiente:

Dato = 36

Ahora veamos esta línea de salida:

```
System.out.println("Dato = \t\t" + Edad ); // Se muestra en pantalla
```

La salida que generará en pantalla será la siguiente:

Dato = 36

Debido a que el carácter \t tabula dos veces antes escribir el contenido de la variable Edad. Tabular es desplazarse 7 espacios en blanco hacia la derecha.

Si se tuviera el siguiente programa Java:

```
public class Clase1 { // clase ppal
    public static void main ( String args [ ] ) { // método ppal
```

```
byte Edad, Dato;      // Se declaran dos variables tipo byte
Edad = 36;           // Se almacena 36 en la variable
Edad
Dato = 18;           // Se almacena 18 en la variable Dato
// Se muestran en pantalla los valores almacenados
System.out.println("Dato = " + Edad + " " + Dato );
}
}
```

La última línea, al ejecutarse, mostrará en pantalla lo siguiente:

Dato = 36 18

Escribe el título “Dato = “, a continuación tabula dos veces para escribir el contenido de la variable Edad (que es el valor 36). Luego tabula otras dos veces para mostrar en pantalla lo que almacena la variable Dato (que corresponde al valor 18).

## 2.6 Ejercicios Propuestos

### PRIMERA PARTE

Subraye el tipo de dato que corresponde al dato (en algunos casos pueden existir varias opciones).

DATO	TIPOS DE DATOS							
25	byte	short	int	long	float	double	char	boolean
90000	byte	short	int	long	float	double	char	boolean
8	byte	short	int	long	float	double	char	boolean
30000	byte	short	int	long	float	double	char	boolean
100000	byte	short	int	long	float	double	char	boolean
500000000	byte	short	int	long	float	double	char	boolean
3.45	byte	short	int	long	float	double	char	boolean

0.000000078	byte	short	int	long	float	double	char	boolean
@	byte	short	int	long	float	double	char	boolean
True	byte	short	int	long	float	double	char	boolean
93	byte	short	int	long	float	double	char	boolean
80000	byte	short	int	long	float	double	char	boolean
7	byte	short	int	long	float	double	char	boolean
40000	byte	short	int	long	float	double	char	boolean
200000	byte	short	int	long	float	double	char	boolean
4000000000	byte	short	int	long	float	double	char	boolean
9.84	byte	short	int	long	float	double	char	boolean
8.948756859	byte	short	int	long	float	double	char	boolean
T	byte	short	int	long	float	double	char	boolean
False	byte	short	int	long	float	double	char	boolean

## SEGUNDA PARTE

Determinar el valor final de las variables en los siguientes ejercicios:

a.

```
int A, B, C;  
A = 10;  
B = 12;  
C = 15;  
A = B;  
B = C;  
C = A;  
A = B;  
B = C;  
C = A;
```

b.

```
int A, B, C;  
A = 13;  
B = 15;  
C = 18;  
A = B;  
B = C;  
C = A;  
A = B;  
B = C;  
C = A;
```

c.

```
int A, B, C;  
A = 20;  
B = 22;  
C = 25;  
A = C;  
B = A;  
C = B;  
A = C;  
B = A;  
C = B ;
```

d.

```
int A, B, C;  
A = 3;  
B = 4;  
C = 5;  
A = B;  
B = C;  
C = A;  
A = B;  
B = C;  
C = A;
```

e.

```
int A, B, C;  
A = 30;  
B = 42;  
C = 55;  
A = B;  
B = C;  
C = A;  
A = B;  
B = C;  
C = A;
```

f.

```
int A, B, C;  
A = 1;  
B = 2;  
C = 3;  
A = B;  
B = C;  
C = A;  
A = B;  
B = C;  
C = A;
```



# **3 Lección**

## **Constantes, expresiones y operadores**



### 3.1 Constantes

Comencemos por saber que en programación absolutamente todo puede cambiar en cualquier momento por esta razón es importante tener en cuenta que, sólo en programación, una constante se define como una variable a la cual no se le está permitido cambiar su contenido dentro de una misma ejecución de un programa. Las constantes son de gran utilidad porque permiten que ciertos valores se mantengan fijos ya que esa es su mayor utilidad. Por ejemplo, para convertir pulgadas en centímetros debemos multiplicar por 2.56 debido a que 1 pulgada equivale a 2,56 centímetros. De la misma manera para convertir centímetros a pulgadas debemos dividir entre 2,56 por la misma razón. Entonces, dado que este valor no cambia, lo podemos concebir como una constante dentro de un programa en el cual vayamos a utilizar la conversión entre los dos sistemas de medidas.

Para declarar una constante en Java todo lo que tenemos que hacer es hacer lo mismo que si fuéramos a declarar una variable normal pero antecediendo dicha declaración por la palabra *final*. De esta manera, si queremos declarar la constante para convertir de pulgadas a centímetros (a la cual llamaremos *pulgacent*), y teniendo en cuenta su naturaleza de número real debido a que tiene decimales, lo podremos hacer de la siguiente forma:

*final double pulgacent;*

Después de esta declaración, podríamos escribir el valor que queremos que se asuma como constante.

*pulgacent = 2.54;*

Para ser un poco más prácticos podríamos hacer todo en una línea, es decir, declarar la constante y asignarle el valor que va a almacenar durante todo el programa.

*final double pulgacent = 2.54;*

Conviene decir que toda constante que se declare dentro de un programa está inhabilitada para que, durante la ejecución de dicho programa, se cambie de valor.

### 3.2 Expresiones

Una expresión es una operación que ese escribe al lado derecho de un signo igual (=) para que su resultado se almacene en la variable que se encuentra al lado izquierdo del mismo signo igual (=). Esta es la forma como se almacena en una variable el resultado de una expresión.

Así como la instrucción

*Num = 18;*

le indica a Java que se almacene el valor 18 en una variable llamada *Num* que, con seguridad hemos declarado previamente de algunos de los tipos enteros, así también si tuviéramos la expresión:

*Num = 5 + 10;*

le indicará a Java que sume los valores 5 y 10 y que, su resultado lo almacene en la variable *Num*. Para esto debe tenerse en cuenta que, cuando se trata de almacenar el resultado de una expresión, primero se obtiene dicho resultado y luego se almacena en la variable que se encuentra al lado izquierdo del signo =.

Si tuviéramos el programa

```
public class Clase1 { // clase ppal
    public static void main ( String args [ ] ) // método ppal
    {
        int N1, N2, N3; ← Línea 1
        N1 = 5; ← Línea 2
        N2 = 6; ← Línea 3
        N3 = 4; ← Línea 4
        N2 = N1 + N2; ← Línea 5
        N3 = N1 + N2 + N3; ← Línea 6
        System.out.println("Resultado = " + N3); ← Línea 7
    }
}
```

Entonces su ejecución sería muy sencilla y entendible. Veamos la explicación línea por línea.

Línea	Explicación
1	Se declaran tres variables, todas de tipo int $byte N1, N2, N3;$
2	Se almacena en la variable N1 el valor 5 $N1 = 5;$
3	Se almacena en la variable N2 el valor 6 $N2 = 6;$
4	Se almacena en la variable N3 el valor 4 $N3 = 4;$
5	Se almacena en N2 el resultado de sumar el contenido de las variables N1 y N2. Como N1 almacena el valor 5 y N2 almacena el valor 6, entonces la expresión $N2 = N1 + N2;$ Equivale a tener $N2 = 5 + 6;$ Lo cual permite que en la variable N2 quede almacenado el valor 11. ¿...y qué pasa con el 6 que estaba almacenado anteriormente en N2? Inmediatamente se borra pues ya se almacenó otro valor.
6	Se almacena en la variable N3 el resultado de sumar los valores almacenados en las variables N1, N2 y N3. Como N1 contiene el valor 5, N2 contiene el valor 11 y N3 almacena el valor 4 entonces la expresión $N3 = N1 + N2 + N3;$ Equivale a $N3 = 5 + 11 + 4;$ Por lo tanto en la variable N3 se almacena el valor 20
7	Se muestra en pantalla el resultado de la última operación acompañado de un título respectivo

	<pre>System.out.println("Resultado = " + N3);</pre>
	En pantalla saldría
	<i>Resultado = 20</i>

### 3.3 Operadores

Los operadores son signos que nos permiten representar, de una forma simple y sencilla, una determinada operación. Cuando usted ve el signo + inmediatamente sabe que se refiere a una suma. Pues bien, Java tiene una cantidad de operadores que simplificarán la representación de las operaciones.

Tipo	Operación	Operador	Ejemplo de Uso	Descripción
<b>Aritmético</b> Estos operadores permiten expresar operaciones simples	Suma	+	$A = B + C;$	Calcula la suma de B y C y la almacena en A
	Resta	-	$A = A - B;$	Calcula la resta entre A y B y la almacena en A
	Multiplicación	*	$A = A * B;$	Calcula el producto entre A y B y lo almacena en A
	División	/	$A = A / B;$	Calcula el cociente de A sobre B y lo guarda en A.
<b>Relacionales</b> Estos operadores permiten obtener respuestas Verdadero o Falso comparando el contenido de las variables	Mayor que	>	( $A > B$ )	Si el contenido de A es mayor que el de B, retorna Verdadero. Si no, retorna Falso
	Menor que	<	( $A < B$ )	Si el contenido de A es menor que el de B retorna Verdadero. Si no, retorna Falso
	Mayor o Igual a	$\geq$	( $A \geq B$ )	Si el contenido de A es mayor o igual que el de B retorna un Verdadero. Si no retorna Falso
	Menor o Igual a	$\leq$	( $A \leq B$ )	Si el contenido de A es menor o igual que el contenido de B retorna un Verdadero. Si no lo es, retorna Falso
	Igual a	$=$	( $A == B$ )	Si el contenido de A es igual al contenido de B retorna Verdadero. Si no, retorna Falso

	Diferente de	$!=$	$(A > B)$	Si el contenido de A es diferente del contenido de B retorna Verdadero. Si no Falso
<b>Booleanos</b> Estos operadores permite conectar expresiones de relación	And	$&&$	$(A > B) \&\& (B > C)$	Retorna Verdadero si las dos expresiones que conecta son Verdaderas. En cualquier otro caso, retorna Falso.
	Or	$  $	$(A > B)     (B > C)$	Retorna Verdadero si alguna de las expresiones que conecta es Verdadera. En otro caso, es retorna Falso.
	Not	!	$! (A > B)$	Invierte el sentido lógico de una expresión relacional. Si es Verdadero, retorna Falso. Si es Falso, retorna Verdadero.
<b>Resumidos</b> Simplifican la forma de expresar operaciones simples	Incremento	$++$	$N++;$	Equivale a escribir $N = N + 1;$
	Decremento	$--$	$N--;$	Equivale a escribir $N = N - 1;$
	Suma	$+=$	$A += 5;$	Equivale a escribir $A = A + 5;$
	Resta	$-=$	$A -= 3;$	Equivale a escribir $A = A - 3;$
	Multiplicación	$*=$	$B *= 5;$	Equivale a escribir $B = B * 5;$
	División	$/=$	$C /= D;$	Equivale a escribir $C = C / D;$
<b>No Numéricos</b> Se aplican para unir cadenas o cadenas con datos	Concatenación	$+$	Permite unir cadenas como cuando se escribe <code>System.out.println("Dato = " + Edad);</code>	

En el caso de la división se cumple que el resultado tendrá o no decimales dependiendo del tipo de datos de A y B. Si son de tipo Entero, no tienen decimales. Si son de tipo Real, tienen decimales. Tenga en cuenta que el operador igual que se usa para comparar se escribe con dos signos igual seguidos ( $==$ ).

### 3.4 Ejercicios Propuestos

Determinar el valor final de las variables en los siguientes ejercicios:

a.

```
int A, B, C;  
A = 10;  
B = 12;  
C = 15;  
A = A + B;  
B = B + C;  
C = C + A;  
A = A + B;  
B = B + C;  
C = C + A;
```

b.

```
int A, B, C;  
A = 13;  
B = 15;  
C = 18;  
A = A + B;  
B = B - C;  
C = C + A;  
A = A - B;  
B = B + C;  
C = C - A;
```

c.

```
int A, B, C;  
A = 20;  
B = 22;  
C = 25;  
A = A + B + C;  
B = B + C - A;  
C = C + A + B;  
A = A + B - C;  
B = B + C + A;  
C = C + A - B ;
```

d.

```
int A, B, C;  
A = 3;  
B = 4;  
C = 5;  
A = A * B;  
B = B / C;  
C = C * A;  
A = A / B;  
B = B * C;  
C = C / A;
```

e.

```
int A, B, C;  
A = 30;  
B = 42;  
C = 55;  
A = A - B;  
B = B * C;  
C = C + A;  
A = A * B;  
B = B - C;  
C = C + A;
```

f.

```
int A, B, C;  
A = 1;  
B = 2;  
C = 3;  
A = A + B;  
B = B - C;  
C = C - A;  
A = A + B;  
B = B - C;  
C = C + A;
```

**Lección  
4**

**Clase math y  
cambios de  
tipo**



#### 4.1 Algunos métodos de la Clase Math

Algunas operaciones no se pueden expresar de forma tan simple como se ha hecho hasta ahora escribiendo un operador entre dos operandos. Una operación como la raíz cuadrada requiere un tratamiento un poco diferente pero igual de sencillo, como todo en Java. La tabla siguiente muestra algunos ejemplos y cómo utilizarlos.

Operación	Uso en Java	Descripción
Conversión a grados	Math.toDegrees(Valor)	Retorna el equivalente en grados de un <i>Valor</i> expresado en radianes
Conversión a radianes	Math.toRadians(Valor)	Retorna el equivalente en radianes de un <i>Valor</i> expresado en grados
Coseno	Math.cos(ángulo)	Retorna el valor del Coseno del <i>ángulo</i> especificado
Exponente de Euler	Math.exp(Valor)	Retorna el resultado de elevar el número e al valor que se envíe como parámetro, o sea $e^x$ siendo x el <i>Valor</i> escrito
Logaritmo	Math.log(Valor)	Retorna el logaritmo natural (base e) del valor escrito
Logaritmo Natural	Math.log10(Valor)	Retorna el logaritmo (base 10) del valor escrito
Número Aleatorio	Math.random()	Retorna un número cualquier comprendido entre 0.0 y 1.0
Número $\pi$	Math.PI	Retorne el valor del número $\pi$
Número $e$	Math.E	Retorne el valor de la constante $e$
Potenciación	Math.pow(base, exp)	Retorne el valor de elevar una <i>base</i> a un <i>exponente</i>
Raíz Cuadrada	Math.sqrt(Valor)	Retorna el valor equivalente a la raíz cuadrada del <i>Valor</i> que se escriba entre paréntesis o del contenido de la variable que allí se escriba
Raíz Cúbica	Math.cbrt(Valor)	Retorna la raíz cúbica del valor que se escribe como parámetro
Redondeo	Math.round(número)	Redondea el <i>número</i> que se especifique, es decir, retorna el número entero más cercano al valor especificado (siempre que este tenga decimales)
Seno	Math.sin(ángulo)	Retorna el valor del Seno del <i>ángulo</i> que se especifique

Tangente	Math.tan(ángulo)	Retorna el valor de la Tangente del <i>ángulo</i> que se especifique
Valor Absoluto	Math.abs(Valor)	Retorna el valor absoluto del valor que se envía como parámetro, independiente del tipo de dato que sea

Una de las destrezas que vale la pena adquirir en un lenguaje tan amplio como Java es el conocimiento de las clases que ya están construidas y, especialmente, de sus métodos pues mucho de lo que queremos hacer ya está hecho. No se vaya a dar a la tarea de aprender de memoria todas las clases y todos los métodos pues son demasiados y, además, es completamente innecesario. Si usted consulta la dirección <https://docs.oracle.com/javase/7/docs/api/> podrá encontrar allí cada clase con sus respectivos métodos y, entonces, podrá darle un uso más práctico a lo que Java ya tiene construido. Si quiere encontrar las clases y sus métodos usando cualquier navegador, simplemente busque “Java API” y haga click en el primer enlace. Lo llevará al mismo sitio web.

La clase *Math*, por ejemplo, tiene muchos métodos que serán muy útiles en la construcción de programas. Tenga en cuenta la letra mayúscula de la clase *Math* pues si escribe *math* (con minúscula) le sale un error ya que Java es *Case Sensitive* o sea que diferencia las letras mayúsculas de las letras minúsculas. Si usted se está preguntando ¿cómo usar estos métodos?, no se preocupe, aquí tiene algunos ejemplos.

## 4.2 Uso de la Clase Math

Con el ánimo de mostrar el uso de estos métodos, se presenta a continuación el siguiente programa Java. Revíselo detenidamente y compruebe la forma como está escrito. Para que el código sea más entendible para usted, revise los comentarios del programa, es decir, las líneas que comienzan por *//*.

```
public class ClaseMath {  
  
    public static void main(String args[]) {  
  
        int Valor = 130;  
  
        // Conversión a grados  
        System.out.println("Grados equivalentes--> " + Math.toDegrees(Valor));  
    }  
}
```

```
// Conversion a radianes
System.out.println("Radianes equivalentes-->" + Math.toRadians(Valor));

// Calculo de e elevado a la x
int x = 3;
System.out.println("e elevado a la " + x + " = " + Math.exp(x));

// Calculo del logaritmo natural
System.out.println("Logaritmo natural de " + x + " = " + Math.log(x));

// Calculo del logaritmo base 10
System.out.println("Logaritmo base 10 de " + x + " = " + Math.log10(x));

// Muestra un número aleatorio comprendido entre 0.0 y 1.0
System.out.println("Número Aleatorio = " + Math.random());

// Muestra el número PI
System.out.println("Número PI = " + Math.PI);

// Muestra el número E
System.out.println("Número E = " + Math.E);

// Elevar una base a un exponente
int base, expon;
base = 4;
expon = 3;
System.out.println(base+" elevado a la "+expon+" = "+Math.pow(base,expon));

// Calcular la raíz cuadrada
int Num = 30;
System.out.println("Raiz Cuadrada de " + Num + " = " + Math.sqrt(Num));

// Calcular la raíz cúbica
System.out.println("Raiz Cúbica de " + Num + " = " + Math.cbrt(Num));

// Redondeo de un número
float Dato = 45.765F;
System.out.println(Dato + " redondeado es = " + Math.round(Dato));

// Calculo del Seno, Coseno y Tangente de un ángulo
double dato = 35;
System.out.println("Seno de " + dato + " = " + Math.sin(dato));
System.out.println("Coseno de " + dato + " = " + Math.cos(dato));
System.out.println("Tangente de " + dato + " = " + Math.tan(dato));

// Cálculo del valor absoluto
int ValorN = -45;
```

```
System.out.println("Valor absoluto de " + ValorN + " = " +  
Math.abs(ValorN));  
}  
}
```

Cuando se ejecuta este programa Java, el resultado en la consola es el siguiente:

```
Grados equivalentes--> 7448.451336700702  
Radianes equivalentes-->2.2689280275926285  
e elevado a la 3 = 20.085536923187668  
Logaritmo natural de 3 = 1.0986122886681098  
Logaritmo base 10 de 3 = 0.47712125471966244  
Número Aleatorio = 0.42067676530486775  
Número PI = 3.141592653589793  
Número E = 2.718281828459045  
4 elevado a la 3 = 64.0  
Raiz Cuadrada de 30 = 5.477225575051661  
Raiz Cúbica de 30 = 3.1072325059538586  
45.765 redondeado es = 46  
Seno de 35.0 = -0.428182669496151  
Coseno de 35.0 = -0.9036922050915067  
Tangente de 35.0 = 0.473814720414451  
Valor absoluto de -45 = 45
```

Es de anotar que si se ejecutara de nuevo este programa Java, los resultados serían los mismos excepto en la línea que dice

```
Número Aleatorio = 0.42067676530486775
```

Puesto que cada vez que se ejecute, se generará un número aleatorio nuevo (un número aleatorio es un número al azar comprendido entre 0.0 y 1.0).

### 4.3 Cambios de tipo

En algunos casos se requiere convertir un tipo de dato en otro. Para ello se hace necesario anteponer al resultado que se espera, el tipo de dato en el cual se quiere convertir. Debe tenerse en cuenta que si el tipo de dato de una expresión ocupa más bytes que el tipo de dato al cual se quiere convertir, es muy posible que se pierda información. Si tenemos el siguiente conjunto de instrucciones:

```
public class Clase1 { // clase ppal
    public static void main ( String args [ ] ) { // método ppal
        double Num1, Num2;
        int Resultado;

        Num1 = 235465;
        Num2 = 435433;
        Resultado = Num1 + Num2;
    }
}
```

Nos generaría un error en la última línea pues se está tratando de guardar en una variable tipo *int* el resultado de sumar dos datos de tipo *double*. En este caso, el cambio de tipo de datos (también conocido como casting) se hace muy útil. Suponiendo que, por razones del programa, requerimos almacenar este resultado en una variable de tipo entero entonces lo hacemos de la siguiente manera:

```
public class Clase1 { // clase ppal
    public static void main ( String args [ ] ) { // método ppal
        double Num1, Num2;
        int Resultado;

        Num1 = 235465;
        Num2 = 435433;
        Resultado = (int) Num1 + Num2;
    }
}
```

Note usted que, a la suma de las variables *Num1* y *Num2*, se le antepone el tipo de dato en el cual se quiere convertir el resultado de dicha suma. Recuerde que es posible que se pierda información.

Finalmente recuerde que cuando quiera almacenar utilizar un dato de tipo *float* debe adicionar, al dato, la letra F.

*float R;*

$$R = 243.56F;$$

De esta forma, Java distingue que se trata de un dato de tipo *float* y lo procesa sin ningún problema.

#### 4.4 Ejercicios Propuestos

##### PRIMERA PARTE

- a. Construir un programa que convierta 2 radianes a su equivalente en grados
- b. Construir un programa que convierta 38 grados a su equivalente en radianes
- c. Construir un programa que muestre en pantalla el seno de 45
- d. Construir un programa que muestre en pantalla el coseno de 57
- e. Construir un programa que muestre en pantalla la tangente de 60
- f. Construir un programa que muestre en pantalla la tangente de 60 pero calculándola a partir de la expresión  $\tan(x) = \frac{\text{seno}(x)}{\text{coseno}(x)}$
- g. Construir un programa que calcule el valor de elevar el número e a la potencia 4
- h. Construir un programa que calcule el logaritmo natural (base e) de 50
- i. Construir un programa que calcule el logaritmo (base 10) de 30
- j. Construir un programa que muestre en pantalla 3 números aleatorios comprendidos entre 0.0 y 1.0
- k. Construir un programa que muestre en pantalla 3 números aleatorios comprendidos entre 1 y 10
- l. Construir un programa que muestre en pantalla 3 números aleatorios comprendidos entre 100 y 200
- m. Construir un programa que muestre en pantalla el resultado de sumar el número  $\pi$  y el número  $e$
- n. Construir un programa que muestre en pantalla el resultado de elevar 5 a la potencia 4
- o. Construir un programa que calcule la raíz cuadrada de 90
- p. Construir un programa que calcula la raíz cúbica de 100
- q. Construir un programa que redondee el número 45.678

##### SEGUNDA PARTE

- a. Construir un programa que muestre en pantalla la mitad de su edad

- b. Construir un programa que muestre en pantalla su nombre y a continuación, tabulado, muestre su edad
- c. Construir un programa que muestre en pantalla su nombre y después, tabulado tres veces, su primer apellido
- d. Construir un programa que muestre en pantalla el resultado de sumar, restar, multiplicar y dividir dos números que usted asigne a unas variables
- e. Construir un programa que muestre en pantalla la suma de las edades de sus 4 mejores amigos
- f. Construir un programa que muestre en pantalla la tabla de multiplicar del 2
- g. Construir un programa que muestre en pantalla los resultados de realizar las siguientes operaciones:

$$1 \times 2$$

$$1 \times 2 \times 3$$

$$1 \times 2 \times 3 \times 4$$

$$1 \times 2 \times 3 \times 4 \times 5$$

$$1 \times 2 \times 3 \times 4 \times 5 \times 6$$

- h. Construir un programa que muestre en pantalla la raíz cuadrada de un determinado valor
- i. Construir un programa que calcula el resultado de elevar  $5^8$
- j. Construir un programa que calcule la siguiente suma:

$$1^1 + 2^2 + 3^3 + 4^4 + 5^5$$



# 5 Lección

# Cadenas



## 5.1 Definición

Una cadena se define como un conjunto de caracteres que están delimitados por comillas dobles tanto al inicio como al final. Ejemplos de cadenas son las siguientes: “Libro”, “Omar Trejos”, “Calle 23 No. 15-51”, “567”. Por estar delimitadas por comillas dobles, estas cadenas no tienen ninguna otra interpretación que la de ser, simplemente, conjuntos de caracteres.

Esto quiere decir lo siguiente: la cadena “567” es diferente al número 567 debido a que el número 567 puede ser usado en operaciones aritméticas mientras que la cadena “567” no se puede usar en esas mismas operaciones, sin embargo, Java provee las posibilidades para que se pueda convertir la cadena “567” en el número 567. Eso lo veremos más adelante.

## 5.2 Declaración de Cadenas

En el lenguaje de programación Java no existe un tipo de datos para cadenas pues los tipos de datos son los que ya vimos. En Java, por ser un lenguaje eminentemente orientado a objetos, existe una clase para cadenas llamada *String*. Esta clase está conformada, entre otros elementos, por métodos que posibilitan el manejo de cadenas dentro de un programa. Por ejemplo, si queremos definir una variable de “tipo” cadena entonces lo deberemos hacer así:

```
String MiCadena;
```

La palabra “tipo” está escrita entre comillas porque en realidad no es un tipo de datos sino una Clase aunque debo admitir que, en términos prácticos, pareciera, para quien está comenzando con Java, como si fuera la misma cosa. Una vez declarada esta cadena, la podemos utilizar para asignarle algo y tenga en cuenta que ese “algo” tiene que ser una cadena. La instrucción

```
MiCadena = 345;
```

Generará un error debido a que 345 es un número entero y MiCadena es una variable “tipo” String (o sea cadena). En cambio, la asignación

*MiCadena = “Omar”;*

Es una asignación válida debido a que se está almacenando una cadena en una variable de “tipo” cadena. Ahora sí, pongámosle a las cosas el nombre que es. Cuando se declarar una variable con un “tipo” de datos que corresponde al nombre de una Clase (como en el caso de String) entonces esa variable en realidad se conoce como un *Objeto* o una *Instancia* de la clase *String*. Estos serán los términos que seguiremos usando para hacer honor a las características del lenguaje de programación Java.

Ahora bien, en algunas ocasiones se ha utilizado la palabra método que son parte de la esencia de la programación orientada a objetos. El diseño y desarrollo de los métodos, así como de los atributos, es lo que distingue a la programación orientada a objetos. Un método es, en palabras sencillas, un conjunto de instrucciones que permiten lograr un pequeño objetivo y que está asociado a una *Clase* ya definida. Un concepto muy parecido al de función (en la programación imperativa o en la programación funcional) pero esta vez con una estrecha e íntima relación con el concepto de *Clase*.

### 5.3 La clase String

En Java, los métodos asociados a las cadenas se encuentran en una clase llamada *String* y, aunque son muchos los métodos que tienen implementados, a continuación se relacionarán algunos de ellos y posteriormente se verá el uso de estos métodos en programas construidos en Java. Para utilizar cualquiera de estos métodos, se requiere anteponerles el nombre de la cadena (variable tipo *String*) con la cual se quiere usar determinado método.

Método	Descripción
length()	Retorna la longitud de una cadena
charAt(pos)	Retorna el carácter de una cadena que se encuentra en la posición <i>pos</i>
substring(ini, num)	Retorna la cadena que comienza en el carácter <i>ini</i> y que tiene una longitud de <i>num</i> caracteres. Tenga en cuenta que en Java la 1 <sup>a</sup> posición es la posición 0
equals(Cad)	Retorna Verdadero si la cadena de referencia y la cadena <i>Cad</i> son iguales. Mayúsculas y minúsculas las considera diferentes. Si las cadenas son diferentes retorna Falso

equalsIgnoreCase(cadena)	Retorna Verdadero si la cadena de referencia y la cadena <i>Cad</i> son iguales. Mayúsculas y minúsculas las considera iguales. Si las cadenas son diferentes retorna Falso
compareTo(cadena)	Compara lexicográficamente dos cadenas
concat(Cad)	Concatena la cadena <i>Cad</i> a la cadena de referencia
contains(Cad)	Retorna Verdadero si la cadena de referencia contiene a la cadena Cad
endsWith(Sufix)	Retorna Verdadero si la cadena de referencia termina en la cadena <i>Sufix</i>
isEmpty( )	Retorna Verdadero si la longitud de la cadena de referencia es igual a 0
startsWith(prefijo)	Retorna Verdadero si la cadena comienza con el prefijo <i>Prefix</i>
toCharArray( )	Convierte una cadena en un arreglo de caracteres. Más adelante trataremos este interesantísimo tema
toLowerCase( )	Convierte todas las letras de la cadena de referencia a minúsculas
toUpperCase( )	Convierte todas las letras de la cadena de referencia a mayúsculas

## 5.4 Uso de la Clase String

A continuación se presenta un programa en donde se utilizan los métodos para cadenas de forma que sea más claro para usted comprender la utilidad de este tipo de facilidades.

```
public class Cadenas1 {
    public static void main(String args[]) {

        String Cad1 = "Universidad";
        System.out.println("Cadena --> " + Cad1);

        // Calcula la longitud (número de caracteres) de una cadena
        System.out.println("Cant. de caracteres = "+Cad1.length());

        // Muestra un carácter en determinada posición de una cadena

        System.out.println("Posición 3, Caracter --> " + Cad1.charAt(3));
        System.out.println("Posición 7, Caracter --> " + Cad1.charAt(7));
        System.out.println("Posición 9, Caracter --> " + Cad1.charAt(9));

        // Muestra una subcadena que comienza en ini y tiene num caracteres
        System.out.println("Subcadena Inicio 4 Longitud 3 --> "+Cad1.substring(2,6));

        // Retorna Verdadero si dos cadenas son iguales
    }
}
```

```
String Cad2 = "Universo";
System.out.println(Cad2 + "Igual a " + Cad1 + "? " + Cad1.equals(Cad2));

// Retorna Verdadero si dos cadenas son iguales (asume mayúsc y minúsc iguales)
String Cad3 = "UnIvErSiDaD";
System.out.println(Cad3 + " Igual a " + Cad1 + "?" + Cad1.equalsIgnoreCase(Cad3));

// Comparación lexicográfica de dos cadenas
System.out.println(Cad1 + " comparada con " + Cad3 + "? " + Cad1.compareTo(Cad3));

// Concatena dos cadenas
System.out.println("Dos cadenas concatenadas --> " + Cad1.concat(Cad3));

// Verifica si una cadena está contenida en otra
String Cad4 = "ver";
System.out.println(Cad4 + " está contenida en " + Cad1 + "? " + Cad1.contains(Cad4));

// Verifica si una cadena finaliza con un sufijo
System.out.println("Cadena " + Cad1 + " Sufijo = idad " + Cad1.endsWith("idad"));

// Determina si una cadena está vacía
String Cad5 = "";
System.out.println("Cadena Vacia? " + Cad5.isEmpty());

// Verifica si una cadena comienza con un prefijo
System.out.println("Cadena " + Cad1 + " Prefijo = univ " + Cad1.startsWith("univ"));
    }
}
```

Al ejecutarse este programa Java, el resultado que aparece en la consola es el siguiente:

```
Cadena --> Universidad
Cant. de caracteres = 11
Posición 3, Caracter --> v
Posición 7, Caracter --> i
Posición 9, Caracter --> a
Subcadena Inicio 4 Longitud 3 --> iver
UniversoIgual a Universidad? false
UnIvErSiDaD Igual a Universidad?true
Universidad comparada con UnIvErSiDaD? 32
Dos cadenas concatenadas --> UniversidadUnIvErSiDaD
ver está contenida en Universidad? true
Cadena Universidad Sufijo = idad true
Cadena Vacía? true
Cadena Universidad Prefijo = univ false
```

## 5.5 Ejercicios Propuestos

- a. Construir un programa que muestre la longitud de las palabras “casa”, “amigo” y “universitario”
- b. Construir un programa que muestre el carácter que está en la posición 5<sup>a</sup>, 7<sup>a</sup> y 9<sup>a</sup> de la palabra “desencantamiento”
- c. Construir un programa que muestre la palabra “versi” tomado de la cadena “Universidad”
- d. Construir un programa que determine si las cadenas “estudiante” y “programación” son iguales
- e. Construir un programa que determine si las cadenas “amigo” y “amigo” son iguales
- f. Construir un programa que determine si las cadenas “Amigo” y “AmIgO” son iguales
- g. Construir un programa que determine si las cadenas “amigo” y “AMIGO” son iguales
- h. Construir un programa que concatene las cadenas “Uni” y “verso”
- i. Construir un programa que determine si la palabra “ver” está contenida en la cadena “Universidad”
- j. Construir un programa que determine si la cadena “enamorado” termina en la palabra “morado”
- k. Construir un programa que determine si la cadena “palabra” comienza con la cadena “pala”



# 6 Lección

## Entrada de datos



## 6.1 Paquetes, Clases y Métodos

Ya sabemos que en Java todo está organizado en forma de *Clases* que, dicho de una manera sencilla pero incorrecta, cada *Clase* es como si fuera un programa independiente. Como más adelante profundizaremos en el concepto de *Clase* entonces allí veremos por qué esta definición es incorrecta pero por ahora tomémosla así.

Para facilitar la labor de programación con Java, se han diseñado una serie de paquetes, clases y métodos de forma que mucho de lo que usted requiere hacer ya lo encuentre hecho y usted solo tenga que usar las instrucciones en donde lo requiera. Con el tiempo, se han creado tantos *Métodos* que se ha hecho necesario organizarlos en *Clases* y éstos, a su vez, en *Paquetes*. Un Paquete es, como quien dice, una carpeta en la cual se encuentran agrupadas varias clases para hacerlas más manejables y más fáciles de ubicar. Si usted entra a Java API observará que la cantidad de *Clases* y *Métodos* es tan abrumadora que sería imposible aprendérselas de memoria (además no tendría sentido).

Los métodos que hemos utilizado para realizar cálculos aritméticos corresponden a la *Clase Math* que se encuentra en el *Paquete Java.lang*. Este paquete es el que, por defecto, queda activo para ser usado siempre que comenzamos a hacer un programa en Java. Otros paquetes deben activarse si los vamos a utilizar. La razón es muy sencilla: en *Java.lang* se han incluido las *Clases* y los *Métodos* que, estadísticamente, son los más utilizados. Por fuera de *Java.lang* se tienen otras *Clases*, otros *Métodos* y otros *Paquetes* para cuando sean necesarios.

En Java existen dos tipos de métodos: los métodos estáticos y los métodos no estáticos. Se definen como métodos estáticos aquellos que, al utilizarse, se les debe anteponer el nombre de la clase. Tal es el caso del método *round()* de la *Clase Math* pues cuando se va a utilizar se requiere escribir *Math.round()*. Los otros métodos son los no estáticos y éstos requieren asociarse con un objeto predefinido como por ejemplo el método *length()* de la Clase String. Cuando se quiere utilizar se requiere declarar una variable tipo String y ahí si usarse, tal como se muestra a continuación:

```
String Nombre = "Omar";
```

```
System.out.println("Mi nombre tiene " + Nombre.length() + " letras");
```

## 6.2 La Clase Scanner

Hasta el momento hemos guardado los datos en las variables de los ejemplos usando el signo =. Esto quiere decir que hemos cargado las variables por el método de asignación. Pero usted habrá notado que muchas veces un programa le pregunta algo, por ejemplo, un programa le dice “Digite su Clave” y a continuación le abre un espacio para que usted escriba su clave con el teclado y luego presione la tecla <Enter> o su equivalente (cuando se trata de cajeros automáticos). Una entrada de datos no es más que el ingreso de información por parte del usuario del programa. En el lenguaje de programación Java para recibir un dato escrito por el usuario y almacenarlo en una variable debemos realizar varias operaciones. Lo primero que debemos hacer es incluir la Clase Scanner (que se encuentra en el paquete *Java.util*) al inicio del programa. Para ello debemos hacerlo así:

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

o también así

```
import java.util.*;
```

Si usted escribe *import java.util.Scanner;* esto le indicará a Java que incluya, en este programa, todos los métodos de la *Clase Scanner*. Si lo hace de la otra forma, o sea *import java.util.\*;*; le estará diciendo a Java que incluya todas las *Clases* de paquete *java.util* dentro de las cuales se encuentra la *Clase Scanner* que es la que vamos a utilizar para leer datos por el teclado. Precisamente, además de crear un objeto de tipo Scanner debemos asociarlo con el teclado que, en Java, se identifica como *System.in*. Luego para “inicializar” el teclado debemos escribir:

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

Recuerde que *Teclado* es el nombre que le he puesto al objeto tipo Scanner. Puede ser cualquier otro nombre que usted escoja y que le sirva para identificar el teclado. Luego de esto usted puede declarar las variables donde vaya a almacenar los datos con sus respectivos tipos de datos, por ejemplo:

```
String Nombre;      // Declara una variable tipo String (Cadena)
int Edad;          // Declara una variable tipo Entero
double Estatura;   // Declara una variable tipo double (con decimales)
```

Teniendo el sistema de entrada reconocido (System.in) y las variables declaradas ahora sí puede solicitar los datos que considere necesarios, por ejemplo:

```
System.out.println("Por favor, escriba su nombre: ");
Nombre = Teclado.nextLine();
```

Con estas dos instrucciones usted podrá leer una cadena. Con la primera de ellas le está avisando al usuario que escriba el nombre y en la segunda línea usted está leyendo (a través del *Teclado*) una línea de caracteres que finalizan con <Enter>. De manera equivalente, se puede leer un valor entero y un valor real, tal como sigue:

```
System.out.println("Por favor, escriba su edad: ");
Edad = Teclado.nextInt();
```

```
System.out.println("Por favor, escriba su estatura: ");
Estatura = Teclado.nextDouble();
```

En ambos casos, usted solicita el dato y a continuación lo lee para almacenarlo en la variable correspondiente. Como se puede observar cada método lee un dato diferente. El método nextLine( ) retorna una cadena que se ha leído y que ha finalizado con <Enter>, el método nextInt( ) retorna un número entero y el método nextDouble( ) retorna un número real. Como cada uno está almacenando lo que retorna en una variable de un tipo pertinente, entonces los valores quedan almacenados de manera apropiada. Finalmente podría pensarse en que una instrucción de salida fuera como sigue:

```
System.out.println("Nombre: \t " + Nombre);
System.out.println("Edad: \t " + Edad);
System.out.println("Estatura: \t " + Estatura );
```

De esta manera, se podrá visualizar la información recibida. El programa completo sería el siguiente:

```
import java.util.Scanner;

public class Lectura1 {
    public static void main(String args[] ) {

        Scanner Teclado = new Scanner(System.in);
        String Nombre;          // Declara una variable tipo String (Cadena)
        int Edad;              // Declara una variable tipo Entero
        double Estatura;        // Declara una variable tipo double

        System.out.println("Por favor, escriba su nombre: ");
        Nombre = Teclado.nextLine();

        System.out.println("Por favor, escriba su edad: ");
        Edad = Teclado.nextInt();

        System.out.println("Por favor, escriba su estatura: ");
        Estatura = Teclado.nextDouble();

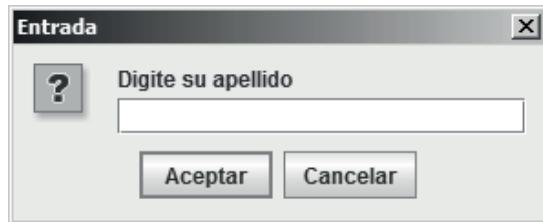
        System.out.println("Nombre: " + Nombre);
        System.out.println("Edad: " + Edad);
        System.out.println("Estatura: " + Estatura );
    }
}
```

Al ejecutar este programa tenga en cuenta que un valor real (float o double) debe digitarse con coma y no con punto. Si digita 3.6 saldrá un error pero si digita 3,6 lo leerá correctamente. Al ejecutar este programa, un resultado posible podría ser el siguiente:

```
"Por favor, escriba su nombre:
Omar
"Por favor, escriba su edad:
52
"Por favor, escriba su estatura:
1,72
Nombre: Omar
Edad: 52
Estatura: 1.72
```

## 6.3 La Clase JOptionPane

Seguramente usted habrá utilizado un programa y habrá que, cuando se requiere escribir un valor, aparecen ventanas de este tipo:



Esta es una ventana más estética y más agradable a la hora de escribir un dato. ¿Cómo hacemos para que un programa Java muestre este tipo de ventanas? Usted se va a sorprender de lo fácil que se hace.

### 6.3.1 Leyendo una cadena

Vamos a asumir que, efectivamente, usted necesita leer el apellido de una persona. Comencemos por recordar que el apellido de una persona es una cadena, o sea, un dato de tipo String.

La construcción de este tipo de ventanas corresponde a un método que se encuentra en el Paquete *javax.swing Clase JOptionPane*. Por lo tanto lo primero que debemos hacer en el programa es escribir la siguiente línea:

```
import javax.swing.JOptionPane;
```

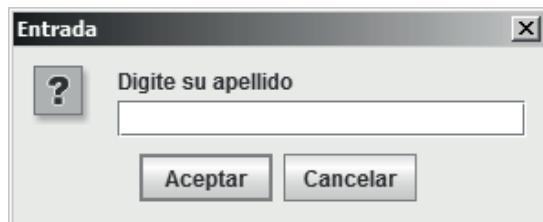
Seguidamente procedemos a escribir la parte inicial del programa o sea:

```
public class Ventana1 {  
    public static void main(String args[ ]) {
```

Ahora, como nuestro programa todo lo que va a hacer es leer una cadena a través de una ventana y mostrarla en la consola entonces declaramos un dato de tipo String y, aprovechamos su declaración para decirle que solicite una cadena y que almacene la cadena leída en el objeto que hemos declarado.

```
String Apellido = JOptionPane.showInputDialog("Digite su apellido");
```

En esta instrucción la declaración de la variable String es muy normal, tal como la hemos explicado anteriormente. Lo nuevo es el método *JOptionPane.showInputDialog*. Como usted puede ver el hecho de que tenga el nombre de la clase antepuesto significa que es un método estático. El método *showInputDialog* lo que hace es que muestra en pantalla una ventana como la que sigue:



Ubicando el cursor en la línea en blanco para que nosotros digitemos, en este caso, el apellido. Note que, cuando se usa el método *showInputDialog*, el aviso que se escribe entre paréntesis (que en este caso es “Digite su apellido”) es el título que sale encima de la línea en blanco en donde va a quedar la cadena que leamos. Automáticamente este método muestra la ventana con opción de cerrar (botón x al lado superior derecho de la ventana) y además le pone dos botones más: un botón de *Aceptar* y otro botón de *Cancelar*. Eso se hace automáticamente.

El método *showInputDialog* retorna la cadena que fue digitada y, como se está asignando a la variable *Apellido*, entonces esta cadena queda almacenada en dicha variable. Por lo tanto podemos proceder a mostrarla en la consola de la siguiente forma:

```
System.out.println("Su apellido es " + Apellido);
```

Finalmente cerramos tanto la llave del método principal (*main*) como la llave de la *Clase Ventana1*.

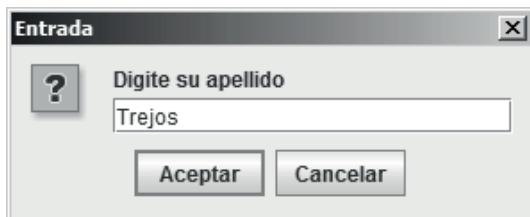
```
}
```

El programa completo sería el siguiente:

```
import javax.swing.JOptionPane;
```

```
public class Ventana1 {  
  
    public static void main(String args[ ]) {  
  
        String Cadena = JOptionPane.showInputDialog("Digite su apellido");  
  
        System.out.println("Su apellido es " + Cadena);  
  
    }  
}
```

Si se ejecutara el programa entonces saldría en pantalla la ventana que sigue.



Si se hubiere digitado el apellido del autor, al oprimir el botón Aceptar aparecerá en la consola el siguiente aviso:

Su apellido es Trejos

Que fue lo que se le dijo que hiciera en la última línea de código del programa. Así de sencillo se usan las ventanas en el lenguaje de programación Java.

### 6.3.2 Leyendo un entero

Ahora veamos cómo sería para leer un entero. Tenga en cuenta que toda información que ingrese a través de una ventana construida con `showInputDialog`, entra en forma de *Cadena* (o sea como un dato de tipo *String*). Para leer un entero utilizando una ventana debemos proceder de una forma similar a la utilizada anteriormente. En primer lugar, vamos a importar el paquete *java.swing* con todas sus utilidades:

```
import javax.swing.JOptionPane;
```

Seguidamente se escribe la clase y el método principal, como se ha hecho en todos los programas construidos hasta ahora:

```
public class Ventana2 {  
  
    public static void main(String args[ ]) {
```

Vamos a asumir en este ejemplo que el objetivo es leer la cantidad de hijos de una persona y determinar si tuviera un hijo más, con cuántos hijos quedaría en total. Para ello lo primero que vamos a declarar es una instancia de la clase *String* (que es como si se declarara una variable “tipo” *String* pero con más utilidades). En esta variable vamos a almacenar la cadena que retorne la ventana de lectura. Es posible que usted piense que el número de hijos es un valor numérico, ¿por qué se debe crear una variable “tipo” *String*? Muy sencillo.

Recuerda que le comenté que todo lo que entra por una ventana ingresa en formato de cadena entonces debemos recibir la cantidad de hijos primero como una cadena y luego lo convertimos al valor numérico correspondiente. ¿Siempre debe hacerse así (primero se recibe como cadena y luego se convierte al número correspondiente? La respuesta es SI mientras esté usando una ventana de lectura. Entonces declaramos la variable “tipo” *String* y a continuación abrimos la ventana para recibir el valor (en formato de cadena):

```
String Hijos;
```

```
Hijos = JOptionPane.showInputDialog("Cuántos hijos tiene?");
```

Ahora vamos a declarar una variable de tipo entero (*int*) en donde vamos a almacenar el número equivalente a la cadena que acabamos de leer. Si hemos leído la cadena “4” entonces ahora vamos a almacenar el número 4. De una vez vamos a declarar la variable y vamos a convertir la cadena en el entero correspondiente.

```
int NumHijos = Integer.parseInt(Hijos);
```

El *Método parseInt* de la *Clase Integer* convierte una cadena que contiene dígitos numéricos al número correspondiente. Note usted que no hay necesidad de incluir la *Clase Integer* debido a que esta forma parte del paquete *java.lang* que es el paquete que Java incluye por defecto para la construcción de cualquier programa.

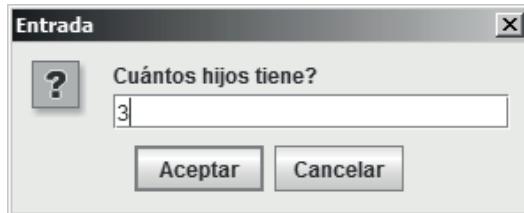
Por último, mostramos en pantalla el resultado que se espera. Tenga en cuenta que la variable “tipo” **String** tiene un nombre (*Hijos*) y la variable tipo **int** tiene otro nombre (*NumHijos*). Esto debe ser así para que no se cree una confusión en el intérprete de Java. La razón por la cual se hizo la conversión de cadena a entero fue para poder realizar la operación de sumarle 1. Si esta operación se hiciera con una variable “tipo” String aparecería el contenido de la cadena y al lado aparecería el número 1 pero como se está haciendo con un valor numérico aparecerá el resultado de sumarle 1 a dicho número.

```
System.out.println("Con un hijo más tendrás " + (NumHijos + 1));
    }      }
```

El programa completo, con todos los elementos explicados, es el siguiente:

```
import javax.swing.JOptionPane;
public class Ventana2 {
    public static void main(String args[ ]) {
        String Hijos;
        Hijos = JOptionPane.showInputDialog("Cuántos hijos tiene?");
        int NumHijos = Integer.parseInt(Hijos);
        System.out.println("Con un hijo más tendrás " + (NumHijos + 1));
    }      }
```

Al ejecutar este programa, saldrá la siguiente ventana:



Si se le hubiera digitado el valor 3, entonces lo que hace el programa es que recibe la cadena “3” y la almacena. Posteriormente convierte esa cadena “3” en el número 3 para poder sumarle 1. En la consola aparecerá:

Con un hijo más tendrás 4

Que cumple con el objetivo que se planteó para el desarrollo de este programa.

### 6.3.3 Leyendo un real

Finalmente ¿cómo hacemos para leer un valor real a través de una ventana? Debemos hacer lo mismo: abrir una ventana, leer una cadena que represente un valor real, convertir la cadena que representa un valor real al número real correspondiente y, ahí sí, usarlo para realizar operaciones. Supongamos que vamos a mostrar la raíz cúbica de un número real, entonces el programa Java (debidamente comentado) podría ser el siguiente:

```
// Se incluye la Clase JOptionPane para crear ventanas
import javax.swing.JOptionPane;

public class Ventana3 { // Clase principal
    public static void main(String args[]) { // Método principal

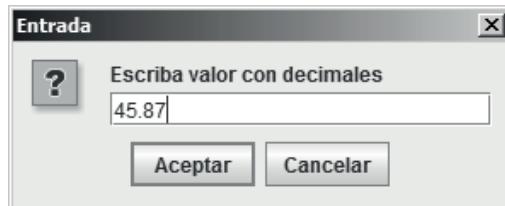
        // Se declara una variable String para recibir lo que retorne la ventana
        String N1 = JOptionPane.showInputDialog("Escriba valor con
decimales");

        // Se declara una variable double para convertir la cadena recibida
        // en su equivalente número con decimales
        double N2 = Double.parseDouble(N1);
```

```
// Se muestra (por consola) el resultado de la raíz cúbica del número  
leído  
System.out.println("La raíz cúbica de " + N1 + " es " + Math.cbrt(N2))  
;  
    }      }
```

Note usted que en la última línea de código se usan N1 y N2. N1 se usa para mostrar el número original que se leyó; en este caso pudo haberse mostrado también el contenido de la variable N2. Para calcular la raíz cúbica (método *Math.cbrt*) debe usarse solamente la variable N2 debido a que esta variable es de tipo *double* y este método requiere, como argumento, un dato tipo *double*.

El método *parseDouble* forma parte de la *Clase Double* que se encuentra en el paquete *java.lang* que, a su vez, es el paquete que se incluye por defecto en cualquier programa que se quiera construir con Java. Al ejecutar este programa aparecerá una ventana como la siguiente:



Si se le escribiera el número 45.87 entonces el resultado por consola sería el siguiente:

La raíz cúbica de 45.87 es 3.5796693513999793

Note usted que el número que se muestra como resultado de calcular la raíz cúbica al valor leído tiene una cantidad de decimales excesiva. Cómo hacer para que salga solamente con 2 decimales? Ese es el tema del siguiente numeral.

#### 6.4 Formato de números con decimales

Para darle formato a un número real simplemente se le debe especificar a Java, a través del método *printf*, la cantidad de decimales que se requieren.

Si se quisieran solamente 2 decimales, veamos cómo quedaría el programa con este cambio:

```
import javax.swing.JOptionPane;  
  
public class Ventana3 {  
    public static void main(String args[]) {  
  
        String N1 = JOptionPane.showInputDialog("Escriba valor con  
decimales");  
  
        double N2 = Double.parseDouble(N1);  
  
        System.out.print("La raíz cúbica de " + N1 + " es ");  
        System.out.printf("%1.3f", Math.cbrt(N2));  
  
    }  
}
```

Las dos últimas líneas son las únicas diferentes. En la penúltima línea se ha utilizado `System.out.print` para que al finalizar de escribir lo que está entre paréntesis, se escriba lo de la siguiente instrucción en la misma línea donde iba. En la última línea se ha acudido a `System.out.printf` que permite definir la cantidad de decimales que queremos que salgan en pantalla. En este caso hemos indicado que muestre la raíz cúbica del valor almacenado en la variable `N2` con tan solo 3 decimales. Al ejecutar el programa, si digitamos el mismo número real, aparecerá en la consola:

La raíz cúbica de 45.87 es 3,580

Que como podemos ver es un número real pero con tan solo 3 decimales tal como se le indicó (`%1.3f`) que le indica a Java que se requiere el dato con 3 decimales y advierte que el dato es real. En este lenguaje de programación existen otros formatos pero serán revisados más adelante.

## 6.5 Ejercicios Propuestos

- Construir un programa Java que permita leer su edad y mostrar la mitad de su edad

- b. Construir un programa Java que permita leer su edad y determina cuántos años tendrá dentro 10 años
- c. Construir un programa Java que permita leer su nombre, después leer su apellido y mostrar en pantalla su nombre concatenado con su apellido en una sola variable
- d. Construir un programa Java que permita leer su nombre y mostrar solamente las dos primeras letras
- e. Construir un programa Java que permita leer su estatura y determinar cuánto mediría usted si tuviera 10 cms. más.
- f. Construir un programa Java que permita leer su salario y determinar cuánto ganará el próximo año si el incremento se sabe que será del 10%
- g. Construir un programa Java que permita leer su estatura y determinar a cuánto es igual la raíz cuadrada de su edad
- h. Construir un programa Java que permita leer su salario y determinar cuánto le descuentan por salud si se sabe que este descuento equivale al 12% de dicho salario
- i. Construir un programa Java que permita leer su edad y determinar a cuánto es igual la raíz cuadrada de dicha edad
- j. Construir un programa Java que recibe tres de sus datos personales y los muestre por consola de forma organizada



# 7 Lección

# Condicionales



## 7.1 El flujo de la ejecución

Cuando se escribe un programa Java, éste se ejecuta línea por línea de arriba hacia abajo. El intérprete de Java siempre respeta el orden en el cual hemos escrito nuestro código y seguirá, una a una, las instrucciones que hayamos incluido. Al orden en el cual se ejecuta un programa se le conoce como el flujo de la ejecución y dependerá de la forma como nosotros mismos hayamos concebido al solución. En algunos casos este orden debe alterarse bien porque lo necesitemos o bien porque el objetivo del programa así lo exija. La primera forma de alterar el flujo de la ejecución son los condicionales.

## 7.2 Condicional Simple

Un condicional es una instrucción que permite que se escoja uno de entre dos caminos lógicos o uno de entre varios caminos lógicos. Si usted va a salir de su casa y se asoma a la ventana para ver si el día está nublado o no y, sobre estaba base, toma la decisión de llevar paraguas entonces usted podría representar la situación de la siguiente forma:

*Si está nublado?*

*Llevar el paraguas (por si llueve)*

*Si no*

*No llevar el paraguas*

Note usted que llevar o no llevar el paraguas depende exclusivamente de que **esté nublado** y precisamente esa es la *Condición* que determina cuál acción realizar. Pues bien, es posible que usted adopte otras acciones como se muestra a continuación:

*Si está nublado?*

*Llevar el paraguas (por si llueve)*

*Ponerse ropa para el frío*

*Llevar el termo con una bebida caliente*

*Si no*

*No llevar el paraguas*

*Ponerse ropa ligera*

*Llevar una botella con agua*

En este caso se puede observar que existen varias acciones en caso de que **esté nublado** y también existen varias acciones en caso de que no sea así. Para hacerlo más entendible podemos adicionarle unas llaves que indiquen cuál es el conjunto de acciones a seguir en el caso de que la *Condición* sea *Verdadera* y cuál es el conjunto de acciones a seguir en el caso de que la *Condición* sea *Falsa*. Esto nos permitiría reescribir este secuencia de acciones de la siguiente forma:

*Si (está nublado?)*  
{     *Llevar el paraguas (por si llueve)*  
        *Ponerse ropa para el frío*  
        *Llevar el termo con una bebida caliente*  
}  
*Si no*  
{     *No llevar el paraguas*  
        *Ponerse ropa ligera*  
        *Llevar una botella con agua*  
}

Esta forma de escribirlo nos asegurará que tendremos muy claro qué debemos hacer en un caso u otro, dependiente de que la condición se cumpla. Ahora sí podemos determinar cuál es la estructura de un condicional simple en Java:

```
if (Condición)
{
    Instrucciones a ejecutar en caso de que la Condición sea Verdadera
}
else
{
    Instrucciones a ejecutar en caso de que la Condición sea Falsa
}
```

Es de anotar que cuando las instrucciones a ejecutar (por cualquiera de los dos lados) corresponden solamente a UNA instrucción, no se requieren las llaves dado que éstas agrupan dos o más instrucciones. Una sola instrucción no requiere ser agrupada. También vale la pena tener en cuenta que no es obligatorio el *else* (o lo que para nosotros representaría el *Si no*)

puesto que muchas veces se requiere evaluar una condición solamente por el lado **Verdadero**.

Vamos a realizar el siguiente ejemplo: Construir un programa que lea un número entero y determine si este número es mayor o menor que 100.

Para resolverlo lo primero que vamos a hacer en nuestro programa será importar la *Clase Scanner* del Paquete *java.util* dado que vamos a recibir los datos por consola (si quisiéramos hacerlo a través de una ventana entonces deberemos importar la *Clase JOptionPane*). Luego nuestra primera instrucción es:

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

Ahora vamos a darle declarar nuestra clase y su método principal.

```
public class Condicional1 {  
    public static void main(String args[ ]) {
```

Luego declaramos una instancia de la *Clase Scanner* para poder recibir información a través del teclado (*System.in*).

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

Seguidamente declaramos una variable tipo **int** (entera) para almacenar en ella el dato que vamos a recibir del usuario.

```
int valor;
```

Escribimos un aviso para solicitar el número, luego lo leemos y lo almacenamos en la variable *valor*.

```
System.out.print("Por favor, escriba un número: ");  
valor = Tec.nextInt();
```

Como el dato leído ya se encuentra almacenado en la variable *valor* entonces evaluamos si dicho dato es mayor que 100. En caso de que sea **Verdadero** entonces mostramos en pantalla el aviso pertinente en consonancia con el enunciado y en caso de que sea **Falso** entonces avisamos lo pertinente.

```
if(valor > 100)
    System.out.println( valor + " es mayor que 100");
else
    System.out.println( valor + " NO es mayor que 100");
```

Terminamos el programa cerrando las llaves tanto del método principal como de la Clase Condicional1.

```
}
```

El programa completo sería el siguiente:

```
import java.util.Scanner;

public class Condicional1 {
    public static void main(String args[ ]) {

        Scanner Tec=new Scanner(System.in);

        int valor;

        System.out.print("Por favor, escriba un número: ");
        valor = Tec.nextInt();

        if(valor > 100)
            System.out.println( valor + " es mayor que 100");
        else
            System.out.println( valor + " NO es mayor que 100");
    }
}
```

Al ejecutar este programa Java, si se digitara el valor 354, el resultado sería el siguiente:

```
Por favor, escriba un número: 354
354 es mayor que 100
```

...y si se digitara un valor menor que 100 entonces saldría en la consola:

Por favor, escriba un número: 35  
35 NO es mayor que 100

### 7.2.1 Ejercicios Propuestos

- a. Construir un programa que lea un número entero y determine si es menor que 500
- b. Construir un programa que lea dos números enteros y determine si el 1º valor leído es mayor que el 2º valor leído
- c. Construir un programa que lea dos números enteros y determine si la suma de los dos números es mayor que 100
- d. Construir un programa que lea dos números enteros y determine si el producto de estos dos números es menor que 300
- e. Construir un programa que lea un número entero y determine cuántos dígitos tiene
- f. Construir un programa que lea un número entero y determine si la mitad es mayor que 100
- g. Construir un programa que lea tres números enteros y determine cuál es el mayor
- h. Construir un programa que lea tres números enteros y determine cuál es el menor
- i. Construir un programa que lea tres números enteros y los muestre ascendentemente
- j. Construir un programa que lea tres números enteros y los muestre descendentemente
- k. Construir un programa que lea tres números enteros y muestre el número intermedio (o sea el que no es el mayor ni es el menor)

### 7.3 Condicional Múltiple

El condicional simple permite escoger uno de entre dos caminos lógicos. Si conectamos un condicional simple con otro condicional simple y así sucesivamente se puede obtener que se tomen decisiones en referencia a varios procesos. Java ofrece la posibilidad de escoger uno de entre varios caminos lógicos.

Tomemos este enunciado: Brindar la posibilidad de que un usuario escriba un número y determine qué quiere hacer con dicho número. Si quiere elevarlo al cuadrado, deberá digitar 1. Si quiere calcular su raíz cuadrada,

deberá digitar 2. Si quiere calcular su raíz cúbica, deberá digitar 3. Si quiere calcular su mitad, deberá digitar 4.

Como se puede observar, el usuario deberá escribir un número y escoger una de entre cuatro posibilidades de operarlo. Entonces lo vamos a hacer basado en lo que hemos visto hasta ahora. Lo primero que vamos a hacer es incluir las clases que se necesitan para que los métodos que se requieran, funcionen sin ningún problema.

```
import java.util.*;  
import javax.swing.*;
```

El paquete *java.util* se requiere para poder utilizar la clase Scanner y el paquete *javax.swing* se necesita para poder utilizar ventanas. Vamos a leer el dato a través de una ventana y vamos a mostrar los resultados por consola. Seguidamente definimos la clase y el método principal de esta clase.

```
public class Condicional2 {  
    public static void main(String args[ ] ) {
```

Ahora declaramos una variable tipo *double* que vamos a llamar *Num*. Esta variable almacenará el número que se va a utilizar para hacer los cálculos.

```
    double Num;
```

Seguidamente declaramos una variable de tipo *int* que llamaremos *Opcion* y que será la que almacenará la opción que el usuario escoja.

```
    int Opcion;
```

En esta parte también vamos a declarar una instancia del objeto *Scanner* y la vamos a asociar con el teclado (*System.in*)

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

Ahora lo que vamos a hacer es recibir el número con el cual el usuario quiere hacer los cálculos. Nos valemos de *JOptionPane* para utilizar una ventana a través de la cual leemos el número. Debe recordarse que, cuando

se digita un valor en una ventana construida con *JOptionPane*, éste entra como una cadena por lo tanto debemos convertir la cadena al correspondiente. Para ello acudimos al método *parseDouble* que realiza este proceso.

```
String N = JOptionPane.showInputDialog("Escriba un número");
Num = Double.parseDouble(N);
```

A continuación, sabiendo que ya tenemos el número real almacenado, mostramos en pantalla los posibles cálculos que puede hacer el usuario con el número digitado. Esto es lo que comúnmente se conoce como un *Menú de Opciones*.

```
System.out.println("Esoja una opción");
System.out.println("=====");
System.out.println("1. Elevarlo al cuadrado");
System.out.println("2. Obtener su raiz cuadrada");
System.out.println("3. Obtener su raíz cúbica");
System.out.println("4. Obtener su mitad");
```

Seguidamente solicitamos al usuario que digite la opción que le interese. Acudimos a la consola para leer la opción escogida por el usuario.

```
System.out.print("\n\nOpción: ");
Opcion = Teclado.nextInt();
```

El carácter \n representa un salto de línea para que la información salga en la consola de una forma más clara. Como ya tenemos el número que el usuario digitó y la opción que quiere realizar, entonces procedemos a evaluar lo que el usuario escogió. Damos la orden de evaluar lo que contiene la variable *Opcion*.

```
switch(Opcion) {
```

En caso de que la variable *Opcion* almacene el valor 1 (o sea en caso de que el usuario quiera calcular el cuadrado del número) procedemos a escribir el código correspondiente de forma que se haga efectivo dicho cálculo.

```
case 1:
```

```
System.out.println(Num+" al cuadrado es = a "+(Num*Num));  
break;
```

La instrucción *break*, que aparece al final de este caso, le indica al intérprete de Java que hasta ahí llega lo que tiene que hacer en caso de que se haya escogido la opción 1. En caso de que la opción escogida por el usuario sea la opción 2 (o sea que quiera obtener la raíz cuadrada del número digitado) entonces se escribe lo pertinente y, de nuevo, se escribe la instrucción *break*.

**case 2:**

```
System.out.println("Raiz Cuadrada de "+Num+" = "+Math.sqrt(Num));  
break;
```

En el caso de que el usuario quiera obtener la raíz cúbica del número que digitó entonces se acude al método correspondiente (que se encuentra en la Clase Math) y se muestra en la consola el resultado. Una vez más se escribe *break* al final de esta opción.

**case 3:**

```
System.out.println("Raiz Cúbica de "+Num+" = " + Math.cbrt(Num));  
break;
```

Si el usuario lo que quiere hacer es mostrar la mitad del número en la consola (opción 4) entonces se escribe la instrucción que realiza el cálculo y que lo muestra. Al final de esta opción se escribe *break*.

**case 4:**

```
System.out.println("La mitad de " + Num + " = " + (Num/2));  
break;
```

Finalmente, en el caso de que el usuario no haya digitado ni la opción 1, ni la opción 2, ni la opción 3, ni la opción 4 (que son las que se programaron) entonces significa que el usuario escribió un número que no está considerado dentro de las opciones programadas. En este caso, sea cual fuere el valor, podemos avisar de que el usuario se equivocó. Para ello se usa la instrucción *default*.

**default:**

```
System.out.println("Escoja solo una de las opciones posibles");
```

Finalmente cerramos la llave del condicional **switch**, del método principal *main()* y de la *Clase Condicional2*.

```
        }  
    }  
}
```

Ahora sí, veamos cómo quedaría el programa completo según lo que hemos explicado.

```
import java.util.*;  
import javax.swing.*;  
  
public class Condicional2 {  
    public static void main(String args[ ] ) {  
  
        double Num;  
        int Opcion;  
        Scanner Teclado = new Scanner(System.in);  
  
        String N = JOptionPane.showInputDialog("Escriba un número");  
        Num = Double.parseDouble(N);  
  
        System.out.println("Esoja una opción");  
        System.out.println("=====");  
        System.out.println("1. Elevarlo al cuadrado");  
        System.out.println("2. Obtener su raiz cuadrada");  
        System.out.println("3. Obtener su raíz cúbica");  
        System.out.println("4. Obtener su mitad");  
  
        System.out.print("\n\nOpción: ");  
        Opcion = Teclado.nextInt();  
  
        switch(Opcion) {  
  
            case 1:  
                System.out.println(Num+" al cuadrado es = "+(Num*Num));  
                break;
```

```
case 2:  
    System.out.print("Raiz Cuadrada de ");  
    System.out.println(Num + " = " + Math.sqrt(Num));  
    break;  
  
case 3:  
    System.out.print("Raiz Cúbica de " + Num + " = ");  
    System.out.println(Math.cbrt(Num));  
    break;  
  
case 4:  
    System.out.println("La mitad de " + Num + " = " + (Num/2));  
    break;  
  
default:  
    System.out.println("Escoja solo una de las opciones posibles");  
}  
}  
}
```

## 7.4 Ejercicios Propuestos

- Construir un programa Java que lea dos números enteros y que ofrezca el siguiente menú: 1) Sumar los dos números leídos, 2) Restar los dos números leídos, 3) Multiplicar los dos números leídos, 4) Dividir los dos números leídos, 5) Elevar el 1º número a la potencia que representa el 2º número
- Construir un programa Java que lea dos números enteros y que ofrezca el siguiente menú: 1) calcular el Seno del 1º número leído, 2) calcular el coseno del 2º número leído, 3) calcular la tangente del 1º número leído, 4) comprobar que la tangente del 1º número leído es igual al resultado de dividir el seno del 1º número entre el coseno del 2º número
- Construir un programa Java que lea una cadena y que ofrezca el siguiente menú: 1) mostrar sus dos primeras letras, 2) mostrar sus tres primeras letras, 3) mostrar sus cuatro primeras letras, 4) mostrar sus cuatro últimas letras, 5) mostrar sus tres últimas letras, 6) mostrar sus dos últimas letras, 7) mostrar su última letra

# 8 Lección

# Ciclos



## 8.1 Definición

Comencemos por revisar lo siguiente: usted está sentado en un comedor y tiene al frente un plato de sopa. Toma una cucharada y se la lleva a la boca. Inconscientemente mira si ya terminó. Como aún tiene sopa en el plato, toma otra cucharada y se la toma. Mira de nuevo al plato, observa que aún no ha terminado, toma otra cucharada y se la lleva a la sopa. Mira de nuevo al plato, observa que aún no ha terminado, toma otra cucharada y se la lleva a la sopa. Cuando observa que no hay más sopa en el plato entonces separa el plato y da por terminado el plato de sopa.

Veamos cómo escribiríamos esto de una forma un poco más algorítmica:

*Observa el plato*

*Mira que no ha terminado la sopa*

*Toma una cucharada de sopa*

*Se la lleva a la boca y se la toma*

*Si aún no ha terminado la sopa*

*Observa el plato*

*Mira que no ha terminado la sopa*

*Toma una cucharada de sopa*

*Se la lleva a la boca y se la toma*

*Si aún no ha terminado la sopa*

*Observa el plato*

*Mira que no ha terminado la sopa*

*Toma una cucharada de sopa*

*Se la lleva a la boca y se la toma*

*Si aún no ha terminado la sopa*

*Observa el plato*

*Mira que no ha terminado la sopa*

*Toma una cucharada de sopa*

*Se la lleva a la boca y se la toma*

Para el proceso solo cuando haya terminado la sopa

En el lugar en donde están los puntos suspensivos usted podrá repetir el bloque de acciones siguiente:

*Si aún no ha terminado la sopa*

*Observa el plato*

*Mira que no ha terminado la sopa*

*Toma una cucharada de sopa*

*Se la lleva a la boca y se la toma*

Tantas veces como lo requiera teniendo en cuenta que este proceso repetitivo terminará cuando termine la sopa. Otra forma como podríamos escribir este proceso podría ser la siguiente:

***Mientras*** *No se haya terminado la sopa*

*Toma una cucharada de sopa*

*Se la lleva a la boca*

*Se la toma*

***Fin\_Mientras***

Note usted que la palabra ***Mientras*** y ***Fin\_Mientras*** (que son los delimitadores del conjunto de acciones) están destacadas en negrilla. La palabra ***Mientras*** indica en donde comienza el conjunto de instrucciones a repetir y la palabra ***Fin\_Mientras*** indica en donde termina dicho conjunto de acciones.

¿Cómo se ejecuta? Muy sencillo, se evalúa la condición que está al lado de la palabra ***Mientras***. Si esta condición es *Verdadera*, entonces realiza el conjunto de instrucciones (que para este caso son tres instrucciones) que están dentro del ciclo.

Cuando llega al ***Fin\_Mientras*** entonces vuelve a evaluar la condición que está al lado de la palabra ***Mientras*** y, dependiendo de su evaluación (es decir, si es *Verdadera* o *Falsa*), vuelve a realizar las acciones del ciclo o definitivamente se sale del ciclo.

Eso es un ciclo en programación. Ahora supongamos que usted quiere llegar a su casa y se ha ido a pie. Entonces usted da un paso en la dirección de su casa, si no ha llegado a ella, da otro paso, si aún no ha llegado, da otro paso, y así sucesivamente sigue dando pasos hasta que llega a la casa. Este ciclo podría escribirse de la siguiente forma:

*Si no ha llegado*

*Dar un paso en dirección a la casa*

*Si aún no ha llegado*

*Dar un paso en dirección a la casa*

*Si aún no ha llegado*

*Dar un paso en dirección a la casa*

.

.

.

*Si llegó pare*

Los puntos suspensivos indican que no sabemos con exactitud cuántos pasos debe dar, por lo tanto una forma de hacerlo como un ciclo podría ser la siguiente:

***Mientras no haya llegado a la casa***

*Dar un paso en dirección a la casa*

**Fin\_Mientras**

En el lenguaje Java existen cuatro tipos de instrucciones con las cuales se pueden implementar ciclos:

- Ciclo while
- Ciclo do - while
- Ciclo for
- Ciclo for – each

Vamos a estudiarlos de una forma detallada para que podamos utilizarlos en los procesos cíclicos que requieran nuestros programas.

## 8.2 Ciclo **while**

Este es el más simple de los ciclos y, además, el más fácil de implementar. Su estructura es muy sencilla:

```
while ( Condición )
{
    Cuerpo del Ciclo
}
```

La instrucción **while** le da inicio al ciclo. La *Condición* le indica la expresión que debe evaluar y, por cada vez que dicha *Condición* sea *Verdadera*, entonces se deberá ejecutar el *Cuerpo del Ciclo*. El cuerpo del ciclo es el conjunto de instrucciones que se van a repetir. Las llaves que abren y cierran indican donde comienza el *Cuerpo del Ciclo* y donde termina. Tenga en cuenta que normalmente un ciclo tiene un valor de inicio, una condición que evalúa la variable que comienza con el valor de inicio y una operación que hace que el contenido de la variable se aproxime al valor que evalúa la condición.

### 8.2.1 Ejemplo resuelto

Construir un programa Java que lea un número entero y muestre todos los enteros comprendidos entre 1 y el número leído.

En este enunciado lo primero que vamos a hacer es importar la Clase Scanner para poder utilizar la entrada por el teclado.

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

Seguidamente iniciamos la *Clase Ciclos1* y el método principal de esta clase (método main).

```
public class Ciclos1 {
    public static void main(String args[]){
```

ahora vamos a declarer una variable tipo entero (int) para almacenar el número que debemos leer y después creamos una instancia del objeto Scanner para asociarlo con el teclado y poder leer el dato que digite el usuario.

```
int num;  
Scanner Teclado = new Scanner(System.in);
```

Ahora escribimos un título en consola solicitando el número, después leemos el número y lo almacenamos en la variable num. Utilizamos el método **print** para que el valor sea leído a continuación del título en la misma línea.

```
System.out.print("Digite un número: ");  
num=Teclado.nextInt();
```

Avisamos, a continuación, con un título que vamos a mostrar los números comprendidos entre 1 y el número leído. El carácter \n le indica que pase a la siguiente línea. Dos veces este carácter indica que deje dos líneas.

```
System.out.println("\n\nNúmeros de 1 hasta " + num );
```

Seguidamente declaramos una variable que vamos a iniciar en 1 y le vamos a ir incrementando su contenido de 1 en 1 hasta llegar a num. Por cada valor que tome esta variable iremos mostrando por consola su contenido, para cumplir con el objetivo del programa.

```
int i = 1;
```

Planteamos el ciclo de la siguiente forma: mientras el valor almacenado en la variable *i* sea menor o igual que el valor almacenado en la variable *num* entonces muestre el contenido de *i* e increméntela en 1.

```
while( i <= num)  
{  
    System.out.print (i + "\t");  
    i++;  
}
```

El carácter \t permite que los datos salgan separados a una distancia de 7 espacios para que la información se vea más presentable. Finalmente cerramos el método principal y la Clase.

```
}  
}
```

El programa completo correspondería al código que se muestra a continuación:

```
import java.util.Scanner;

public class Ciclos1 {
    public static void main(String args[ ] ) {

        int num;
        Scanner Teclado = new Scanner(System.in);

        System.out.print("Digite un número: ");
        num=Teclado.nextInt();

        System.out.println("\n\nNúmeros de 1 hasta " + num );
        int i = 1;
        while(i<=num)
        {
            System.out.print(i + "\t");
            i++;
        }
    }
}
```

Al ejecutar este programa, si se digitara el número 10, el resultado sería el siguiente:

Digite un número: 10

Números de 1 hasta 10  
1      2      3      4      5      6      7      8      9      10

Que cumple con el objetivo que se propuso al inicio que era leer un número entero y mostrar todos los enteros comprendidos entre 1 y el número leído. Si se hubiera digitado 6 entonces el resultado sería el siguiente:

Digite un número: 10

Números de 1 hasta 10  
1      2      3      4      5      6

### 8.2.2 Ejercicios Propuestos

- a. Construir un programa que permita leer un número entero y mostrar todos los enteros comprendidos entre 1 y la mitad del número leído
- b. Construir un programa que permita leer un número entero y mostrar todos los enteros comprendidos entre 1 y la parte entera de la raíz cuadrada del número leído
- c. Construir un programa que permita leer un número entero y mostrar todos los enteros comprendidos 1 y la parte entera de la raíz cúbica del número leído
- d. Construir un programa que permita leer dos números enteros y mostrar todos los números enteros comprendidos entre el número menor y el número mayor
- e. Construir un programa que permita leer dos números enteros y mostrar todos los números enteros comprendidos entre la raíz cuadrada del número menor y la raíz cuadrada del número mayor
- f. Construir un programa que permita leer dos números enteros y mostrar todos los números enteros comprendidos entre la raíz cúbica del número menor y la raíz cúbica del número mayor
- g. Construir un programa que permita leer una cadena y mostrarla por consola carácter por carácter (verticalmente)
- h. Construir un programa que muestre por consola 10 números aleatorios comprendidos entre 1 y 100
- i. Construir un programa que permita leer dos números enteros y muestre 10 números aleatorios comprendidos entre el menor y el mayor
- j. Construir un programa que lea un número entero y muestre todos los múltiplos de 3 comprendidos entre 1 y el número leído

### 8.3 Ciclo **do-while**

En realidad no es que existan varios tipos de ciclos, lo que sucede es que el lenguaje provee varias formas sintácticas de escribirlos de forma que se pueda ajustar a la lógica de cada persona. El ciclo **do – while** es una variante muy interesante del ciclo **while** con la única diferencia de que en el ciclo **while** la condición es lo primero que se evalúa y, si su respuesta es *Verdadera*, entonces se ejecuta el cuerpo del ciclo mientras que en el ciclo **do – while** primero se ejecuta el cuerpo del ciclo y, al final, se evalúa la condición.

La estructura de un ciclo ***do – while*** es muy similar a la del ciclo ***while*** con la diferencia que la condición se evalúa al final del ciclo.

```
do
{
    Cuerpo del Ciclo
} while (Condición);
```

La palabra *do* inicia un ciclo ***do - while***. A continuación se abre la llave que encierra el *Cuerpo del Ciclo* o sea el conjunto de instrucciones que deben repetirse mientras la condición sea Verdadera. La llave de cierre indica en donde termina el *Cuerpo del Ciclo* y mientras la *Condición* sea *Verdadera* se volverá a ejecutar.

### 8.3.1 Ejemplo resuelto

Vamos a desarrollar un ejemplo que evidencia el uso del ciclo ***do – while***. Construir un programa que permita leer un número entero y calcular su número factorial. El factorial de un número es el resultado de multiplicar todos los enteros comprendidos entre 1 y el número leído para lo cual el número leído debe ser mayor que 0. En caso de que el número sea igual a 0 su número factorial es igual a 1. No están definido un proceso para calcular el número factorial de números negativos.

Lo primero que haremos será importar la librería Scanner para poder leer datos a través del teclado.

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

A continuación damos inicio tanto a la *Clase Ciclos2* como al método principal de dicha clase o sea al método *main*.

```
public class Ciclos2 {
    public static void main(String args[] ) {
```

Luego creamos una instancia del objeto Scanner y lo asociamos con el teclado (*System.in*).

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

De la misma forma declaramos una variable de tipo entero (int) en donde vamos a almacenar el número que queremos leer.

**int num;**

Escribimos un título que solicite el número y lo leemos a través del teclado.

```
System.out.print("Número : ");
num = Tec.nextInt();
```

Ahora vamos a iniciar dos variables: una que llamamos *facto* y que comenzamos en 1 y otra que llamamos *i* y que también iniciamos en 1. En la variable *facto* vamos a almacenar el resultado de multiplicar sucesivamente los números comprendidos entre 1 y el valor leído y en la variable *i* vamos a almacenar cada uno de los valores enteros comprendidos entre 1 y dicho valor leído.

**int facto=1, i=1;**

Iniciamos el ciclo con la instrucción **do** a continuación de lo cual abrimos una llave que da inicio al cuerpo del ciclo.

```
do
{
```

A continuación guardamos en la variable *facto* el resultado de multiplicar lo que tenga almacenado *facto* por lo que almacene la variable *i*. Después de realizar esta operación, incrementamos en 1 el valor de la variable *i* a través del operador doble **++**.

```
facto = facto * i;
i++;
```

Cerramos el cuerpo del ciclo **do – while** de este ejercicio y establecemos la condición de forma que el ciclo se mantenga mientras el valor almacenado en *i* sea menor o igual que el valor almacenado en *num*.

```
} while( i <= num );
```

Por último, mostramos por consola el título que nos indica que el factorial del número leído es igual al cálculo que se almacenó, progresivamente a través del ciclo **do – while**, en la variable *facto*.

```
System.out.println("El factorial de "+num+" = "+facto);
```

Finalmente cerramos la llave de la función principal main y la llave de la Clase Ciclos2.

```
}
```

El código completo de este programa se presenta a continuación:

```
import java.util.Scanner;

public class Ciclos2 {

    public static void main(String args[ ] ) {

        Scanner Teclado = new Scanner(System.in);
        int num;

        System.out.print("Número : ");
        num = Teclado.nextInt();

        int facto=1, i=1;

        do
        {
            facto = facto * i;
            i++;
        }while( i <= num);

        System.out.println("El factorial de "+num+" = "+facto);
    }
}
```

Si se digitara el número 5 entonces el resultado que aparece en la consola es:

Número : 8

El factorial de 8 = 40320

Este resultado efectivamente es el producto de multiplicar  $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8$  que corresponde al cálculo del número factorial de 8. Usted deberá ajustar este programa para que tenga en cuenta los casos en que digiten un valor negativo o digiten el valor 0.

### 8.3.2 Ejercicios Propuestos

Utilizando el ciclo ***do – while*** desarrollar una solución para los siguientes enunciados:

- a. Construir un programa Java que permita leer un número entero y determinar cuántos dígitos tiene
- b. Construir un programa Java que permita leer un número entero y determinar cuántos dígitos pares tiene
- c. Construir un programa Java que permita leer un número entero y determinar cuántos dígitos impares tiene
- d. Construir un programa Java que lea un número entero y determine cuántos dígitos primos tiene (los dígitos primos son 2, 3, 5 y 7)
- e. Construir un programa Java que lea dos números enteros y determina cuál de los dos tiene más dígitos impares
- f. Construir un programa Java que lea dos números enteros y determina cuál de los dos tiene más dígitos pares
- g. Construir un programa Java que lea dos números enteros y determine cuál de los dos tiene más dígitos primos
- h. Construir un programa Java que lea una cadena y determine cuántas vocales tiene
- i. Construir un programa Java que lea una cadena y determine cuántas veces está la vocal e
- j. Construir un programa Java que lea una cadena y determine cuántas veces está una vocal que digite el mismo usuario
- k. Construir un programa Java que lea una cadena y que determine cuántas sílabas tiene

### 8.4 Ciclo *for*

El ciclo *for* es un ciclo mucho más fácil de concebir para los programadores. Normalmente un ciclo contiene tres partes: un valor de

inicio, una condición que evalúa el final del ciclo y un incremento (o decremento) que permite que se avance desde el valor del ciclo hasta el valor final. Esas tres partes se reflejan claramente en la estructura del ciclo for.

```
for ( Valor_Inicial ; Condición ; Incremento (Decremento) )  
{  
    Cuerpo del Ciclo  
}
```

En esta estructura se observa que la palabra *for* le da inicio al ciclo y seguidamente, entre paréntesis, se escriben las tres partes que se han comentado: el valor inicial del ciclo, la condición y el incremento o decremento. En condiciones normales, estas tres partes están relacionadas con una variable que se conoce como *índice del ciclo* pues es la que permite que el ciclo avance o finalice dependiendo del valor que almacene.

#### 8.4.1 Ejemplo resuelto

Construir un programa que permita leer dos números enteros: el primero servirá como base y el segundo servirá como exponente de manera que se pueda calcular el resultado de elevar dicha base a dicho exponente.

Al igual que en los otros programas, lo primero que vamos a hacer es importar la Clase *JOptionPane* que es la que permite que podamos utilizar las ventanas para lectura de datos.

```
import javax.swing.JOptionPane;
```

A continuación damos inicio a la Clase Ciclos3 y al método principal de esta clase.

```
public class Ciclos3 {  
  
    public static void main(String args[ ] ) {
```

Ahora vamos a declarar dos variables *String* y dos variables enteras. En las variables *String* vamos a almacenar las cadenas que se reciben a través de las ventanas de captura. Recuerde que la información que se recibe a través de una ventana *JOptionPane* siempre entra en forma de cadena. En

las variables enteras (*int*) vamos a almacenar la conversión de dichas cadenas en los respectivos números enteros que representan para poder realizar las operaciones y hacer el cálculo que se requiere.

```
String bas, ex;           // Variables Cadena  
int base, expon;         // Variables Enteras
```

Seguidamente, desplegamos dos ventanas (una seguida de la otra): la primera para leer la base y la segunda para leer el exponente.

```
bas = JOptionPane.showInputDialog("Digite la base");  
ex = JOptionPane.showInputDialog("Digite el  
exponente");
```

A continuación, convertimos la cadena que acabamos de leer como base (*bas*) en el número entero que represente. De la misma forma, convertimos la cadena que acabamos de leer como exponente (*ex*) en el número entero que representa.

```
base = Integer.parseInt(bas);  
expon = Integer.parseInt(ex);
```

Como ya tenemos la base y el exponente que necesitamos para resolver el enunciado, entonces procedemos a hacer el cálculo. Para ello vamos a declarar una variable que nos va a servir como índice del ciclo, es decir, es la que va a controlar el proceso por lo tanto será la variable que tomará valores desde 1 hasta el valor del exponente (*expon*) incrementando de 1 en 1. También declaramos una variable (*result*) que almacenará el valor de la multiplicación que se debe realizar, o sea, el resultado de elevar la base al exponente.

```
int i, result=1;
```

Planteamos un ciclo *for* que nos va a permitir llevar la variable *I* desde 1 hasta el valor del exponente con incrementos de 1. Por cada valor que tome la variable se realizará la multiplicación *result\*base* y ese resultado se irá almacenando progresivamente en la misma variable *result*.

```
for ( i=1 ; i <= expon ; ++i )  
    result *= base;
```

Finalmente, mostramos el resultado en la consola de la forma más clara posible para el usuario y cerramos tanto el paréntesis del método principal como el paréntesis de la Clase Ciclos3.

```
System.out.println(base+" elevado a la "+expon+" = "+result);  
}  
}
```

Tenga en cuenta algunos detalles. El uso de paréntesis determina el agrupamiento de instrucciones. Si el ciclo *for* requiere que se ejecuten varias instrucciones como *Cuerpo de Ciclo* entonces se necesita encerrar dicho bloque de instrucciones entre paréntesis. Los llamados pueden resumirse de una forma más práctica pero eso lo haremos más adelante cuando ya se haya alcanzado un nivel mayor de destreza en el uso de este lenguaje de programación. Por ejemplo, el conjunto de instrucciones:

```
bas = JOptionPane.showInputDialog("Digite la base");  
ex = JOptionPane.showInputDialog("Digite el exponente");  
  
base = Integer.parseInt(bas);  
expon = Integer.parseInt(ex);
```

Se pudo haber escrito de la siguiente forma:

```
base = Integer.parseInt ( JOptionPane.showInputDialog("Digite la base") );  
expon= Integer.parseInt ( JOptionPane.showInputDialog("Digite el  
exponente") );
```

Debido a que lo que retorna el método *showInputDialog* (de la Clase *JOptionPane*) es una cadena e inmediatamente esa cadena sirve como argumento al método *parseInt* (de la Clase *Integer*) para que sea convertida en el número correspondiente. Esto sucede tanto para la variable *base* como para la variable *expon*. Es claro que lo que se obtiene en este programa es un resultado que se pudo haber calculado acudiendo al método *exp* de la Clase *Math*, sin embargo es un ejercicio muy interesante plantear soluciones que no solo dependan de los métodos que ya tiene predefinido Java sino que sean soluciones propias. Sobra decir que para leer un dato se puede hacer tanto acudiendo a la Clase *JOptionPane* como a la Clase *Scanner*.

### 8.4.2 Ejercicios Propuestos

Utilizando la sintaxis del ciclo *for*, plantear soluciones para cada uno de los siguientes enunciados:

- a. Construir un programa Java que permita leer un número entero positivo y mostrar todos los números comprendidos entre 1 y el número leído
- b. Construir un programa Java que permita leer un número entero positivo y mostrar todos los dígitos del número (uno por uno) de manera vertical
- c. Construir un programa Java que permita leer un número entero positivo y mostrar, por consola, la suma de los dígitos del número
- d. Construir un programa Java que permita leer un número entero positivo y mostrar la cantidad de dígitos del número
- e. Construir un programa Java que permita leer un número entero positivo y mostrar la cantidad de dígitos pares del número
- f. Construir un programa Java que permita leer un número entero positivo y mostrar la cantidad de dígitos impares del número
- g. Construir un programa Java que permita leer un número entero positivo y mostrar la cantidad de dígitos primos del número
- h. Construir un programa Java que permita leer un número entero positivo y mostrar la suma de los dígitos pares del número
- i. Construir un programa Java que permita leer un número entero positivo y mostrar la suma de los dígitos impares del número
- j. Construir un programa Java que permita leer un número entero positivo y mostrar la suma de los dígitos primos del número



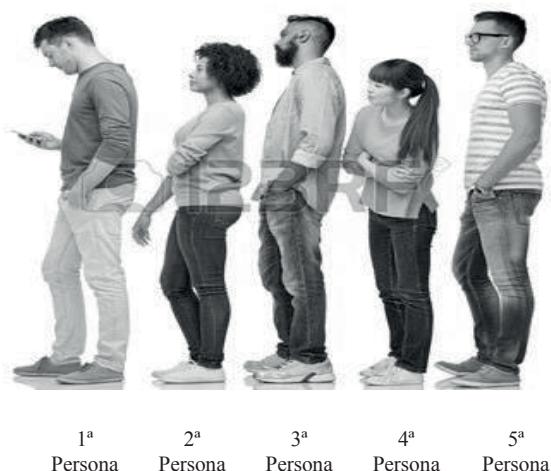
# 9 Lección

# Vectores



## 9.1 Definición

Un arreglo se define como un conjunto de datos del mismo tipo que están almacenados en un espacio de memoria subdividido en donde cada dato se puede referenciar por la posición que ocupa dentro del arreglo. Al igual que cuando usted está haciendo una fila en un banco para pagar algo. De repente el cajero dice “Venga la 3<sup>a</sup> persona de la fila”. Usted, que está en la 2<sup>a</sup> posición, mira extrañado a la 3<sup>a</sup> persona pues a ella la van a atender primero que a usted sabiendo que usted había llegado antes que ella. Usted no duda en mirar la persona que está detrás de usted porque está totalmente seguro que si usted ocupa el 2º puesto en la fila entonces quien le sigue ocupa el 3º puesto en esa misma fila.



En este caso es claro que cada persona ocupa un puesto determinado dentro de la fila y, como no existe duda de la posición de cada uno y cada miembro se puede referenciar por su posición, entonces se podría decir que este es un arreglo de personas.

## 9.2 Utilidad

Los vectores son muy útiles cuando necesitamos manipular conjuntos de datos que tienen relación entre sí y que corresponden al mismo tipo de datos. Por ejemplo, si a usted le dicen que debe calcular el promedio entero

de las edades de los estudiantes de un curso de Programación y le dicen que el siguiente es el conjunto de edades:

$$\text{Edades} = \{15, 16, 14, 15, 16, 16, 16, 17, 15, 14, 15, 15, 16, 16, 14, 16, 15, 16, 17, 17, 18\};$$

Entonces esto podría indicarle a usted que se encuentra ante un arreglo de datos pues son del mismo tipo y cada uno ocupa una posición específica dentro del arreglo. Nota usted que hay datos repetidos pero eso no importa pues al fin y al cabo dos o más estudiantes del curso de programación pueden tener la misma edad.

¿Qué podría hacer usted con este conjunto de datos? Podría determinar a cuánto es igual el promedio de las edades del curso, como inicialmente se dijo, pero también podría determinar cuál es la edad del más joven, la edad del más viejo, cuántos estudiantes tienen la mínima edad, cuántos tienen la máxima edad en el curso, cuántos estudiantes hay, en cuántos estudiantes su edad es igual al promedio entero de las edades, etc. Todos estos manejos de los datos se pueden facilitar si en vez de manipularlos uno por uno, se manejan en conjunto como si fueran un arreglo.

### 9.3 Tipos de arreglos

La organización de un conjunto de datos, desde la perspectiva de la programación, puede tener varias formas pues cuando se tiene una fila de datos es una de ellas, cuando los datos están organizados en varias filas es otra. Por esta razón, y solo para facilitar la concepción de los arreglos, se han establecido unos tipos en donde todos acuden a que el concepto central sea el de Matriz.

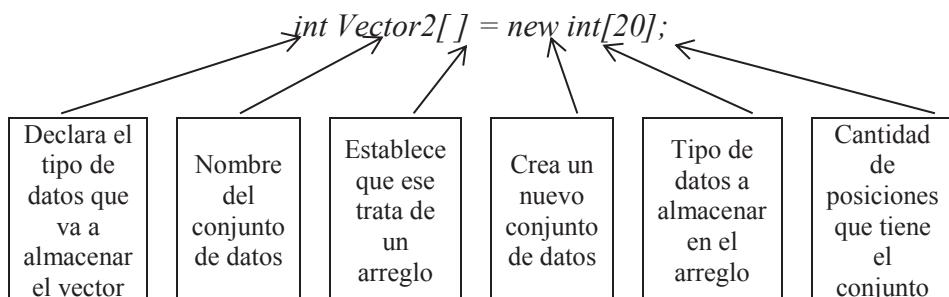
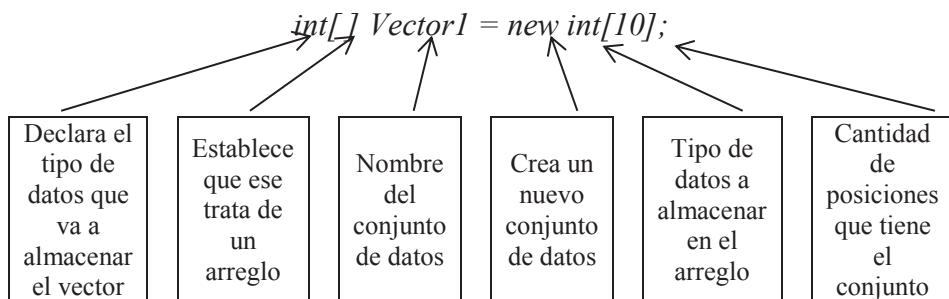
Una matriz es, por lo tanto, un arreglo de datos homogéneos (o sea que son del mismo tipo) organizado en forma de filas y columnas. Si bien todos los arreglos están formados por un conjunto de datos, la diferencia entre uno y otro radica en la cantidad de filas que tengan. De allí que los tipos de datos giran alrededor de la cantidad de filas tal como se describe en la siguiente tabla:

Nombre	Descripción	Ejemplo
Arreglo Unidimensional	Consiste en una matriz con una sola fila.	Una fila de personas esperando pagar un recibo en un banco

	También se conocen como <i>Vectores</i>	
Arreglo Bidimensional	Consiste en una matriz con varias filas y varias columnas	Un salón de clases en donde los estudiantes están organizados en forma de filas y columnas
Arreglo Tridimensional	Consiste en varias matrices cada una con varias filas y varias columnas. Se conocen también como Cubos Lógicos	Cada uno de los elementos que conforman el cubo de Rubik pues cada cuadrito pertenece a una cara y, en conjunto, las seis caras conforman el cubo
Arreglo N - Dimensional	Consiste en varios conjuntos de matrices en donde cada una tiene varias filas y varias columnas	La ubicación de un estudiante en una fila y una columna en un salón determinado de un piso de un edificio de algunas de las sedes de una universidad

#### 9.4 Declaración de un arreglo

En el lenguaje de programación Java, los arreglos se pueden definir de dos formas que son equivalentes:



Una vez que se ha declarado una matriz, que en este caso es de una sola fila, se puede utilizar para almacenar los datos.

## 9.5 Carga de datos en un arreglo

Para facilitar el llenado de datos del arreglo se pueden acudir a cuatro formas:

- Carga de datos en conjunto
- Carga de datos individual
- Carga de datos por digitación
- Carga de datos por operación

### 9.5.1 Carga de datos en conjunto

Al declarar un arreglo se puede, al mismo tiempo, cargar los datos que se quieren almacenar en el arreglo.

```
int [] VectorI = { 4, 5, 6, 7, 6, 8, 7, 5, 6, 4, 5};
```

Esta inicialización declara un vector y éste toma la dimensión que permite que se le almacenen los datos que están dentro de las llaves. En este caso, se trata de 11 datos y por lo tanto el vector quedará con esa dimensión (es decir, con esa capacidad de datos). Más adelante veremos cómo hacer para tener clara la dimensión del vector valiéndonos de un método de la clase Vector.

### 9.5.2 Carga de datos individual

Otra forma de inicializar un vector podría ser la siguiente:

```
int [] VectorI = new int[6];
```

```
VectorI[0]=5;  
VectorI[1]=4;  
VectorI[2]=6;  
VectorI[3]=8;  
VectorI[4]=2;  
VectorI[5]=3;
```

Tenga en cuenta que la primera posición de un arreglo es la posición 0. En este caso podría ser bastante dispendioso llenar un arreglo (sea cual fuera

su característica) en caso de que se hubiera pensado en muchos datos para almacenar. Sin embargo, en algunos casos, es bueno conocer esta opción para llenar un arreglo cuando se trata de pocos datos.

### 9.5.3 Carga de datos por digitación

Otra forma de llenar los datos en un vector es a través de la información que digite un usuario. Ya sabemos que esta información puede entrar usando la *Clase JOptionPane* o también la *Clase Scanner*. Mostraremos como cargamos los datos usando la Clase Scanner.

```
int [] Vector1 = new int [10];
Scanner Tecla = new Scanner(System.in);

for ( int i = 0; i < 10; ++i )
    Vector1[i] = Tecla.nextInt();
```

Como se puede observar, se declarar un vector de 10 posiciones (esto quiere decir que va desde la posición 0 hasta la posición 9), se declara también una instancia de la *Clase Scanner* que se asocia con el teclado (*System.in*). Luego, valiéndonos de una variable tipo entera que oficia como índice del ciclo, le vamos dando valores entre 0 y 9 y, por cada valor de esta variable, vamos almacenando un dato en el vector, producto de lo que digite el usuario.

Tenga en cuenta que, si bien en este ejemplo se ha utilizado la *Clase Scanner* para la lectura de datos, en cualquier momento se puede utilizar la *Clase JOptionPane* para recibir los datos a través de una ventana. Recuerde que se recibe dato por dato. También recuerde que lo que entra por una ventana tipo *JOptionPane* llega como tipo *String* y por lo tanto debe convertirse. Si es un entero debe convertirse al entero correspondiente a través de *Integer.parseInt()* y si es un *double* debe convertirse al número real correspondiente a través de *Integer.parseDouble()*. Ambos métodos requieren que se les envíe como argumento el nombre de la cadena a convertir.

### 9.5.4 Carga de datos por operación

Una forma más de guardar datos en un vector es utilizando una operación que sea coherente con los objetivos del programa. Supongamos que se quiere almacenar datos en un vector y que cada dato almacenado en una

casilla del vector es el doble del dato almacenado en la casilla anterior. Por lo tanto podríamos hacer lo siguiente:

```
int [] Vector1 = new int[10];  
  
Vector1[0] = 1;  
  
for ( int i = 1; i <= 9 ; ++i )  
    Vector1[i] = 2 * Vector1[i-1];
```

Recuerde que el nombre Vector1 es un nombre cualquiera que se le ha dado al vector que hemos utilizado para los ejemplos. Este nombre puede ser cualquiera que usted le quiera conferir. De esta forma podríamos llenar el vector tal como se ha indicado en el enunciado.

## 9.6 Despliegue de datos de un arreglo

Como ya vimos cómo se entran datos a un vector, ahora veremos cómo se muestran los datos que ya están almacenados en un vector. Si conocemos la dimensión del vector, esta podría ser una forma de mostrar los datos almacenados, a través de la consola:

```
for ( int i = 0; i < 5 ; ++i )  
    System.out.println("Vector [ " + i + " ] = " + Vector1[ i ]);
```

Asumimos en este ejemplo que el vector ya tiene los datos almacenados y que tiene capacidad para almacenar 5 datos. Si se ejecutaran estas dos instrucciones y formaran parte de un programa, una salida podría ser:

```
Vector[0] = 4  
Vector[1] = 5  
Vector[2] = 8  
Vector[3] = 5  
Vector[4] = 9
```

Estamos asumiendo que los datos almacenados son los que aparecen en la pantalla. Ahora bien, en caso de que no se conozca la dimensión del vector, entonces una forma recorrerlo para mostrar sus datos es la siguiente:

```
for ( int i = 0; i < Vector1.length ; ++i )  
    System.out.println("Vector [ " + i + " ] = " + Vector1[ i ]);
```

De esta forma, Java encuentra el tamaño del vector y muestra los datos de una forma similar a como se mostraron en el ejemplo anterior.

### 9.7 Ejercicios Propuestos

- a. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine en qué posición está el número menor
- b. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine en qué posición está el número mayor
- c. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine en qué posición está el menor número par
- d. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine en qué posición está el menor número impar
- e. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine en qué posición está el menor número terminado en 3
- f. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine cuántos números pares hay en el vector
- g. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine cuántos números impares hay en el vector
- h. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine cuántos números terminados en 5 hay en el vector
- i. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine cuántas veces está el número menor
- j. Construir un programa que lea un conjunto de 10 datos y, después de leídos, determine cuántas veces está el número mayor

### 9.8 Ciclo for – each

Este es un ciclo relativamente nuevo que facilita el acceso a los datos de un arreglo y simplifica notoriamente el código. Supongamos que tenemos un vector con datos y queremos calcular el cuadrado de cada uno de los datos. La solución la podríamos plantear como sigue.

En primer lugar damos inicio tanto a la *Clase* (que en este caso la llamaremos *Cicloforeach*) como a su método principal (que siempre se llama *main*).

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

Seguidamente, asumamos que hemos declarado un arreglo unidimensional (vector o matriz de una sola fila) y lo hemos llenado con datos.

```
int [] Vec = { 2, 3, 4, 5, 6, 1, 8, 7, 9, 11 };
```

Ahora vamos a declarar una variable tipo *int* (que llamaremos *num*) y la vamos a asociar con el vector que hemos llamado *Vec*. De forma que esta variable tome cada valor del vector *Vec* y, por cada vez que tome un valor que esté en una posición diferente, va a ejecutar el cuerpo del ciclo, de la siguiente forma:

```
for(int num:Vec)  
    System.out.println(num+" a la 2 = "+Math.pow(num,2));
```

En este caso el cuerpo del ciclo está conformado por una sola línea que mostrará tanto cada dato del vector como el resultado de elevarlo al cuadrado. Tenga en cuenta que la conexión entre la variable *num* y el vector *Vec* se hace a través del signo de dos puntos ( : ). Finalmente cerramos tanto la llave de la función principal (*main*) como de la Clase (*Cicloforeach*).

```
}  
}
```

El código completo de este programa se verá de la siguiente forma:

```
public class Cicloforeach {  
  
    public static void main(String args[]){  
  
        int [] Vec = {2,3,4,5,6,1,8,7,9,11};  
  
        for(int num:Vec)  
            System.out.println(num + " a la 2 = " +  
                Math.pow(num,2));  
    }  
}
```

Si se ejecuta este programa Java, el resultado que aparece en consola será el siguiente:

```
2 a la 2 = 4.0
3 a la 2 = 9.0
4 a la 2 = 16.0
5 a la 2 = 25.0
6 a la 2 = 36.0
1 a la 2 = 1.0
8 a la 2 = 64.0
7 a la 2 = 49.0
9 a la 2 = 81.0
11 a la 2 = 121.0
```

En donde se muestra cada valor almacenado en el vector y el resultado respectivo de elevarlo al cuadrado.

Veamos otro ejemplo. Supongamos que tenemos un vector de cadenas y que, en cada posición, tenemos almacenado el nombre de una persona. El objetivo es escribir la longitud de cada cadena, o sea, cuántos caracteres tiene cada una.

Damos inicio a la *Clase* que hemos llamado *Cicloforeach2* y asimismo iniciamos el método principal (*main*).

```
public class Cicloforeach2 {
    public static void main(String args[] ) {
```

Declaramos un arreglo unidimensional y guardamos dentro de dicho arreglo el nombre de los amigos. Note usted que este arreglo no tiene una dimensión específica sino que toma como tamaño la cantidad de datos que se le han guardado.

```
String []
Amigos={"Carlos","Mariana","Alberto","Juan","Julio","Jorge"};
```

Ahora declaramos una variable String que hemos llamado *Friend* y que asociamos con cada uno de los nombres almacenados en el vector *Amigos*.

```
for(String Friend:Amigos)
```

Finalmente mostramos por consola el nombre de cada uno de los amigos con la respectiva longitud, o sea, la cantidad de caracteres que tiene cada nombre. Igualmente cerramos las llaves tanto de la función principal (*main*) como de la Clase *Cicloforeach2*.

```
System.out.println("Nombre "+Friend+"\t"+Friend.length() + "letras");
}
}
```

El programa completo se presenta a continuación:

```
public class Cicloforeach2 {

public static void main(String args[ ] ) {

String [ ] Amigos =
{"Carlos","Mariana","Alberto","Juan","Julio","Jorge"};

for(String Friend:Amigos)
System.out.println("Nombre "+Friend+"\t\t"+ Friend.length() + "letras");
}
}
```

Al ejecutarlo el resultado que se obtiene en la consola es el siguiente:

Nombre Mariana	7 letras
Nombre Alberto	7 letras
Nombre Juan	4 letras
Nombre Julio	5 letras
Nombre Jorge	5 letras

## 9.9 Ejercicios Propuestos

Utilizando la estructura del ciclo *for – each* construir una solución para los siguientes enunciados:

- a. Construir un programa Java que permita señalar en pantalla los números pares de un conjunto de 10 números
- b. Construir un programa Java que permita señalar en pantalla los números impares de un conjunto de 10 números
- c. Construir un programa Java que permita señalar en pantalla los números múltiplos de 2 de un conjunto de 10 números
- d. Construir un programa Java que permita señalar en pantalla los números múltiplos de 5 de un conjunto de 10 números
- e. Construir un programa Java que muestre en pantalla el resultado progresivo de sumar los números almacenados en un arreglo de 15 datos
- f. Construir un programa Java que permita calcular la raíz cuadrada de cada uno de los datos de un conjunto de 20 números
- g. Construir un programa Java que permita calcular la raíz cúbica de cada uno de los datos de un conjunto de 20 números
- h. Construir un programa Java que permita calcular la función seno de cada uno de los datos de un conjunto de 20 números
- i. Construir un programa Java que permita calcular la función coseno de cada uno de los datos de un conjunto de 20 números
- j. Construir un programa Java que permita calcular la función tangente de cada uno de los datos de un conjunto de 20 números



# 10 Lección

# Matrices



## 10.1 Definición

Una matriz es un arreglo de datos homogéneos (del mismo tipo) organizado en forma de filas y columnas de manera que imite, de la manera más cercana posible, la realidad que representa. Cuando un grupo de estudiantes se sientan en un salón, normalmente los pupitres están organizados en forma de filas y columnas de manera que si quisieramos representar el salón en la memoria del computador podría hacerse a través de una matriz.

Debe tenerse en cuenta que la representación de una realidad es solo una esquematización mas no es una instantánea y por ello la matriz es un modelo que se aproxima al entorno que representa para facilitar su manipulación, su intervención y, en algunos casos, la simulación de sus comportamientos.

Por ejemplo, pudiéramos tener un salón de 25 alumnos en un salón. Hipotéticamente podríamos representar dicho salón de la siguiente forma:


Cada casilla corresponde a un estudiante y, de acuerdo a lo establecido por el manejo de los arreglos, podríamos escribir el número de cada fila y de cada columna.

	Col 0	Col 1	Col 2	Col 3	Col 4
Fila 0 →					
Fila 1 →					
Fila 2 →					
Fila 3 →					
Fila 4 →					

Se puede observar que las columnas y las filas comienzan su numeración en 0, la igual que con los vectores. Ahora bien, si quisieramos almacenar las edades de los jóvenes de este curso, ubicándolas en los puestos en donde cada estudiante se sienta, entonces podríamos tener lo siguiente:

	Col 0	Col 1	Col 2	Col 3	Col 4
Fila 0 →	14	16	13	13	16
Fila 1 →	15	14	14	14	14
Fila 2 →	13	15	15	14	16
Fila 3 →	13	15	14	15	14
Fila 4 →	14	16	15	16	15

Como se puede ver, la matriz representa un esquema de la realidad que es el salón de clases. Si bien es posible que los estudiantes no estén, en la vida real, tan organizados como se observa esta matriz, debe admitirse que es una buena aproximación que permite manejar los datos que se involucran en el contexto del salón. Podríamos, por lo tanto, tener otra matriz en donde almacenemos los nombres de los estudiantes.

	Col 0	Col 1	Col 2	Col 3	Col 4
Fila 0 →	Carlos	Pedro	Juan	Luis	Ana
Fila 1 →	Luis	Camilo	José	Jorge	Julio
Fila 2 →	Julio	Kathy	Carlos	Iván	Pablo
Fila 3 →	Juan	Natalia	Luis	Javier	Robert
Fila 4 →	María	José	Omar	José	Pedro

Para diferenciarlas podríamos decir que la matriz que almacena las edades de los estudiantes podría llamarse EDAD y que la matriz que almacena los nombres de los estudiantes podría llamarse NOMBRE.

### 10.3 Declaración de una matriz

Una matriz, por ser un arreglo bidimensional (conformado por filas y columnas), se puede declarar de una forma similar a la forma como se declara un vector.

```
int [][] matriz1 = new int[5][5];
```

Esta línea estaría declarando una matriz de 5 filas con 5 columnas cada una, tal como las que se han utilizado para ejemplificar la matriz EDAD y la matriz NOMBRE. Otra forma de hacerlo es la siguiente:

```
int matriz1[][] = new int[5][5];
```

Esta declaración abre un espacio de 5 filas y 5 columnas para que se almacenen datos de tipo entero.

## 10.4 Carga de datos en una matriz

En referencia con la carga de datos de una matriz la situación es similar a la del vector teniendo en cuenta, eso sí, que una matriz requiere de dos subíndices (para indicar tanto las filas como las columnas) mientras que el vector requiere solamente un subíndice (que indica la posición del dato dentro de la colección de datos).

### 10.4.1 Carga de datos en conjunto

```
int [][] mat1 = { { 3, 4, 6, 5, 4 }, { 5, 3, 4, 2, 3 }, { 4, 4, 3, 3, 4 },
{ 3, 5, 4, 2, 3 }, { 3, 5, 5, 4, 5 } };
```

### 10.4.2 Carga de datos individual

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

```
mat[0][0]=3;
mat[0][1]=4;
mat[0][2]=6;
mat[0][3]=5;
mat[0][4]=4;
```

```
mat[1][0]=5;
mat[1][1]=3;
mat[1][2]=4;
mat[1][3]=2;
mat[1][4]=3;
```

```
mat[2][0]=4;  
mat[2][1]=4;  
mat[2][2]=3;  
mat[2][3]=3;  
mat[2][4]=4;
```

```
mat[3][0]=3;  
mat[3][1]=5;  
mat[3][2]=4;  
mat[3][3]=2;  
mat[3][4]=3;
```

```
mat[4][0]=3;  
mat[4][1]=5;  
mat[4][2]=5;  
mat[4][3]=4;  
mat[4][4]=5;
```

#### 10.4.3 Carga de datos por digitación

```
int [ ] [ ] matriz = new int [5][5];  
Scanner Teclado = new Scanner(System.in);  
  
System.out.println("Digite 24 números enteros:");  
for(int i = 0; i <= 4; ++i )  
    for (j=0; j <= 4; ++j )  
        matriz [i] [j] = Teclado.nextInt();
```

En el caso de la lectura por digitación sobra decir que se puede utilizar tanto la *Clase Scanner* como la *Clase JOptionPane*.

#### 10.5 Despliegue de datos en una matriz

En cuanto al despliegue de datos de la matriz, éstos se pueden mostrar valiéndonos de dos ciclos anidados (es decir, un ciclo dentro de otro) de forma que se recorra la matriz fila por fila y, dentro de cada fila, se recorra posición por posición.

```
for (int i = 0; i <= 4; ++i )  
    for (j=0; j <= 4; ++j )  
    {  
        System.out.print("Matriz fila " +i+ " columna "+j+ " = ");  
        System.out.println(matriz [i][j] );  
    }
```

## 10.6 Ejercicios Propuestos

- a. Construir un programa Java que llene una matriz de 5 filas por 3 columnas y que calcule la suma de todos los datos almacenados en la matriz
- b. Construir un programa Java que llene una matriz de 3 filas por 6 columnas (3x6) y que calcule la suma de cada fila de datos de la matriz
- c. Construir un programa Java que llene un matriz de 6x4 y que determine cuál es el valor menor almacenado en toda la matriz
- d. Construir un programa Java que llene una matriz 3x5 y que determine cuál es el valor menor de cada fila de la matriz
- e. Construir un programa Java que llene una matriz 4x6 y que determine en qué posición (en qué fila y en qué columna) está el valor menor de toda la matriz
- f. Construir un programa Java que llene una matriz 3x8 y que determine en qué posición está el valor menor de cada fila
- g. Construir un programa Java que llene un matriz de 6x4 y que determine cuál es el valor mayor almacenado en toda la matriz
- h. Construir un programa Java que llene una matriz 3x5 y que determine cuál es el valor mayor de cada fila de la matriz
- i. Construir un programa Java que llene una matriz 4x6 y que determine en qué posición (en qué fila y en qué columna) está el valor mayor de toda la matriz
- j. Construir un programa Java que llene una matriz 3x8 y que determine en qué posición está el valor mayor de cada fila



# 11 Lección

## Aproximación a la POO



## 11.1 Paradigmas de programación

Un *paradigma* puede definirse como un conjunto de teorías (o incluso una teoría) cuyo planteamiento central es aceptado sin ningún cuestionamiento que es el que brinda la base y el modelo que posibilitan la resolución de problemas y, de paso, el avance en el conocimiento dentro de un área determinada. Un *paradigma* también es una forma de concebir determinada situación de manera que dicha forma es aceptada por una comunidad determinada que, normalmente, tiene una relación directa con la situación estudiada.

Como *Programación de Computadores* (conocida simplemente como Programación) se conoce el arte de lograr que el computador alcance unos objetivos, que normalmente se cristalizan en resultados numéricos, valiéndose de unas instrucciones que, dentro del marco de un lenguaje de programación, son entendibles y ejecutables por un computador. La programación de computadores es una tendencia que busca aprovechar al máximo las capacidades de un dispositivo electrónico con capacidades de almacenamiento, con capacidades de procesamiento, con capacidades de respuesta a nivel de información y con capacidades de recepción de información. El dispositivo que mejor cumple con estas características es el computador.

Un *paradigma de programación* es un enfoque determinado que permite tener unos parámetros específicos, definidos, estandarizados y aceptados para plantear posibles soluciones a problemas que sean computables, es decir, que se puedan resolver apoyándose en el computador. En la evolución de los computadores, y específicamente en el desarrollo de los lenguajes de programación, se conocen dos enfoques: el enfoque de la programación que se fundamenta en una orientación a los procedimientos y el enfoque de la programación orientado a objetos.

El enfoque de programación orientado a procedimientos prioriza los estados, los procesos, el concepto de función y destaca como base para dichos procesos el sistema de almacenamiento a través de variables o el sistema de procesamiento a través del concepto de función. Por su parte, el enfoque orientado a objetos prioriza la relación entre los métodos y los

atributos como mecanismo de aproximarse a la realidad que se intenta describir desde la programación de computadores.

El enfoque de programación orientado a objetos ha ido tomando mucha más fuerza, por encima de los otros paradigmas, debido a que pareciera ser el que más se aproxima a la descripción de la realidad tal como la concebimos e interactuamos con ella. Normalmente la realidad debe ser ajustada para pensar en representarla o simularla a través de un computador sin embargo bajo el paradigma de la programación OO (orientada a objetos) los ajustes de interpretación son pocos y, por lo tanto, la representación en muchos casos es idéntica a la realidad, tal como sucede en los juegos de computador.

## 11.2 Objetos en el mundo real

Un *Objeto*, en el mundo real, no es más que todo aquello tangible o intangible que puede ser descrito por sus características y/o por sus usos. Todo lo que nos rodea tiene usos y características. Los usos lo podríamos llamar *Métodos* y las características las podríamos llamar *Atributos*. Sobre esta base podríamos decir, a la luz de la POO, que un objeto es el conjunto formado por la unión de unos atributos y los métodos de manera que abren el camino para interacciones y características adicionales.

Veamos un ejemplo: un lápiz tiene características (atributos) y usos (métodos). Entre las características de un lápiz podríamos decir que éste tiene un peso, un color, un sabor, una textura, una densidad, un tamaño, un grosor, etc. y entre los usos podríamos decir que un lápiz sirve para escribir, para señalar, para indicar algo, para rascarnos, para masticar, para aplanar algo, etc. Usted puede notar que si bien algunas características y algunos usos son los convencionales también algunos podrían un poco informales. Sin embargo son características y usos que son válidos. Una *Clase* es la forma como se puede declarar un dato con las características de un *Objeto* de manera que posteriormente se puedan crear tantas instancias como sean necesarias. Una instancia de una clase es un objeto en sí.

## 11.3 Características propias de la POO

La programación orientada a objetos ha permitido que los objetos tengan unas características que los distinguen y que posibilitan la representación de la realidad tangible e intangible que conocemos. Entre las principales

características de los objetos, establecidas por la POO podemos citar las siguientes:

- **Modularización.** Un objeto puede ser descompuesto en partes y, a su vez, un objeto siempre es una parte de otro objeto mayor
- **Herencia.** Un objeto podría parecerse a otro en muchos de sus atributos (características) y en muchos de sus métodos (usos) sin embargo. Asimismo puede tomar características de otro y puede ser usado en lo que usan otro objeto
- **Encapsulamiento.** Muchas veces un objeto es una incógnita para uno pues no siempre podemos saber cómo está construido, por dentro y detalladamente, dicho objeto
- **Tratamiento de excepciones.** En cualquier momento a un objeto se le puede dar un uso que jamás se hubiera pensado y eso le puede conferir nuevas características
- **Polimorfismo.** Un objeto puede adoptar varias formas diferentes y seguir siendo el mismo objeto

### 11.3 ¿Qué es la POO?

La POO consiste en describir, a partir de las características (atributos) y usos (métodos) de las cosas, la realidad que nos rodea y, de esta forma, poder simular las relaciones que se pueden dar entre dichas cosas (llamadas aquí *Objetos*). Luego programar bajo el enfoque orientado a objetos dependerá, en gran medida, de la aproximación que tengamos de la realidad, de la forma como describamos dichos objetos tanto en sus atributos como en los usos que éstos pueden tener.

Así como para declarar una variable de tipo entera primero tenemos que escribir el tipo de dato (*int*) asimismo para declarar un *Objeto* primero debemos describir la *Clase* a la cual pertenece dicho *Objeto*, es decir, un *Objeto* es una instancia de determinada *Clase*. Por ejemplo, la *Clase Prenda\_de\_Vestir* podría tener una instancia llamada *Camisa*, otra instancia llamada *Camiseta*, otra podría ser *Chaqueta* y entonces cada una de éstas corresponderá a un *Objeto* de la *Clase* mencionada. Por esta razón, lo primero que vamos a ver es cómo declaramos una Clase.

## 11.4 Creación de una Clase

Para crear una Clase lo primero que debemos hacer, por organización de la información debido a que podríamos llegar a tener muchas Clases declaradas, es asociarla con un paquete. Normalmente los ambientes integrados de desarrollo (IDE – Integrated Development Environment) son programas que facilitan muchas de las operaciones que deben hacerse, desde el código, para que el programa quede bien escrito.

Algunos de esos ambientes son BlueJ, Eclipse o Netbeans. Hay muchos más pero éstos podrían estar entre los más comerciales. Si usted va a trabajar con alguno de ellos, notará que le brindan muchas facilidades para que se haga más simple escribir programas Java que, de por sí, tiene momentos en donde la construcción del código es un poco dispendiosa.

Para declarar una Clase, lo primero que hacemos es asociarla con un paquete (como dijimos anteriormente, solo por organización de la información). Internamente, en el disco duro, un paquete es una carpeta en donde se van a almacenar todas las *Clases* que escribamos y que queramos que formen parte de él. Por tal motivo, aprovechamos las facilidades del entorno que estemos utilizando y veremos que nos pondrá automáticamente la instrucción:

```
package poo;
```

Seguidamente, y tal como lo hemos hecho hasta ahora, declaramos una Clase.

```
public class ClaseBase {
```

Ahora, supongamos que nuestra *ClaseBase* (que es la clase que utilizaremos de ejemplo) describe algo que tiene tres atributos: un dato entero, un dato real y una cadena. Entonces realizamos las declaraciones pertinentes teniendo en cuenta los tipos respectivos de datos.

```
int Dato1;           // Declara un atributo de tipo entero
double Dato2;        // Declara un atributo de tipo double
String Nombre;       // Declara un atributo de tipo cadena
```

Con esto sería suficiente para declarar una Clase y ya podríamos declarar Objetos (o instancias de Clase) basados en este nuevo “tipo” de datos. Sin

embargo cuando construimos una clase podríamos, de una vez, darle a sus atributos valores de inicio y hacer otras operaciones para que se hagan efectivas en el momento en que se crea un nuevo Objeto.

Para ello, escribimos un primero método que se conoce como *Constructor*. Su nombre se debe a que tiene el mismo nombre de la Clase, es decir, si la clase se llama *ClaseBase* entonces su método *Constructor* se llama también *ClaseBase()* solo que, al acompañarlo con paréntesis, eso indica que es un método.

```
public ClaseBase() {  
    Dato1 = 10;  
    Dato2 = 20.0;  
    Nombre = "Omar I";
```

De esta forma, al declarar un Objeto del “tipo” *ClaseBase* automáticamente se carga el número 10 en la variable **int** *Dato1*, el número 20.0 en la variable **double** *Dato2* y la cadena “Omar I” en la variable tipo **String** *Nombre*. Solo restaría cerrar la llave del método *Constructor* y de la Clase *ClaseBase*.

```
    }  
}
```

El código completo de esta Clase (que hemos llamado *ClaseBase*) se muestra a continuación:

```
package poo;  
  
public class ClaseBase {  
  
    int Dato1;  
    double Dato2;  
    String Nombre;  
  
    public ClaseBase() {  
  
        Dato1 = 10;  
        Dato2 = 20.0;  
        Nombre = "Omar I";
```

```
}
```

## 11.5 Creación de un Objeto

Como ya tenemos creada la **Clase ClaseBase** entonces ya podemos crear un Objeto que tenga las características de esta clase. Para ello, lo primero que hacemos es abrir espacio para crear una nueva clase. Por ahora no se preocupe mucho por el uso de los términos que parecieran ser confusos pero realmente son mucho más simples de lo que parecen.

Cuando se habla en este libro de crear una nueva clase, me refiero a abrir un archivo para crear lo que podríamos llamar un nuevo programa Java. Como lo que vamos a hacer es utilizar la **Clase ClaseBase** entonces lo primero que debemos realizar es incluir el paquete con el cual se quiere trabajar. En este caso se refiere a un paquete que, en este ejemplo, hemos llamado **poo**.

```
package poo;
```

Esto nos permitirá usar la **Clase ClaseBase** cuyo código forma parte del paquete **poo** (es decir, está almacenada en la carpeta **poo**). A continuación procedemos como lo hemos hecho hasta ahora. Iniciamos una nueva Clase. Recuerde que en Java todo se basa en Clases. Igualmente declaramos, como lo habíamos venido haciendo, el método principal (*main*) que en la **ClaseBase** no se utilizó.

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

Como la **ClaseBase** está lista para ser usada la utilizamos como si fuera un tipo de dato. Procedemos a declarar la instancia **Objeto1** de “tipo” **ClaseBase** y a continuación invocamos el constructor para indicar que se crea una nueva clase con el “tipo” especificado.

```
    ClaseBase Objeto1 = new ClaseBase();
```

El primer uso de la **ClaseBase**, en la línea anterior, le indica a Java que se va a crear un objeto con las características que se describieron y el segundo

uso de la *ClaseBase()* le indica a Java que **Objeto1** comenzará, en sus atributos, con los valores que se almacenan en su método constructor.

Como ya creamos un objeto llamado Objeto1 y éste es una instancia de la Clase ClaseBase entonces ya podemos mostrar en pantalla los datos almacenados en dicho objeto. ¿En qué momento se almacenaron? Esa es la labor que se hace con el método constructor. Para ello se usa la notación punto que implica escribir el nombre del objeto y el nombre del atributo que queremos referenciar, separándolos con un punto.

```
System.out.println("Dato Entero:\t" + Objeto1.Dato1);
System.out.println("Dato Real:\t" + Objeto1.Dato2);
System.out.println("Cadena :\t" + Objeto1.Nombre);
```

Finalmente, cerramos la llave del método principal (*main*) y de la *Clase ClaseEjem* que es como hemos llamado a esta Clase.

```
}
```

El código completo de este programa Java es el siguiente:

```
package poo;

public class ClaseEjem {
    public static void main(String args[]) {

        ClaseBase Objeto1 = new ClaseBase();

        System.out.println("Dato Entero:\t" + Objeto1.Dato1);
        System.out.println("Dato Real:\t" + Objeto1.Dato2);
        System.out.println("Cadena :\t" + Objeto1.Nombre);
    }
}
```

Tenga en cuenta que *ClaseBase* no se ejecuta, se ejecuta *ClaseEjem* que es la que tiene un método principal llamado *main()*. Al ejecutar este programa (o sea ClaseEjem) en consola aparecerá lo siguiente:

Dato Entero: 10

Dato Real: 20.0  
Cadena : Omar I

Que corresponde a los datos almacenados en cada uno de los atributos del *objeto Objeto1* declarado bajo las características y el constructor de la *Clase ClaseBase*.

Veamos ahora un ejemplo un poco más práctico. Vamos a construir una clase que se va a llamar **ClaseTriángulo**. Para ello primero vamos a construir la clase y después la usaremos para construir una clase con función principal que use la *ClaseTriángulo* y, de esa forma, veamos qué otras cosas podemos hacer en este naciente y fascinante mundo de la POO.

En este caso hemos creado un paquete llamado figuras que va a contener la Clase ClaseTriangulo. Por esta razón lo primero que hacemos es incluir el paquete y “abrir” la Clase para comenzar a construirla.

```
package figuras;  
  
public class ClaseTriangulo {
```

A continuación, tratándose de un triángulo (teóricamente), vamos a declarar los tres atributos del triángulo: la base, la altura y el área. Es de anotar que la base y altura son los atributos que distinguen cada triángulo y el área es el resultado de operar la base y la altura a partir de la fórmula  $\frac{(base \times altura)}{2}$ . Todos estos atributos son de tipo entero (*int*).

```
int base;  
int altura;  
int area;
```

Seguidamente creamos el método *Constructor* de la *ClaseTriangulo* que lleva el mismo nombre de la clase. Abrimos la llave y dentro de ellas escribimos lo que consideramos importante realizar cada que se llegue a crear una instancia de esta Clase. En este caso hemos optado porque se almacene en la variable **base** el valor 5, en la variable **altura** el valor 10 y en la variable **area** el resultado de realizar la fórmula del área del triángulo.

```
public ClaseTriangulo( ) {
```

```
base = 5;  
altura = 10;  
  
area = (base * altura) / 2;
```

Finalmente cerramos las llaves tanto del método *Constructor* como de la Clase *ClaseTriangulo*.

```
}
```

El código completo de esta Clase es el siguiente:

```
package figuras;  
  
public class ClaseTriangulo {  
  
    int base;  
    int altura;  
    int area;  
  
    public ClaseTriangulo() {  
  
        base = 5;  
        altura = 10;  
  
        area = (base * altura) / 2;  
    }  
}
```

Ahora vamos a utilizar esta Clase para construir un programa Java. Para ello de nuevo, lo primero que hacemos es incluir el nombre del *paquete* *figuras* y abrimos tanto la *Clase* (que en este caso la llamamos *Ejemplo*) como la función principal (*main*).

```
package figuras;  
  
public class Ejemplo {  
  
    public static void main(String args[] ) {
```

Seguidamente creamos una instancia de la Clase *ClaseTriangulo* a la cual llamamos *Trian1* y, después del signo =, llamamos al método *Constructor* de esta clase hacer efectiva esta instancia.

*ClaseTriangulo Trian1 = new ClaseTriangulo();*

Al crear la instancia de clase *Trian1*, o sea el objeto *Trian1*, automáticamente se asignan los valores que aparecen en el método *ClaseTriangulo* gracias a la acción del método *Constructor*. Por esta razón todo lo que tenemos que hacer es mostrarlos en pantalla pues ya se encuentran almacenados.

```
System.out.println("Base = " + Trian1.base);
System.out.println("Altura:" + Trian1.altura);
System.out.println("Area =" + Trian1.area);
```

Finalmente cerramos las llaves que aún están abiertas y queda concluido este programa Java.

```
}
```

El código completo de este programa es el siguiente:

```
package figuras;

public class Ejemplo {

    public static void main(String args[ ] ) {

        ClaseTriangulo Trian1 = new ClaseTriangulo();

        System.out.println("Base = " + Trian1.base);
        System.out.println("Altura:" + Trian1.altura);
        System.out.println("Area =" + Trian1.area);
    }
}
```

Al ejecutar este programa, aparecerá en consola lo siguiente:

Base =	5
Altura:	10
Area =	25

Que efectivamente corresponde a la ejecución del código que se programó. Con estos elementos conceptuales, ya puede usted mismo crear sus primeros programas Java basados en POO. No se vaya a afanar porque a este enfoque de programación todavía le falta mucho por conocerle. Por ahora, disfrute lo visto y realice los ejercicios.

## 11.6 Ejercicios Propuestos

- a. Construir una clase con tres atributos de una casa, crear la Clase Casa y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase Casa y muestre sus resultados
- b. Construir una clase con tres atributos de una fruta, crear la Clase Fruta y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
- c. Construir una clase con cuatro atributos de un empleado, crear la Clase Empleado y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
- d. Construir una clase con tres atributos de una silla, crear la Clase Silla y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
- e. Construir una clase con cinco atributos de un carro, crear la Clase Carro y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
- f. Construir una clase con tres atributos de un computador, crear la Clase Computador y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
- g. Construir una clase con tres atributos de un ser humano, crear la Clase SerHumano y crear su método Constructor que le asigne valores a los

- atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
- h. Construir una clase con cuatro atributos de una ciudad, crear la Clase Ciudad y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
  - i. Construir una clase con tres atributos de una trompeta, crear la Clase Trompeta y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados
  - j. Construir una clase con tres atributos de un libro, crear la Clase Libro y crear su método Constructor que le asigne valores a los atributos que se crearon. Adicional construya un programa Java que utilice esta Clase y muestre sus resultados

# 12 Lección

## Método constructor con parámetros



## 12.1 Reflexión

Si en el programa anterior se hicieran las siguientes declaraciones:

```
ClaseTriangulo Trian1 = new ClaseTriangulo();  
ClaseTriangulo Trian2 = new ClaseTriangulo();  
ClaseTriangulo Trian3 = new ClaseTriangulo();  
ClaseTriangulo Trian4 = new ClaseTriangulo();
```

Todos los objetos declarados (o sea *Trian1*, *Trian2*, *Trian3* y *Trian4*) tendrían los mismos valores en sus atributos *base*, *altura* y *area*. ¿Sería útil que fuera así? Lo más posible es que su respuesta sea negativa pues lo útil es que pudiéramos declarar triángulos con diferentes medidas, es decir, diferentes bases, diferentes alturas y diferentes áreas. Eso es precisamente lo que vamos a estudiar. La idea de poder crear un objeto, que es una imitación de la realidad, es poder lograr que los objetos tengan sus diferencias. Si quisiéramos crear 4 triángulos sirve de muy poco el hecho de que todos tengan la misma *base* y la misma *altura* (y que, por ende, tengan la misma *área*). Lo interesante es que se puedan construir triángulos con medidas diferentes y que, con estos datos, se puedan calcular las áreas respectivas de cada uno.

¿Recuerda usted en qué consiste el **método Constructor** en la declaración de una *Clase*? Este método, que tiene el mismo nombre de la *Clase*, permite que realicemos operaciones básicas que consideremos pertinentes en la creación de un *Objeto* de determinada *Clase*. Por ejemplo, supongamos que en la creación de un *Objeto* de la *Clase ClaseTriangulo* nos interesa que éste se pueda crear escribiendo nosotros los atributos por cada *Objeto* creado, es decir, construyendo los triángulos con medidas diferentes.

## 12.2 Método Constructor de Clase con parámetros

Lo primero que debemos hacer es definir el paquete con el cual vamos a trabajar. Recuerde que un paquete es algo así como una carpeta en donde quedarán almacenados los archivos pertinentes. Este paquete lo vamos a llamar *param*.

```
package param;
```

A continuación iniciamos la *Clase* que vamos a llamar *ClaseconParam* (como haciendo referencia a que es una *Clase con Parámetros*).

```
public class ClaseconParam {
```

Ahora vamos a declarar los atributos de esta Clase de ejemplo que, para el efecto, corresponden a una variable de tipo **int**, una de tipo **double** y otra de tipo **String**.

```
    int Dato1;  
    double Dato2;  
    String Cadena;
```

Ahora vamos a construir el método Constructor de *ClaseconParam*. Para este efecto, debe llamarse como la misma clase o sea *ClaseconParam()* que lo declaramos **public** por razones que más adelante revisaremos. A diferencia de los métodos constructores de las clases que hemos declarado en los otros ejemplos, esta vez vamos a declarar unas variables dentro de los paréntesis que acompañan la declaración del método Constructor.

Entre estos paréntesis vamos a declarar un parámetro de tipo **int** que llamaremos *Var1*, otro parámetro de tipo **double** que llamaremos *Var2* y otro de tipo **String** que llamaremos *Cad*. Estos son los parámetros que utilizaremos para que reciban los valores que queremos usar para crear cada *Objeto* de la *Clase*.

Seguidamente, almacenamos cada parámetro en su atributo respectivo. Esto es, almacenamos el valor recibido en *Var1* en el atributo *Dato1*, almacenamos el valor recibido en *Var2* en el atributo *Dato2* y almacenamos la cadena recibida en *Cad* en el atributo *Cadena*. De esta forma, el *Objeto* que se quiera crear asumirá como propios los valores que se le envíen.

```
public ClaseconParam(int Var1, double Var2, String Cad) {
```

```
    Dato1 = Var1;  
    Dato2 = Var2;
```

*Cadena = Cad;*

Cerramos la llave del método *Constructor* y también cerramos la llave de la Clase *ClaseconParam*.

```
}
```

El código completo de esta Clase creada es el siguiente:

```
package param;
```

```
public class ClaseconParam {
```

```
    int Dato1;  
    double Dato2;  
    String Cadena;
```

```
    public ClaseconParam(int Var1, double Var2, String Cad) {
```

```
        Dato1 = Var1;  
        Dato2 = Var2;  
        Cadena = Cad;
```

```
}
```

Como puede observar, el método *Constructor* tiene unos parámetros que son los que usaremos en el programa Java que vamos a construir. Vamos a suponer que ahora queremos crear varios *Objetos* tipo *ClaseconParam* entonces procedemos de la siguiente forma. Debemos escribir un programa Java que lo permita. Para ello lo primero que hacemos es incluir el paquete en el que estamos trabajando.

```
package param;
```

A continuación iniciamos la Clase *EjemconParam* que es la que vamos a utilizar como ejemplo. Igualmente iniciamos el método principal (main) para que el programa se ejecute. A esta altura usted habrá notado que una diferencia, al menos en primera instancia, que se percibe entre una Clase y un programa Java (que siempre debe hacerse dentro de una Clase) es que

el programa Java tiene un método principal llamado *main()* mientras que la Clase no lo tiene.

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

Ahora vamos a declarar dos Objetos de tipo *ClaseconParam* y, al momento de crear cada *Objeto*, llamamos al método *ClaseconParam( )* (que es su método constructor) y le enviamos parámetros con valores que queremos almacenar en los atributos de la *Clase* en mención.

```
ClaseconParam CP1 = new ClaseconParam(10, 20.0, "Hola");  
ClaseconParam CP2 = new ClaseconParam(20, 40.0, "Adios");
```

El Objeto CP1 ha sido creado enviándole como parámetros el valor entero 10, el número double 20.0 y la cadena “Hola”. El Objeto CP2 ha sido creado enviándole como parámetros el valor entero 20, el número double 40.0 y la cadena “Adiós”. Estos valores se cargan, como lo indicamos en el método constructor, en los atributos del objeto. De forma que podemos mostrarlos por consola tal como se presenta a continuación:

```
System.out.println("Primer Objeto");  
System.out.println("Dato1 =|t" + CP1.Dato1);  
System.out.println("Dato2 =|t" + CP1.Dato2);  
System.out.println("Cadena=|t" + CP1.Cadena);  
  
System.out.println("\n\nSegundo Objeto");  
System.out.println("Dato1 =|t" + CP2.Dato1);  
System.out.println("Dato2 =|t" + CP2.Dato2);  
System.out.println("Cadena=|t" + CP2.Cadena);
```

Se acude a la notación punto para referenciar cada uno de los atributos de los objetos creados a partir de la Clase *ClaseconParam*. Finalmente cerramos tanto la llave del método principal (*main*) de esta Clase como la llave de la Clase *EjemconParam*.

```
}  
}
```

El código completo de este programa Java es el siguiente:

```
package param;

public class EjemconParam {

    public static void main(String args[]) {

        ClaseconParam CP1 = new ClaseconParam(10, 20.0, "Hola");
        ClaseconParam CP2 = new ClaseconParam(20, 40.0, "Adios");

        System.out.println("Primer Objeto\n");
        System.out.println("Dato1 =|t" + CP1.Dato1);
        System.out.println("Dato2 =|t" + CP1.Dato2);
        System.out.println("Cadena=|t" + CP1.Cadena);

        System.out.println("\n\nSegundo Objeto\n");
        System.out.println("Dato1 =|t" + CP2.Dato1);
        System.out.println("Dato2 =|t" + CP2.Dato2);
        System.out.println("Cadena=|t" + CP2.Cadena);
    }
}
```

Al ejecutar este programa, el resultado es el siguiente:

Primer Objeto

```
Dato1 =      10
Dato2 =      20.0
Cadena=     Hola
```

Segundo Objeto

```
Dato1 =      20
Dato2 =      40.0
Cadena=     Adios
```

Como se puede observar, y tal como lo indicamos en el código, se muestran los valores almacenados en cada uno de los atributos del **Primer**

Objeto (*CP1*) y los valores almacenados en cada uno de atributos del Segundo Objeto (*CP2*).

### 12.3 Construcción del Ejemplo ClaseTriangulo

Ahora vamos a construir el ejemplo del triángulo pero en una versión más práctica de forma que podamos configurar los atributos de manera que cada Objeto declarado sea diferente a los demás. Lo primero que haremos será incluir el paquete con el cual estamos trabajando.

```
package figuras;
```

A continuación creamos la Clase Triángulo con sus atributos (base, altura y area).

```
public class Triangulo {  
    int base;  
    int altura;  
    int area;
```

Seguidamente construimos el *método Constructor* de la *Clase Triangulo* que incluirá dos parámetros: *bas* (que representa la base) y *alt* (que representa la altura). Estos parámetros se cargarán con los valores que se escriban al momento de crear un nuevo Objeto y de invocar el método Constructor.

```
public Triangulo(int bas, int alt) {  
    base = bas;  
    altura = alt;  
    area = (base * altura)/2;
```

Note usted que se reciben, como parámetros, la base y la altura del triángulo y el área se calcula a partir de los valores almacenados en estos dos parámetros. Con esto cerramos el paréntesis del *método Constructor* y el paréntesis de la *Clase Triángulo*.

```
}
```

El código completo de esta Clase se muestra a continuación:

```
package figuras;  
  
public class Triangulo {  
    int base;  
    int altura;  
    int area;  
  
    public Triangulo(int bas, int alt) {  
        base = bas;  
        altura = alt;  
        area = (base * altura)/2;  
    }  
}
```

Ahora vamos a construir un programa Java que incluya la clase que ya se construyó (o sea la *Clase Triangulo*). Recuerde que un programa Java implica la presencia de un método llamado *main()* que corresponde al método principal. Iniciamos escribiendo el paquete que vamos a utilizar, iniciando una clase (que vamos a llamar *Triangulos*, con una letra s al final para diferenciarla de la *Clase Triangulo*). Igualmente creamos el método principal (*main*).

```
package figuras;  
  
public class Triangulos {  
    public static void main(String args[] ) {
```

Vamos a crear tres *Objetos* (que son instancias de la *Clase Triangulo*). En la creación de cada uno, escribimos los parámetros con los cuales se quieren crear los Objetos mencionados.

```
    Triangulo Trian1 = new Triangulo(20,15);  
    Triangulo Trian2 = new Triangulo(10,40);  
    Triangulo Trian3 = new Triangulo(40,20);
```

Crear los tres objetos implica que el 1º de ellos tiene 20 de base y 15 de altura, el 2º de ellos tiene 10 de base y 40 de altura y el 3º de ellos tiene 40 de base y 20 de altura. Eso implica que, posiblemente, el cálculo del

área de cada triángulo podría ser diferente. Procedemos a mostrar por los atributos de cada triángulo.

```
System.out.println("\nPrimer Triangulo\n");
System.out.println("Base:\t" + Trian1.base);
System.out.println("Altura\t" + Trian1.altura);
System.out.println("Area:\t" + Trian1.area);

System.out.println("\nSegundo Triangulo\n");
System.out.println("Base:\t" + Trian2.base);
System.out.println("Altura\t" + Trian2.altura);
System.out.println("Area:\t" + Trian2.area);

System.out.println("\nTercer Triangulo\n");
System.out.println("Base:\t" + Trian3.base);
System.out.println("Altura\t" + Trian3.altura);
System.out.println("Area:\t" + Trian3.area);

}

}
```

Finalmente cerramos las llaves del método principal (*main*) y las llaves de la Clase *Triangulos*.

El código completo de este programa Java se presenta a continuación:

```
package figuras;

public class Triangulos {
    public static void main(String args[]) {
        Triangulo Trian1 = new Triangulo(20,15);
        Triangulo Trian2 = new Triangulo(10,40);
        Triangulo Trian3 = new Triangulo(40,20);

        System.out.println("\nPrimer Triangulo\n");
        System.out.println("Base:\t" + Trian1.base);
        System.out.println("Altura\t" + Trian1.altura);
        System.out.println("Area:\t" + Trian1.area);

        System.out.println("\nSegundo Triangulo\n");
        System.out.println("Base:\t" + Trian2.base);
```

```
System.out.println("Altura\t" + Trian2.altura);
System.out.println("Area:\t" + Trian2.area);

System.out.println("\nTercer Triangulo\n");
System.out.println("Base:\t" + Trian3.base);
System.out.println("Altura\t" + Trian3.altura);
System.out.println("Area:\t" + Trian3.area);
}

}
```

Al ejecutar este programa, los resultados que aparecen en consola son los siguientes:

Primer Triangulo

```
Base: 20
Altura 15
Area: 150
```

Segundo Triangulo

```
Base: 10
Altura 40
Area: 200
```

Tercer Triangulo

```
Base: 40
Altura 20
Area: 400
```

Cumpliendo con lo que se había programado. Nótese la gran ventaja que se tiene de poder construir, por el mecanismo del *método Constructor* con parámetros, triángulos con diferentes características. Ahora vamos a pensar ¿cómo haríamos si se quisiera que los atributos de cada Objeto (o sea, cada triángulo) fueran digitados por el usuario? Eso es lo que veremos en el siguiente numeral.

### 12.3 Método Constructor de Clase con parámetros digitados

A diferencia de los otros ejemplos, esta vez usted mismo revisará el código tanto de la *Clase* que se va a utilizar como del programa Java que la utiliza. Verifique la documentación (o sea los comentarios en el programa) y podrá

entender claramente cómo se reciben los parámetros por el teclado y cómo se configuran, con ellos, los *Objetos* que se declaran.

```
// *****
// Código de la Clase
// *****
// Se incluye el paquete respectivo
package figuras;

// Se inicia la Clase Triángulo y se declaran los atributos
public class Triangulo {
    int base;
    int altura;
    int area;

    // Se crea el método Constructor con dos parámetros
    // y con su contenido se "cargan" los atributos
    public Triangulo(int bas, int alt) {
        base = bas;
        altura = alt;
        area = (base * altura)/2;

    }                                // Fin del método Constructor
                                    // Fin de la Clase Triángulo

// *****
// Código del Programa Java
// *****
// Inclusión del paquete con el cual estamos trabajando
package figuras;

// Se importa el paquete requerido para leer por el teclado
import java.util.Scanner;

// Declaración de la Clase
public class TrianconParamDigitxUsu {

    // Método Principal
```

```
public static void main(String args[ ] ) {  
  
    // Se declaran las variables locales del método principal  
    int b1, a1, b2, a2; // corresponden a altura y base  
  
    // Se crea una instancia del Objeto Scanner  
    // Para leer por el teclado  
    Scanner Tec = new Scanner(System.in);  
  
    // Se leen los parámetros del 1er objeto triángulo  
    System.out.print("Base 1: ");  
    b1 = Tec.nextInt();  
    System.out.print("Altura 1: ");  
    a1 = Tec.nextInt();  
  
    // Se leen los parámetros del 2o objeto triángulo  
    System.out.print("Base 2: ");  
    b2 = Tec.nextInt();  
    System.out.print("Altura 2 : ");  
    a2 = Tec.nextInt();  
  
    // Se crean dos objetos (triángulos)  
    // cada uno con valores leidos por el teclado  
    Triangulo T1 = new Triangulo(b1, a1);  
    Triangulo T2 = new Triangulo(b2, a2);  
  
    // Se muestran en pantalla los datos  
    // del 1er objeto triángulo  
    System.out.println("\nTriangulo 1\n");  
    System.out.println("Base:\t" + T1.base);  
    System.out.println("Altura\t" + T1.altura);  
    System.out.println("Area:\t" + T1.area);  
  
    // Se muestran en pantalla los datos  
    // del 2o objeto triángulo  
    System.out.println("\nTriangulo 2\n");  
    System.out.println("Base:\t" + T2.base);  
    System.out.println("Altura\t" + T2.altura);  
    System.out.println("Area:\t" + T2.area);
```

```
    }           //fin del método principal  
}
```

Note usted la importancia de la documentación (comentarios) para entender claramente un programa con la preparación previa requerida.

## 12.4 Ejercicios Propuestos

- Construir una Clase (y su programa Java respectivo) que permita crear Cuadrados y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Rectángulos y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Pentágonos y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Hexágonos y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Octágonos y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Decágonos y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Dodecágonos y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Pirámides y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Cubos y que muestre, apropiadamente, sus atributos, su área y su perímetro
- Construir una Clase (y su programa Java respectivo) que permita crear Octaedros y que muestre, apropiadamente, sus atributos, su área y su perímetro

# 13 Lección

## Modularización y encapsulación



### 13.1 ¿Qué es la Modularización?

El término *modularizar* podría interpretarse como la posibilidad de construir un programa a partir de las relaciones que se puedan establecer entre diferentes partes. En programación orientada a objetos, así como en otros paradigmas, es de gran importancia pensar en que la solución de un problema computable debe concebirse por la interacción de dichas partes.

Si bien es posible construir una solución a partir de un programa completo, es mucho mejor pensar en que se construyan módulos y se relacionen para que se pueda lograr el objetivo de resolver el problema computable por un camino mucho más sencillo. En programación de computadores debe uno tender a buscar siempre la solución más sencilla pues de eso se trata aprovechar la lógica deliberativa humana para encontrar e implementar una solución computacional. Veamos el siguiente código:

```
package poo;  
  
public class Clase2 {  
  
    int dato1;  
    int dato2;  
    String cad;  
  
    public Clase2() {  
  
        dato1=10;  
        dato2=20;  
        cad="Hola";  
    }  
  
    public static void main(String args[] ) {  
  
        Clase2 Obj1 = new Clase2();  
  
        System.out.println("Dato1 = " + Obj1.dato1);  
        System.out.println("Dato2 = " + Obj1.dato2);  
        System.out.println("Cadena = " + Obj1.cad);  
    }  
}
```

```
}
```

Observe usted que es un solo archivo que contiene todo lo que se necesita para que el programa funcione. Incluye el paquete *poo*, declara una Clase llamada Clase2 que tiene como atributos tres variables (*dato1*, *dato2* y *cad*). Luego se declara el método Constructor que carga unos valores en los atributos del objeto y finalmente, en el método principal, se declara un objeto tipo Clase2 y se muestran dichos atributos en la consola.

Si bien este es un programa demasiado sencillo, permitirá ver la diferencia entre *modularizar* y no hacerlo pues, en esta primera versión, se puede observar que todo el código se encuentra en un solo archivo. Ahora veamos la versión *modularizada* de este programa tan sencillo.

// PRIMER ARCHIVO

```
package poo;  
  
public class Clase2 {  
  
    int dato1;  
    int dato2;  
    String cad;  
  
    public Clase2() {  
  
        dato1=10;  
        dato2=20;  
        cad="Hola";  
    }  
}
```

// SEGUNDO ARCHIVO

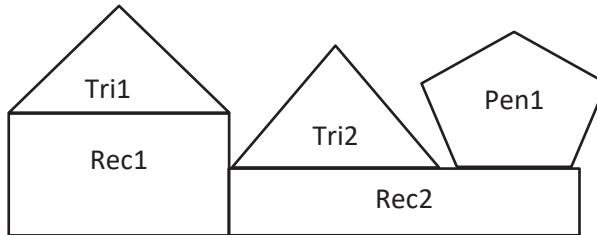
```
package poo;  
  
public class Ejemplo2 {
```

```
public static void main(String args[ ] ) {  
    Clase2 Obj1 = new Clase2();  
  
    System.out.println("Dato1 = " + Obj1.dato1);  
    System.out.println("Dato2 = " + Obj1.dato2);  
    System.out.println("Cadena = " + Obj1.cad);  
}  
}
```

El *Primer Archivo* contiene la declaración de la Clase, sus atributos y el método constructor. El *Segundo Archivo* contiene un programa Java que utiliza la clase y, gracias al método constructor, carga sus atributos y muestra su contenido en pantalla. Ahora vamos a ver un ejemplo mucho más útil en la construcción de un programa aprovechando la *modularización*.

### 13.2 Ejemplo Resuelto

Construir un programa Java que permita calcular el área total de la siguiente figura:



En esta figura las medidas de cada polígono son las siguientes:

Nombre	Tri1
Figura	Triángulo
Base	5
Altura	3

Nombre	Rec1
Figura	Rectángulo

Base	5
Altura	4

Nombre	Tri2
Figura	Triángulo
Base	5
Altura	4

Nombre	Rec2
Figura	Rectángulo
Base	8
Altura	2

Nombre	Pen1
Figura	Pentágono
Lado	3
Apotema	2

Si tuviéramos que hacerlo manualmente, la solución consistiría en calcular el área de cada uno de las figuras, sumar dichas áreas y tendríamos el resultado del área total que nos solicitan, pero como se trata de calcular el área de toda la figura pero a través de un programa, entonces, vamos a acudir a la *modularización* para evidenciar la utilidad de unir partes para resolver un problema computable. Veamos una versión altamente modularizada.

Para facilitar la comprensión de este código, se ha designado cada elemento que forma parte de la solución como un ARCHIVO y un número que lo identifique. De esta forma, el *ARCHIVO 1* contiene la declaración de la *Clase Triangulo* cuyos atributos (*base*, *altura* y *área*) describen las variables que se pueden involucrar en el cálculo del área de un triángulo convencional de lados rectos. De la misma manera, el método constructor de esta *Clase*, almacena en los atributos *base* y *altura*, los parámetros que le sean enviados desde la creación de un nuevo objeto. El atributo *área* se calcula con la aplicación de la fórmula del área de un triángulo.

// ARCHIVO 1

*package* figuras;

```
public class Triangulo {  
  
    int base;  
    int altura;  
    double area;  
  
    public Triangulo(int b, int a) {  
        base = b;  
        altura = a;  
        area = (double)(base * altura)/2;  
    }  
}
```

Por su parte, el *ARCHIVO 2* contiene la declaración de la *Clase Rectangulo* cuyos atributos *base*, *altura* y *área* (al igual que en el *ARCHIVO 1*) posibilitan el cálculo del área del triángulo. Aunque los atributos del *ARCHIVO 2* tienen el mismo nombre de los atributos del *ARCHIVO 1*, Java distingue uno de otros al momento de la creación de los objetos correspondientes.

El método constructor de la *Clase Rectangulo* recibe dos parámetros que, a su vez, se almacenan en los atributos *base* y *altura* para poder realizar el cálculo del área del rectángulo utilizándolos a la luz de la fórmula que lo permite. Es de anotar que tanto en el *ARCHIVO 1* como en el *ARCHIVO 2* cálculo del área se realiza con variables de tipo *int* pero se requiere que el resultado se obtenga con decimales (tipo *double*) para que se pueda almacenar en la variable *area* que es del mismo tipo.

```
// ARCHIVO 2  
  
package figuras;  
  
public class Rectangulo {  
  
    int base;  
    int altura;  
    double area;  
  
    public Rectangulo(int b, int a) {
```

```
base = b;  
altura = a;  
area = (double) base * altura;  
}  
}
```

El ARCHIVO 3 tiene una descripción muy similar a la de cada uno de los archivos anteriores. En este archivo se declara la *Clase Pentagono* que, apoyado en sus atributos y gracias a su *método Constructor*, permite calcular el área de un pentágono basado en la teoría de que un pentágono equivale a la unión lateral de 5 triángulos iguales que tienen como base el lado del pentágono y como altura, la apotema (que es la distancia entre el centro de dicha figura geométrica y la mitad de cualquiera de sus lados. El área del pentágono se calcula basado en este postulado.

```
// ARCHIVO 3  
  
package figuras;  
  
public class Pentagono {  
  
    int lado;  
    int apotema;  
    double area;  
  
    public Pentagono(int l, int a) {  
  
        lado = l;  
        apotema = a;  
        area = (double) 5 * ((lado*apotema)/2);  
    }  
}
```

En este ejemplo, lo que se hace es crear un programa Java en donde en su método principal (*main*) se declaran dos triángulos (*Tri1* y *Tri2*), dos rectángulos (*Rec1* y *Rec2*) y un pentágono (*Pen1*) con las características de sus respectivas clases. Al crearlos, aprovechando las bondades del método constructor de cada uno, se pasan los parámetros que fueron dados al inicio del enunciado y que corresponden a las medidas requeridas en cada uno de los polígonos para poder calcular su respectiva área.

El cálculo del área total corresponderá a la suma de las áreas de *Tri1*, *Tri2*, *Rec1*, *Rec2* y *Pen1*. Este valor se almacena en una variable que se ha llamado *AreaTotal* y que es de tipo *double*. Al final, se muestra en pantalla el resultado del cálculo del área total.

// ARCHIVO 4

```
package figuras;  
  
public class AreaTotal {  
  
    public static void main(String args[] ) {  
  
        Triangulo Tri1 = new Triangulo(5,3);  
        Triangulo Tri2 = new Triangulo(5,4);  
        Rectangulo Rec1 = new Rectangulo(5,4);  
        Rectangulo Rec2 = new Rectangulo(8,2);  
        Pentagono Pen1 = new Pentagono(3,2);  
  
        double AreaTotal =   Tri1.area + Tri2.area + Rec1.area +  
                            Rec2.area + Pen1.area;  
  
        System.out.println("Area Total = " + AreaTotal);  
    }  
}
```

Recuerde que, de todos estos programas, solamente el ARCHIVO 4 corresponde a un programa Java en sí dado que es el único que tiene un método principal (*main*). Al ejecutar este programa, el resultado es el siguiente:

Area Total = 68.5

Que era lo que se quería inicialmente. Si se hubiera querido mostrar todos los parámetros de cada uno de los polígonos, para documentar mejor el resultado, se hubiera podido hacer fácilmente dado que se cuenta con el acceso a todos ellos. Adecúe usted este programa para que muestre en pantalla todos los parámetros de todos los polígonos antes de mostrar el resultado del cálculo del área total.

### 13.4 Ejercicios Propuestos

- Construir un programa que permita leer dos números enteros y, basado en clases, calcule la suma de los dos números, la resta de los dos números, la multiplicación de los dos números y el cociente de los dos números
- Construir un programa que permita leer dos números enteros y, basado en clases, determinar si el 1º número es mayor que el 2º, si el 1º número es menor que el 2º, si los dos números son iguales y si los dos números son diferentes
- Construir un programa que permita leer un número entero y, basado en clases, determinar cuántos dígitos tiene, cuántos dígitos pares tiene, cuántos dígitos impares tiene, cuántos dígitos primos tiene y cuántas veces contiene el dígito 6
- Construir un programa que permita leer un número entero de tres dígitos y, basado en clases, mostrar solamente el 1º dígito, mostrar solamente el 2º dígito, mostrar solamente el 3º dígito, determinar si tiene algún dígito repetido y determinar si los tres dígitos son iguales
- Construir un programa que permita leer dos números enteros dos números enteros de tres dígitos y, basado en clases, determinar si tienen dígitos comunes, si todos los dígitos son diferentes o si la suma de los dígitos de un número es igual a la suma de los dígitos del otro

### 13.3 ¿Qué es la Encapsulación?

Recordemos el código del programa que construimos al inicio de este capítulo. Se declaró una clase llamada *Clase2* que tenía tres atributos: dos de tipo *int* y otro de tipo *String*. Su método constructor asigna a la variable *dato1* el valor 10, a la variable *dato2* el valor 20 y a la variable *cad* la palabra “*Hola*”.

```
package poo;  
  
public class Clase2 {  
  
    int dato1;  
    int dato2;  
    String cad;
```

```
public Clase2() {  
  
    dato1=10;  
    dato2=20;  
    cad="Hola";  
}  
}
```

Al declarar un objeto de tipo *Clase2*, el método constructor indica que debe iniciar con los valores 10, 20 y “Hola” en las variables *dato1*, *dato2* y *cad*, respectivamente. Tal como está el código, el programa *Ejemplo2* crea un objeto tipo *Clase2* llamado *Obj1*. Sin embargo, si se quisieran alterar los valores que asigna el método constructor a las variables propias de la clase *Clase2* se podría hacer sin ningún problema ya que un código como el que sigue sería completamente posible.

```
package poo;  
  
public class Ejemplo2 {  
  
    public static void main(String args[ ] ) {  
  
        Clase2 Obj1 = new Clase2();  
  
        Obj1.dato1 = 33;           // Línea 1  
        Obj1.cad="Adios";         // Línea 2  
  
        System.out.println("Dato1 = " + Obj1.dato1);  
        System.out.println("Dato2 = " + Obj1.dato2);  
        System.out.println("Cadena = " + Obj1.cad);  
  
    }  
}
```

Nótese que en las líneas marcadas como *Línea 1* y *Línea 2* aparecen dos asignaciones: una para almacenar el valor 33 en la variable *dato1* del objeto *Obj1* y otra para asignar la cadena “Adios” en la variable *cad* del mismo objeto. Con estas asignaciones se están alterando los valores que se asignaron desde el método constructor de manera que cuando se visualizan

los datos almacenados en las variables del objeto Obj1 aparecerá en pantalla lo siguiente:

```
Dato1 = 33  
Dato2 = 20  
Cadena = Adios
```

Que es una alteración de los valores iniciales establecidos en el método constructor. En este ejemplo esto no sería mayor problema pero tengamos en cuenta que la POO busca imitar la realidad tal como es luego si creamos un objeto determinado con unos valores iniciales (que obedecen a unas especificaciones determinadas por la realidad) no es muy conveniente que desde cualquier programa se puedan modificar dichos valores iniciales, al menos de una forma tan directa y sencilla.

**Encapsular** consiste en evitar que los datos almacenados en las variables propias de un objeto, a través del método constructor, sean modificados por cualquier programa que acceda y utilice la clase que los incluye. La encapsulación se logra haciendo que las variables propias de una clase solamente puedan ser reconocidas y modificadas desde la misma clase. Para ello nos valemos del identificador de acceso *private*.

Usted habrá notado que hasta el momento hemos utilizado el modificador *public* indicando que cualquier persona puede acceder a lo que hasta ahora hemos construido. Con el modificador de acceso *private* no solo establecemos restricciones para su acceso y modificación sino que lo ocultamos de forma que ni siquiera se conoce el nombre. Veamos ahora el siguiente ejemplo.

Si la declaración de la clase Clase2 se hiciera anteponiendo a cada tipo de dato de los atributos la palabra *private*, entonces el programa generaría errores en las líneas que se identificaron como *Línea 1* y *Línea 2* indicando que no se puede tener acceso a esas variables y, por lo tanto, no se puede modificar su contenido tan fácilmente.

```
package poo;  
  
public class Clase2 {  
  
    private int dato1;
```

```
private int dato2;  
private String cad;  
  
public Clase2() {  
  
    dato1=10;  
    dato2=20;  
    cad="Hola";  
}  
  
public int muestra_dato1() {  
    return dato1;  
}  
  
public String muestra_cad() {  
    return cad;  
}  
}
```

Note usted la estructura de las funciones que muestran los datos que queremos ver. Contienen primero el modificador de acceso (que en este caso es *public*), después el tipo de dato que se va a retornar y que puede ser de cualquier tipo de datos (primitivos o de usuario) que tenga Java para declarar variables u objetos. Seguidamente tiene el nombre del método acompañado con paréntesis.

Después aparece la llave que da inicio al cuerpo del método y, junto con las instrucciones que correspondan (que aquí se han omitido), se escribe al final la instrucción *return* seguida de la variable que se quiere mostrar. Es de anotar que el tipo de dato a retornar de este tipo de métodos (o sea el tipo de dato que aparece después del modificador de acceso *public*) debe coincidir con el tipo de la variable que acompaña la instrucción *return*.

### 13.4 Métodos tipo *Get* o Métodos *Getter*

Si queremos asignar valores desde adentro de la clase, este tipo de métodos podrán llevar sus paréntesis vacíos. Este tipo de métodos, o sea los que muestran valores internos de una clase que está encapsulada, se conocen como métodos tipo *Get* o métodos *Getter*.

De esta manera, se puede decir que en esta clase se cuenta con dos métodos tipo *Getter*: el método *muestra\_dato1()* que retorna un entero (*int*) y que es de acceso público y el método *muestra\_cad()* que retorna una cadena (*String*) y que también es de acceso público. El papel de estos métodos es mostrar el contenido de las variables que queremos mostrar y la manera como queremos mostrar dicho contenido. Para utilizarlos debe ajustarse el programa para que acceda a los métodos *Getter* que hemos diseñado. El siguiente programa lo ejemplifica:

```
package poo;  
  
public class Ejemplo2 {  
    public static void main(String args[] ) {  
        Clase2 Obj1 = new Clase2();  
  
        System.out.println("Dato1 = " + Obj1.muestra_dato1());  
        System.out.println("Cadena = " + Obj1.muestra_cad());  
    }  
}
```

Como se puede observar en el programa, para acceder a los métodos asociados a un objeto (exceptuando el método constructor) simplemente se recurre a la notación punto, es decir, se escribe el nombre del objeto y, después de un signo punto, se escribe el nombre del método que se quiere invocar. Nótese que, como se está invocando un método, los nombres de las variables no aparecen sino simplemente el resultado que retorne cada método invocado. Vale la pena anotar que en este ejemplo solamente se han construido dos métodos tipo *Getter* para mostrar el contenido de la variable *dato1* y el contenido de la variable *cad*, ambos atributos del objeto *Obj1*. Al ejecutar este programa, el resultado es el siguiente:

```
Dato1 = 10  
Cadena = Hola
```

### 13.5 Métodos tipo *Set* o Métodos *Setter*

Possiblemente usted se esté preguntando, ¿cómo se haría para cambiar los valores iniciales, asignados por el método constructor de una determinada clase, por otros valores que se requieran sabiendo que con el modificador

de acceso *private* eso resulta imposible? Para eso existen los métodos *Setter* que facilitan esa operación y cuyo objetivo es poder modificar los valores de las variables propias de una clase que tienen modificador de acceso *private*.

Veamos un ejemplo de este tipo de métodos que almacenan valores en las variables con modificador de acceso *private* y que corresponden a atributos de una clase. En el código que hemos construido, hemos incorporado un nuevo método llamado *establece\_dato()* y cuyo objetivo es recibir un parámetro tipo *int* y almacenar su valor en la variable *dato1* declarada con modificador de acceso *private*. Este método está escrito al final del siguiente código. El método *establece\_dato()* ha sido declarado como tipo *void* para indicar que es un método que no retorna nada ya que, efectivamente, su objetivo es almacenar un dato en uno de los parámetros propios de la clase *Clase2*.

```
package poo;  
  
public class Clase2 {  
  
    private int dato1;  
    private int dato2;  
    private String cad;  
  
    public Clase2() {  
  
        dato1=10;  
        dato2=20;  
        cad="Hola";  
    }  
  
    public int muestra_dato1() {  
        return dato1;  
    }  
  
    public String muestra_cad() {  
        return cad;  
    }  
  
    public void establece_dato1(int d) {
```

```
dato1=d;  
}  
}
```

A continuación vamos a mostrar la manera de cambiar el valor almacenado en un atributo (cuyo contenido fue asignado en el método constructor) a través de un método tipo *Setter*.

```
package poo;  
  
public class Ejemplo2 {  
  
    public static void main(String args[ ] ) {  
  
        Clase2 Obj1 = new Clase2();  
  
        System.out.println("Dato1 = " + Obj1.muestra_dato1());  
        System.out.println("Cadena = " + Obj1.muestra_cad());  
  
        Obj1.establece_dato1(333);  
        System.out.println("\n\nNuevo dato1 = " + Obj1.muestra_dato1());  
    };  
}
```

Note usted que las dos últimas líneas de este programa, o sea, las siguientes:

```
Obj1.establece_dato1(333);  
System.out.println("\n\nNuevo dato1 = " + Obj1.muestra_dato1());
```

En la 1<sup>a</sup> de estas dos líneas, se asigna el valor 333 a la variable *dato1* a través del método *establece\_dato1()* que es un método tipo *Setter* y seguidamente se muestra en pantalla el nuevo valor almacenado en *dato1* a través del método *muestra\_dato1()* que es de tipo *Getter*. El valor 333 se le envía al parámetro d del método *establece\_dato1()* y de allí, a través del cuerpo del método, se almacena en la variable *dato1* que es uno de los atributos de la clase *Clase2*. Al ejecutar este programa, el resultado es el siguiente:

```
Dato1 = 10  
Cadena = Hola
```

```
El nuevo dato1 es 333
```

Según este resultado, se muestra primero el valor almacenado en la variable dato1 (que es igual a 10) que fue asignado por el método constructor y luego, cuando se cambia el valor de la variable a través del método *Setter*, se muestra el nuevo valor en consola (valor que corresponde al número 333). El proceso de encapsulación oculta procesos, variables y muchos detalles de la clase que no se quieren dar a conocer y, de esta forma, podemos garantizar que solamente se realicen los cambios que autoricemos desde adentro de las mismas clases.

### 13.6 Ejercicios Propuestos

- Construir un programa con clase encapsulada que permita mostrar las dimensiones de un triángulo. Estas dimensiones corresponderán a una base igual a 20 y una altura igual a 40. Estos valores serán válidos solamente si no se establecen valores diferentes
- Construir un programa con clase encapsulada que permita calcular el área de un triángulo mostrando solamente el resultado del área sin dejar acceder a sus dimensiones
- Construir un programa con clase encapsulada que permita mostrar las dimensiones de un pentágono. Estas dimensiones corresponderán a un lado igual a 20 y una apotema igual a 17. Estos valores serán válidos solamente si no se establecen valores diferentes
- Construir un programa con clase encapsulada que permita calcular el área de un pentágono a partir de la concepción de que el área de un pentágono es igual a la suma de las áreas de los cinco triángulos isósceles que lo conforman. Mostrar las dimensiones del pentágono junto con su área
- Construir un programa con clases encapsuladas que reciba un valor que representa una temperatura en grados Celsius (o grados Centígrados) y los convierta en su equivalente a grados Farenheit y viceversa
- Construir un programa con clases encapsuladas que reciba un valor que representa una cantidad de dinero en dólares y la convierta en su equivalente a pesos colombianos y viceversa



# 14 Lección

## Modificadores de acceso



## 14.1 Constantes

Una variable cuyo contenido no debe cambiar dentro de la ejecución de un mismo programa se conoce como Constante. Las constantes son muy necesarias en programación debido a que son muchos las variables cuyo contenido debe ser incambiable para darle coherencia al manejo de los datos desde la perspectiva de un lenguaje de programación.

Veamos un ejemplo sencillo. Supongamos que usted debe llenar un formulario que le solicita el número de identificación, el nombre y el salario. Cada dato tiene unas características propias tanto en su caracterización como en su uso. En lo práctico, es decir, en su uso el salario puede incrementarse con el tiempo bien porque la empresa donde usted trabajar toma la decisión de aumentar anualmente el salario o bien porque así lo dispone el Gobierno y promulga una ley que incremente el salario de los empleados. Esto significa que, a nivel de datos, el salario no es un valor constante y por lo tanto almacenarlo en una variable sería muy conveniente dada la dinámica que éstas tienen cuando se manejan a través de un lenguaje de programación.

En cuanto al nombre propio de una persona, la ley ya permite en muchos países que una persona se cambie el nombre e incluso que cambie el orden de sus dos apellidos. Esto ha posibilitado que las personas que cambian de sexo puedan tener una existencia legal digna. De acuerdo a esto, aunque no es muy usual, el nombre de una persona podría ser variable y entonces convendría también almacenarlo en una variable tipo *String* que hayamos pensado para tal fin. El caso especial lo constituye el número de identificación que, bajo ninguna circunstancia, debe cambiar puesto que, en todos los países, este número es único e irrepetible. Esto conlleva a que pensemos en que al construir una clase que vaya a almacenar los datos número de identificación, nombre y salario deberemos asegurarnos que el primero de ellos no pueda ser modificado una vez se le haya asignado un valor.

## 14.2 El modificador *final*

Cuando se declara una variable con cualquiera de los tipos primitivos propios del lenguaje de programación, se le está habilitando para que su

contenido pueda ser modificado en cualquier momento. De esta manera si se declara una variable de la siguiente forma:

**String** Nombre;

Se está indicando al intérprete de Java que en cualquier momento se le puede hacer una asignación como:

Nombre = “Omar”;

...y en cualquier otra parte del programa se puede asignar otro nombre a esta cadena como por ejemplo:

Nombre = “Iván”;

No hay restricciones en cuanto a la asignación del contenido de una variable declarada en las condiciones que conocemos actualmente. Sin embargo si se declara una variable con el modificador *final* de la siguiente forma:

**final String** Nombre = “Omar”;

Se le está indicando a Java que la variable llamada *Nombre* almacena la cadena “Omar” y que no se le puede cambiar su contenido dentro de una misma ejecución del programa. De acuerdo a esto, si se quisiera almacenar en la variable *Nombre* la cadena “Iván”, apenas se escriba la instrucción

Nombre = “Iván”;

Inmediatamente el entorno integrado con el cual estemos trabajando generará un error puesto que reconoce que, en la realidad, *Nombre* no es una *variable* sino una *constante* por la connotación que se ha indicado en esta lección de no podérsele cambiar su contenido durante una misma ejecución del programa.

Dos constantes típicas, por sus características matemáticas, que encontramos en el lenguaje de programación Java son el número  $\pi$  y el número  $e$ . Si las buscamos en la documentación de la Java API, encontramos que se encuentra en la librería Math y su declaración es de la siguiente forma:

*public static final double PI*

*public static final double E*

Esta declaración indica que son tipo **public** para que desde cualquier Clase se pueda utilizar bien sea una Clase de usuario o bien una Clase de algún paquete propio de Java; son tipo **static** para que el valor se mantenga dentro y fuera de cualquier clase, sea cual fuere su uso; tiene el modificador **final** para indicarle a Java que los valores almacenados en estas constantes no pueden ser modificados desde ninguna clase puesto que es sabido que, en matemáticas, tanto la constante PI como la constante E son números que corresponde a valores definidos, sin período, con cantidad de decimales infinitos e incambiables.

El declarador **double** le indica a Java que son números que pueden tener hasta 16 decimales de precisión.

### 14.3 Ejercicio Resuelto con *final*

Construir una clase que permita crear objetos que reciban por cada empleado su identificación, su nombre y su salario pero asegurándose que el número de la identificación no podrá ser modificado desde ninguna parte del programa.

Lo primero que vamos a hacer será crear una clase llamada *Empleado* con las características que implica el enunciado. Debe acotarse que se ha creado un nuevo paquete llamado *Clase3* y por tal motivo la primera línea así se lo indica al intérprete de Java.

*package Clase3;*

La Clase Empleado, en consonancia con el enunciado, contiene tres atributos que corresponden a las variables número de identificación (*NumId*), nombre del empleado (*Nombre*) y sueldo del empleado (*Sueldo*). Nótese que la “variable” *NumId* tiene el modificador **final** indicándole al intérprete de Java que, después de que se le haya hecho la primera asignación, dicho contenido no puede ser modificado como lo indica la vida práctica en relación con el número de identificación de un empleado o de cualquier otra persona.

```
public class Empleado {  
  
    private final String NumId;  
    private String Nombre;  
    private double Sueldo;
```

Seguidamente creamos el constructor de la Clase que, como ya sabemos, tiene el mismo nombre de la Clase. Este es un método **public** para que pueda ser invocado desde cualquier otra clase y recibe tres parámetros: una cadena, otra cadena y un número tipo **double** que corresponden al número de identificación, al nombre del empleado y al sueldo del empleado respectivamente. La idea es que, cuando se vaya a crear un objeto tipo Empleado, se le pasen los parámetros correspondientes de forma que dicho empleado quede con la identificación, el nombre y el sueldo que le hayamos enviado como parámetros.

```
public Empleado(String Id, String Nom, double Sue) {  
  
    NumId = Id;  
    Nombre = Nom;  
    Sueldo = Sue;  
}
```

Por razones prácticas, vamos a construir los métodos que nos permiten visualizar los datos y modificar los datos que se puedan modificar dado que los atributos fueron declarados (también) de tipo **private** lo cual le indica que solo pueden ser modificados desde adentro de su propia clase tal como lo indica el principio de la encapsulación.

El método *MuestraDatos()* todo lo que hace es mostrar los datos que contiene un objeto que haya sido declarado de tipo *Empleado* de forma que aparezcan en pantalla de una manera organizada, presentable y estética. Este método retorna una cadena que la conforma la unión de las cadenas y los datos correspondiente. Se han utilizado los caracteres especiales \n y \t para indicarle al intérprete de Java que cambie de línea y que tabule, ambos efectos en consola, según nuestro criterio estético.

```
String MuestraDatos() {
```

```
return "\nId : " + NumId + "\nNombre : " + Nombre + "\nSueldo : " +  
Sueldo;  
}
```

El método *CambiaNombre()*, que recibe una cadena, permite reasignar un nuevo nombre a un objeto tipo *Empleado* ya construido en consonancia con la realidad de que una persona, legalmente, puede cambiar su nombre ante las instancias pertinentes.

```
public void CambiaNombre(String Nom) {  
  
    Nombre = Nom;  
}
```

El método *CambiaSueldo()*, que recibe un valor tipo **double**, permite cambiar el valor de la variable *Sueldo* que es uno de los tres atributos del objeto *Empleado*.

```
public void CambiaSueldo(double Sue) {  
  
    Sueldo = Sue;  
}  
}
```

Con este último método termina la Clase *Empleado*. A continuación construimos el programa que hace uso de la clase *Empleado* a través de la cual crea dos empleados nombrados como *Emp1* y *Emp2*. En este programa (que se distingue de la Clase simple porque contiene un método llamado *main*) se crean los dos empleados y a cada uno se le asignan valores pertinentes de acuerdo a los parámetros que recibe su constructor.

```
package Clase3;  
  
public class ProgEmpleado {  
    public static void main(String[] args) {  
  
        Empleado Emp1 = new Empleado("111", "Omar Iván", 1200000);  
        Empleado Emp2 = new Empleado("222", "Juana María", 1300000);  
    }  
}
```

Después de haber creado los dos objetos de tipo Empleado, se muestran por consola los datos de cada uno de ellos acudiendo al método *getter* que presenta los datos y que hemos llamado *MuestraDatos()*.

```
System.out.println(Emp1.MuestraDatos());  
System.out.println(Emp2.MuestraDatos());
```

A continuación, para entender un poco mejor el programa que hemos construido, acudimos al método *CambiaNombre()* para que el objeto Emp1 tenga un nuevo nombre y ya no se llame “Omar Iván” sino “José Luis”. Seguidamente volvemos a mostrar los datos por consola debidamente actualizados.

```
Emp1.CambiaNombre("José Luis");  
System.out.println(Emp1.MuestraDatos());
```

De la misma forma invocamos el método *CambiaSueldo()* que permite reasignar un nuevo valor tipo **double** a la variable Sueldo que es uno de los atributos del objeto Emp1. Igualmente se muestran los datos debidamente actualizados en pantalla pues el objeto Emp1 ahora tiene un sueldo de 2500000.

```
Emp1.CambiaSueldo(2500000);  
System.out.println(Emp1.MuestraDatos());  
}  
}
```

Note usted que no se ha creado un método CambiaId() pues la variable NumId no se puede cambiar debido a que tiene el modificador final lo cual hace que, después de que se le ha asignado el primer valor, no se le puede asignar ningún otro valor. El código completo de la Clase Empleado y del programa se presenta a continuación:

```
// CLASE EMPLEADO  
  
package Clase3;  
  
public class Empleado {  
  
    private final String NumId;  
    private String Nombre;
```

```
private double Sueldo;  
  
public Empleado(String Id, String Nom, double Sue) {  
  
    NumId = Id;  
    Nombre = Nom;  
    Sueldo = Sue;  
}  
  
String MuestraDatos() {  
  
return "\nId : " + NumId + "\nNombre : " + Nombre + "\nSueldo : " +  
    Sueldo;  
}  
  
public void CambiaNombre(String Nom) {  
  
    Nombre = Nom;  
}  
  
public void CambiaSueldo(double Sue) {  
  
    Sueldo = Sue;  
}  
  
// PROGRAMA QUE USA LA CLASE EMPLEADO  
  
package Clase3;  
  
public class ProgEmpleado {  
  
    public static void main(String[ ] args) {  
  
        Empleado Emp1 = new Empleado("111", "Omar Iván", 1200000);  
        Empleado Emp2 = new Empleado("222", "Juana María", 1300000);  
  
        System.out.println(Emp1.MuestraDatos());  
        System.out.println(Emp2.MuestraDatos());  
    }  
}
```

```
    Emp1.CambiaNombre("José Luis");
    System.out.println(Emp1.MuestraDatos());
}

Emp1.CambiaSueldo(2500000);
System.out.println(Emp1.MuestraDatos());
}
}
```

Al ejecutar este programa el resultado por consola, tal como lo habíamos previsto, es el siguiente:

```
Num Id : 111
Nombre : Omar Iván
Sueldo : 1200000.0

Num Id : 222
Nombre : Juana María
Sueldo : 1300000.0

Num Id : 111
Nombre : José Luis
Sueldo : 1200000.0

Num Id : 111
Nombre : José Luis
Sueldo : 2500000.0
```

#### 14.4 Ejercicios Propuestos

- Construir un programa Java que permita crear el registro inicial de unos estudiantes de un colegio de manera que contenga los datos Código del Estudiante, Nombres del Estudiante, Apellidos del Estudiante y Fecha de Nacimiento. Se requiere que el programa garantice que, una vez digitado el código del estudiante, éste no podrá ser modificado desde otra clase
- Construir un programa Java que permita crear los artículos de una ferretería (almacén donde se venden herramientas) de manera que cada uno tenga un Código de Referencia, un Nombre, una Descripción y un

Precio de Venta. Se necesita que el código de referencia de cada herramienta no pueda ser modificada desde otra clase

- Construir un programa Java que permita inscribir ciudadanos para un proceso electoral. Se necesita almacenar el número de identificación de cada ciudadano, su nombre completo, su fecha y ciudad de nacimiento. Se requiere que el número de identificación del ciudadano no se pueda modificar una vez se haya registrado
- Construir un programa Java que permita registrar los libros que entran en una librería de forma que se pueda almacenar el ISBN de cada libro, el título del libro, el autor, el año de edición y el nombre de la editorial. Se necesita que el número ISBN de cada libro no se pueda modificar después de que éste se haya registrado
- Calcular el área de un triángulo equilátero teniendo en cuenta que la altura de un triángulo equilátero se puede calcular basado en la dimensión de uno de sus lados y que éste valor es constante para cualquier triángulo equilátero. Escribir el programa de forma que dicho valor se trate como una constante

## 14.5 Variables Estáticas

Vamos a revisar una situación que, siendo hipotética, está muy ajustada a la realidad y que puede darle indicios de situaciones similares en donde se requiere que el computador nos ayude a ejercer control. Supongamos que se necesita vender el aforo de un teatro de 300 sillas para una presentación de un artista. Los productores del espectáculo han indicado que, por razones de organización, cada silla tiene un número asignado y que éste siempre debe quedar registrado así como el nombre de la persona que ha adquirido la boleta respectiva.

De la misma manera se necesita registrar el precio de cada boleta debido a que, dependiente de la ubicación en el teatro, se manejan 6 precios diferentes y se requiere tener registro inmediato de cada boleta que se vende. Se requiere, igualmente, que se lleve un control de la cantidad de sillas que se han vendido para saber en qué momento se ha completado todo el aforo del teatro.

Debo comenzar por admitir que la solución que se va a plantear no es la óptima y ni siquiera es la más práctica pero será la suficiente para entender el concepto de lo que se quiere explicar en relación con el manejo de las variables.

Cuando se crea una instancia de una clase (es decir, un objeto), cada objeto tiene sus propias variables de forma que los atributos de un objeto sean completamente diferentes de los atributos de otro objeto así lleven el mismo nombre. De esta forma, se pueden diferenciar unos de otros y se pueden manejar de manera independiente. Existen situaciones en donde requerimos que algún dato de un objeto pueda ser utilizado por otro objeto y luego por otro y así sucesivamente.

Con lo que hemos visto hasta el momento sería un poco completo lograrlo y por ello se necesita que una Clase, como tal, tenga sus propias variables, es decir, que existan algunas variables que no pertenezcan a un objeto específico (o a una instancia de la clase) sino que pertenezcan a la clase como tal y, por lo tanto, que puedan ser utilizadas por todos los objetos sin ninguna dificultad.

En el ejemplo que nos compete vemos que se requiere, de una parte, tener un control de la cantidad de sillas que se han vendido hasta completar el aforo del teatro de manera que no se vayan a vender más sillas de las que tiene y, por otra parte, se requiere llevar un control instantáneo del valor que se esté recaudando por cada venta de las boletas dado que, como se dijo, se manejan 6 precios diferentes de boletas dependiendo de la ubicación de la silla dentro del teatro. Esta es la situación que vamos a resolver con la ayuda del lenguaje de programación Java y con el apoyo del paradigma de programación orientada a objetos teniendo en cuenta que no es la solución óptima ni la más práctica pero que nos servirá para entender la gran utilidad que tiene el manejo de variables de tipo estática.

#### **14.6 El modificador *static***

El modificador *static* permite que se declaren variables que sean de la clase donde se encuentren de forma que el valor que almacenen perdure independiente de las instancias que se estén creando para que su contenido se pueda reconocer y utilizar en cualquier momento.

#### **14.7 Ejercicio Resuelto con *static***

Construir un programa que atienda la necesidad situacional que se planteó al inicio del numeral 14.5 apoyándose en las características que adquiere una variable cuando se declara tipo *static*.

Lo primero que vamos a hacer es declarar una Clase a la cual llamaremos Silla que contendrá, como atributos, el número de la silla dentro del teatro, el nombre de la persona que la va a ocupar, el precio de la boleta que debe pagarse para utilizar determinada silla (esto teniendo en cuenta que se cuenta con 6 precios diferentes de boletas dependiendo de la ubicación dentro del teatro) y un número de orden que indique cuál fue la posición en la cual se vendió la determinada silla. Debe aclararse que, en esta solución y con el ánimo de hacer hincapié en las características del modificador *static*, no se hará control del precio, de la ubicación y de otros elementos que se involucran en la vida real.

```
package Taquilla;  
  
public class Silla {  
  
    private int Num;          // Número de la silla en el teatro  
    private String Nom;       // Nombre del Usuario  
    private double Precio;   // Precio boleta para esa silla  
    private int Orden;        // Determina el número de orden  
                            // de venta de la silla
```

Con el ánimo de poder llevar el control de la cantidad de sillas y del valor recaudado durante el proceso de venta, se declaran dos variables de tipo *private* para atender las bondades de la encapsulación, de tipo *static* para que se puedan realizar operaciones y procesos con dichas variables en la medida en que se vayan vendiendo las boletas.

La variable *CantSillas* será de tipo entero dado que siempre se venden sillas completas (no se vende un pedazo de una silla en un teatro) y la variable *VrRecaudo* será de tipo *double* para poder contabilizar el valor total por concepto de la venta de las boletas. Por las razones del uso que se le piensa dar a estas variables, ambas variables se inicializan en 0 indicando que, al inicio del proceso, aún no se ha vendido ninguna boleta y no se ha recaudado ningún dinero.

```
private static int CantSillas=0;  
private static double VrRecaudo=0;
```

A continuación se crea el método constructor (que como sabemos debe tener el mismo nombre de la Clase). Este método recibe tres parámetros:

un valor **int** que corresponde al número de la silla, un dato tipo **String** que corresponde al nombre del usuario que compra la boleta y un valor tipo **double** que corresponde al precio de la boleta correspondiente dependiendo de la silla que se esté vendiendo.

En este método constructor se tomarán los parámetros recibidos y se almacenarán en las variables respectivas que pertenecen al objeto. Se ha utilizado el apuntador **this** para indicar que se refiere al objeto que se haya creado debido a que los parámetros y los atributos tienen los mismos nombres y esta es una forma clara y apropiada de diferenciarlos.

De la misma manera, se incrementa en 1 la variable que almacena la cantidad de sillas para que, por cada silla que se venda, se pueda indicar que una silla más se ha vendido. Este valor se almacena en la variable *Orden* que pertenece al objeto (teniendo en cuenta que la variable *CantSillas* pertenece a la Clase) para que quede el registro de la posición ordinal en la cual la silla fue vendida. Por cada vez que se cree un nuevo objeto tipo Silla se incrementará en 1 el contenido de la variable *CantSillas* y de esa forma se puede llevar un control seguro necesario para no vender más sillas de las que tiene el teatro.

Igualmente, en consonancia con el enunciado a resolver, la variable *VrRecaudo* acumulará el valor del *Precio* de cada boleta para que se vaya contabilizando en el momento en que se venda. De la misma manera se acude al apuntador **this** para que se reconozca que se refiere a la variable *Precio* del objeto y no a la variable *Precio* que llega como parámetro aunque debe aceptarse que, en este caso y solo en este caso, daría el mismo resultado pero debe tenerse en cuenta cuándo se usa la variable del objeto y cuándo se usa la variable de la clase. Con esto termina la declaración del método constructor de manera que, por cada boleta que se venda, no solo se van a registrar los datos pertinentes que llegan por la vía de los parámetros sino que se van a acumular, en las variables estáticas, los resultados necesarios (cantidad de sillas vendidas y valor recaudado) para atender las necesidades del enunciado.

```
public Silla(int Num, String Nom, double Precio) {  
  
    this.Num = Num;  
    this.Nom = Nom;  
    this.Precio = Precio;
```

```
CantSillas = CantSillas + 1;  
this.Orden = CantSillas;  
  
VrRecaudo = VrRecaudo + this.Precio;  
}
```

El otro método que se declara en esta clase es una función *getter* que muestra los resultados pertinentes por cada vez que se venda una boleta incluyendo el número de orden de la venta, la cantidad de sillas que quedan disponibles (que equivale a restarle al aforo completo –que son 300 sillas– el valor de las sillas vendidas) y el valor recaudado por cada boleta que se vende. Este método retorna un dato tipo *String* con la información solicitada.

```
public String MuestraDatos() {  
  
    return "\n\nSilla No.\t" + Num +  
        "\nUsuario\t" + Nom +  
        "\nPrecio\t" + Precio +  
        "\nOrden\t" + Orden +  
        "\n\nQuedan " + (300-CantSillas) + " sillas libres" +  
        "\nVr Recaudado hasta ahora:\t" + VrRecaudo +  
        "\n-----";  
}  
}
```

Con esta función *getter* termina la clase que se requiere para la solución que se plantea en este ejercicio. A continuación veamos el programa que utiliza la clase que se ha declarado para hacer efectiva la venta de las boletas y la contabilización del recaudo. No se olvide que esta no es la mejor solución pero que es muy útil para resaltar la gran ventaja que tiene la utilización de variables tipo *static*.

Recuerde que un programa Java se diferencia de una Clase Java porque el programa tiene un método llamado *main* desde donde se ejecuta la solución que se haya implementado para resolver el enunciado. En este programa se declaran varios objetos y por cada objeto declarado se muestra tanto la contabilización de las sillas como el recaudo. Cada objeto creado tiene un número de silla (que corresponde al número que tiene la silla

asignado dentro del teatro), un nombre completo de la persona que piensa ocupar dicha silla y el precio de la boleta pagado para usar la silla dependiendo de la ubicación (no se hace validación de la relación entre la ubicación y el precio, solo se reciben los datos). La utilización de las variable *static* son lo que permite que, por cada boleta vendida y registrada, se pueda llevar la contabilidad del valor recaudado y se pueda calcular cuántas sillas disponibles quedan.

```
package Taquilla;  
  
public class Venta {  
  
    public static void main(String[ ] args) {  
  
        Silla S1 = new Silla(18, "Pedro Perez", 15000);  
        System.out.println(S1.MuestraDatos());  
  
        Silla S2 = new Silla(23, "Carlos Gomez", 15000);  
        System.out.println(S2.MuestraDatos());  
  
        Silla S3 = new Silla(105, "Ana Diaz", 20000);  
        System.out.println(S3.MuestraDatos());  
  
        Silla S4 = new Silla(106, "Pablo Trejos", 15000);  
        System.out.println(S4.MuestraDatos());  
  
        Silla S5 = new Silla(30, "María Ramirez", 20000);  
        System.out.println(S5.MuestraDatos());  
    }  
}
```

El código completo tanto de la clase que se ha construido para implementar la solución como del programa que la usa se presenta a continuación:

```
// CLASE REQUERIDA  
package Taquilla;  
  
public class Silla {  
    private int Num;      // Número de la silla en el teatro  
    private String Nom;   // Nombre del Usuario
```

```
private double Precio;           // Precio boleta para esa silla
private int Orden;              // Determina el número de orden
//de venta de la silla

private static int CantSillas=0;
private static double VrRecaudo=0;

public Silla(int Num, String Nom, double Precio) {

    this.Num = Num;
    this.Nom = Nom;
    this.Precio = Precio;

    CantSillas = CantSillas + 1;
    this.Orden = CantSillas;

    VrRecaudo = VrRecaudo + this.Precio;
}

public String MuestraDatos() {

    return "\n\nSilla No.\t" + Num +
        "\nUsuario\t" + Nom +
        "\nPrecio\t" + Precio +
        "\nOrden\t" + Orden +
        "\n\nQuedan " + (300-CantSillas) +
        " sillas disponibles" +
        "\nVr Recaudado hasta ahora:\t" + VrRecaudo +
        "\n-----";
}

}

// PROGRAMA QUE USA LA CLASE REQUERIDA
package Taquilla;
```

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

        Silla S1 = new Silla(18, "Pedro Perez", 15000);
```

```
System.out.println(S1.MuestraDatos());  
  
Silla S2 = new Silla(23, "Carlos Gomez", 15000);  
System.out.println(S2.MuestraDatos());  
  
Silla S3 = new Silla(105, "Ana Diaz", 20000);  
System.out.println(S3.MuestraDatos());  
  
Silla S4 = new Silla(106, "Pablo Trejos", 15000);  
System.out.println(S4.MuestraDatos());  
  
Silla S5 = new Silla(30, "Maria Ramirez", 20000);  
System.out.println(S5.MuestraDatos());  
}  
}
```

Al ejecutar este programa, se presentan a continuación los resultados que se obtienen, en consonancia con las necesidades que se han planteado en el enunciado.

```
Silla No. 18  
Usuario Pedro Perez  
Precio 15000.0  
Orden 1
```

```
Quedan 299 sillas disponibles  
Vr Recaudado hasta ahora: 15000.0
```

---

```
Silla No. 23  
Usuario Carlos Gomez  
Precio 15000.0  
Orden 2
```

```
Quedan 298 sillas disponibles  
Vr Recaudado hasta ahora: 30000.0
```

---

Silla No. 105  
Usuario Ana Diaz  
Precio 20000.0  
Orden 3

Quedan 297 sillas disponibles  
Vr Recaudado hasta ahora: 50000.0

-----

Silla No. 106  
Usuario Pablo Trejos  
Precio 15000.0  
Orden 4

Quedan 296 sillas disponibles  
Vr Recaudado hasta ahora: 65000.0

-----

Silla No. 30  
Usuario María Ramirez  
Precio 20000.0  
Orden 5

Quedan 295 sillas disponibles  
Vr Recaudado hasta ahora: 85000.0

-----

Nótese que por cada boleta que se venda, aparece el número de la silla que se ha adquirido, aparece el nombre de la persona que la va a ocupar y el precio que ha pagado por la boleta. De la misma manera aparece la posición ordinal en la cual se vendió la silla. Al final aparece la cantidad de sillas que quedan disponibles y el valor que se va recaudando en la medida en que se van vendiendo las sillas, ambas situaciones gracias a las bondades de la utilización de variables que se han declarado como *static*.

## 14.8 Ejercicios Propuestos

- Construir un programa Java que permita comprar y vender una cantidad de huevos de manera que se pueda tener control de los huevos que se venden, que no se puedan vender más de los que hay y que cada cantidad de huevos adquiridos incrementen el inventario para vender
- Construir un programa Java que permita registrar los movimientos de consignación y retiro sobre una cuenta bancaria por parte de su propietario, desde la óptica del cuentahabiente, de manera que se pueda controlar el saldo que se tiene en el banco
- Construir un programa Java que permita registrar la cantidad de estudiantes que ingresan a un colegio de forma que además de sus propios datos se pueda tener, en cualquier momento, la cantidad de estudiantes matriculados
- Construir un programa Java que permita registrar los libros que ingresan a una librería teniendo en cuenta que sus estantes solo tienen capacidad para 100 libros
- Construir un programa Java que permita registrar los gastos diarios que se tienen en una empresa sabiendo que se cuenta con una caja menor diaria de 500.000 pesos

## 14.9 Acceso a variables *static*

Tengamos en cuenta que una variable tipo *static* es una variable propia de la clase en la cual se haya declarado lo cual quiere decir que cuando se crear un nuevo objeto (también llamado instancia de clase) no se crea una copia de esta variable como parte de los atributos del objeto. Precisamente las variables tipo *static* sirven para algunos procesos sean compartidos por las diferentes instancias de clase de forma que se puedan mantener ciertos datos comunes entre ellas.

Aunque no es obligatorio, tiene mucho sentido que el acceso a las variables *static* sea *private* puesto que están asociadas a la clase como tal y eso pareciera invitar a que son las primeras llamadas a acudir al concepto de encapsulación como forma de ser protegidas en su contenido. Surge entonces una pregunta ¿si estas variables son de tipo *static*, entonces cómo podemos acceder a su contenido desde un programa Java? Existen dos formas:

La primera es declarar la variable *static* (propia de la clase) con acceso *public*. De esta forma, por ejemplo, si la variable se ha declarado de la siguiente manera;

```
public static int CantSillas;
```

Entonces, desde el programa, podríamos usar una línea como la siguiente:

```
System.out.println("nCantidad de Sillas = " + Silla.CantSillas);
```

Dentro del marco del ejemplo que hemos construido, es decir, considerando que la variable se llama *CantSillas* y que pertenece a una clase llamada *Silla*. Esto demuestra que la variable es realmente una variable de clase y no una variable que se reproduce en la medida en que se crea una nueva instancia de la clase (objeto). Siempre que se vaya a hacer referencia a una variable de clase, y ésta es tipo *static* con acceso *public*, se debe escribir el nombre de la clase, después el signo punto y por último el nombre de la clase. Esa es la razón por la cual, cuando se va a utilizar el valor de  $\pi$  debemos escribir *Math.PI* puesto que *PI* es una variable de la clase *Math* y su declaración es la siguiente:

```
static double PI;
```

Se asume que es de acceso público (*public*) cuando no se declara nada. La otra manera de acceder a las variables de clase es a través de los métodos estáticos (o métodos tipo *static*) que son métodos que, normalmente, retornan un valor que accede a una variable *static*. Es de anotar que tanto las variables tipo *static* como los métodos *static* son propios de la clase. Si en el ejemplo que hemos venido desarrollando se incorpora un nuevo método tipo *static* para acceder a la variable *VrRecaudo* con el ánimo de poder mostrar el valor total recaudado (al final del programa) sin tener que depender de ningún objeto, entonces el código podría ser el siguiente:

```
package Taquilla;
```

```
public class Silla {
```

```
    private int Num;      // Número de la silla dentro del teatro  
    private String Nom;  // Nombre del Usuario  
    private double Precio; // Precio boleta para esa silla
```

```
private int Orden; // Número de orden venta de silla

private static int CantSillas=0;
private static double VrRecaudo=0;

public Silla(int Num, String Nom, double Precio) {

    this.Num = Num;
    this.Nom = Nom;
    this.Precio = Precio;

    CantSillas = CantSillas + 1;
    this.Orden = CantSillas;

    VrRecaudo = VrRecaudo + this.Precio;
}

// NUEVO MÉTODO INCORPORADO *****
public static String MuestraVrRecaudo() {
    return "\n\n*****" +
        "\nValor Total Recaudado" +
        "\n*****\n\t" +
        VrRecaudo;
}

public String MuestraDatos() {

    return "\n\nSilla No.\t" + Num +
        "\nUsuario\t" + Nom +
        "\nPrecio\t" + Precio +
        "\nOrden\t" + Orden +
        "\n\nQuedan " + (300-CantSillas) +
        " sillas disponibles" +
        "\nVr Recaudado hasta ahora:\t" + VrRecaudo +
        "\n-----";
}
```

Nótese que se ha incorporado un método llamado MuestraVrRecaudado cuyo objetivo es retornar (con un título adecuado) el valor total que se ha

recaudado y que se espera mostrar al final del programa. El código de este método es el siguiente:

```
public static String MuestraVrRecaudo() {  
    return "\n\n*****\n" +  
        "\nValor Total Recaudado" +  
        "\n*****\n\t" +  
        VrRecaudo;  
}
```

Como se puede ver, este método se ha declarado *public* para que cualquier objeto pueda acceder a ella y también es *static* para que sea un método de la clase y no un método de cada instancia de ella. Ahora ¿cómo haríamos para invocar este método? Todo lo que tenemos que hacer es utilizarlo como se hace al final del programa al cual se le adiciona una línea de la siguiente forma:

```
package Taquilla;  
  
public class Venta {  
  
    public static void main(String[] args) {  
  
        Silla S1 = new Silla(18, "Pedro Perez", 15000);  
        System.out.println(S1.MuestraDatos());  
  
        Silla S2 = new Silla(23, "Carlos Gomez", 15000);  
        System.out.println(S2.MuestraDatos());  
  
        Silla S3 = new Silla(105, "Ana Diaz", 20000);  
        System.out.println(S3.MuestraDatos());  
  
        Silla S4 = new Silla(106, "Pablo Trejos", 15000);  
        System.out.println(S4.MuestraDatos());  
  
        Silla S5 = new Silla(30, "María Ramirez", 20000);  
        System.out.println(S5.MuestraDatos());  
  
        System.out.println(Silla.MuestraVrRecaudo());  
    }*****
```

```
}
```

La línea marcada con los asteriscos es la que se ha adicionado. En esta línea se usa la salida estándar *System.out.println* para que muestre por consola el resultado que retorne el método *MuestraVrRecaudo()* que pertenece a la clase *Silla*. Nótese como se invoca el método asociándolo directamente a la clase y no a un objeto en particular. Con esto se cumple con el objetivo de mostrar, solo al final, el valor total recaudado por concepto de venta de boletas (sin dejar de olvidar que esta no es la mejor solución al problema). Al ejecutar el programa se mostrará en pantalla lo siguiente:

```
Silla No. 18
Usuario    Pedro Perez
Precio     15000.0
Orden 1
```

```
Quedan 299 sillas disponibles
Vr Recaudado hasta ahora: 15000.0
-----
```

```
Silla No. 23
Usuario    Carlos Gomez
Precio     15000.0
Orden 2
```

```
Quedan 298 sillas disponibles
Vr Recaudado hasta ahora: 30000.0
-----
```

```
Silla No. 105
Usuario   Ana Diaz
Precio    20000.0
Orden 3
```

```
Quedan 297 sillas disponibles
Vr Recaudado hasta ahora: 50000.0
```

Silla No. 106  
Usuario Pablo Trejos  
Precio 15000.0  
Orden 4

Quedan 296 sillas disponibles  
Vr Recaudado hasta ahora: 65000.0

---

Silla No. 30  
Usuario María Ramirez  
Precio 20000.0  
Orden 5

Quedan 295 sillas disponibles  
Vr Recaudado hasta ahora: 85000.0

---

\*\*\*\*\*  
Valor Total Recaudado  
\*\*\*\*\*  
85000.0

Que corresponde a la salida por consola que ya conocíamos pero adicionándole lo último, es decir, un reporte del valor total recaudado.

#### 14.10 Ejercicios Propuestos

Completar los ejercicios del numeral 14.8 escribiendo el método static respectivo que permita visualizar las variables de clase utilizadas.



# 15 Lección

## Sobrecarga de métodos



## 15.1 Definición

Para continuar con el enunciado que venimos desarrollando desde la lección anterior, supongamos que algunas sillas del teatro se han entregado a manera de “Cortesía”, algo que usualmente se hace en las presentaciones, que consiste en que se regalan algunas entradas a personas ilustres o a empresas que patrocinan la presentación.

En este caso lo único que necesitamos es el número de la silla para reservarla y no se requiere ni el nombre de la persona que la va a ocupar (y que es quien escoge la silla) ni el valor real de la boleta según su ubicación. Para estas boletas se cobra un valor figurativo de \$5000 que lo cancela el beneficiado de la boleta. Aquí tendríamos dos formas de registrar la información respectiva. La primera es hacerle el llamado al constructor de clase que ya tenemos en el código con parámetros fijos. Por ejemplo, si viniera una persona que ha escogido la silla 121 y trae boleta de cortesía, podríamos llamar al constructor de la siguiente forma:

*Silla S9 = new Silla(121, "Cortesía", 5000);*

Si, posteriormente, viene otra persona con una boleta de cortesía y escoge la silla 87 entonces podríamos registrarla así:

*Silla S11 = new Silla(87, "Cortesía", 5000);*

Cada que venga una persona con una boleta de cortesía podríamos hacerlo de esta forma. Sin embargo podemos acudir a la sobrecarga de constructores para simplificar esta tarea que, en el caso específico del ejemplo, pareciera ser innecesario pues ya hemos mostrado una forma de resolverlo sin embargo serán muchos los casos en donde acudir a esta facilidad que brinda el lenguaje de programación Java resulta ser de una inmensa utilidad.

## 15.2 Sobrecarga de Constructores

Ya sabemos que el método constructor es el que permite crear una instancia de una clase específica y que, para tal fin, se requiere que el

método tenga el mismo nombre de la clase. A este método se le pueden enviar los parámetros que se requieran para su construcción. Pues bien, Java permite que escribamos varios métodos con el mismo nombre del constructor siempre y cuando difieran en la cantidad de parámetros que se le envíen.

De esta forma, si queremos registrar la venta de una boleta que ha sido adquirida en condiciones normales y de la cual tenemos todos los datos, simplemente acudimos al método constructor que ya conocemos.

```
public Silla(int Num, String Nom, double Precio) {  
  
    this.Num = Num;  
    this.Nom = Nom;  
    this.Precio = Precio;  
  
    Cantsillas = Cantsillas + 1;  
    this.Orden = Cantsillas;  
  
    VrRecaudo = VrRecaudo + this.Precio;  
}
```

Pero si se trata de registrar la venta de una boleta que ha sido entregada como “Cortesía” entonces podemos acudir a un método constructor con el siguiente código:

```
public Silla(int Num) {  
    this(Num, "Cortesía", 5000);  
}
```

Nótese que este método, por ser constructor, tiene el mismo nombre de la clase al igual que el método anterior.

La diferencia radica en que este método constructor solamente requiere un parámetro que es el número de la silla que el espectador quiere ocupar. Asumimos, en este ejemplo, que todo aquel que presente una boleta de cortesía puede sentarse en cualquier ubicación.

De la misma forma nótese que el cuerpo del método contiene una sola instrucción:

```
this(Num, "Cortesía", 5000);
```

En esta instrucción estamos llamando con **this** el objeto que se está creando y que, automáticamente, Java lo asocia con el método constructor que inicialmente habíamos utilizado, es decir, el método que requería como parámetros el número de la silla, el nombre del espectador y el valor de la boleta dependiendo de la ubicación.

Como en este caso se trata de una boleta de cortesía entonces solo se necesita el número de la silla, se asume como nombre del asistente la palabra “Cortesía” y se cobra como precio de la boleta un valor figurativo de \$5000 para cualquier ubicación que deberá ser cancelado por la persona que compra la boleta. El apuntador **this** lo que hace es que llama, internamente, al constructor que ya conocíamos:

```
public Silla(int Num, String Nom, double Precio) {  
  
    this.Num = Num;  
    this.Nom = Nom;  
    this.Precio = Precio;  
  
    CantSillas = CantSillas + 1;  
    this.Orden = CantSillas;  
  
    VrRecaudo = VrRecaudo + this.Precio;  
}
```

Con este llamado, procede a almacenar en *Num* el número de la silla, a almacenar en *Nom* la palabra “Cortesía” y a almacenar en *Precio* el valor 5000, registrando la boleta a través de su constructor como si se tratara de una venta normal. Al tiempo, se contabiliza la silla (pues al fin y al cabo una boleta de cortesía ocupa silla) y se registra el valor recaudado. De esta manera el código completo de la clase quedaría de la siguiente forma:

```
package Taquilla;  
  
public class Silla {  
    private int Num;      // Número de la silla dentro del teatro  
    private String Nom;  // Nombre del Usuario  
    private double Precio; // Precio de la boleta para esa silla  
    private int Orden;   // Número de orden de venta de la silla
```

```
private static int CantSillas=0;
private static double VrRecaudo=0;

public Silla(int Num, String Nom, double Precio) {

    this.Num = Num;
    this.Nom = Nom;
    this.Precio = Precio;

    CantSillas = CantSillas + 1;
    this.Orden = CantSillas;

    VrRecaudo = VrRecaudo + this.Precio;
}

public Silla(int Num) {
    this(Num, "Cortesía", 5000);
}

public static String MuestraVrRecaudo() {
    return "\n\n*****\n" +
           "\nValor Total Recaudado" +
           "\n*****\n" +
           VrRecaudo;
}

public String MuestraDatos() {

    return "\n\nSilla No.\t" + Num +
           "\nUsuario\t" + Nom +
           "\nPrecio\t" + Precio +
           "\nOrden\t" + Orden +
           "\n\nQuedan " + (300-CantSillas) +
           " sillas disponibles" +
           "\n\nVr Recaudado hasta ahora:\t" + VrRecaudo +
           "\n-----";
}

}
```

Por su parte, el código completo del programa quedaría como se presenta a continuación:

```
package Taquilla;  
  
public class Venta {  
  
    public static void main(String[] args) {  
  
        Silla S1 = new Silla(18, "Pedro Perez", 15000);  
        System.out.println(S1.MuestraDatos());  
  
        Silla S2 = new Silla(23, "Carlos Gomez", 15000);  
        System.out.println(S2.MuestraDatos());  
  
        Silla S5 = new Silla(30, "María Ramirez", 20000);  
        System.out.println(S5.MuestraDatos());  
  
        Silla S6 = new Silla(100);  
        System.out.println(S6.MuestraDatos());  
  
        Silla S7 = new Silla(101);  
        System.out.println(S7.MuestraDatos());  
  
        System.out.println(Silla.MuestraVrRecaudo());  
    }  
}
```

En donde se puede observar que las sillas 100 y 101 son sillas que se ocuparán con boletas de cortesía. Para simplificar el código se han omitido la venta de algunas sillas que aparecía en el programa que se usó en la lección anterior. Al ejecutar el programa, el resultado obtenido será el siguiente:

Silla No.	18
Usuario	Pedro Perez
Precio	15000.0
Orden	1

Quedan 299 sillas disponibles  
Vr Recaudado hasta ahora: 15000.0

-----

Silla No. 23  
Usuario Carlos Gomez  
Precio 15000.0  
Orden 2

Quedan 298 sillas disponibles  
Vr Recaudado hasta ahora: 30000.0

-----

Silla No. 30  
Usuario María Ramirez  
Precio 20000.0  
Orden 3

Quedan 297 sillas disponibles  
Vr Recaudado hasta ahora: 50000.0

-----

Silla No. 100  
Usuario Cortesía  
Precio 5000.0  
Orden 4

Quedan 296 sillas disponibles  
Vr Recaudado hasta ahora: 55000.0

-----

Silla No. 101  
Usuario Cortesía  
Precio 5000.0  
Orden 5

Quedan 295 sillas disponibles  
Vr Recaudado hasta ahora: 60000.0

-----

\*\*\*\*\*

**Valor Total Recaudado**

\*\*\*\*\*

**60000.0**

### **15.3 Ejercicios Propuestos**

- Construir un programa Java que permita calcular el área de un cuadrado, de un triángulo o de un pentágono apoyándose en el concepto de sobrecarga de constructores
- Construir un programa Java que permita sumar dos o tres o cuatro operandos apoyándose en el concepto de sobrecarga de constructores
- Constituir un programa Java que permita registrar un nuevo estudiante en un colegio con los datos Nombres, Apellidos, Edad y Año de Nacimiento. Cuando no se tengan todos los datos, deberá registrarse en Edad el valor 1 y en el Año de Nacimiento el valor 1000
- Construir un programa Java que permita registrar nuevos libros en una librería con los datos título, autor, año de edición y editorial. Cuando no se tengan todos los datos entonces el autor deberá quedar “Pendiente”, el año de edición deberá ser 1000 y la editorial también deberá quedar “Pendiente”
- Construir un programa Java que permita registrar nuevos empleados en una empresa de forma que se obtengan los datos número de identificación, nombre completo y edad. Cuando no se tengan todos los datos, el nombre completo deberá quedar como “Nuevo” y la edad deberá quedar como 99

Tenga en cuenta que en el lenguaje de programación Java no sólo se pueden sobrecargar los métodos constructores sino cualquier otro método se puede sobrecargar también. Lo único que debe tener en cuenta es que, así dos métodos tengan el mismo nombre, deben diferir en la cantidad de parámetros que se le deben enviar cuando se invoque.



# 16 Lección

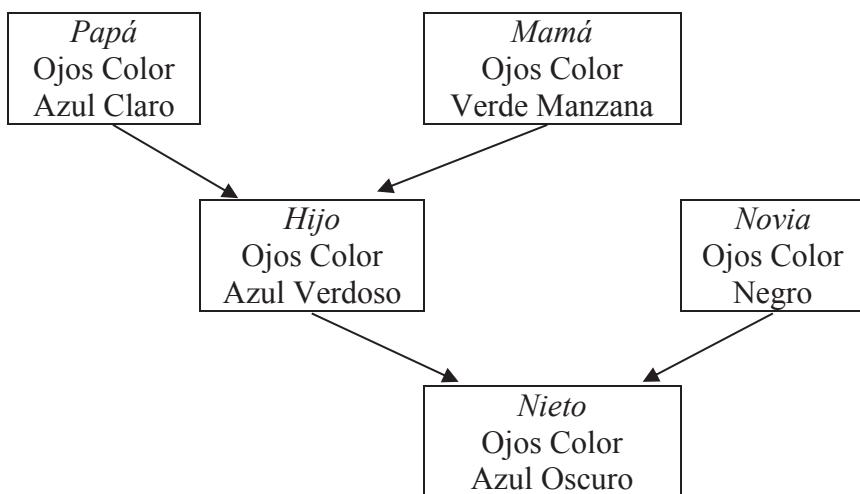
## Herencia



## 16.1 Concepto

Possiblemente uno de los conceptos más interesantes con que cuenta la Programación Orientada a Objetos es el de la herencia puesto que posibilita un conjunto de relaciones jerárquica que se pueden establecer entre diferentes clases. Vamos a ver un ejemplo que permita comprender el concepto de herencia. Un hombre con ojos de color azul claro se casa con una mujer con ojos de color verde manzana. Tienen un hijo y los ojos de este hijo son de color azul verdoso, producto de la combinación del color de los ojos de los padres. Cuando el niño crece, se casa con una muchacha que tiene ojos de color negro y su hijo mayor nace con unos ojos de color azul oscuro al parecer producto de la combinación de los dos colores. El esquema 2 siguiente muestra la relación entre los colores de los ojos.

Esquema 2. Relación de colores en los ojos



Como puede observarse en el esquema 2, el *Hijo* hereda algo del *Papá* y algo de la *Mamá* lo cual hace que el color de sus ojos sea producto de la combinación de ellos pero, además, tengan un tono propio. De la misma forma el *Nieto* hereda el color de sus ojos basado en el color de los ojos del *Hijo* (que es su padre) y de la *Novia* (que es su madre) sin embargo también le imprime algo propio al color de sus ojos.

La herencia puede definirse como el traspaso de algunas características que se transmiten a nuevas generaciones además de las características propias de dichas generaciones. En el caso de los Objetos (o instancias de clase), un Objeto hereda las características de una Clase que ha servido como base para construir su propia Clase. Es de anotar que la Clase de un Objeto también puede tener sus atributos, características o variables propias. En términos de POO supongamos que se tiene una Clase general llamada Vehículo y que la hemos declarado de forma que contenga las características de todo vehículo: el número de llantas, el color externo, la marca del vehículo (por ejemplo) son algunas características que son comunes a todos los vehículos. Sin embargo una bicicleta, una moto, un bus y una camioneta, sabiendo que comparten estos atributos, también tienen atributos propios.

La bicicleta podría tener cambios o no y además podría ser de carreras o de turismo; la moto podría tener radiador para el enfriamiento del agua o podría tener compartimientos adicionales para guardar cosas; un bus tiene como característico el número de pasajeros que, normalmente, es superior o igual a 10 y una camioneta normalmente no puede llevar mas de 3 o 4 pasajeros y tiene un cilindraje superior al de un vehículo común. En términos simples, una bicicleta, una moto, un bus y una camioneta son vehículos, por lo tanto, si quisieramos verlos desde la perspectiva de la POO deberíamos declarar una clase general llamada VEHICULO y de ella derivar, como como clases heredadas, la clase bicicleta, la clase moto, la clase bus y la clase camioneta. ¿Por qué deberíamos hacerlo así? Precisamente porque los diferentes vehículos comparten algunas características (que son las que se declaran en la clase general) pero cada uno de estos tipos de vehículos tiene características propias que podrían tener relación con la forma, con algunos aditamentos que solo determinado vehículo podría tener o con la estructura sobre la cual está montado.

El concepto de herencia permite que se puedan unificar algunos procesos y atributos y que, a partir de ellos, se puedan diferenciar otros procesos y atributos que tengan que ver con las clases heredadas de manera específica. A continuación desarrollaremos un ejemplo teniendo en cuenta el código Java para que se puedan comprender las herramientas que brinda este potente lenguaje de programación cuando se trata de establecer nexos entre las clases a partir del concepto de herencia.

## 16.2 Ejemplo Resuelto

Vamos a considerar el desarrollo de un programa que permita construir una clase para identificar un dispositivo electrónico que permita lectura, almacenamiento, proceso y despliegue de resultados. En esta categoría vamos a incurir, solo por las razones del ejemplo, dos dispositivos: los celulares tipo SmartPhone y los computadores portátiles. Para ello lo primero que vamos a hacer es declarar la clase *Dispositivo* debido a que tanto el celular *SmartPhone* como el computador *portátil* son *dispositivos* y no al contrario.

Comenzamos con la inclusión en un paquete que hemos llamado *Herencia* dentro del cual declaramos una Clase llamada *Dispositivo*. Hemos reconocido algunas características generales de todo dispositivo electrónico como es el caso de la marca del dispositivo, el peso, el color, decir si el dispositivo incluye maleta, el tamaño de su memoria RAM, el tipo de procesador y el tamaño de su memoria auxiliar. Como hemos dicho, estas son características comunes a todos los dispositivos electrónicos que permitan las operaciones de lectura, procesamiento, almacenamiento y despliegue de datos. Por lo tanto declaramos los atributos de esta clase:

```
package Herencia;  
  
public class Dispositivo {  
    private String Marca;  
    private double Peso;  
    private String Color;  
    private String Maleta;  
    private int MemRam;  
    private String Procesador;  
    private int MemAux;
```

A continuación, declaramos el método constructor para esta clase. Este método tiene un conjunto de parámetros que son las características que hemos declarado como comunes para todos los dispositivos electrónicos que cumplen con las características mencionadas. Este método constructor recibe 7 parámetros, cada uno de ellos equivalente a sus atributos. Seguidamente almacenamos el contenido de cada parámetro en el respectivo atributo. Debido a que tanto los parámetros como los atributos tienen el mismo nombre, se ha acudido al apuntador *this* para

diferenciarlos y de esta forma saber cuándo se refiere a atributo de clase y cuándo se refiere a parámetro del método constructor.

```
public Dispositivo(String Marca, double Peso, String Color,  
String Maleta, int MemRam, String Procesador, int MemAux) {  
    this.Marca = Marca;  
    this.Peso = Peso;  
    this.Color = Color;  
    this.Maleta = Maleta;  
    this.MemRam = MemRam;  
    this.Procesador = Procesador;  
    this.MemAux = MemAux;  
}
```

De la misma forma construimos un método tipo *getter* que permita mostrar la información almacenada en los atributos de la clase *Dispositivo*. Este método retorna una cadena conformada por toda esta información presentada de una manera estética apropiada.

```
public String MuestraDatosGrales() {  
    return "\nMarca = " + Marca +  
           "\nPeso = " + Peso +  
           "\nColor = " + Color +  
           "\nMaleta = " + Maleta +  
           "\nMem RAM= " + MemRam +  
           "\nProcesador= " + Procesador +  
           "\nMem Aux= " + MemAux;  
}
```

Con esto termina la declaración de la clase *Dispositivo*. El código completo de esta clase se muestra a continuación:

```
// CLASE DISPOSITIVO  
  
package Herencia;  
public class Dispositivo {  
    private String Marca;  
    private double Peso;  
    private String Color;
```

```
private String Maleta;
private int MemRam;
private String Procesador;
private int MemAux;

public Dispositivo(String Marca, double Peso, String Color,
String Maleta, int MemRam, String Procesador, int MemAux) {
    this.Marca = Marca;
    this.Peso = Peso;
    this.Color = Color;
    this.Maleta = Maleta;
    this.MemRam = MemRam;
    this.Procesador = Procesador;
    this.MemAux = MemAux;
}

public String MuestraDatosGrales() {
    return "\nMarca = \t" + Marca +
           "\nPeso = \t" + Peso +
           "\nColor = \t" + Color +
           "\nMaleta = \t" + Maleta +
           "\nMem RAM= \t" + MemRam +
           "\nProcesador= \t" + Procesador +
           "\nMem Aux= \t" + MemAux;
}
}
```

Como ya tenemos una clase que se llama Dispositivo que contiene todos los atributos de un dispositivo electrónico con las características enunciadas pero que no tiene TODAS las características de TODOS los dispositivos, entonces procedemos a declarar dos clases: una para los dispositivos tipo Smart Phones (celulares inteligentes) y otra para los dispositivos tipo Computadores Portátiles. En primer lugar vamos a declarar la clase SmarPhone que, aunque tiene las características de todo dispositivo, también tiene algunas características que le son propias.

Para este caso hemos seleccionado solo tres de ellas: la determinación de confirmar si el teléfono es del tipo inteligente o no, la confirmación de que tiene un protector de pantalla y se ha designado un atributo llamado Tipo en donde vamos a almacenar la palabra “SmartPhone” de manera que

podamos identificar que se trata de este tipo de dispositivo. Esta variable llamada *Tipo* pudimos declararla entre los atributos generales de la clase *Dispositivo* sin embargo quise declararla aquí para tener una copia de esta variable en cada objeto declarado pero con un contenido propio.

En la declaración de la clase SmartPhone para indicarle a Java que se van a tomar como propios los atributos y los métodos de la clase Dispositivo se utiliza la instrucción extends con eso se podrá asumir que un objeto SmarPhone es un Dispositivo y no al contrario. Después de esto se declaran los atributos propios de la clase SmartPhone, todos con modificador de acceso private para que se aprovechen las bondades de la encapsulación.

```
package Herencia;
```

```
public class SmartPhone extends Dispositivo {  
    private String Inteligente;  
    private String Protector;  
    private String Tipo = "SmartPhone";
```

A continuación, estructuramos el método constructor de la clase SmartPhone que, por las razones explicadas a lo largo de este libro, debe tener el mismo nombre de la clase. En este método recibimos todos los parámetros que caracterizan tanto a cualquier dispositivo como a un celular inteligente específicamente. Los parámetros que identifican a todo dispositivo se los pasamos al constructor de la clase *Dispositivo* (que es como quien dice la clase padre de la clase *SmartPhone*) a través de la instrucción super que lo que hace es que llama al constructor de la clase *Dispositivo*. Los parámetros que le son propios a la clase *SmartPhone* se almacenan aprovechando el apuntador *this* debido a que atributos y parámetros del método constructor tienen el mismo nombre.

```
public SmartPhone(String Marca, double Peso, String Color,  
String Maleta, int MemRam, String Procesador, int MemAux, String  
Inteligente, String Protector) {  
    super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
    this.Inteligente = Inteligente;  
    this.Protector = Protector;  
}
```

Seguidamente construimos una función *getter* que muestre los datos del Dispositivo tipo SmartPhone. Esta clase retorna una cadena conformada por toda la información que incluye la información de todo dispositivo y la información que aquí hemos categorizado como propia de un objeto SmartPhone. Nótese que en el cuerpo de este método se acude al método MuestraDatosGrales que es propio de la clase Dispositivo pero debido a que la clase SmartPhone heredó de ésta sus características y métodos, puede acceder a ellos como si fueran propios. Recuerde que los caracteres \n y \t son para pasar a la siguiente línea y para tabular, respectivamente.

```
public String MuestraDatosSP() {
    return "\n*****" +
        "\nTipo = \t" + Tipo + "\n" +
        this.MuestraDatosGrales() +
        "\nInteligente= \t" + Inteligente +
        "\nProtector=\t" + Protector;
    }
}
```

Con esto termina la clase *SmartPhone*. De manera similar construimos la clase *Portatil* que es un dispositivo electrónico que se ajusta a la descripción que se hizo inicialmente pero que, en su declaración, hereda características y métodos de la clase dispositivo, lo cual se hace con la instrucción **extends**. En esta clase, además de las características heredadas de *Dispositivo*, también se tienen algunas variables propias que, para el ejemplo, se han identificado como la confirmación de si el portátil es de tipo modular, es decir, si la pantalla se puede separar del teclado, la confirmación de si la pantalla es sensible al tacto y se ha utilizado de nuevo la variable *Tipo* para indicar que se trata de un portátil. Esta es la misma variable que se usó en la clase *SmartPhone* pero que se ha declarado como variable de clase hija y no como variable de clase padre por razones de fortalecer el presente ejemplo.

```
package Herencia;

public class Portatil extends Dispositivo{
    private String Modular;
    private String Tactil;
    private String Tipo = "Portatil";
```

Se codifica a continuación el método constructor de la clase *Portatil* y para ello se reciben todos los parámetros que la distinguen y que incluye tanto a los valores que caracterizan a cualquier dispositivo (Clase *Dispositivo*) como a las que se han adoptado en este ejemplo como características de un computador portátil (Clase *Portatil*). Las variables propias de la clase *Dispositivo* (que es la clase padre) se pasan a través del llamado con la instrucción super que ya sabemos que invoca al método constructor de esta clase. Los otros parámetros se almacenan directamente en las variables de la clase *Portatil*. De nuevo se ha acudido al apuntador this dado que los parámetros del método constructor como las variables tanto de la clase padre (*Dispositivo*) como de la clase hijo (*Portatil*) tienen el mismo nombre.

```
public Portatil(String Marca, double Peso, String Color, String Maleta,  
int MemRam, String Procesador, int MemAux, String Modular, String  
Tactil)  
{  
super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
this.Modular=Modular;  
this.Tactil=Tactil;  
}
```

Se ha codificado un método tipo getter que muestre todos los datos del dispositivo tipo portátil y para ello, una vez más, se ha acudido a la función *MuestraDatosGrales()* que es propia de la clase *Dispositivo* pero se ha heredado desde la clase *Portatil*. Aprovechando los caracteres de cambio de línea y de tabulación, la información se intenta presentar de una manera decorosamente estética.

```
public String MuestraDatosPortatil() {  
return "\n*****\n" +  
"\nTipo = " + Tipo + "\n" +  
this.MuestraDatosGrales() +  
"\nModular = " + Modular +  
"\nTactil = " + Tactil;  
}  
}
```

Finalmente, con las clases construidas (clase padre *Dispositivo*, clase hija *SmartPhone* y clase hija *Portatil*) se construye el programa que aprovecha

las características de la herencia dentro del contexto de la Programación Orientada a Objetos. El programa incluye el paquete *Herencia* y la función *main()* (función principal que caracteriza un programa Java). Dentro de esta función se ha declarado una instancia de clase tipo *SmartPhone* llamada *S1* y otra instancia de clase (u Objeto) tipo *Portatil* llamada *P1*. Ambas instancias de clase heredan atributos y métodos de la clase *Dispositivo*. Como se puede observar en el código Java, se crea cada uno de los objetos y se pasan valores que se cargan en las respectivas variables de cada uno. Al final, se muestran los datos aprovechando las respectivas funciones getter de cada uno de los objetos. Vale la pena recordar que estos métodos invocan, internamente, al método *MostrarDatosGrales()* que es un método propio de la clase padre *Dispositivo*.

```
package Herencia;

public class Programa {
    public static void main(String [] args)
    {
        SmartPhone S1 = new SmartPhone ("Samsung", 150, "Negro", "Si",
        100000, "T1", 500000, "Si", "No");

        Portatil P1 = new Portatil ("Hp", 1500, "Gris", "No", 1000000, "P9",
        5000000, "Si", "No");

        System.out.println(S1.MuestraDatosSP());
        System.out.println(P1.MuestraDatosPortatil());
    }
}
```

El código completo tanto de las clases como del programa se presenta a continuación:

```
// CLASE DISPOSITIVO
package Herencia;

public class Dispositivo {
    private String Marca;
    private double Peso;
    private String Color;
    private String Maleta;
```

```
private int MemRam;
private String Procesador;
private int MemAux;

public Dispositivo(String Marca, double Peso, String Color,
String Maleta, int MemRam, String Procesador, int MemAux) {
    this.Marcas = Marca;
    this.Peso = Peso;
    this.Color = Color;
    this.Maleta = Maleta;
    this.MemRam = MemRam;
    this.Procesador = Procesador;
    this.MemAux = MemAux;
}

public String MuestraDatosGrales() {
    return "\nMarca = " + Marca +
           "\nPeso = " + Peso +
           "\nColor = " + Color +
           "\nMaleta = " + Maleta +
           "\nMem RAM= " + MemRam +
           "\nProcesador= " + Procesador +
           "\nMem Aux= " + MemAux;
}

}

// CLASE SMARTPHONE QUE HEREDA DE DISPOSITIVO

package Herencia;

public class SmartPhone extends Dispositivo {
    private String Inteligente;
    private String Protector;
    private String Tipo = "SmartPhone";
    public SmartPhone(String Marca, double Peso, String Color,
String Maleta, int MemRam, String Procesador, int MemAux, String
Inteligente, String Protector)
    {
```

```
super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
this.Inteligente = Inteligente;  
this.Protector = Protector;  
}
```

```
public String MuestraDatosSP() {  
    return "\n*****\n" +  
        "\nTipo = " + Tipo + "\n" +  
        this.MuestraDatosGrales() +  
        "\nInteligente= " + Inteligente +  
        "\nProtector=" + Protector;  
}  
}
```

// CLASE PORTATIL QUE HEREDA DE DISPOSITIVO

```
package Herencia;  
  
public class Portatil extends Dispositivo{  
    private String Modular;  
    private String Tactil;  
    private String Tipo = "Portatil";  
  
    public Portatil(String Marca, double Peso, String Color, String  
    Maleta, int MemRam, String Procesador, int MemAux, String Modular,  
    String Tactil)  
    {  
        super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
        this.Modular=Modular;  
        this.Tactil=Tactil;  
    }  
  
    public String MuestraDatosPortatil() {  
        return "\n*****\n" +  
            "\nTipo = " + Tipo + "\n" +  
            this.MuestraDatosGrales() +  
            "\nModular = " + Modular +  
            "\nTactil = " + Tactil;  
    }  
}
```

}

// PROGRAMA PRINCIPAL

```
package Herencia;  
  
public class Programa {  
    public static void main(String [] args) {  
  
        SmartPhone S1 = new SmartPhone ("Samsung", 150, "Negro", "Si",  
        100000, "T1", 500000, "Si", "No");  
  
        Portatil P1 = new Portatil ("Hp", 1500, "Gris", "No", 1000000, "P9",  
        5000000, "Si", "No");  
  
        System.out.println(S1.MuestraDatosSP());  
        System.out.println(P1.MuestraDatosPortatil());  
    }  
}
```

Al ejecutar este programa, se obtienen los resultados que se presentan a continuación:

\*\*\*\*\*

```
Tipo =      SmartPhone  
Marca =     Samsung  
Peso =      150.0  
Color =     Negro  
Maleta =    Si  
Mem RAM=   100000  
Procesador=T1  
Mem Aux=   500000  
Inteligente=Si  
Protector= No
```

\*\*\*\*\*

```
Tipo =      Portatil
```

Marca = Hp  
Peso = 1500.0  
Color = Gris  
Maleta = No  
Mem RAM= 1000000  
Procesador=P9  
Mem Aux= 5000000  
Modular = Si  
Tactil = No

Como se puede observar, aprovechando las características de la herencia que provee la POO se pueden lograr los resultados que se esperaban. Tenga en cuenta que si, desde una clase Hija, se quiere invocar un método de la clase Padre todo lo que tenemos que hacer es escribir la palabra **super** y después de un punto escribir el método que se quiere invocar.

### 16.3 Ejercicios Propuestos

- Construir un programa Java que permita mostrar por consola las características propias y heredadas de una familia compuesta por abuelo, padre e hijo
- Construir un programa Java que permita mostrar por consola las características propias y heredadas de los empleados de una empresa categorizados entre Gerentes, Jefes de Sección, Administrativo y Operarios
- Construir un programa Java que permita mostrar por consola las características propias y heredadas de tres tipos de vehículos: bicicleta, motocicleta y carro
- Construir un programa Java que permita mostrar por consola las características propias y heredadas de los miembros de una comunidad académica categorizados en Rector, Coordinador, Profesor, Estudiante y Administrativo
- Construir un programa Java que permita mostrar por consola las características propias y heredadas de varios tipos de alimentos: frutas, verduras, harinas y azúcares



# 17 Lección

## Polimorfismo



## 17.1 Polimorfismo

El concepto de *Polimorfismo* es uno de los conceptos mas importantes de la programación orientada a objetos pues junto con la herencia, y otros términos que veremos mas adelante, distinguen de manera precisa la esencia de dicho paradigma de programación. La palabra *Polimorfismo*, según la Real Academia Española, tiene varias interpretaciones pero todas ellas adoptan la misma esencia de su etimología. *Poli* significa varios y *morfos* significa formas luego *Polimorfismo* indica una tendencia a adoptar varias varias formas. En biología, por ejemplo, la RAE define la palabra como la propiedad de las especies de seres vivos cuyos individuos pueden presentar diferentes formas o aspectos, bien para diferenciarse en castas, como las termitas, bien por tratarse de distintas etapas del ciclo vital, como la oruga y la mariposa.

Por su parte en bioquímica, se considera como polimorfismo la propiedad de los ácidos nucleicos y las proteínas que pueden presentarse bajo varias formas moleculares y en química se define como la propiedad de los elementos y sus compuestos, que pueden cambiar de forma sin variar su naturaleza.

Pero veamos este concepto de manera más sencilla desde la perspectiva de la POO. Pensemos en un lápiz como un objeto. Un lápiz tiene unos atributos y unos métodos asociados. Entre sus atributos podríamos contar su longitud, color, material, grosor, peso y olor, entre otras. Entre los métodos asociados a un lápiz están todos aquellos procedimientos en donde es útil, por ejemplo, un lápiz sirve para rayar, para escribir, para señalar, para marcar y para lanzar como un dardo, entre otros métodos. Si quisiéramos describir el lápiz en términos de la POO podríamos definir una clase de la siguiente forma:

```
public class Lapiz {  
  
    public double longitud;  
    private String color;  
    private String material;  
    private double grosor;
```

```
private double peso;  
private String olor;  
  
// Método Constructor  
  
public Lapiz(double longitud, String color, String material,  
             double grosor, double peso, String olor) {  
    this.longitud = longitud;  
    this.color = color;  
    this.material = material;  
    this.grosor = grosor;  
    this.peso = peso;  
    this.olor = olor;  
}  
  
// Métodos de Clase  
  
public String rayar(){  
}  
  
public void señalar(){  
}  
  
public String marcar(){  
}  
  
public void lanzar(){  
}  
}
```

No olvide que este es tan solo un ejemplo y por lo tanto los métodos, exceptuando el constructor, no tienen un cuerpo con instrucciones. Faltaría llenarlo para que efectivamente se complete la clase *Lapiz*. Lo importante es que quede claro que cuando se quiere definir un objeto de la vida real, podríamos interpretarlo tal como lo usamos o lo percibimos aprovechando las características para declarar atributos y métodos que la POO provee. Nótese también que un lápiz no solo sirve para escribir o rayar, tal como uno en primera instancia pensaría, sino que también puede servir para otras cosas (o, en términos de POO, otros métodos) y eso es lo que hace pensar

en el concepto de *Polimorfismo*, es decir, un mismo objeto puede adoptar varias formas dependiendo del contexto en el cual se necesite.

## 17.2 Principio de Sustitución

La POO, y específicamente el lenguaje Java, posibilitan un principio muy interesante conocido como el principio de sustitución y que consiste en que un objeto de una subclase se puede utilizar en cualquier parte en donde debería ir un objeto de la superclase. Esto implica que siempre que necesitemos utilizar un objeto de una superclase, es posible, sin ninguna restricción en su funcionamiento que se utilice un objeto de una subclase que herede las características de la superclase. El efecto contrario no es válido y la razón es muy sencilla. Normalmente un objeto de una subclase, que ha heredado de una superclase y debido al concepto de la herencia, involucra tanto las características (atributos) y métodos de dicha superclase. Para la utilización de los objetos que se involucren en este tipo de operaciones debe tenerse en cuenta que es muy posible que un objeto de la subclase tenga atributos que no contenga la superclase y, por lo tanto, se debe tener cuidado cuando éstos se usen desde los métodos pues generaría errores que no siempre son fáciles de encontrar.

## 17.3 Polimorfismo POO

De acuerdo a lo explicado en relación con el concepto de Polimorfismo, cuando una superclase contiene un método con un nombre determinado y una subclase también tiene un método con el mismo nombre, el lenguaje de programación Java utiliza uno u otro dependiendo del contexto, es decir, de lo que se requiera según sea lo apropiado. La forma como Java reconoce cuál es el método que se necesita, aún teniendo el mismo nombre y perteneciendo a clases diferentes (superclase y subclase), se basa en los llamados que se hagan y en la relación que dichos llamados tengan con los objetos declarados. Si un proceso hace referencia a los objetos de la superclase, entonces Java utilizará el método que pertenece a ella y si otro proceso hace referencia a los objetos de una subclase, entonces Java acudirá a los métodos de dicha subclase. En determinados casos, como veremos en el ejemplo que explicamos a continuación, Java determina acertadamente si aún utilizando un objeto de subclase acude al método de superclase o viceversa.

De acuerdo a lo explicado, el polimorfismo se evidencia en Java en dos situaciones específicas: en la declaración de objetos y la invocación de los métodos. Cuando se declara un objeto cuyo tipo corresponde a una superclase, es posible crear (con *new*) y asignar un objeto cuyo tipo corresponde a la subclase.

En cuanto al uso, es posible invocar en cualquier momento un método que, teniendo el mismo nombre de otro equivalente en la superclase, pertenezca a la subclase, como ya se ha explicado. Ahí se evidencia la particularidad de Java de encontrar el método apropiado dependiendo del contexto del programa. Este proceso de enlazarse con el método apropiado dependiendo del contexto cuando se trata de dos métodos que tienen el mismo nombre en donde uno pertenece a la superclase y otro pertenece a la subclase, es lo que se conoce como *enlazado dinámico* pues es la manera como Java reonocé uno y otro ejecutando el mas apropiado.

## 17.4 Anotaciones

En este aparte solo quiero hacer dos aclaraciones que, en algún momento pueden ser de gran utilidad. Ya sabemos que cuando una variable se declara como *final* significa que una vez se le haya hecho la primera asignación, es decir, se le haya almacenado un valor, éste no se podrá modificar porque de allí en adelante la *variable* se comporta como una *constante* o sea un espacio de almacenamiento con contenido constante durante una misma ejecución del programa.

De la misma manera ¿qué significa que un método se declare como *final* o que una clase se declare como *final*? Cuando un método se declara como *final*, en caso de que esta clase sea heredada por otra, la subclase no podrá contener un método con el mismo nombre. En condiciones normales, una superclase y una subclase pueden tener métodos con el mismo nombre, según lo que se ha explicado en el principio de sustitución, sin embargo cuando en la subclase un método se declara como *final*, la subclase no podrá contener un método con el mismo nombre. Esto se utiliza debido a que hay nombres muy comunes que se le ponen a los métodos en el mundo de la POO. Muchas clases acuden a este mecanismo para que sus métodos, posiblemente de un gran uso, no tengan un equivalente en las subclases que heredan y, de esta forma, se ejecuten procesos lejanos a los objetivos logrados con dicho método en la superclase. De la misma forma, cuando se declara una clase como *final*, esto significa que dicha clase no puede

ser heredada por ninguna otra clase, es decir, no puede ser superclase de ninguna subclase.

### 17.5 Ejemplo Resuelto

Para mostrar lo que se ha explicado en relación con el concepto de polimorfismo que caracteriza en gran medida la programación orientada a objetos, realizaremos un ejercicio completo basado en el ejercicio de la lección anterior. En este ejercicio vamos a retomar la clase Dispositivo que ya conocemos y que contiene, como atributos, las características generales de un dispositivo electrónico genérico (marca, peso, color, maleta, memoria ram, procesador y memoria auxiliar). El método constructor se ha mantenido tal como lo vimos en la lección anterior y en él se reciben 7 parámetros que se almacenan en las respectivas variables de la clase.

```
package Herencia;  
  
public class Dispositivo {  
    public String Marca;  
    private double Peso;  
    private String Color;  
    private String Maleta;  
    private int MemRam;  
    private String Procesador;  
    private int MemAux;  
  
    public Dispositivo(String Marca, double Peso, String Color, String  
    Maleta, int MemRam, String Procesador, int MemAux) {  
        this.Marcas = Marca;  
        this.Peso = Peso;  
        this.Color = Color;  
        this.Maleta = Maleta;  
        this.MemRam = MemRam;  
        this.Procesador = Procesador;  
        this.MemAux = MemAux;  
    }  
}
```

En esta oportunidad, con el ánimo de explicar el concepto central de esta lección, hemos incorporado un método que llamaremos *MuestraMarca()* y que despliega en pantalla la palabra *Dispositivo* seguido por la marca de

dicho dispositivo. La palabra dispositivo la hemos incluido para distinguir cuando estamos accediendo al método *MuestraMarca()* de la superclase.

```
public String MuestraMarca() {  
    return "\nDispositivo Marca " + Marca;  
}  
}
```

De la misma manera, se declara la clase *SmartPhone* que hereda atributos y métodos de la clase *Dispositivo*. Adicionalmente, la clase *SmartPhone* tiene tres atributos más.

```
package Herencia;  
  
public class SmartPhone extends Dispositivo {  
    private String Inteligente;  
    private String Protector;  
    private String Tipo = "SmartPhone";
```

Se ha codificado un método constructor para la clase *SmartPhone* que recibe 9 parámetros que corresponden a los 7 parámetros de la superclase *Dispositivo*, a dos parámetros que se asignan por esta vía de la clase *SmartPhone* dado que el 3<sup>er</sup> parámetro propio de esta subclase se asigna directamente con la palabra “*SmartPhone*”.

```
public SmartPhone(String Marca, double Peso, String Color,  
String Maleta, int MemRam, String Procesador, int MemAux, String  
Inteligente, String Protector)  
{  
    super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
    this.Inteligente = Inteligente;  
    this.Protector = Protector;  
}
```

Por su parte, al igual que en la superclase *Dispositivo*, se ha codificado un método llamado *MuestraMarca()*, que tiene el mismo nombre del respectivo método de la superclase. La única diferencia es que este método, para reconocer cuando se invoque, tiene (en la cadena que despliega) la palabra “*SmartPhone*”.

```
public String MuestraMarca()
{
    return "\nSmartPhone Marca " + Marca;
}
}
```

Por su parte, en lo que corresponde a la definición de la subclase *Portatil* (que también hereda atributos y métodos de la superclase *Dispositivo*), se han codificado 3 atributos propios de esta subclase. Dos de ellos se asignarán a través de los parámetros de su constructor y el 3º se asignará directamente con la palabra “Portátil”.

```
package Herencia;

public class Portatil extends Dispositivo{
    private String Modular;
    private String Tactil;
    private String Tipo = "Portatil";
}
```

El método constructor, cuyo nombre debe ser el mismo de la clase, contiene 9 parámetros que corresponden a los 7 de la superclase y a 2 de los parámetros de la subclase pues el 3º de ellos se cargó directamente.

```
public Portatil(String Marca, double Peso, String Color, String Maleta,
int MemRam, String Procesador, int MemAux, String Modular, String
Tactil)
{
super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);
this.Modular=Modular;
this.Tactil=Tactil;
}
```

En esta subclase, al igual que en las dos anteriores, se ha incorporado un método llamado *MuestraMarca()* que tiene el mismo nombre de un método equivalente tanto en la superclase *Dispositivo* como en la subclase *SmartPhone* precisamente para evidenciar la particularidad del polimorfismo y del enlace dinámico que hace Java cuando se presenta esta situación, invocando el método que mas conviene de acuerdo al contexto. Para distinguir este método de los otros, se ha incorporado en el título que despliega la palabra “Portátil”. De esta forma sabremos plenamente cual

de los tres métodos *MuestraMarca()* estaremos invocando; si es el de la superclase *Dispositivo*, el de la subclase *SmartPhone* o el de la subclase *Portatil*.

```
public String MuestraMarca()
{
    return "\nPortatil Marca " + Marca;
}
```

Con estos elementos podemos escribir, ahora sí, un programa en Java en el cual definimos un vector del tipo de la superclase *Dispositivo* de 5 elementos.

```
package Herencia;

public class Programa {
    public static void main(String [] args)
    {
        Dispositivo [] DV = new Dispositivo[5];
```

Igualmente creamos, en esta función main, un objeto tipo *SmartPhone* y un objeto tipo *Portatil* de manera independiente.

```
SmartPhone DS = new SmartPhone ("Nokia", 100, "Blanco", "Si",
2000000, "T6", 600000, "Si", "No");
```

```
Portatil DP = new Portatil ("Lenovo", 1000, "Rojo", "No",
10000000, "P5", 5000000, "Si", "No");
```

Ahora asignamos a la primera posición del vector *DV* (que es de tipo *Dispositivo*) el objeto *DS* (que es de tipo *SmartPhone*). Aquí vemos el principio de sustitución en una de sus expresiones pues, a pesar de que el vector es de tipo que provee la superclase *Dispositivo*, se puede almacenar en él un objeto de tipo *SmartPhone* que es una subclase heredada de *Dispositivo*. Lo mismo sucede con el objeto que se almacena en la 2<sup>a</sup> posición del vector (o sea en la posición *DV[1]*). Allí estamos almacenando un objeto de tipo *Portatil* a pesar de que el vector es de tipo *Dispositivo*.

```
DV[0] = DS;
```

*DV[1] = DP;*

Otra forma de hacerlo es asignando directamente a un dato de tipo de una superclase un objeto nuevo de una subclase que la hereda como se ve a continuación en donde en las posiciones 2 y 3 del vector se almacenan objetos de la subclase *SmartPhone* y la subclase *Portatil*, a pesar de que el vector fue declarado con el tipo de la superclase *Dispositivo*.

*DV[2] = new SmartPhone ("Compaq", 200, "Amarillo", "No",  
300000, "T2", 600000, "Si", "No");*

*DV[3] = new Portatil ("IBM", 3000, "Azul", "Si", 4000000, "P5",  
5000000, "Si", "No");*

De la misma manera, y de siendo prácticamente obvio, se ha almacenado en la posición 4 del vector un nuevo objeto del tipo de la superclase *Dispositivo*, pues al fin y al cabo ese es el tipo de datos con el cual se declaró la clase.

*DV[4] = new Dispositivo ("Compaq", 200, "Amarillo", "No",  
300000, "T2", 600000);*

Ahora declaramos una variable tipo *Dispositivo* que hemos llamado *D* y que la asociamos, a través de un ciclo *for each*, con el vector *DV* que también es de tipo *Dispositivo* aunque contiene en su almacenamiento datos tipo *SmartPhone* y tipo *Portatil* que son subclases que han heredado de la superclase *Dispositivo*. En este ciclo lo que vamos a hacer es invocar el método *MuestraMarca()* recordando que existe un método con ese nombre tanto en la superclase *Dispositivo* como en las subclases *SmartPhone* y *Portatil*.

```
for(Dispositivo D: DV)
    System.out.println(D.MuestraMarca());
}
```

Para efectos prácticos recordemos que el método *MuestraMarca()* tiene, en el título que despliega, la palabra *Dispositivo* cuando se trata del método que forma parte de la superclase *Dispositivo*, la palabra *SmartPhone* cuando se trata del método que forma parte de la subclase *SmartPhone* que hereda a la superclase *Dispositivo* y la palabra *Portatil* cuando se trata del

método que forma parte de la subclase *Portatil* que también hereda de la superclase *Dispositivo*. El código completo es el que se presenta a continuación.

```
// SUPERCLASE DISPOSITIVO
package Herencia;

public class Dispositivo {

    public String Marca;
    private double Peso;
    private String Color;
    private String Maleta;
    private int MemRam;
    private String Procesador;
    private int MemAux;

    public Dispositivo(String Marca, double Peso, String Color, String
Maleta, int MemRam, String Procesador, int MemAux) {
        this.Marca = Marca;
        this.Peso = Peso;
        this.Color = Color;
        this.Maleta = Maleta;
        this.MemRam = MemRam;
        this.Procesador = Procesador;
        this.MemAux = MemAux;
    }

    public String MuestraMarca() {
        return "\nDispositivo Marca " + Marca;
    }
}

// SUBCLASE SMARTPHONE
package Herencia;

public class SmartPhone extends Dispositivo {
    private String Inteligente;
    private String Protector;
```

```
private String Tipo = "SmartPhone";  
  
public SmartPhone(String Marca, double Peso, String Color,  
String Maleta, int MemRam, String Procesador, int MemAux, String  
Inteligente, String Protector)  
{  
super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
this.Inteligente = Inteligente;  
this.Protector = Protector;  
}  
  
public String MuestraMarca()  
{  
return "\nSmartPhone Marca " + Marca;  
}  
}  
  
// SUBCLASE PORTATIL  
package Herencia;  
  
public class Portatil extends Dispositivo{  
    private String Modular;  
    private String Tactil;  
    private String Tipo = "Portatil";  
  
public Portatil(String Marca, double Peso, String Color, String Maleta,  
int MemRam, String Procesador, int MemAux, String Modular, String  
Tactil)  
{  
super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
this.Modular=Modular;  
this.Tactil=Tactil;  
}  
  
public String MuestraMarca()  
{  
return "\nPortatil Marca " + Marca;  
}  
}
```

```
// CLASE PRINCIPAL (PROGRAMA)
package Herencia;

public class Programa {
    public static void main(String [] args)
    {
        Dispositivo [] DV = new Dispositivo[5];

        SmartPhone DS = new SmartPhone ("Nokia", 100, "Blanco", "Si",
200000, "T6", 600000, "Si", "No");
        Portatil DP = new Portatil ("Lenovo", 1000, "Rojo", "No",
1000000, "P5", 5000000, "Si", "No");

        DV[0] = DS;
        DV[1] = DP;
        DV[2] = new SmartPhone ("Compaq", 200, "Amarillo", "No",
300000, "T2", 600000, "Si", "No");
        DV[3] = new Portatil ("IBM", 3000, "Azul", "Si", 4000000, "P5",
5000000, "Si", "No");
        DV[4] = new Dispositivo ("Compaq", 200, "Amarillo", "No",
300000, "T2", 600000);

        for(Dispositivo D: DV)
            System.out.println(D.MuestraMarca());
    }
}
```

El resultado de la ejecución de este programa se presenta a continuación y en él vemos el enlazado dinámico y el polimorfismo en toda su plenitud dado que Java reconoce cuál es el método *MuestraMarca()* que debe llamar según sea el caso.

SmartPhone Marca Nokia

Portatil Marca Lenovo

SmartPhone Marca Compaq

**Portatil Marca IBM**

**Dispositivo Marca Compaq**

Revise detenidamente el código de este programa y analice las diferentes implicaciones y ventajas que tiene el concepto de polimorfismo en la construcción de programas con Java.

### **17.6 Ejercicios Propuestos**

- Construir un programa Java que muestre las características de un martillo, un serrucho y un taladro, todas ellas como objetos heredados de la clase Herramienta
- Construir un programa Java que muestre el tamaño de un libro, de un cuaderno y de una libreta de notas, todas ellas como objetos heredados de la clase Documento
- Construir un programa Java que muestre las características de un apartamento, una casa y un chalet, todas ellas como objetos heredados de la clase Aposento
- Construir un programa Java que muestre el tamaño y afinación de una trompeta, un trombón y una tuba, todas ellas como objetos heredados de la clase Instrumento
- Construir un programa Java que muestre la marca de un carro, una bicicleta y una moto, todas ellas como objetos heredados de la clase Vehículo



# 19 Lección

## Clases abstractas



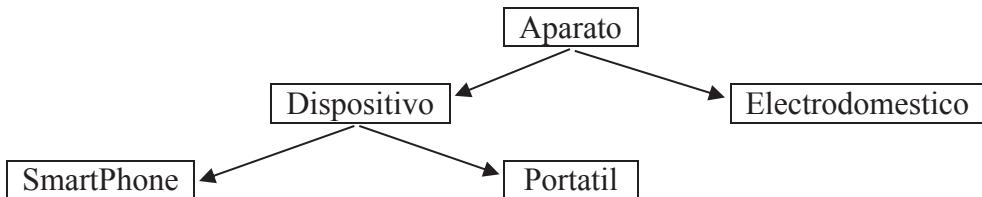
### 19.1 ¿Qué es una Clase Abstracta?

Cuando diseñamos un esquema de clases basado en donde unas clases heredan de otras sus atributos y características, es decir, en donde existen una superclases y unas subclases, es posible que nos preguntemos hasta dónde podríamos seguir hacia arriba y hasta dónde podríamos seguir hacia abajo, o sea, hasta dónde podríamos seguir diseñando superclases y hasta dónde podríamos seguir derivando subclases?. Esta pregunta la responde la vida práctica y la realidad en la cual vivimos. En el ejemplo que hemos venido manejando, con las clases *Dispositivo*, *SmartPhone* y *Portatil*, podemos ver claramente (en la vida práctica) que un smartphone es un dispositivo y que un portátil es un dispositivo. El caso contrario no es tan cierto, es decir, un dispositivo no necesariamente tiene que ser un Smartphone ni un portatil. Esta reflexión tan simple es la que nos permite entender por qué *Dispositivo* es la superclase y por qué *SmartPhone* y *Portátil* son las subclases que heredan de *Dispositivo*.

Si analizamos con mas detenimiento la situación, podríamos preguntarnos si un dispositivo es algo más, es decir, si es posible que la superclase *Dispositivo* podría heredar de otra clase y, para el ejemplo que hemos venido construyendo, vamos a asumir que sí, que si es posible. Un dispositivo es, en últimas, un aparato entonces vamos a construir una superclase llamada *Aparato* que podría hacer referencia a cualquier aparato que encontremos en el mercado. Esto permitirá pensar en que *Aparato* es la superclase, *Dispositivo* es la subclase que hereda de *Aparato* y *SmartPhone* y *Portatil* son las subclases que heredan de *Dispositivo* (y, por ende, de *Aparato*).

Según la vida práctica también podríamos pensar que existen otro tipo de aparatos como por ejemplo electrodomésticos que no son dispositivos pero si son aparatos que sirven en el hogar y que permiten que las tareas sean mucho más sencillas. Con una licuadora, por ejemplo, es mucho más fácil la tarea de hacer un jugo que si se hiciera a mano (como hicieron nuestras abuelas). Visto desde la perspectiva de la POO, esto significaría que se podría declarar una clase llamada *Electrodomestico* que herede de *Aparato* y que no tenga nada que ver con la línea de herencia de la clase *Dispositivo*. El esquema 3 hace una representación de lo explicado.

Esquema 3. Relación entre clases



Por los principios de la herencia, es claro que los atributos (variables) y métodos que contenga la clase *Aparato* serán heredados tanto por los objetos que sean de la clase *Dispositivo* y de la clase *Electrodoméstico*. De la misma manera los objetos de las clases *SmartPhone* y *Portatil* heredarán atributos y métodos de la clase *Dispositivo* y también los atributos y métodos de la clase *Aparato*.

Ahora pensemos en que quisiéramos construir un método llamado Descripción que nos permita, textualmente, describir el objeto del cual estemos hablando. ¿Dónde ubicaríamos este método para que sea general para todos? Bueno, ahí está el primer error pues este método no podrá ser general para todos ya que describir un aparato (en general) es diferente a describir un dispositivo (en términos generales) o describir un SmartPhone o describir un Portatil o describir un Electrodoméstico. Lo que sí podemos aceptar es que sería muy útil que todos estos métodos se llamaran Descripción para facilitar su uso. ¿Cómo hacemos entonces para tener un método que sea obligatorio en todas las subclases y que tenga el mismo nombre a pesar de que internamente haga algo diferente? Esa es la razón que justifica los métodos abstractos. Un método abstracto es un método cuyo prototipo se declara en una clase (que oficia en algún momento de superclase) y que indica que todas las subclases derivadas de ella deben OBLIGATORIAMENTE contener un método con ese nombre. Es de anotar que el lenguaje de programación Java exige que si una clase contiene un método abstracto entonces esa clase también tiene que ser declarada como abstracta.

Las clases abstractas y los métodos abstractos facilitan mucho el diseño de clases en un programa Java pues indican características que las subclases,

derivadas de ellas, deben tener de manera obligatoria. Vamos a desarrollar un ejemplo completo, basado en el código que hemos venido trabajando en las últimas dos lecciones. En este ejemplo que vamos a codificar, se construirán las clases que se presentaron en el esquema 3 (relación de clases).

## 19.2 Ejemplo Resuelto

Lo primero que vamos a hacer es declarar la clase *Aparato*. Debido a que esta clase va a establecer características de diseño (a nivel de métodos) para las clases que la hereden, entonces esta clase debe declararse como abstracta (**abstract**). En esta clase vamos a incluir los atributos *Marca* y *Peso*, dado que todo aparato, sea cual fuere, tiene estos dos atributos.

```
// CLASE ABSTRACTA APARATO  
package Herencia;
```

```
public abstract class Aparato  
{  
    public String Marca;  
    public double Peso;
```

Seguidamente codificamos el método constructor de la clase *Aparato* (que ya sabemos que debe tener el mismo nombre de la clase) y que, para el caso, recibe dos parámetros: una marca y un peso. El cuerpo de este método almacena en las respectivas variables de clase, los valores que se reciben en los parámetros.

```
public Aparato(String Marca, double Peso)  
{  
    this.Marca = Marca;  
    this.Peso = Peso;  
}
```

A continuación vamos a codificar una función tipo getter que retorne la marca que se recibió a través de los parámetros del método constructor.

```
public String MuestraMarca()  
{  
    return "\nMarca = " + Marca;
```

}

Por último, en esta clase, declaramos el método abstracto que necesitamos y que llamaremos *Descripción()* lo cual le indica a Java que cualquier clase que se derive de esta clase *Aparato*, deberá contener OBLIGATORIAMENTE un método llamado *Descripción()* que se utilizará para describir el aparato sea cual fuere. Por ser este un método tipo **abstract** y sabiendo que su cuerpo es completamente diferente en cada uno de las clases que lo hereden, entonces solo tendremos que escribir su prototipo, es decir, la restricción de acceso (**public**), su declaración de ser un método abstracto (**abstract**), el tipo de dato que va a retornar (**String**, para este caso) y el nombre del método (que hemos llamado **Descripción**).

Al terminar de escribir los paréntesis que acompañan el nombre del método *Descripción*, debemos escribir un signo de punto y coma ( ; ) para indicar que allí termina la declaración de esta método abstracto y que su contenido debe desarrollarse en cada una de las clases que la hereden.

```
public abstract String Descripcion();  
}
```

Con esto finaliza la declaración de la superclase *Aparato* y podemos comenzar a utilizarla para heredar sus variables (atributos) y sus métodos. Ahora vamos a declarar una clase que llamaremos *Dispositivo* y que heredará características y métodos de la clase *Aparato* debido a que un dispositivo es un aparato. Procedemos a indicar que la clase *Dispositivo* es una subclase de *Aparato*. La clase *dispositivo* tiene los atributos color, existencia de maleta, tamaño de la memoria ram, tipo de procesador y tamaño de la memoria auxiliar.

```
// SUBCLASE DISPOSITIVO  
package Herencia;  
  
public class Dispositivo extends Aparato  
{  
    private String Color;  
    private String Maleta;  
    private int MemRam;  
    private String Procesador;  
    private int MemAux;
```

A continuación codificamos el método constructor de la clase *Dispositivo* que recibirá 7 parámetros que corresponden tanto a los atributos de su propia clase como a los atributos heredados de la clase *Aparato*. Todo lo que haremos será almacenar los valores recibidos en los parámetros, en sus respectivas variables de clase y de superclase.

```
public Dispositivo(String Marca, double Peso, String Color, String
Maleta, int MemRam, String Procesador, int MemAux)
{
    super(Marca, Peso);
    this.Color = Color;
    this.Maleta = Maleta;
    this.MemRam = MemRam;
    this.Procesador = Procesador;
    this.MemAux = MemAux;
}
```

A continuación construimos un método tipo getter que hemos llamado *MuestraMarca()* y cuyo objetivo es retornar la marca del dispositivo.

```
public String MuestraMarca( )
{
    return "\nDispositivo Marca " + Marca;
}
```

Por último, en esta clase, codificamos el método *Descripción* que corresponde al método abstracto (y por lo tanto obligatorio) de la clase abstracta *Aparato*.

```
public String Descripcion()
{
    return "Un Dispositivo es un aparato eléctrico o electrónico que
facilita la vida de las personas";
}
```

De esta forma finalizamos la declaración de la subclase *Dispositivo*, que hereda características y métodos de la superclase abstracta *Aparato*. Recuerde que la superclase *Aparato* es abstracta porque contiene un

método abstracto y esa es una condición que pone Java para este tipo de facilidades. Nótese que el método descripción retorna una cadena que describe, en palabras muy simples, lo que es un dispositivo. Continuamos con la declaración de la clase *SmartPhone* que es un dispositivo, lo cual hace concluir que también es un aparato. Esto que acabamos de decir, y que se evidencia en la vida práctica, lo reflejamos en Java tal y como lo vemos en la realidad, es decir, la clase *SmartPhone* hereda atributos y métodos de la clase *Dispositivo* y ésta, a su vez, hereda atributos y métodos de la clase *Aparato*.

La clase *SmartPhone*, además de las variables (atributos) heredados de las clases *Dispositivo* y *Aparato*, tiene sus propios atributos como son la definición de si el teléfono es inteligente o no, la adición de un protector y una variable que hemos llamado tipo que indica que es un *SmartPhone* para efectos de facilitar los avisos en pantalla cuando se trate de este objeto.

```
package Herencia;  
  
// SUB SUBCLASE SMARTPHONE  
public class SmartPhone extends Dispositivo  
{  
    private String Inteligente;  
    private String Protector;  
    private String Tipo = "SmartPhone";
```

Seguidamente codificamos el método constructor de la clase *SmartPhone* el cual recibe parámetros que van a permitir que se carguen tanto las variables propias de esta clase como las variables heredadas de la superclase *Dispositivo* y las que esta a su vez hereda de la superclase *Aparato*. Precisamente el contenido del método constructor es almacenar los datos en las respectivas variables bien sea a través del constructor de su superclase o a través de las variables propias de esta clase.

```
public SmartPhone(String Marca, double Peso, String Color,  
String Maleta, int MemRam, String Procesador, int MemAux, String  
Inteligente, String Protector)  
{  
    super(Marca, Peso, Color, Maleta, MemRam, Procesador,  
MemAux);
```

```
this.Inteligente = Inteligente;  
this.Protector = Protector;  
}
```

Tal como en las otras clases, codificamos una función que hemos llamado *MuestraMarca()* y cuyo objetivo es retornar el nombre de la marca del *SmartPhone*.

```
public String MuestraMarca( )  
{  
    return "\nSmartPhone Marca " + Marca;  
}
```

Por último, en esta clase, codificamos el método *Descripción()* que es obligatorio dado que este es el método abstracto que proviene de la clase abstracta *Aparato* y cuyo contenido es completamente diferente en cada clase donde se incluye. Este método, en esta clase, describe fácilmente qué es un aparato celular tipo SmartPhone.

```
public String Descripcion()  
{  
    return "Un SmartPhone es un aparato celular con aplicaciones y  
utilidades de uso muy frecuente";  
}
```

Ahora vamos a declarar la clase *Portatil*, que es similar a la clase *SmartPhone* pero no es igual. Esta clase hereda de la clase *Dispositivo* y, además de heredar sus atributos (variables) y métodos, también tiene sus propias variables que en este caso corresponden a la posibilidad de que el portátil sea modular, que tenga pantalla táctil y se ha adicionado una variable llamada *Tipo* que almacena la palabra *Portatil* para indicar el tipo de dispositivo al cual nos estamos refiriendo cuando debamos definir un objeto de esta clase.

```
// SUB SUBCLASE PORTATIL  
package Herencia;  
  
public class Portatil extends Dispositivo{  
    private String Modular;
```

```
private String Tactil;  
private String Tipo = "Portatil";
```

A continuación codificamos el método constructor de la clase *Portatil* que recibe 9 parámetros cuyo contenido se almacena en las variables de clase tanto las variables propias de esta clase como en las variables heredadas de las clases *Dispositivo* y la superclase *Aparato*.

```
public Portatil(String Marca, double Peso, String Color, String Maleta, int MemRam, String Procesador, int MemAux, String Modular, String Tactil)  
{  
super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
this.Modular=Modular;  
this.Tactil=Tactil;  
}
```

Construimos el método *MuestraMarca()* que, aprovechando el concepto de polimorfismo, nos permite mostrar la marca, esta vez, del portátil.

```
public String MuestraMarca()  
{  
return "\nPortatil Marca " + Marca;  
}
```

Por último, en esta clase, codificamos el método *Descripción()* que, como su nombre lo indica, describe de una forma textual y simple qué es un computador portátil. De nuevo debe recordarse que el contenido de este método, o sea la descripción en sí, es completamente diferente a la de los otros métodos similares, es decir, los que tienen el mismo nombre.

```
public String Descripcion()  
{  
return "Un Portatil es un computador que se puede  
transportar fácilmente a cualquier lugar";  
}  
}
```

Ahora vamos a declarar una subclase que llamaremos *Electrodomestico* y que hereda de *Aparato* sus variables y sus métodos. Esta es una derivación diferente de las clases abordadas hasta el momento. Servirá revisar el

esquema 3 de esta lección en caso de que se presenten dudas en cuando a la relación entre las clases mencionadas. La clase *Electrodomestico* contiene una variable propia, para simplificar el ejemplo, que se llama *Tipo* y que servirá para determinar a qué tipo de electrodoméstico nos estamos refiriendo ya que son muchos los que existen en una casa (licuadora, equipo de sonido, nevera, ventilador, etc.). El método constructor de la clase Electrodoméstico recibe tres parámetros y almacena uno de ellos en la variable ***Tipo*** que es propia de la clase y los otros dos parámetros los almacena en las variables heredadas de la superclase *Aparato* por medio de su respectivo constructor.

```
// SUBCLASE ELECTRODOMESTICO
package Herencia;

public class Electrodomestico extends Aparato
{
    String Tipo;

    public Electrodomestico(String Marca, double Peso, String Tipo)
    {
        super(Marca, Peso);
        this.Tipo = Tipo;
    }
}
```

En esta clase, el método *Descripcion()* tiene un contenido diferente al de los métodos equivalentes en las otras clases. Este método describe brevemente qué es un electrodoméstico.

```
public String Descripcion()
{
    return "Un Electrodoméstico es un aparato eléctrico que facilita
labores en el hogar";
}
```

A continuación veamos cómo construimos el programa Java. Este programa tiene una clase llamada *Programa* y un método llamado *main()* que es el método cuya presencia diferencia una declaración de una clase normal de una clase que se convierte en el programa como tal.

// PROGRAMA EN JAVA

**package** Herencia;

**public class** Programa {

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

Declaramos un objeto tipo SmartPhone que llamaremos DS, otro objeto tipo Portatil que llamaremos DP, otro objeto tipo Dispositivo que llamaremos DD y un objeto tipo Electrodomestico llamado DE.

*SmartPhone DS = new SmartPhone ("Nokia", 100, "Blanco", "Si",  
2000000, "T6", 600000, "Si", "No");*

*Portatil DP = new Portatil ("Lenovo", 1000, "Rojo", "No",  
1000000, "P5", 5000000, "Si", "No");*

*Dispositivo DD = new Dispositivo ("Compaq", 200, "Amarillo",  
"No", 300000, "T2", 600000);*

*Electrodomestico DE = new Electrodomestico ("General  
Electric", 1500.0 , "Licuadora");*

Seguidamente mostramos, por consola, los datos que retorna el método MuestraMarca( ) de cada objeto declarado y la información que se retorna cuando se invoca el método Descripcion( ) que es el método abstracto de la clase abstracta Aparato que es obligatorio y que, para cada tipo de objeto, tiene un contenido diferente.

*System.out.println (DS.MuestraMarca() + ". " + DS.Descripcion());*

*System.out.println (DP.MuestraMarca() + ". " + DP.Descripcion());*

*System.out.println (DD.MuestraMarca() + ". " + DD.Descripcion());*

*System.out.println (DE.MuestraMarca() + ". " + DE.Descripcion());*

*}*

*}*

Verifique el contenido del método Descripcion( ) de cada objeto declarado.  
El código completo de este programa se presenta a continuación:

```
// CLASE ABSTRACTA APARATO
package Herencia;

public abstract class Aparato
{
    public String Marca;
    public double Peso;

    public Aparato(String Marca, double Peso)
    {

        this.Marca = Marca;
        this.Peso = Peso;
    }

    public String MuestraMarca()
    {
        return "\nMarca = " + Marca;
    }

    public abstract String Descripcion();
}

// SUBCLASE DISPOSITIVO
package Herencia;

public class Dispositivo extends Aparato
{
    private String Color;
    private String Maleta;
    private int MemRam;
    private String Procesador;
    private int MemAux;

    public Dispositivo(String Marca, double Peso, String Color, String
Maleta, int MemRam, String Procesador, int MemAux)
    {
        super(Marca, Peso);
```

```
this.Color = Color;
this.Maleta = Maleta;
this.MemRam = MemRam;
this.Procesador = Procesador;
this.MemAux = MemAux;
}

public String MuestraMarca()
{
    return "\nDispositivo Marca " + Marca;
}

public String Descripcion()
{
    return "Un Dispositivo es un aparato eléctrico o
electrónico que facilita la vida de las personas";
}

package Herencia;

// SUB SUBCLASE SMARTPHONE
public class SmartPhone extends Dispositivo
{
    private String Inteligente;
    private String Protector;
    private String Tipo = "SmartPhone";

    public SmartPhone(String Marca, double Peso, String Color,
String Maleta, int MemRam, String Procesador, int MemAux, String
Inteligente, String Protector)
    {
        super(Marca, Peso, Color, Maleta, MemRam, Procesador,
MemAux);
        this.Inteligente = Inteligente;
        this.Protector = Protector;
    }

    public String MuestraMarca()
```

```
{  
    return "\nSmartPhone Marca " + Marca;  
}  
  
public String Descripcion()  
{  
    return "Un SmartPhone es un aparato celular con aplicaciones y  
utilidades de uso muy frecuente";  
}  
}  
  
// SUB SUBCLASE PORTATIL  
package Herencia;  
  
public class Portatil extends Dispositivo{  
    private String Modular;  
    private String Tactil;  
    private String Tipo = "Portatil";  
  
    public Portatil(String Marca, double Peso, String Color, String  
    Maleta, int MemRam, String Procesador, int MemAux, String Modular,  
    String Tactil)  
    {  
        super(Marca, Peso, Color, Maleta, MemRam, Procesador, MemAux);  
        this.Modular=Modular;  
        this.Tactil=Tactil;  
    }  
  
    public String MuestraMarca()  
    {  
        return "\nPortatil Marca " + Marca;  
    }  
  
    public String Descripcion()  
    {  
        return "Un Portatil es un computador que se puede  
transportar fácilmente a cualquier lugar";  
    }  
}
```

```
// SUBCLASE ELECTRODOMESTICO
package Herencia;

public class Electrodomestico extends Aparato
{
    String Tipo;

    public Electrodomestico(String Marca, double Peso, String Tipo)
    {
        super(Marca, Peso);
        this.Tipo = Tipo;
    }

    public String Descripcion()
    {
        return "Un Electrodoméstico es un aparato eléctrico que facilita
labores en el hogar";
    }
}
```

```
// PROGRAMA EN JAVA
package Herencia;

public class Programa {
    public static void main(String [] args) {

        SmartPhone DS = new SmartPhone ("Nokia", 100, "Blanco", "Si",
200000, "T6", 600000, "Si", "No");

        Portatil DP = new Portatil ("Lenovo", 1000, "Rojo", "No",
1000000, "P5", 5000000, "Si", "No");

        Dispositivo DD = new Dispositivo ("Compaq", 200, "Amarillo",
"No", 300000, "T2", 600000);

        Electrodomestico DE = new Electrodomestico ("General
Electric", 1500.0 , "Licuadora");
    }
}
```

```
System.out.println (DS.MuestraMarca() + ". " + DS.Descripcion());  
System.out.println (DP.MuestraMarca() + ". " + DP.Descripcion());  
System.out.println (DD.MuestraMarca() + ". " + DD.Descripcion());  
System.out.println (DE.MuestraMarca() + ". " + DE.Descripcion());  
}  
}
```

Al ejecutar este programa, el resultado que aparece en la consola es el siguiente:

SmartPhone Marca Nokia. Un SmartPhone es un aparato celular con aplicaciones y utilidades de uso muy frecuente

Portatil Marca Lenovo. Un Portatil es un computador que se puede transportar fácilmente a cualquier lugar

Dispositivo Marca Compaq. Un Dispositivo es un aparato eléctrico o electrónico que facilita la vida de las personas

Marca = General Electric. Un Electrodoméstico es un aparato eléctrico que facilita labores en el hogar

Nótese la diferencia de avisos que se despliegan con los diferentes métodos *Descripcion()* que corresponden a cada objeto declarado.

### 19.3 Ejercicios Propuestos

- Construir un programa Java que permita implementar una función que muestre la marca en diferentes aparatos que pertenecen, por niveles, a una misma clase
- Construir un programa Java que permita implementar una función que describa las características propias de cada dispositivo que pertenecen a clases heredadas de una clase abstracta

- Construir un programa Java que permita implementar la función DescripcionFisica( ) cuyo objetivo es describir los diferentes niveles de una clase llamada Persona de la cual se derivan los cargos de una empresa
- Construir un programa Java que permita implementar la función Habilidades( ) cuyo objetivo es describir las habilidades propias de estudiantes de colegios públicos y privados que se derivan de una clase abstracta común

#### 19.4 Acerca de los modificadores de acceso

Uno de los elementos interesantes que aparecen en un lenguaje orientado a objetos como Java corresponde a los modificadores de acceso, es decir, las palabras que se escriben antes de una variable o de un método. En Java existen 4 modificadores: **public**, **private**, **protected** y **(por defecto)**. El modificador por defecto es el que se asume, por parte del lenguaje, cuando no se escribe nada antes de la variable o del método.

Un modificador de acceso es una especie de instrucción que le permite al intérprete de Java conocer desde donde se puede acceder a las variables, a las clases o a los métodos. Recuérdese el concepto de encapsulación que, a través del modificador **private**, permite que las variables (cuando éste se ha antepuesto a ellas) solo sean visibles dentro de una misma clase y no se pueda acceder a su contenido, bien para modificarlo o para visualizarlo, desde otra clase.

Esto genera un nivel de seguridad que garantiza que los datos sólo se pueda acceder a los datos a partir de las instrucciones que nosotros mismos programemos y que, por accidente o de manera intencional, no se acceda a cambiar contenidos que, en un momento dado, pueden a su vez modificar el desarrollo y resultados de un programa en pos de un determinado objetivo.

El primero de los modificadores, que hemos trabajado ya en este libro, es el modificador **public**. Cuando se antepone este modificador de acceso, bien sea a una variable o bien a un método, se puede acceder al contenido de dicha variable o se puede invocar el método desde cualquier otro método de la misma clase, desde cualquier método de cualquier clase del mismo paquete o desde cualquier subclase que haya heredado el método o

la variable. El modificador **public** le proporciona acceso tanto a métodos como variables prácticamente desde cualquier ubicación.

Por su parte, el modificador **private**, que ya hemos utilizado en el concepto de encapsulación, permite que la variable o el método al cual se le anteponga, puede ser accedido desde la misma clase pero no puede ser accedido desde cualquier otra clase del mismo paquete así como no se puede acceder desde una subclase. Debe entonces tenerse particular cuidado cuando se heredan clases porque es posible que este modificador de acceso genere errores que impidan la correcta ejecución del programa. Vale la pena anotar que es poco frecuente encontrar el modificador de acceso **private** antes de un método. En el caso de las variables no solo es útil sino que, por momentos, es necesario.

Cuando no se declara, de manera específica, ningún modificador de acceso se dice que se asume el modificador **por defecto**. Si no se restringe concretamente el acceso a una variable o a un método, es decir, si se acude al modificador de acceso **por defecto** entonces las variables o los métodos que así se hayan configurado permitirán acceso desde la misma clase así como acceso desde el mismo paquete. Sin embargo no permitirán acceso desde las subclases que hereden dichos métodos o dichas variables. Tanto en esta como en las otras especificaciones, tenga en cuenta que cuando se habla de una subclase, se refiere a subclases que se encuentren dentro de otro paquete más no dentro del mismo paquete.

El cuarto modificador de acceso es **protected** que, si bien aún no lo hemos utilizado, cuanto lo anteponemos a una variable o a un método, tanto el método como la variable serán accesibles desde la clase (por supuesto) y también desde cualquier otra clase que se encuentren dentro del mismo paquete. Sin embargo, y a diferencia del modificador por defecto, el modificador **protected** permite acceso desde una subclase que se encuentre en otro paquete tanto a las variables como a los métodos que lo tengan antepuesto.

Si tenemos en cuenta estas restricciones no solo vamos a encontrarles la gran utilidad que tienen sino que vamos a capitalizar sus bondades para garantizar cierto nivel de seguridad en el acceso a los datos y a los métodos lo cual, sin lugar a dudas, se convierte en un rédito bastante significativo al momento de programar.

## 19.5 Acerca de la clase Object

Hemos hablado que cada clase es posible que herede de otra y eso nos ha llevado a definir el concepto de *clase abstracta*. Sin embargo, tal vez usted se pregunte ¿y no hay una clase que sea tan genérica que todas (absolutamente todas las clases) hereden de ella en algún momento? Y la respuesta es un rotundo y absoluto SI. Esta clase se llama la clase Object que podríamos decir, en un lenguaje muy informal, que es (como quien dice), la “mamá” de todas las clases lo cual quiere decir que toda clase, sea cual fuere, sea nuestra o sea propia de Java, siempre va a heredar de la clase Object.

Usted se preguntará que en ningún momento hemos escrito en alguno de nuestros programas un código como

```
public class MiClase extends Object {
```

y efectivamente no sólo no lo hemos escrito sino que nunca lo tendremos que escribir porque automáticamente Java asume que toda clase heredará de dicha *Object*. La clase Object se ha creado para que contenga unos métodos que, a juicio de los creadores de Java, son propios de absolutamente cualquier clase que se declare. Si, valiéndose de un buscador, usted solicita que se encuentre el filtro “java api”, usted podrá observar el listado tan inmenso de clases con que cuenta este lenguaje y, cuando haga click en una de ellas sea cual fuere, notará que dicha clase siempre heredará en algún momento de *Object*. ¿Por qué digo que en algún momento? Porque es posible que la clase que usted quiera observar herede de otra y esa de otra y esa de otra y, finalmente, esa clase heredará de Object. Entiende ahora ¿por qué se dice que es la “mamá” de todas las clases?

La clase *Object* contiene métodos que son absolutamente genéricos o comunes a todos los objetos. Consulte en “java api” la clase Object y verifique los métodos que allí aparecen como parte constitutiva de su cuerpo de clase. Esta es la razón por la cual cuando se trabaja con un IDE (Integrated Development Environment), que es un programa que facilita la codificación de programas en diferentes lenguajes de programación, y usted escribe el nombre de un objeto (o instancia de clase) usted notará que inmediatamente el IDE no solo le sugiere los métodos que usted ha

codificado sino otros que usted posiblemente no sepa de donde salen. Esos son los métodos que se heredan de la clase *Object* o de sus intermediarios.

La tabla 2, tomada de Java API, presenta los métodos que pertenecen a la superclase cósmica *Object*. El nombre de “superclase cósmica” es muy utilizado en los libros de POO para hacer referencia a la clase *Object*.

Tabla 2. Métodos de la clase Object

Modificador y Tipo	Método	Descripción
<i>protected Object</i>	<i>clone()</i>	Crea y retorna una copia de este objeto
<i>boolean</i>	<i>equals(Object obj)</i>	Indica si otro objeto es igual a este
<i>protected void</i>	<i>finalize()</i>	Llamado por el recolector de basura sobre objetos, cuando la recolección de basura determina que no hay más referencias al objeto
<i>class</i>	<i>getClass()</i>	Retorna la clase runtime de este objeto
<i>int</i>	<i>hashCode()</i>	Retorna un código hash de este objeto
<i>void</i>	<i>notify()</i>	Inicia un hilo individual de ejecución que está monitoreando el objeto
<i>void</i>	<i>notifyAll()</i>	Inicia todos los hilos individuales de ejecución que están monitoreando el objeto
<i>String</i>	<i>toString()</i>	Retorna una representación del objeto en formato de cadena
<i>void</i>	<i>wait()</i>	Genera que el hilo de ejecución actual espere hasta que otro hilo invoque el método <i>notify()</i> o <i>notifyAll()</i> de este objeto
<i>void</i>	<i>wait( long timeout )</i>	Genera que el hilo de ejecución actual espere hasta que otro hilo invoque el método <i>notify()</i> o <i>notifyAll()</i> de este objeto o que otro objeto interrumpa el hilo actual o que haya expirado una cantidad de tiempo real

Es posible que pocas veces tenga que entenderse con la clase *Object* pero no descarte que así sea puesto que contiene métodos que, en algún momento pueden llegar a ser más útiles de lo que se imagina.



Lección

# 20 Datos enumerados



## 20.1 Concepto General

Es muy usual que algunos datos utilicen ciertas abreviaturas predefinidas para describirlos. Este tipo de datos se conocen como datos enumerados. Generalmente las abreviaturas obedecen a convenciones que se utilizan casi mundialmente y que posibilitan que en diferentes culturas se puedan entender unas con otras. Dentro de este contexto es igualmente importante que los datos como tales estén siempre bien escritos y que sus respectivas abreviaturas también estén bien escritas dado que, de otra forma, podrían hacerse interpretaciones inapropiadas dentro de determinados contextos.

Este es el caso que se presenta, por ejemplo, con los símbolos químicos en donde a cada nombre de un elemento químico se le ha asignado un único símbolo químico (que no es más que una abreviatura de su propio nombre). De esta forma el elemento químico llamado *Nitrógeno* tiene el símbolo *N*, el elemento llamado *Oxígeno* tiene el símbolo *O*, el elemento llamado *Hidrógeno* tiene el símbolo *H* y el elemento llamado *Calcio* tiene el símbolo *Ca*. De la misma manera sucede con todos los demás elementos químicos que en total son 118 y que se listan tanto en su nombre, su símbolo químico y sus características en un instrumento conocido como Tabla Periódica.

En un programa en donde se requiera trabajar con elementos químicos, conviene tener un mecanismo ágil para identificar el nombre y asociarlo con su respectivo símbolo químico. Podríamos pensar que eso se haga a través de una clase que contenga un método en la cual, a partir de un uso intensivo de condicionales *if*, podamos resolver la asociación entre nombre del elemento y su respectivo símbolo. Sin embargo, el lenguaje de programación Java ofrece un mecanismo, a través de los datos enumerados, para que este proceso se realice de una forma simple y sencilla de manera que, como siempre, se simplifique nuestra labor como programadores a partir del aprovechamiento de las bondades que posibilita la POO.

## 20.2 Ejemplo Resuelto

El tipo de dato *enum* tiene un tratamiento muy similar al de una clase normal, de las que hasta ahora hemos estudiado, es decir, se puede declarar, puede tener método constructor, se le pueden declarar variables

y valores, puede tener otros métodos, se le pueden construir métodos getter y setter, etc., en fin, se le puede dar el tratamiento normal de una clase. Adicional se puede utilizar un dato tipo **enum** para declarar una variable que reciba sus atributos y propiedades. Por tener el tratamiento idéntico al de una clase debe tenerse en cuenta que la clase **enum** debe declararse por fuera del método *main()* en caso de que estemos codificando clases y método *main()* en un solo archivo. Vamos a construir un ejemplo sobre los compuestos químicos y sus símbolos respectivos tomando solamente algunos de ellos pero entendiendo que, de hacerlo completo, deberíamos escribir los 118 existentes. Una de las diferencias entre la clase **enum** y una clase convencional es que no se puede construir un objeto con el tipo de la clase **enum** que hayamos construido.

Construyamos entonces un tipo de dato que contenga los elementos químicos con sus respectivos símbolos. Aprovechando las bondades de la POO con Java podríamos hacerlo como lo explicaremos a continuación. Primero vamos a seleccionar un paquete que llamaremos *Enumerados*.

**package** *Enumerados*;

Ahora vamos a declarar la clase pública *ElemQuim* dentro de la cual declararemos los datos enumerados que necesitamos en este ejercicio.

**public class** *ElemQuim* {

A continuación creamos la clase Elementos que será de tipo enum o, lo que es lo mismo, creamos el tipo de datos enumerados enum.

**enum** *Elementos* {

Lo primero que hacemos es construir las diferentes cadenas con sus equivalencias. En este caso, cada cadena corresponde a un elemento químico (de los cuales solo tomaremos 5 de ellos) y cada uno lo asociaremos con su respectivo símbolo químico. La sintaxis que proporciona Java es la que se utiliza en esta línea, es decir, nombre del símbolo y entre paréntesis y comillas dobles el símbolo químico. Debe aclararse que en este caso se utilizan comillas dobles debido a que algunos símbolos químicos como el Calcio (Ca) o como el Sodio (Na) tienen dos caracteres. Las comillas dobles representan datos de tipo cadena.

**Nitrógeno("N"), Oxígeno("O"), Hidrógeno("H"),  
Calcio("Ca"), Sodio("Na");**

Ahora declaramos el atributo de esta clase que, en este caso, será una variable de tipo cadena (*String*) a la cual le pondremos el nombre *Simbolo*.

```
private String Simbolo;
```

A continuación, debido a que los datos enumerados tienen un tratamiento muy similar a las clases, codificamos el método constructor. Como la clase enum tiene el nombre *Elementos*, entonces el método constructor también se llamará *Elementos*. En este método todo lo que haremos será recibir una cadena y asociarla con el símbolo correspondiente de esa cadena. Este es el momento en donde Java hace el transfer de reconocer la cadena, para este caso como un nombre de un elemento químico, y asociarlo con otra cadena que corresponde a su respectivo símbolo químico.

```
private Elementos(String Simbolo) {  
    this.Simbolo = Simbolo;  
}
```

Para facilitar el manejo de esta clase, y teniendo en cuenta que el método constructor es de tipo *private* así como la variable *Simbolo*, entonces creamos un método público tipo getter que permita mostrar el símbolo químico asociado al nombre del elemento químico. Con este último método, damos por finalizada la “clase” enumerada que hemos llamado *Elementos*.

```
public String MuestraSimbolo() {  
  
    return Simbolo;  
}  
}
```

Seguidamente, construirmos el respectivo método *main()*. Tenga en cuenta que se hubiera podido construir la clase *Elementos* con el tipo de datos enumerados en otro archivo del mismo paquete y se hubiera podido utilizar en un archivo que constituyera el programa principal. Por facilidades se ha codificado de esta manera. Vamos a hacerlo de una manera muy sencilla.

Creamos cuatro variables tipo *String* para almacenar cada uno de los elementos químicos. Esta parte se hubiera podido hacer permitiendo que el usuario escribiera el nombre del elemento químico bien a través de un objeto *Scanner* o bien a través de la utilidad *JOptionPane*. Sin embargo, para facilitar la explicación y solo por ello, asignaremos directamente los nombres de los elementos químicos que usaremos como ejemplo.

```
public static void main(String[] args) {  
  
    String E1 = "Nitrógeno";  
    String E2 = "Oxígeno";  
    String E3 = "Calcio";  
    String E4 = "Sodio";
```

A continuación, creamos 4 instancias objeto del tipo *Elementos* (o sea, instancias del tipo de dato enumerado). En cada una almacenamos el símbolo equivalente a cada una de los elementos almacenados en las variables tipo *String*. Esto lo logramos a través del método *valueOf* que es un método que, por ser *static*, requiere que se le anteponga el nombre del tipo de clase, que en este caso es *Enum* tal como lo indica Java API.

Este método requiere dos parámetros: el primero es el nombre de la clase enumerada seguida por la palabra *class* y el segundo es el nombre de la cadena a la cual se le quiere obtener la subcadena equivalente. Para este ejemplo, el segundo corresponde al elemento químico al cual se le quiere encontrar su símbolo químico según lo establecimos al inicio de la declaración de la clase enumerada *Elementos*. También, para efectos del ejemplo, se hace esto mismo con cada uno de los 4 elementos seleccionados.

```
Elementos S1=Enum.valueOf(Elementos.class, E1);  
Elementos S2=Enum.valueOf(Elementos.class, E2);  
Elementos S3=Enum.valueOf(Elementos.class, E3);  
Elementos S4=Enum.valueOf(Elementos.class, E4);
```

Finalmente, mostramos por consola tanto el nombre del elemento químico (almacenado en la variable de tipo *String*) como el símbolo químico correspondiente que lo obtenemos a través del método *MuestraSimbolo* y que, a su vez, él lo obtiene de los datos enumerados que declaramos al

inicio de la clase **enum**. De esta forma, finalizamos el programa que necesitamos para ejemplificar el uso de datos enumerados.

```
System.out.println("Elemento: " + E1 + "\tSimbolo: " + S1.MuestraSimbolo());
System.out.println("Elemento: " + E2 + "\tSimbolo: " + S2.MuestraSimbolo());
System.out.println("Elemento: " + E3 + "\tSimbolo: " + S3.MuestraSimbolo());
System.out.println("Elemento: " + E4 + "\tSimbolo: " + S4.MuestraSimbolo());  
}  
}
```

El código completo de este programa se presenta a continuación:

```
package Enumerados;
public class ElemQuim {
    enum Elementos {
        Nitrógeno("N"), Oxígeno("O"), Hidrógeno("H"),
        Calcio("Ca"), Sodio("Na");

        private String Simbolo;
        private Elementos(String Simbolo) {
            this.Simbolo = Simbolo;
        }

        public String MuestraSimbolo() {
            return Simbolo;
        }
    }

    public static void main(String[] args) {
        String E1 = "Nitrógeno";
        String E2 = "Oxígeno";
        String E3 = "Calcio";
        String E4 = "Sodio";

        Elementos S1=Enum.valueOf(Elementos.class, E1);
        Elementos S2=Enum.valueOf(Elementos.class, E2);
        Elementos S3=Enum.valueOf(Elementos.class, E3);
        Elementos S4=Enum.valueOf(Elementos.class, E4);
```

```
System.out.println("Elemento: " + E1 + "\tSimbolo: " + S1.MuestraSimbolo());  
System.out.println("Elemento: " + E2 + "\tSimbolo: " + S2.MuestraSimbolo());  
System.out.println("Elemento: " + E3 + "\tSimbolo: " + S3.MuestraSimbolo());  
System.out.println("Elemento: " + E4 + "\tSimbolo: " + S4.MuestraSimbolo())  
    }  
}
```

Al ejecutar este programa, la salida por consola se presenta seguidamente:

```
Elemento: Nitrógeno  Simbolo: N  
Elemento: Oxígeno    Simbolo: O  
Elemento: Calcio     Simbolo: Ca  
Elemento: Sodio      Simbolo: Na
```

Sobra decir que de esta forma se le puede dar cualquier otro uso a los datos enumerados referenciándolos a través de la función *getter()* que hayamos construido para tal fin. Como puede ver, el uso de los datos enumerados constituye una solución simple, sencilla y práctica que, en algunos casos, resuelve situaciones que ahorran mucho código y hacen más sencilla la comprensión de los programas. Si bien este código también se hubiera podido construir con un uso intensivo de condicionales if, no se puede negar que el uso de datos enumerados condensa y simplifica la lógica y posibilita un uso más adecuado de datos que tengan estas características.

### 20.3 Ejercicios Propuestos

- Construir un programa Java que, a través de datos enumerados, permita manejar cinco marcas de carros y una abreviatura por cada una que no exceda tres letras
- Construir un programa Java que, a través de datos enumerados, permita manejar las abreviaturas formales de seis instrumentos musicales
- Construir un programa Java que, a través de datos enumerados, permita manejar los códigos de las asignaturas de un pensum escolar
- Construir un programa Java que, a través de datos enumerados, permita manejar diez ciudades con sus respectivas abreviaturas según los estándares internacionales
- Construir un programa Java que, a través de datos enumerados, permita manejar 8 equipos de futbol con sus respectivas abreviaturas comerciales o siglas

# Lección 21 Interfaces



## 21.1 Concepto General

Ya hemos dicho que cuando se hereda una clase, en realidad se están heredando sus atributos (variables) y sus métodos, sin embargo, el lenguaje de programación Java no permite la herencia múltiple, es decir, que una clase pueda heredar de varias clases al tiempo como podría ser posible en la realidad dado que, si lo llevamos a un caso verdadero, un hijo puede heredar el carro de su padre biológico pero también podría heredar el carro de su padre putativo. En Java esto no se puede hacer y por lo tanto se cuenta con un recurso que permite hacer algo muy aproximado que resuelve el problema en caso de que se requiera implementar este tipo de situaciones, al fin y al cabo, la programación está diseñada para representar la realidad en el computador.

Esta facilidad se utiliza debido a que algunas veces se hace necesario caracterizar ciertos métodos para que se incluyan siempre en las clases que los vayan a utilizar. En este caso no hablaremos de superclases sino que hablaremos de Interfaces que podríamos definirlas como un determinado conjunto de indicaciones a nivel de código que deben ser de obligatorio cumplimiento por parte de las clases que las implementen. De alguna manera el comportamiento de las Interfaces tiene muchas similitudes con las superclases. Dado que las clases implementan a las *Interfaces*, vale la pena tener en cuenta que los métodos que estén indicados en las Interfaces son de obligatorio cumplimiento en dichas clases, es decir, es obligatorio que implementen unos métodos con el prototipo que se indique en las *Interfaces*. El prototipo de un método es la primera línea del método, es decir, el modificador de acceso, el indicador de invocación, el tipo de dato a retornar, el nombre del método y los parámetros que se requieran. Por ejemplo:

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

corresponde al prototipo del método *main()*. En esta línea se indica que esta función es pública (*public*), estática (*static*), no retorna nada (*void*), se llama *main* y requiere como parámetro un vector de tipo *String* llamado *args*. Como se puede observar, esta línea describe, de manera muy resumida, el objetivo de este método. Es fácil asumir que si se llama *main*

es porque es el método principal debido a que *main* es una palabra del inglés que, en español, significa lo más importante, lo principal. Esto indica que la relación entre una clase que implemente una Interfaz y dicha Interfaz es muy similar a la relación entre una clase heredada y una clase abstracta (especialmente con sus métodos abstractos). Las interfaces determinan, entonces, los comportamientos con los cuales deben cumplir las clases que las implementen.

Al igual que las clases, las interfaces también pueden estar definidas por la API de Java, es decir, pueden venir con todo el paquete de utilidades que se incluyen en el lenguaje de programación Java y que se pueden estudiar si busca “Java API” con un buscador, valga esta redundancia. También se pueden diseñar, como lo haremos en esta lección, interfaces definidas por nosotros mismos para facilitar la labor de la programación y aprovechar las ventajas que ofrece la POO. Es bueno tener en cuenta que las interfaces solamente pueden contener métodos que sean abstractos y no pueden contener variables, es decir, siempre deberán contener constantes. En la parte operativa, las interfaces se almacenan en un archivo de texto con extensión .class y, a pesar de tener todas las características aparentemente de una clase, no se pueden crear objetos con el tipo que indique la interface, es decir, no se pueden crear instancias de dicha clase. Solamente se pueden implementar a través de una clase, aquellos métodos que aparecen descritos como obligatorios dentro de la interface.

Los métodos de las interfaces siempre son públicos y también son abstractos, sea que se definan de manera explícita o sea que se utilice el modificador de acceso por defecto. Vale la pena tener en cuenta, para efectos de capitalizar las ventajas que ofrece la POO a través del lenguaje de programación Java, que una clase puede implementar tantas interfaces como se requieran. No hay restricción.

Si, dentro de una clase, se tuviera la siguiente línea:

```
class Egresado extends Estudiante implements Interfaz_a, Interfaz_b {
```

se le está indicando a Java que se está definiendo una clase llamada Egresado que hereda atributos y métodos de una superclase llamada Estudiante y que implementa los métodos que describe la Interfaz\_a y la Interfaz\_b. La instrucción implements permite establecer la relación entre una clase y la interfaz que se requiera. La línea

```
class Dispositivo implements Interfaz1 {
```

le indica a Java que se está declarando una clase que hemos llamado *Dispositivo* y que implementa, en su interior, los métodos descritos en la interfaz que hemos llamado *Interfaz1*.

## 21.2 Ejemplo Resuelto

Vamos a asumir que necesitamos un “convertidor de mes” para el desarrollo de un programa. Un “convertidor de mes” es un mecanismo a través del cual digitamos el número de un mes y, cuando se requiera, el programa muestra el nombre del mes. Por ejemplo, el mes 1 lo identifica como *Ene*, el mes 2 lo identifica como *Feb* y así sucesivamente. Lo primero que vamos a aclarar es que esto lo podríamos hacer a través de una sentencia **switch** sin embargo, por facilidad y para asegurar el proceso, lo haremos a través de la implementación de una interfaz que posibilite el proceso requerido. Esta vez vamos a trabajar en un paquete que llamaremos *Interfaces*. Lo primero que vamos a realizar es declarar la interface cuyo nombre, para el ejemplo, será *Interface1*.

```
package Interfaces;  
  
public interface Interface1 {
```

En esta interfaz vamos a declarar como constante (final), un vector de cadenas que van a almacenar el nombre abreviado de cada uno de los meses.

```
final String[] Mes = {"Ene", "Feb", "Mar", "Abr", "May", "Jun",  
"Jul", "Ago", "Sep", "Oct", "Nov", "Dic"};
```

A continuación escribimos el método que deberá ser incluido por las clases que vayan a implementar esta interface. En este caso se trata del método *MuestraMes* que es público (*public*), también es abstracto (*abstract*), retorna una cadena (*String*) y recibe como parámetro un valor de tipo entero.

```
public abstract String MuestraMes(int n);  
}
```

Note usted que solo se escribe el prototipo del método. No se desarrolla puesto que esa es precisamente la misión que se tiene en cada uno de las clases que quieran implementar esta interface. En este caso hemos declarado el método como público (cualquier puede invocarlo) y abstracto (es obligatorio para toda clase que implemente la interface), sin embargo si no se hubiera declarado, el hecho de ser el prototipo de un método escrito dentro de una interface, automáticamente se asumirá, por parte del lenguaje de programación, como público y abstracto. En este caso solamente describiremos un solo método dentro de la interface *Interface1*, por lo tanto, después de describirlo cerramos la declaración de la interface con una llave. Ahora vamos a declarar una clase que implemente esta interface. De nuevo trabajamos dentro del paquete *Interfaces*, por facilidad. Seguidamente declaramos la clase pública *Estudiante* que va a implementar (**implements**) la interface *Interface1*.

```
package Interfaces;  
  
public class Estudiante implements Interface1 {
```

Para que el ejemplo sea lo mas completo posible, vamos a declarar los atributos (variables) de esta clase. En este caso estos atributos corresponden a un código (tipo cadena), un nombre (tipo cadena), una fecha en donde cada uno de sus componentes (el día, el mes y el año) son de tipo entero.

```
String Cod;  
String Nom;  
int dd;  
int mm;  
int aa;
```

Dado que se trata de la declaración de una clase, procedemos a codificar su método constructor que, por las razones que ya conocemos, debe tener el mismo nombre de la clase. Este método recibirá como parámetros dos cadenas y tres enteros que corresponden al código, el nombre y la fecha del estudiante a crear. Almacenamos cada uno de los parámetros recibidos en cada uno de los atributos de la clase en donde, a través del indicador *this*, se hace referencia al objeto que se haya creado. Con esto concluimos la codificación del método constructor.

```
public Estudiante(String Cod, String Nom, int dd, int mm, int aa) {  
  
    this.Cod = Cod;  
    this.Nom = Nom;  
    this.dd = dd;  
    this.mm = mm;  
    this.aa = aa;  
}
```

Ahora, en consonancia con lo visto hasta el momento sobre las interfaces y debido a que la interface *Interface1* tiene como obligatorio un método llamado *MuestraMes* que recibe un valor entero como parámetro, implementamos dentro de la clase el método *MuestraMes*. Este método lo que hará será retornar la cadena correspondiente al nombre abreviado del mes que se escriba en la fecha. Para ello se apoya en el vector de nombres abreviados que hemos declarado como constante dentro de la interface *Interface1*.

```
public String MuestraMes(int n) {  
  
    return Mes[n - 1];  
}  
}
```

Con esto finalizamos tanto la codificación del método obligatorio *MuestraMes* como la declaración de la clase.

Ahora vamos a codificar el programa Java que permite utilizar la clase *Estudiante* que implementa la interface *Interface1*. Acudimos al paquete *Interfaces* para que reconozca los recursos que vamos a utilizar. Asimismo importamos la clase *Scanner* de la librería *java.util* para poder leer un dato a través del teclado.

```
package Interfaces;  
  
import java.util.Scanner;
```

Seguidamente declaramos la clase *ProgInterface1* que contiene el método principal *main()*. En esta clase declaramos dos variables tipo **String** (cadena) y tres variables tipo **int** (enteros).

```
public class ProgInterface1 {  
  
    public static void main(String[ ] args) {  
  
        String Cod, Nom;  
        int dd, mm, aa;
```

A continuación creamos una instancia del objeto Scanner y lo asociamos con el teclado (*System.in*) para poder leer un dato a través de este periférico.

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

Ahora vamos a presentar unos títulos y leeremos, a través del teclado, un código y un nombre (ambos de tipo cadena) con el método que para tal fin provee la función *Scanner* (*nextLine()* )

```
System.out.println("ESTUDIANTE\n=====");  
System.out.println("\nCódigo: ");  
Cod = Teclado.nextLine();  
  
System.out.println("\nNombre: ");  
Nom = Teclado.nextLine();
```

Seguidamente, apoyándonos en unos títulos apropiados, leemos tres valores enteros, el 1º corresponde a un día, el 2º corresponde a un mes y el 3º corresponde a un año y los almacenamos en las variables que para tal fin hemos declarado.

```
System.out.println("\nDia: ");  
dd = Teclado.nextInt();  
System.out.println("\nMes: ");  
mm = Teclado.nextInt();  
System.out.println("\nAño: ");  
aa = Teclado.nextInt();
```

Ahora vamos a declarar una instancia de la clase Estudiante que llamaremos Est1 para lo cual aprovechamos su constructor.

*Estudiante Est1 = new Estudiante(Cod, Nom, dd, mm, aa);*

Por último, en este programa, mostramos todos los datos leídos con un formato que sea estéticamente agradable. Nótese que al momento de mostrar el mes no mostramos el número que introdujo el usuario de este programa sino que se muestra la abreviatura del mes correspondiente. Para ello invocamos el método MuestraMes (que era obligatorio pues estaba definido en Interface1) y que retorna el nombre abreviado del mes.

```
System.out.println("\nCodigo\t" + Est1.Cod + "\nNombre\t" +
Est1.Nom + "\nFecha \t" + Est1.dd + "-" + Est1.MuestraMes(mm) + "-"
+ Est1.aa);
}
}
```

Finalmente cerramos tanto la llave del método principal como la llave de la clase que lo encierra y allí finaliza el programa. El código completo del programa se presenta a continuación.

```
// INTERFACE

package Interfaces;

public interface Interface1 {
final String[] Mes = {"Ene", "Feb", "Mar", "Abr", "May", "Jun", "Jul",
"Ago", "Sep", "Oct", "Nov", "Dic"};

```

```
    public abstract String MuestraMes(int n);
}
```

*// CLASE QUE IMPLEMENTA LA INTERFACE*

```
package Interfaces;

public class Estudiante implements Interface1 {
    String Cod;
    String Nom;
    int dd;
    int mm;
    int aa;
```

```
public Estudiante(String Cod, String Nom, int dd, int mm, int aa) {  
  
    this.Cod = Cod;  
    this.Nom = Nom;  
    this.dd = dd;  
    this.mm = mm;  
    this.aa = aa;  
}
```

```
public String MuestraMes(int n) {  
  
    return Mes[n - 1];  
}  
}
```

// PROGRAMA JAVA

```
package Interfaces;  
import java.util.Scanner;  
public class ProgInterface1 {  
  
    public static void main(String[] args) {  
        String Cod;  
        String Nom;  
        int dd, mm, aa;  
  
        Scanner Teclado = new Scanner(System.in);  
  
        System.out.println("ESTUDIANTE\n=====");  
        System.out.println("\nCódigo: ");  
        Cod = Teclado.nextLine();  
  
        System.out.println("\nNombre: ");  
        Nom = Teclado.nextLine();  
  
        System.out.println("\nDia: ");  
        dd = Teclado.nextInt();  
        System.out.println("\nMes: ");
```

```
mm = Teclado.nextInt();
System.out.println("\nAño: ");
aa = Teclado.nextInt();

Estudiante Est1 = new Estudiante(Cod, Nom, dd, mm, aa);

System.out.println("\nCodigo\t" + Est1.Cod + "\nNombre\t" +
Est1.Nom + "\nFecha \t" + Est1.dd + "-" + Est1.MuestraMes(mm) + "-"
+ Est1.aa);
}
}
```

Al ejecutar este programa, si se le digita la información que se presenta, el resultado es el siguiente:

ESTUDIANTE

=====

Código:

32802

Nombre:

Omar I Trejos B

Dia:

4

Mes:

10

Año:

2017

Codigo      32802

Nombre      Omar I Trejos B

Fecha      4-Oct-2017

Nótese que en la fecha aparece el nombre abreviado del mes a pesar de que se recibió el número de dicho mes.

### 21.3 Ejercicios Propuestos

- Construir un programa Java que permita cobrar el valor del impuesto IVA (que equivale al 19%) de un producto cualquiera usando la facilidad de las interfaces
- Construir un programa que permita descontar de un producto en promoción el 20% del precio del producto usando la facilidad de las interfaces
- Construir un programa que permita pagar a un empleado el 5% adicional sobre el valor de su salario por cada hijo que tenga, usando la facilidad de las interfaces
- Construir un programa que permita aumentar el salario de un empleado en un porcentaje dado a partir del uso de interfaces
- Construir un programa que permita descontar el valor por salud del salario de un empleado (que equivale al 3%) usando la facilidad de las interfaces

### 21.4 Herencia entre Interfaces

Hasta el momento hemos hablado de la posibilidad de que una clase herede de otra sus atributos (variables) y sus métodos. Dado que las interfaces tienen un comportamiento muy similar a las clases, la propiedad de la herencia se puede concebir solamente entre interfaces, es decir, una interfaz puede heredar de otra interfaz. Esto implica que lo que una interfaz hereda de otra interfaz corresponde a los métodos que contiene lo cual quiere decir que si una clase implementa la interfaz hija, entonces deberá implementar tanto los métodos que contiene dicha interfaz como los métodos que contiene la interfaz padre que, en este caso, actuaría como si fuera una superclase. Veámoslo con un ejemplo.

Supongamos que debemos establecer una interfaz para calcular el impuesto a pagar por determinado producto y que también debemos establecer el descuento a tener en cuenta ya que, por políticas del almacén, en ciertas épocas algunos productos son beneficiados con descuentos para promover su adquisición por parte de los clientes. Para ello debemos partir del hecho de que todos los productos pagan impuesto pero no todos los productos reciben el beneficio de descuento cuando son adquiridos. Por esta razón, vamos a proceder a construir una interfaz padre (lo que se

parecería a una superclase) que describa el impuesto a cobrar debido a que éste es general para todos y también vamos a construir una interfaz hija (que podría precerse a una subclase) que describa el descuento a realizarle al producto dependiendo de las políticas del almacén. Esta vez vamos a trabajar con el paquete *Interfaces2* que lo hemos llamado así para que no se confunda con el paquete del ejemplo anterior. Recuerde que en estos ejemplos hemos estado trabajando con archivos diferentes pero todo el código, incluyendo interfaces y clases, podría escribirse en un solo archivo.

```
package Interfaces2;
```

Definimos la interface como pública (**public**) y la llamamos *InterfaceBase*. En esta interface todo lo que vamos a hacer es establecer que la clase que la implemente deberá tener un método llamado *Impuesto* que retorna un valor doble (**double**) y que recibe también un valor de tipo doble (**double**) que hemos llamado *Porc* como haciendo referencia a un porcentaje.

```
public interface InterfaceBase {
```

```
    double Impuesto(double Porc);  
}
```

Ahora vamos a construir una interfaz, que también pertenece al paquete *Interfaces2*, y que describirá un método llamado *Descuento* que será obligatorio para toda clase que la implemente. La interface *InterfaceHeredada* hereda de *InterfaceBase*. Esto significa que cualquier método que implemente el método *Descuento* (que es el método que se describe en *InterfaceHeredada*) también deberá implementar el método *Impuesto* que es el método que se describe en la interface *InterfaceBase* puesto que esta actúa a manera de superclase.

```
package Interfaces2;
```

```
public interface InterfaceHeredada extends InterfaceBase{
```

```
    double Descuento (double Desc);  
}
```

Seguidamente vamos a codificar la clase *Producto* que implementa la interface *InterfaceHeredada* y, debido a que ésta implementa a su vez la interface *InterfaceBase*, obliga a que dicha clase *Producto* tenga codificado tanto el método *Descuento* como el método impuesto. Esta clase, en formato de archivo independiente, lo relacionamos con el paquete *Interfaces2*.

```
package Interfaces2;
```

```
public class Producto implements InterfaceHeredada {
```

Esta clase tiene declarados como atributos una variable entera llamada *Codigo*, una variable tipo cadena llamada *Descripción* y dos variables tipo double llamadas *Precio* y *PrecioReal* respectivamente.

```
int Codigo;  
String Descripción;  
double Precio;  
double PrecioReal;
```

Como es de esperarse, el primer método que codificamos es el constructor de la clase que, por las razones que se han explicado en otras lecciones, debe tener el mismo nombre de la clase. Este método constructor recibe como parámetros un entero, una cadena y un valor doble. Estos valores los almacena en las respectivas variables de la clase.

```
public Producto(int Codigo, String Descripción, double Precio) {  
    this.Codigo = Codigo;  
    this.Descripción = Descripción;  
    this.Precio = Precio;  
}
```

Ahora vamos a codificar el método *Descuento* que está descrito en la interface *InterfaceHeredada* y que, por tal motivo, es obligatoria su inclusión en la clase *Producto* debido a que ésta implementa aquella interface.

```
public double Descuento(double Desc) {  
    return (this.Precio * Desc)/100;  
}
```

Como la interface *InterfaceHeredada* es una clase hija de *InterfaceBase*, entonces vamos a codificar el método Impuesto que está descrito en *InterfaceBase* por las razones que implica la herencia entre interfaces. Con la codificación de este método, finalizamos la clase Producto.

```
public double Impuesto(double porcentaje) {  
    return (this.Precio * porcentaje)/100;  
}  
}
```

Ahora avamos a construir el programa que utilice la estructura de interfaces y clase que explicamos anteriormente. Este programa también esta inscrito en el paquete *Interfaces2* y lo hemos llamado *ProgInterface* intentando hacer referencia a que es un programa que incluye interfaces. Debido a que esta es la clase principal, entonces contiene un método llamado *main* desde donde se codifica la acción del programa Java como tal.

```
package Interfaces2;  
  
public class ProgInterface{  
  
    public static void main(String [] args) {
```

Declaramos dos instancias de la clase Producto que hemos llamado P1 y P2 haciendo uso del respectivo constructor a través del cual enviamos como parámetros un valor entero, una cadena y un valor real.

```
Producto P1 = new Producto (1122, "Grabadora Sonyo", 100000);  
Producto P2 = new Producto (2233, "Televisor Elnovo", 40000);
```

A continuación calculamos el contenido de la variable *PrecioReal* a partir de tomar el precio del producto restarle el descuento y sumarle el impuesto correspondiente. Es en este caso en donde usamos los métodos obligatorios que han sido implementados de las interfaces *InterfaceHeredada* e *InterfaceBase*.

```
P1.PrecioReal = P1.Precio - P1.Descuento(20.0) + P1.Impuesto(15.0);  
P2.PrecioReal = P2.Precio - P2.Descuento(20.0) + P2.Impuesto(15.0);
```

Por último, mostramos en pantalla los datos de cada producto almacenados en las variables respectivas de cada instancia de la clase *Producto*.

```
System.out.println("\nCodigo :|t" + P1.Codigo);
System.out.println("Nombre :|t" + P1.Descripción);
System.out.println("Precio :|t" + P1.Precio);
System.out.println("P Real :|t" + P1.PrecioReal);

System.out.println("\nCodigo :|t" + P2.Codigo);
System.out.println("Nombre :|t" + P2.Descripción);
System.out.println("Precio :|t" + P2.Precio);
System.out.println("P Real :|t" + P2.PrecioReal);
    }
}
```

Si queremos observar el código Java completo, éste se presenta a continuación:

```
// INTERFAZ PADRE
package Interfaces2;

public interface InterfaceBase {
    double Impuesto(double Porc);
}

// INTERFAZ HIJA
package Interfaces2;

public interface InterfaceHeredada extends InterfaceBase{
    double Descuento (double Desc);
}

// CLASE QUE IMPLEMENTA LA INTERFAZ HIJA
// Y POR ENDE LA INTERFAZ PADRE
package Interfaces2;

public class Producto implements InterfaceHeredada {
    int Codigo;
    String Descripción;
```

```
double Precio;
double PrecioReal;

public Producto(int Codigo, String Descripción, double Precio) {
    this.Codigo = Codigo;
    this.Descripción = Descripción;
    this.Precio = Precio;
}

public double Descuento(double Desc) {
    return (this.Precio * Desc)/100;
}

public double Impuesto(double porcentaje) {
    return (this.Precio * porcentaje)/100;
}

}

// CLASE (PROGRAMA) PRINCIPAL

package Interfaces2;

public class ProgInterface{

public static void main(String [] args){

Producto P1 = new Producto (1122, "Grabadora Sonyo", 100000);
Producto P2 = new Producto (2233, "Televisor Elnovo", 40000);

P1.PrecioReal = P1.Precio - P1.Descuento(20.0) + P1.Impuesto(15.0);
P2.PrecioReal = P2.Precio - P2.Descuento(20.0) + P2.Impuesto(15.0);

System.out.println("\nCodigo :|t" + P1.Codigo);
System.out.println("Nombre :|t" + P1.Descripción);
System.out.println("Precio :|t" + P1.Precio);
System.out.println("P Real :|t" + P1.PrecioReal);

System.out.println("\nCodigo :|t" + P2.Codigo);
```

```
System.out.println("Nombre :|t" + P2.Descripción);
System.out.println("Precio :|t" + P2.Precio);
System.out.println("P Real :|t" + P2.PrecioReal);
    }
}
```

Al ejecutar el programa con los datos que se han escrito, el resultado es el siguiente:

```
Codigo : 1122
Nombre : Grabadora Sonyo
Precio : 100000.0
P Real : 95000.0
```

```
Codigo : 2233
Nombre : Televisor Elnovo
Precio : 40000.0
P Real : 38000.0
```

## 21.5 Ejercicios Propuestos

- Construir un programa Java que permita diseñar una solución para realizar un descuento para todos los estudiantes que se matriculen en un colegio y además para que los estudiantes de estratos 1 y 2 reciban un descuento adicional, aprovechando el concepto de herencia entre interfaces
- Construir un programa Java que permita pagar un 3% adicional como recargo al salario por cada hijo que tenga un empleado de una empresa y que pague 2% más por cada hijo después del 3er hijo, aprovechando el concepto de herencia entre interfaces
- Construir un programa Java que permita asignarle 2 puntos a un equipo de futbol por cada partido ganado y que le adicione 1 punto más cuando dicho triunfo se hace como equipo visitante, es decir, en la casa del otro equipo, aprovechando el concepto de herencia entre interfaces
- Construir un programa Java que permita asignar el valor de una multa de tráfico a una persona y que, además, permita descontarle el 30% de dicha deuda si la persona paga antes de 3 días, aprovechando el concepto de herencia entre interfaces
- Construir un programa Java que permita adquirir un producto por un precio determinado y descontarle el 10% de su precio por pago en efectivo, aprovechando el concepto de herencia entre interfaces

Lección

# 22 Clases internas



## 22.1 Concepto General

Algunas veces la interacción con las variables (atributos) propias de una clase se puede convertir en algo un poco dispendioso dado que, por momentos, deben referenciarse de diferentes formas y eso hace que el código no siempre sea tan entendible y tan legible incluso para personas capacitadas en programación orientada a objetos o sencillamente en lenguaje Java.

Para simplificar esto, entre otros objetivos, el lenguaje de programación Java pone a disposición de los programadores la posibilidad de declarar *Clases Internas*, es decir, clases que se declaran dentro de otra clase para que el código sea más fácil de entender y para que algunos procesos sean mas legibles desde el punto de vista del código.

Debe anotarse que este tipo de facilidades también se pueden hacer con la construcción de métodos como parte de una clase, sin embargo, cada programador evaluará cuál opción le parece más simple para facilitar la construcción de sus programas Java.

Veamos un ejemplo sencillo que nos muestre la facilidad de construir un programa Java aprovechando el concepto de clases internas y las posibilidades que éstas ofrecen en la POO. Vamos a suponer que se tiene un programa que permite crear nuevos empleados en una empresa. Debo advertir que es un programa muy sencillo que se ocupa de destacar el uso y aprovechamiento de las clases internas. En esta empresa, a los empleados se les brinda una bonificación dependiendo de un factor que se asocia con criterios propios de la empresa.

Si dicho factor es menor que 20, al empleado se le hace un recargo de \$100.000,oo. Si el factor es igual a 20, entonces el recargo es de \$300.000,oo y si dicho factor es mayor a 20 el recargo es de \$500.000,oo. Podríamos construir un método que resuelva la situación sin embargo vamos a acudir al uso de clases internas para aprovechar sus bondades.

## 22.2 Ejemplo Resuelto

Esta vez vamos a trabajar con un paquete que hemos llamado *ClasesInternas* y cuyo objetivo es agrupar los dos archivos que conforman el programa en mención. Recuerde que estos dos archivos pueden convertirse en uno solo y, de esta forma, tener todo el código junto para efectos de hacer más fácil, si así usted lo considera, la corrección de errores.

```
package ClasesInternas;
```

A continuación vamos a declarar la clase *ClaseI* que contiene cuatro variables: una de tipo entero (**int**), otra de tipo cadena (**String**) y dos variables de tipo doble (**double**).

```
public class ClaseI {  
    int Codigo;  
    String Nombre;  
    double Salario;  
    double SalNeto;
```

Ahora vamos a codificar el método constructor de *ClaseI* que, por lo que ya sabemos, tiene el mismo nombre de la clase. Este método recibe cuatro parámetros: un entero (**int**) que representa un código, una cadena (**String**) que representa un nombre, un valor doble (**double**) que representa un salario y un valor entero (**int**) que representa un factor.

```
public ClaseI(int Codigo, String Nombre,  
              double Salario, int Factor) {
```

Los valores almacenados en los parámetros del método constructor se almacenan en las variables respectivas de la clase *ClaseI*. La variable *SalNeto* (que es de también de la clase) se calcula dependiendo del valor que se haya almacenado en el parámetro *Factor* que se recibió en el método constructor. Este cálculo se realiza a través del método *CalculaSalNeto* que explicaremos seguidamente y al cual se le envía el contenido de la variable *Factor* como parámetro.

```
this.Codigo = Codigo;  
this.Nombre = Nombre;
```

```
this.Salario = Salario;
CalculaSalNeto(Factor);
}
```

Veamos ahora el método *CalculaSalNeto* que recibe como parámetro el contenido de la variable Factor que se recibió en el método constructor. Para calcular el salario neto del empleado (que sería igual al salario mas la bonificación que se le concede dependiendo del valor del factor) se declara una instancia de la clase *ClaseInterna* que hemos llamado *V1* y cuyo constructor solo requiere un parámetro. Retornamos el valor 1 para mantener el sentido de la declaración en el prototipo del método.

```
public double CalculaSalNeto(int Factor) {

    ClaseInterna V1 = new ClaseInterna(Factor);

    return I;
}
```

Seguidamente construimos una clase que hemos llamado *ClaseInterna*, DENTRO de la clase *ClaseI* para facilitar el cálculo del salario neto (que se almacenará en la variable *SalNeto* que es una de las variables de clase). Nótese usted que, debido a que esta clase *ClaseInterna* se encuentra DENTRO de la clase *ClaseI*, para referenciar cualquiera de las variables de la clase no tenemos que instanciarlas, es decir, podemos hacer referencia a las variables propias de la clase (sean **private** o no) con solo nombrarlas y eso facilita muchísimo la construcción de cualquier programa.

En esta clase *ClaseInterna* declaramos su método constructor el cual recibe como parámetro el factor del cual depende el cálculo del salario neto. Se realiza una evaluación a través de tres condicionales tal como se estableció en el enunciado. Es aquí en donde se acude a las variables *SalNeto* y *Salario* sin necesidad de instanciarlas, es decir, sin asociarlas con una instancia de clase.

```
private class ClaseInterna {
    private ClaseInterna(int Factor) {
        if (Factor < 20)
            SalNeto = Salario + 100000;
    }
```

```
    if (Factor == 20)
        SalNeto = Salario + 300000;
    if (Factor > 20)
        SalNeto = Salario + 500000;
}
}
```

Se cierra la clase interna *ClaseInterna* y se cierra también la clase *ClaseI* que contiene a la clase *ClaseInterna*. Aquí termina la declaración de la clase como tal. Ahora vamos a construir el programa como tal que sea el que aproveche y capitalice las ventajas de la clase interna. Para ello seguimos trabajando con el paquete *ClasesInternas* y declaramos una clase que hemos llamado *ClaseExterna*.

```
package ClasesInternas;

public class ClaseExterna {
```

Declaramos el método principal con sus elementos característicos. Seguidamente declaramos una instancia de la clase *ClaseI* que hemos llamado *P1* y, a través de su constructor, enviamos los parámetros respectivos. Después declaramos otra instancia que hemos llamado *P2* y también le enviamos los parámetros a través de su constructor. El objeto *P1* tiene un factor igual a 15 y el objeto *P2* tiene un factor igual a 40.

```
public static void main(String[ ] args) {
    ClaseI P1 = new ClaseI(111, "Alberto Perez", 1000000, 15);
    ClaseI P2 = new ClaseI(222, "Jorge Ramirez", 2000000, 40);
```

Finalmente presentamos por consola los datos de cada una de las variables (atributos) de los objetos *ClaseI* declarados.

```
System.out.println("\nCódigo:\t" + P1.Codigo);
System.out.println("Nombre:\t" + P1.Nombre);
System.out.println("Salario:\t" + P1.Salario);
System.out.println("Neto:\t" + P1.SalNeto);
```

```
System.out.println("\nCódigo:\t" + P2.Codigo);
```

```
        System.out.println("Nombre:\t" + P2.Nombre);
        System.out.println("Salario:\t" + P2.Salario);
        System.out.println("Neto:\t" + P2.SalNeto);
    }
}
```

El programa completo se presenta a continuación:

```
// DECLARACIÓN DE CLASE

package ClasesInternas;

public class ClaseI {
    int Codigo;
    String Nombre;
    double Salario;
    double SalNeto;

    public ClaseI(int Codigo, String Nombre, double Salario, int Factor) {
        this.Codigo = Codigo;
        this.Nombre = Nombre;
        this.Salario = Salario;
        CalculaSalNeto(Factor);
    }

    public double CalculaSalNeto(int Factor) {
        ClaseInterna V1 = new ClaseInterna(Factor);
        return 1;
    }

    private class ClaseInterna {
        private ClaseInterna(int Factor) {
            if (Factor < 20)
                SalNeto = Salario + 100000;
            if (Factor == 20)
                SalNeto = Salario + 300000;
            if (Factor > 20)
                SalNeto = Salario + 500000;
        }
    }
}
```

}

// PROGRAMA JAVA

```
package ClasesInternas;

public class ClaseExterna {
    public static void main(String[] args) {

        ClaseI P1 = new ClaseI(111, "Alberto Perez", 1000000, 15);
        ClaseI P2 = new ClaseI(222, "Jorge Ramirez", 2000000, 40);

        System.out.println("\nCódigo:\t" + P1.Codigo);
        System.out.println("Nombre:\t" + P1.Nombre);
        System.out.println("Salario:\t" + P1.Salario);
        System.out.println("Neto:\t" + P1.SalNeto);

        System.out.println("\nCódigo:\t" + P2.Codigo);
        System.out.println("Nombre:\t" + P2.Nombre);
        System.out.println("Salario:\t" + P2.Salario);
        System.out.println("Neto:\t" + P2.SalNeto);

    }
}
```

Al ejecutar este programa con los datos suministrados, el resultado sería el siguiente:

Código: 111  
Nombre: Alberto Perez  
Salario: 100000.0  
Neto: 110000.0

Código: 222  
Nombre: Jorge Ramirez  
Salario: 200000.0  
Neto: 250000.0

### 22.3 Ejercicios Propuestos

- Construir un programa que permita realizar descuentos diferenciales a estudiantes de un colegio dependiendo del estrato social donde vive, aprovechando las ventajas de las clases internas
- Construir un programa que permita realizar descuentos diferenciales a los productos de un supermercado dependiendo del precio de éstos, usando el concepto de clases internas
- Construir un programa que permita abonarles una bonificación de puntos a equipos de futbol dependiendo del puesto que ocupen en la tabla de clasificación, usando la facilidad de las clases internas
- Construir un programa que permita calcular el impuesto a pagar por algunos productos de un almacén de cadena dependiendo de su uso (para ello codificar por lo menos cinco tipos de usos), aprovechando las bondades de las clases internas
- Construir un programa que permita calcular el impuesto de una casa de vivienda dependiendo del estrato en donde esté ubicada, usando la facilidad de las clases internas



# Lección

# 23

# Interfaces

# de usuario



### 23.1 Concepto General

Se definen como interfaces de usuario aquellas utilidades que tiene un lenguaje de programación para mejorar sustancialmente la estética de presentación de los datos para el usuario. Una Interfaz de usuario, en términos generales, es el conjunto de elementos que permiten que se visualicen los datos para el usuario. Debe anotarse que de la estética como se presenten los datos en pantalla, asimismo podrán ser éstos muy entendibles para el usuario concibiendo al usuario como la persona que en un momento dado utiliza uno de nuestros programas. Teóricamente, de la claridad con que los datos se presenten al usuario, asimismo será posible reducir los errores de ese mismo usuario y posibilitar que la información desplegada en una pantalla sea de verdadera utilidad.

Las interfaces de usuario también se conocen como *Ventanas*, *Marcos* o *Frames*. Por esta razón en este libro se usará cualquiera de los tres términos indistintamente. Una ventana es, por lo tanto, un marco que permite presentarle al usuario la información que éste requiere, de una forma muy clara y muy estética. Es de anotar que cuando se crea una ventana, instantáneamente éstas no son visibles. Esto quiere decir que no es suficiente con crear una ventana sino que además debe visualizarse pues la creación de la ventana se hace en memoria pero la visualización de la misma se hace sobre la pantalla que es lo que, en realidad, el usuario podrá observar. Para visualizar una ventana, que ya ha sido creada, se utiliza el método *setVisible()* al cual se le debe enviar un parámetro booleano igual a **true** indicando que efectivamente se muestre la ventana en pantalla.

De la misma forma, cuando se crea una ventana debe conferírsele un tamaño que se ajuste a la pantalla. El lenguaje de programación Java contiene métodos que permiten detectar la resolución de la pantalla y, de esa forma, garantizar que las ventanas que se utilicen encajen plenamente en la pantalla cualquiera que fuere su resolución. El tamaño de una ventana depende de dos valores: el ancho y el alto. Ambos valores deben expresarse en *pixels*, que es el patrón de medida que se usa en la pantalla. Para darle el tamaño a la ventana se utiliza el método *setSize()* que recibe como parámetros, precisamente, el ancho y el alto, en ese orden.

Es importante tener en cuenta que una vez que se ha creado una ventana y que se ha visualizado, es necesario (y profundamente útil) especificar la acción a realizar cuando la ventana se cierre. Debe tener en cuenta, querido lector, que las ventanas cuando se crean automáticamente tienen botones de *Minimizar*, *Maximizar* y *Cerrar* y que estos botones funcionan correctamente. Para especificar qué se debe hacer cuando se cierre la ventana, vamos a acudir al método *setDefaultCloseOperation( )* que requiere, como parámetro, un valor entero reflejado en variables figurativas que se relacionan a continuación y que han sido tomadas de Java API:

Variable Figurativa	Descripción	Definido en
DO NOTHING ON CLOSE	Indica que no se haga nada para lo cual se requiere un programa que maneje la operación.	WindowConstants
HIDE ON CLOSE	Automáticamente oculta la ventana después de invocar algún objeto que actúe como un “oidor” de eventos (WindowListener). Está definido en WindowConstants	WindowConstants
DISPOSE ON CLOSE	Automáticamente oculta y dispone de la ventana después de invocar un objeto tipo WindowListener	WindowConstants
EXIT ON CLOSE	Finaliza la aplicación acudiendo al método de salida (exit) del sistema. Este método solo puede usarse en aplicaciones	JFrame

En el primer ejemplo que realizaremos vamos a acudir EXIT\_ON\_CLOSE indicando que se finalice y se cierre la aplicación.

## 23.2 Creación de una ventana

Vamos a analizar el código Java para crear una ventana que acertadamente se conoce también como Marco puesto que esta ventana es solo el marco en donde se ubicarán otros controles para que dicha ventana sea útil dado que, al final, veremos que esta ventana (así como está) es absolutamente inútil. Esta vez vamos a trabajar con un paquete que hemos llamado *Ventanas*. Después de esto vamos a importar todos los métodos de la clase extendida de Java (*javax*) llamada *swing*.

```
package Ventanas;  
import javax.swing.*;
```

Codificamos a continuación la clase *Ventana1* que contendrá el método *main()* lo cual lo convierte en un programa Java.

```
public class Ventana1 {  
    public static void main(String[] args) {
```

Seguidamente se declara un objeto de tipo *Frame1* –que es una clase que declaramos más abajo- y cuyo constructor invocamos para que se cree apropiadamente la instancia requerida.

```
    Frame1 V1 = new Frame1();
```

A continuación se acude al método *setVisible* (con su parámetro *true*) para hacer visible la ventana. Después de esto, especificamos la acción a seguir para cuando dicha ventana sea cerrada.

```
    V1.setVisible(true);  
    V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}
```

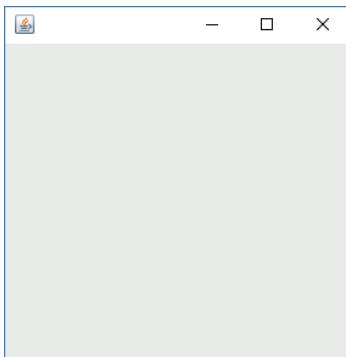
Finalizada la clase principal, creamos la clase *Frame1* que hereda de la clase *JFrame* (que es una clase predefinida de Java que contiene los recursos para crear interfaces gráficas de usuario apropiadas). En esta clase todo lo que vamos a hacer es codificar su método constructor, dentro del cual especificamos el tamaño de la ventana a través del método *setSize* que, en este caso, recibe los parámetros 300 y 300 indicando que será una ventana que tiene 300 pixels de ancho y 300 pixels de alto.

```
class Frame1 extends JFrame {  
  
    public Frame1() {  
        setSize(300,300);  
    }  
}
```

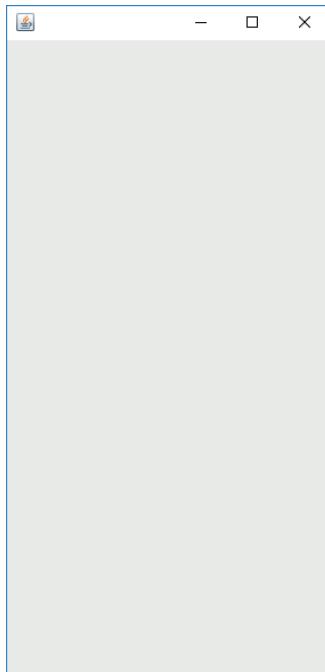
Es de anotar que, dentro del método constructor de la clase *Frame1*, también se hubiera podido utilizar el método *setVisible* para hacer visible la ventana y el método *setDefaultCloseOperation* para especificar la acción a seguir una vez se cierre la ventana. El código completo del programa se presenta a continuación:

```
package Ventanas;  
import javax.swing.*;  
  
public class Ventana1 {  
    public static void main(String[] args) {  
        Frame1 V1 = new Frame1();  
        V1.setVisible(true);  
        V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}  
  
class Frame1 extends JFrame {  
    public Frame1() {  
        setSize(300,300);  
    }  
}
```

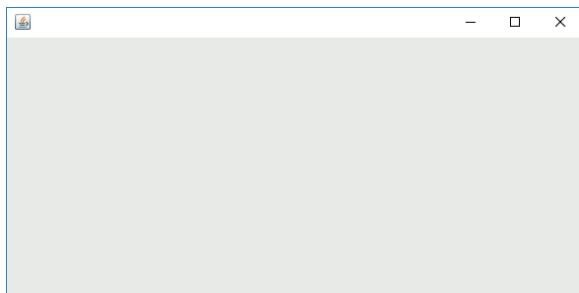
Al ejecutar este programa (en donde se visualiza una ventana de 300 x 300) el resultado es el siguiente:



Nótese que es una ventana que tiene un borde estándar, tres botones en la parte superior derecha que permiten Minimizar, Maximizar y Cerrar y un cuerpo de ventana que no hace nada pues no hemos utilizado los otros recursos que se usan sobre una ventana creada y visualizada. Por eso es que, al inicio de esta lección, decíamos que esta ventana, tal como está, no sirve para nada. Si ejecutáramos el programa pero especificando que la ventana tuviera 300 de ancho y 600 de alto, entonces el resultado sería similar al que se presenta a continuación.



Si se quisiera visualizar una ventana de 600 pixels de ancho y 300 de alto entonces, en pantalla, se mostraría algo similar a lo que se presenta seguidamente:



### 23.3 Cambiando atributos de una ventana

Como ha podido observar hasta el momento, la ventana que hemos construido aún no sirve para nada pues es solo el marco sobre el cual podemos trabajar pero entender cómo funciona y cuáles características podemos intervenir es el primer paso para que sus programas sean notoriamente estéticos, mas agradables y entendibles para el usuario.

En nuestro ejemplo, construir la clase *Frame1* (que hereda de la clase *JFrame*) es la clave para que las ventanas aparezcan tal y cual como nosotros queramos. Para ello debemos entender que la siguiente secuencia corresponde a las clases e interfaces que se van heredando progresivamente hasta llegar a *JFrame* comenzando en la superclase cósmica *Object*. Por ahora no se preocupe por entender detalladamente cada uno de los nombres que aparecen a continuación pero si quiere ampliar un poco más su conocimiento acerca de cada uno de ellos entonces *Java API* le permitirá profundizar hasta donde usted quiera conocerlas.

Object → Component → Container → Window → Frame → JFrame

Aunque en el ejemplo hemos codificado el método constructor *Frame1* asignándole valores a sus atributos desde adentro de la clase, vale la pena recordar que dichos valores también se pueden pasar como parámetros de manera que tengamos una clase un poco mas amplia que nos sirva para desarrollar programas con diferentes tipos de ventanas. En este sentido, también vale la pena recordar *JOptionPane* que es un recurso que se tiene

para recibir datos por parte del usuario y para escribir mensajes, ambos a través del mecanismo de ventanas.

La clase *Component* pone a nuestra disposición dos métodos que son de gran utilidad. Uno de ellos ya lo vimos en el primer ejemplo con ventanas y es el método *setSize* que permite establecer las dimensiones de la ventana (o sea, la anchura y la altura de la ventana). Esta clase también nos provee el método *setLocation* que nos permite ubicar la ventana en algún punto de la pantalla pues no es lo mismo que una ventana aparezca en la esquina superior izquierda de la pantalla a que aparezca hacia el centro o hacia el extremo derecho. Es posible que estos conceptos, que son puramente estéticos, sean importantes en el desarrollo de sus programas.

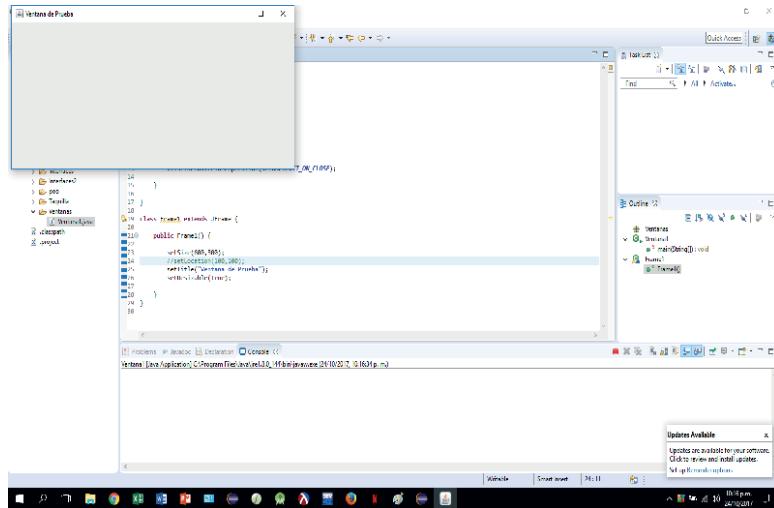
Para utilizar el método *setLocation* todo lo que debemos tener en cuenta es que la esquina superior izquierda de la pantalla corresponde al punto (0,0) de referencia, es decir, es el punto de inicio desde donde se empiezan a contar pixels para ubicar nuestras ventanas apropiadamente. Cuando no especificamos la ubicación de una ventana, Java las despliega a partir del punto (0,0). Cuando le especificamos estos valores, entonces Java las ubica en el lugar correspondiente. En el ejemplo que ya hemos desarrollado y que se presenta a continuación:

```
package Ventanas;
import javax.swing.*;
public class Ventana1 {

    public static void main(String[] args) {
        Frame1 V1 = new Frame1();
        V1.setVisible(true);
        V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    class Frame1 extends JFrame {
        public Frame1() {
            setSize(600,300);
        }
    }
}
```

La ventana se despliega en el extremo superior izquierdo de la pantalla (tal como se muestra en la siguiente imagen) debido a que no hemos especificado ninguna coordenada para su ubicación.



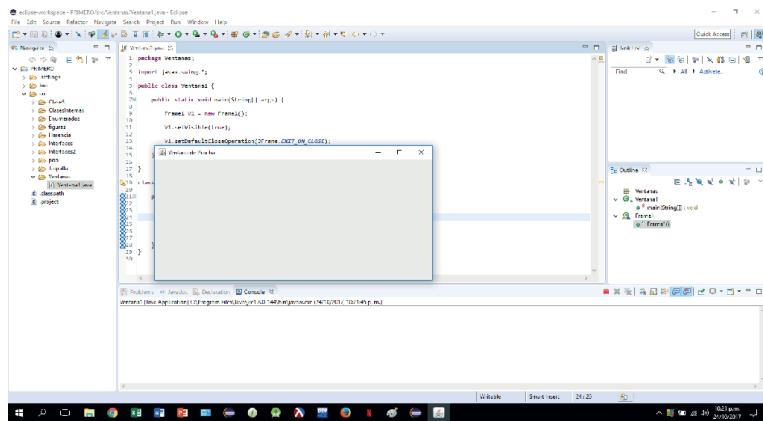
Ahora bien, si establecemos unas coordenadas para ello usando el método *setLocation*, entonces Java la ubica en donde corresponda. Si el código fuera el siguiente:

```
package Ventanas;

import javax.swing.*;
public class Ventana1 {
    public static void main(String[] args) {
        Frame1 V1 = new Frame1();
        V1.setVisible(true);
        V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

class Frame1 extends JFrame {
    public Frame1() {
        setSize(600,300);
        setLocation(300,300); ← Método para ubicar la ventana
    }
}
```

Se le estaría indicando a Java que despliegue la ventana a partir del punto (300,300), es decir, 300 pixels hacia la derecha y 300 pixels hacia abajo. Observe que es la misma ventana pero ubicada un poco mas a la derecha y un poco mas abajo. El resultado sería algo como lo siguiente:



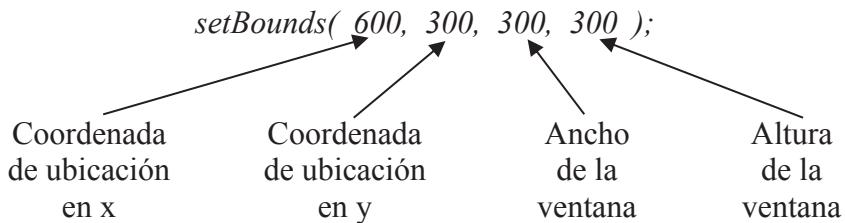
Los métodos *setSize* y  *setLocation* nos permiten dimensionar la ventana y ubicarla dentro de la pantalla a nuestro gusto. El método *setBounds* permite hacer ambas cosas al tiempo, es decir, le podemos especificar la ubicación de la ventana en la pantalla y las dimensiones de la misma. Para ello todo lo que tenemos que hacer es invocar el método y enviarle como parámetros las coordenadas (x,y) de su ubicación en pantalla y las dimensiones (ancho y largo) de la ventana. De acuerdo a lo explicado será lo mismo que la clase Frame1 esté escrita como se presenta a continuación:

```
class Frame1 extends JFrame {
    public Frame1() {
        setSize(600,300);
        setLocation(300,300); ← Método para ubicar la ventana
    }
}
```

Que si se escribiera usando el método *setBounds* de la siguiente forma:

```
class Frame1 extends JFrame {
    public Frame1() {
        setBounds(600,300,300,300);
    }
}
```

Pues al utilizar este método le estamos especificando a Java lo siguiente:

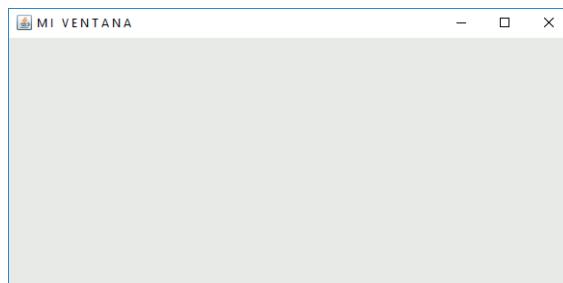


La clase *Frame* nos provee un método que permite escribirle un título a la ventana y se trata del método *setTitle* para lo cual todo lo que tenemos que hacer es invocar el método y enviarle como parámetro una cadena que es el título que aparecerá en la parte superior de la ventana.

Si le incorporamos a la clase *Frame1* el llamado al método *setTitle* y le enviamos la cadena “M I V E N T A N A” tal como sigue:

```
class Frame1 extends JFrame {  
  
    public Frame1() {  
        setSize(600,300);  
        setLocation(300,300);  
        setTitle("M I V E N T A N A");  
    }  
}
```

Entonces aparecerá en pantalla una ventana como la siguiente:



Usted habrá notado que cuando se construye una ventana, a partir de código Java, una vez ésta ha aparecido en la pantalla se puede, valiéndose del mouse, “estirar” o “contraer” la ventana como nosotros queramos, es decir, se puede ubicar el puntero del mouse sobre uno de los lados de la ventana (o sobre una de las esquinas) y, haciendo click en el botón izquierdo, podemos redimensionar la ventana. Es posible que en algunos programas, por razones propias del objetivo del mismo, necesitemos que esta posibilidad no la tenga el usuario y que no tenga la posibilidad de redimensionar la ventana a su propio gusto. La clase *JFrame* cuenta con el método *setResizable* cuyo parámetro podría ser *true* o *false*. Si es *true*, significará que el usuario tendrá la posibilidad de redimensionar la ventana de nuestro programa como el quiera. Si es *false*, entonces dicha posibilidad queda anulada, es decir, no podrá redimensionar la ventana a su propio gusto.

Ahora bien, la clase *JFrame* provee un método llamado *setExtendedState* que permite especificar si queremos que nuestra ventana (ubicando el código desde el método constructor) aparezca con alguno de los siguientes estados.

- *JFrame.NORMAL*: Inicializa la ventana en estado *Normal*, es decir, asumiendo las especificaciones que se le hayan indicado o asumiendo las especificaciones que tiene Java por defecto
- *JFrame.ICONIFIED*: Inicializa la ventana en estado *Minimizado*, es decir, la ventana aparece en sus mínimas dimensiones posibles
- *JFrame.MAXIMIZED\_HORIZ*: Inicializa la ventana en estado *Maximizado Horizontalmente*, es decir, del ancho de toda la pantalla
- *JFrame.MAXIMIZED\_VERT*: Inicializa la ventana en estado *Maximizado Verticalmente*, es decir, de la altura de toda la pantalla
- *JFrame.MAXIMIZED\_BOTH*: Inicializa la ventana en estado *Maximizado* en ambos sentidos, es decir, la ventana queda del tamaño de TODA la pantalla

Todo lo que tenemos que hacer es invocar el método *setExtendedState* y, entre paréntesis, escribirle algunos de los valores figurativos que se acabaron de explicar, por ejemplo:

*setExtenderState (JFrame.MAXIMIZED\_BOTH);*

Recuerde que en Java escribir en mayúscular no es lo mismo que escribir en minúsculas.

### 23.4 Escribiendo texto sobre una ventana

Como usted se ha de imaginar, a partir del contacto que se tiene con el mundo moderno de los computadores personales y dispositivos similares, una ventana es la base para que los programas tengan una apariencia estética suficientemente entendible y fácil de manejar para cualquier usuario (en términos generales). Sobre la ventana podemos ubicar texto, botones, controles, títulos, opciones, listas desplegables, botones de radio, cajas de chequeo y muchas otras facilidades que permiten que nuestros programas no solamente logren el objetivo de ser entendibles sino que sean fáciles de manejar. En este sentido debemos tener en cuenta que si un programa o una aplicación es muy difícil de manejar, posiblemente el usuario no lo llegue a utilizar nunca y lo que se quiere con un programa es que haya usuarios que lo utilicen así como lo que se requiere con un libro es que haya lectores que lo lean.

En el lenguaje de programación Java, siempre que se quiera construir una ventana se requiere construir una clase que herede de JFrame puesto que esta contiene (o ha heredado a su vez) los métodos que permiten que la ventana tenga el aspecto que conocemos y las facilidades para que en ella se puedan ubicar apropiadamente todos los controles que, técnicamente, se conocen como controles GUI que significa Graphics User Interface, es decir, Interfaz Gráfica de Usuario.

Para concebir apropiadamente lo que es una ventana, como la hemos codificado hasta ahora, pensemos en que se solamente un *marco* sobre el cual podemos hacer muchas cosas. Haga de cuenta como si la definición de la ventana que hemos realizado es como si hubiéramos construido un escritorio vacío y estamos sentados frente a él. Nosotros podríamos hacer dibujos sobre el escritorio, podríamos escribir texto directamente en el escritorio y podríamos hacer muchas cosas en él pero no es lo que normalmente hacemos puesto que para poder dibujos, escribir texto o para pintar requerimos de una hoja en blanco. De una parte hacerlo sobre el escritorio directamente, malograría el escritorio (aunque posiblemente eso no es lo que piensen los decoradores) y por otro lado porque si hacemos un dibujo en el escritorio directamente, dicho dibujo tendría que quedarse allí y no podríamos manejarlo con la misma facilidad con que lo haríamos si lo hicéramos sobre una hoja. Pues bien, esa hoja a la que hacemos

referencia es lo que en Java llamaremos un *Panel* que no es más que una especie de transparencia que ponemos sobre la ventana para poder escribir, dibujar, pintar, poner controles y hacer todo lo que nuestros programas necesiten. Esto también implicaría que sobre una transparencia (o *Panel*) podríamos poner otra transparencia y así sucesivamente, dependiendo de las necesidades que tengamos en consonancia con el programa que estemos construyendo. Esto significa que la ventana (o *Marco*) que hemos construido a través de heredar *JFrame* es solo un soporte sobre el cual ponemos la transparencia que necesitemos y, ahí sí, podemos hacer todo lo que queramos. Sobra decir que el *Panel* se coloca por encima de la ventana construida.

Ya hemos visto que para crear una ventana debemos construir una clase que herede de la clase *JFrame*. Pues bien, para construir un panel debemos, de la misma forma, construir una clase que herede de la clase *JPanel* que a su vez hereda de *JComponent* y que, al igual que *JFrame*, hereda de *Container*, *Component* y la super clase *Object*, sucesivamente y ascendenteamente. Para construir una ventana y escribir sobre ella un texto debemos proceder de la siguiente forma. Lo primero que hacemos, si es el caso como sucede en los ejemplos de este libro, es incluir el paquete con el cual se esté trabajando. En este caso, por razones didácticas, trabajaremos en el paquete Ventanas que es el paquete de usuario que se ha creado para desarrollar este ejemplo.

```
package Ventanas;
```

Seguidamente se importan los paquetes y bibliotecas requeridos para lograr el objetivo de poner un texto sobre una ventana. *Swing* es una biblioteca gráfica construida para darle soporte al lenguaje de programación Java que incluye facilidades tipo GUI como botones, listas desplegables, tablas, cajas de texto y otros. El paquete Abstract Window Toolkit (*awt*) contiene un conjunto de recursos que permiten dibujar, pintar, escribir texto, poner imágenes y otras facilidades a través de métodos que tiene previsto para tal fin. En su más simple definición, *awt* es la que representa el contexto gráfico de un componente o una imagen.

```
import javax.swing.*;
import java.awt.*;
```

Lo primero que vamos a hacer en este programa es crear la transparencia sobre la cual vamos a escribir el texto. Para ello creamos una clase que hereda métodos y atributos de *JPanel*.

```
class Panel1 extends JPanel {
```

La clase *JPanel* tiene definido un método que es obligatorio implementar para todas las clases que la hereden y es el método *paintComponent* que, como su nombre lo indica, permite “pintar componentes”, es decir, permite que en la transparencia podamos poner los controles y la partes que requiera nuestro programa. Este método recibe un parámetro (obligatorio) de tipo *Graphics* que por ahora tomaremos como dogma de fe pues mas adelante lo explicaremos con más detalle.

```
public void paintComponent(Graphics g) {
```

El método *paintComponent* hace, entre otras cosas, labores como preparar la transparencia y acondicionarla para que sobre ella se puedan hacer los dibujos, escribir los textos o poner los controles que necesitamos. Por esta razón se requiere que llamemos a la super clase *JPanel* (pues es de la cual se hereda este método) y le indiquemos que el método haga la labor correspondiente.

Al igual que en la definición anterior, el detalle de lo que hace el método *paintComponent* será tema de profundización mas adelante. Para que se haga efectivo este llamado, se requiere que se envíe, como parámetro, la variable tipo *Graphics* que se declaró previamente.

```
super.paintComponent(g);
```

Ahora sí, ya tenemos la forma de definir el tipo de letra que necesitamos, o que queremos utilizar para escribir el texto. De la misma forma podemos definir la característica de esta letra y el tamaño. En el ejemplo, vamos a utilizar letra Times New Roman, con característica (PLAIN) o sea completamente sencilla y de un tamaño de 20 puntos. Estos parámetros se lo indicamos a Java a través del método *setFont* que acude a una instancia de la clase *Font*. Puede consultar en Java API los diferentes valores posibles tanto para el tipo de letra como para la característica de la misma que uno quiere aplicar en un texto.

```
g.setFont(new Font("TimesRoman", Font.PLAIN, 20));
```

Seguidamente escogemos el color que queremos para nuestra letra a través del método setColor. Conviene recordar que los métodos setFont, setColor y drawString son métodos propios de la clase Graphics, razón por la cual para ser utilizados deben invocarse anteponiéndose en el nombre de la instancia respectiva.

Es de anotar que Java tiene predefinidos algunos colores aunque a través de la clase Color se puede definir el color que se quiera acudiendo al estándar RGB (Red - Green – Blue ). Los colores que tiene el lenguaje de programación Java son los siguientes:

Color.white	Color.black	Color.yellow	Color.lightGray
Color.red	Color.green	Color.gray	Color.pink
Color.magenta	Color.darkGray	Color.orange	Color.cyan
Color.blue			

Para el caso del ejemplo se ha escogido el color negro para el texto, por lo tanto, se invoca el método *setColor* (para establecer el color) y se le envía como parámetro *Color.black*.

```
g.setColor(Color.black);
```

Con las características establecidas, ahora si se puede definir cuál es el texto que se quiere mostrar y en qué parte de la ventana se quiere que aparezca. Vale la pena tener en cuenta que las coordenadas para ubicar el texto dentro de la ventana se cuentan en pixels a partir del extremo superior izquierdo útil de dicha ventana.

En este caso que tenemos coordenada en *x* igual a 100 y coordenada en *y* igual a 30 significa que el texto estará escrito a 100 pixels del extremo izquierdo dde la ventana y a 30 pixels del extremo superior de la misma. Entre comillas dobles se escribe el texto que queremos que aparezca. Con esto finalizamos la definición de la transparencia que queremos poner sobre la ventana para que se vea que el texto quedó encima de ella.

```
g.drawString("Primer Texto", 100, 30);  
}  
}
```

Ahora, a continuación, creamos la clase que le da características a la ventana y que hereda de la clase *JFrame*. Para ello procedemos como lo hemos hecho hasta ahora.

```
class Frame1 extends JFrame {
```

Como hemos llamado *Frame1* a esta clase, ese es el nombre que debe tener su método constructor. Este método recibe 6 parámetros que corresponden a las características básicas de una ventana. Debe aclararse que no es absolutamente necesario que el método constructor tenga que codificarse recibiendo parámetros. Esta vez se ha querido hacer así.

Los parámetros del método *Frame1* corresponden al ancho de la ventana, a la altura de la ventana, a las coordenadas *x* y *y* a partir de las cuales debe mostrarse la ventana en la pantalla, al título que irá sobre la parte superior de la ventana y al valor booleano que autoriza que la ventana pueda ser redimensionada por parte del usuario o no (a través del uso del puntero del mouse en los bordes y extremos de la ventana).

```
public Frame1(int Ancho, int Altura, int Coordx, int Coordy,  
String Titulo, boolean ReDim) {
```

Con los valores recibidos como parámetros del método constructor se codifican las pautas que caracterizan a la ventana. En primer lugar la hacemos visible a través del método *setVisible* al cual le enviamos el valor booleano **true** indicando que efectivamente la queremos ver en pantalla.

```
setVisible(true);
```

A continuación, establecemos las dimensiones de la ventana utilizando los valores de Ancho y Altura que llegaron como parámetros del método constructor.

```
setSize(Ancho, Altura);
```

Ubicamos la ventana dentro de la pantalla a través de las coordenadas *x* y *y* que se han recibido en los parámetros *Coordx* y *Coordy*. Recuerde que estos valores corresponden a la distancia, en pixels, desde el lado izquierdo de la pantalla hasta el lado izquierdo de la ventana (coordenada en *x*) y

desde el lado superior de la pantalla hasta el lado superior de la ventana (coordenada en y).

```
setLocation( Coordx, Coordy );
```

Se le coloca el título que recibimos como parámetro y que va ubicado en la parte superior de la ventana.

```
setTitle(Titulo);
```

Se establece que la ventana puede ser redimensionada por el mismo usuario (usando el puntero del mouse) y teniendo en cuenta el valor booleano que se recibió como parte de los parámetros del método constructor.

```
setResizable(ReDim);
```

Con todas las características de la ventana, ahora vamos “poner” encima de ella una instancia de la clase *Panell* (que fue la que heredó métodos y atributos de la clase *JPanel*) y que correspondería, en lenguaje metafórico, a la transparencia que se pone encima de la ventana para que parezca como si se hubiera escrito el texto sobre la ventana.

Para ello declaramos un objeto tipo *Panell*. Tenga en cuenta que cuando no se codifica un método constructor de manera específica, Java asume un constructor por defecto, como en este caso. Esto quiere decir que toda clase siempre tiene un método constructor, sea que lo hayamos codificado explícitamente o que no lo hayamos codificado. Finalmente, tal como haríamos con una transparencia sobre el escritorio, “ponemos” la transparencia sobre la ventana a través del método *add* y con ello finalizamos la declaración de la clase *Frame1* que hereda métodos y atributos de *JFrame*.

```
Panell P = new Panell();
add(P);
}
```

Solo resta construir la clase que contiene el método *main* para ver la ejecución del programa. En esta clase simplemente declaramos un objeto de tipo *Frame1* con los parámetros que requiere su método constructor y

definimos también la acción a seguir en el momento en que se cierre la ventana que, en este caso, corresponde a finalizar el programa (EXIT\_ON\_CLOSE).

```
public class VentanaI {  
  
    public static void main(String[] args) {  
  
        FrameI VI = new FrameI(500, 300, 400, 200, "Mi Ventana", true);  
        VI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

El código completo de este programa se presenta a continuación:

```
package Ventanas;  
  
import javax.swing.*;  
import java.awt.*;  
  
// CLASE PARA LA TRASPARENCIA  
class Panel1 extends JPanel {  
  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.setFont(new Font("TimesRoman", Font.PLAIN, 20));  
        g.setColor(Color.black);  
        g.drawString("Primer Texto", 50, 50);  
    }  
}  
  
// CLASE PARA LA VENTANA  
class FrameI extends JFrame {  
  
    public FrameI(int Ancho, int Altura, int Coordx, int Coordy,  
                 String Titulo, boolean ReDim) {  
        setVisible(true);  
        setSize(Ancho, Altura);  
        setLocation(Coordx, Coordy);  
    }  
}
```

```
setTitle(Titulo);
setResizable(ReDim);
Panel1 P = new Panel1();
add(P);
}
}

// CLASE PARA LA EJECUCIÓN DEL PROGRAMA
public class Ventana1 {

    public static void main(String[] args) {
Frame1 V1 = new Frame1(500, 300, 400, 200, "Mi Ventana", true);
    V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Finalmente, si ejecutamos el programa, el resultado es una ventana con el siguiente aspecto.



Como puede observarse, se cumplen en ella todos los parámetros y propósitos que se codificaron. El texto aparece con el tipo de letra escogido, en el tamaño seleccionado y en la ubicación definida.

### 23.5 Dibujando figuras y líneas

Como usted ha de suponer, el procedimiento para dibujar figuras geométricas sobre una ventana es muy similar al que se utilizó para dibujar un texto. Esto quiere decir que, lo conveniente desde todo punto de vista, es que no se dibuje directamente sobre la ventana (como no sería

conveniente hacer un dibujo directamente sobre el escritorio, primero ponemos una hoja para empezar a dibujar) sino que se dibuje sobre una transparencia que se coloca encima de ella para facilitar el acceso a los recursos que proporciona la clase *Graphics* y la reutilización de la ventana para otros propósitos. Recuerde que para conocer un poco mas de los métodos que brinda Java en esta parte gráfica, simplemente debe abrir su navegador, iniciar un buscador y escribir “Java API” e inmediatamente, el primer enlace que le aparezca, lo llevará al repositorio académico de Java en donde explican cada una de las clases, interfaces, recursos y métodos con ejemplos, fundamentación teórica y descripción en toda su plenitud.

En el ejemplo que haremos a continuación solamente vamos a dibujar un rectángulo convencional con esquinas cuadradas, una línea y un rectángulo con esquinas redondeadas para que usted vea cómo se acude a los métodos que proporciona la clase *Graphics*. Debo aclarar que esto le dará una idea de cómo hacerlo y le permitirá entender, igualmente, cómo acudir a librerías más modernas, como por ejemplo *Rectangle2D*, que permite cambiar otros parámetros y posibilita mucho mas manejo en el dibujo de figuras geométricas. Para explicar cómo dibujar, teniendo en cuenta que los fundamentos ya se dieron en el numeral anterior, se explicarán los cambios que se hacen sobre la clase *Panel1* que heredaba a la clase *JPanel* y que es la que permitía codificar nuestras necesidades de graficación. El resto del programa, que presentaremos al final, es completamente similar al que hemos venido trabajando.

Tal como lo hemos explicado, para construir una transparencia sobre la cual podamos dibujar debemos construir una clase que herede atributos y métodos de la clase *JPanel*. En este caso esa clase la hemos llamado *Panel1*. Seguidamente lo primero que debemos hacer es reescribir el método *paintComponent()* que es el método que permite, como lo indica su nombre, pintar los componentes gráficos. Este método recibe como argumento un objeto de tipo *Graphics*.

```
class Panel1 extends JPanel {  
  
    public void paintComponent(Graphics g) {
```

Tal como ya lo hemos explicado, lo primero que hacemos es llamar a la clase padre para que el método *paintComponent()* prepare todo lo que corresponde para poder realizar los dibujos que se requieran.

```
super.paintComponent(g);
```

Seguidamente, y solo para sintonizarnos con el ejemplo anterior, definimos el tipo de letra, su característica y su tamaño.

```
g.setFont(new Font("TimesRoman", Font.PLAIN, 20));
```

De la misma forma se establece el color de la letra y se especifica el texto que queremos que aparezca dentro de la transparencia que se pondrá encima de la ventana indicando las coordenadas a partir de las cuales se espera visualizar.

```
g.setColor(Color.black);
g.drawString("Figuras Geométricas", 30, 50);
```

Ahora bien, si queremos dibujar un rectángulo convencional con esquinas cuadradas, todo lo que tenemos que hacer es invocar el método *drawRect* (que es un método definido en la clase *Graphics*) cuyos parámetros son las coordenadas de la esquina superior izquierda del rectángulo en relación con la ventana y las dimensiones del rectángulo (ancho y altura).

```
g.drawRect(50, 100, 100, 100);
```

De la misma manera, si queremos dibujar una línea recta invocamos el método *drawLine* y especificamos las coordenadas del punto donde comienza la recta y las coordenadas del punto donde finaliza.

```
g.drawLine(300, 100, 400, 120);
```

Finalmente, para este ejemplo, si queremos dibujar un rectángulo con esquinas redondeadas se indican sus parámetros: coordenadas de inicio esquina superior izquierda del rectángulo, dimensiones del rectángulo y especificaciones del “redondeo” de las esquinas, es decir, caracterización angular de las mismas.

```
g.drawRoundRect(180, 100, 100, 100, 20, 20);
}
```

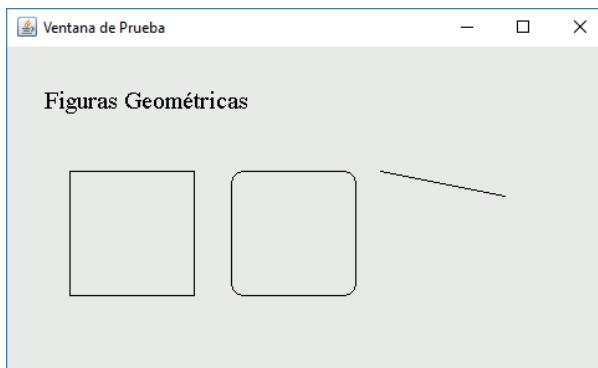
Si se quieren conocer más herramientas que provee *Graphics* para dibujar figuras geométricas, simplemente tenemos que acudir a Java API tal como

se ha indicado al inicio de este numeral y allí podremos conocer no solo el nombre de otros métodos sino también los parámetros que se requieren para que estos métodos funcionen correctamente. El código completo del programa se presenta a continuación.

```
package Ventanas;  
import javax.swing.*;  
import java.awt.*;  
  
class Panel1 extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.setFont(new Font("TimesRoman", Font.PLAIN, 20));  
        g.setColor(Color.black);  
        g.drawString("Figuras Geométricas", 30, 50);  
  
        g.drawRect(50, 100, 100, 100);  
        g.drawLine(300, 100, 400, 120);  
        g.drawRoundRect(180, 100, 100, 100, 20, 20);  
    }  
}  
  
class Frame1 extends JFrame {  
    public Frame1(int Ancho, int Altura, int Coordx, int Coordy,  
                 String Titulo, boolean ReDim) {  
        setVisible(true);  
        setSize(Ancho, Altura);  
        setLocation(Coordx, Coordy);  
        setTitle(Titulo);  
        setResizable(ReDim);  
        Panel1 P = new Panel1();  
        add(P);  
    }  
}  
  
public class Ventana1 {  
    public static void main(String[] args) {  
  
        Frame1 V1 = new Frame1(500,300,400,200,"Ventana de Prueba", true);  
        V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

```
}
```

Al ejecutar el programa, como era de esperarse, aparece una ventana con los títulos y dimensiones que hemos codificado y, dentro de ella, aparece un texto, un rectángulo con esquinas rectas, una segmento de línea recta y un rectángulo con esquinas redondeadas.



Puede suponerse que, a partir de los recursos gráficos que provee la clase *Graphics* y adicionando una buena dosis de creatividad, se pueden realizar dibujos completos usando las figuras geométricas convencionales.

### 23.6 Trabajando con Colores

Un recurso que resulta muy útil, al momento de dibujar, son los colores. Para ello, el lenguaje de programación Java provee unos métodos que facilitan la interacción entre formas, trazos y colores de manera que el programador tenga control sobre todo el código y, de esta forma, pintar en la transparencia respectiva. Se explicará a continuación un cambio que se ha incorporado en la clase *Panel1* para explicar algunas de los métodos utilizados. En esta misma lección se explicarán otros métodos que se pueden utilizar y que provee la clase *Graphics*. Tal como ya lo hemos visto, lo primero que hacemos es declarar la clase *Panel1* que hereda métodos y atributos de *JPanel*. Dentro de esta clase procedemos a reescribir el método *paintComponent* que recibe un argumento de tipo *Graphics*.

```
class Panel1 extends JPanel {  
  
    public void paintComponent(Graphics g) {
```

Dentro de este método, en primer lugar invocamos el método *paintComponent* al cual le enviamos el argumento tipo *Graphics*. Es de recordar que este método lo invocamos llamando a la clase padre (la **super** clase) para que se realice la adecuación correspondiente que posibilite el uso de los métodos gráficos.

```
super.paintComponent(g);
```

Seguidamente, establecemos el tipo de letra que vamos a utilizar (**TimesRoman**), la característica de la letra (**PLAIN**), el tamaño de la letra (**20**) y el color de la letra (**black**), acudiendo a los métodos *setFont* y *setColor*. Para escribir la cadena, invocamos el método *drawString*, escribimos la cadena que queremos desplegar sobre la transparencia encerrándola entre comillas dobles y especificamos las coordenadas *x* y *y*, dentro de la ventana, en donde queremos que aparezca la cadena.

```
g.setFont(new Font("TimesRoman", Font.PLAIN, 20));  
g.setColor(Color.black);  
g.drawString("Figuras Geométricas 2D", 30, 50);
```

En este ejemplo, vamos a dibujar un rectángulo con esquinas cuadradas, una línea, un rectángulo con esquinas redondeadas y un óvalo (elipse). Dado que todo está adecuado para ello, invocaremos los métodos *drawRect*, *drawLine*, *drawRoundRect* y *drawOval* que, respectivamente, harán lo propuesto. El método *drawRect* requiere como argumentos las coordenadas *x* y *y* de la esquina superior izquierda y las coordenadas *x* y *y* de la esquina inferior derecha que se ubicarán dentro de la ventana para realizar los trazos del rectángulo. Por su parte el método *drawLine* requiere las coordenadas *x* y *y* del punto inicial del segmento de recta a dibujar y las coordenadas *x* y *y* del punto final de dicho segmento. El método *drawRoundRect*, debido a que permite dibujar un rectángulo solo que con esquinas redondeadas, requiere las coordenadas *x* y *y* tanto para la esquina superior izquierda como para la esquina inferior derecha de dicho rectángulo. Los dos últimos parámetros indican la caracterización de la “curva” que se quiere para redondear las esquinas. El método *drawOval*, cuyo objetivo es dibujar una elipse, requiere coordenadas *x* y *y* de ubicación y largo y ancho de la elipse.

```
g.drawRect(50, 100, 100, 100);  
g.drawLine(300, 100, 400, 120);
```

```
g.drawRoundRect(180, 100, 100, 100, 20, 20);  
g.drawOval(150, 215, 100, 30);
```

Ahora, si queremos llenar el rectángulo de esquinas cuadradas con un color azul, todo lo que tenemos que hacer es establecer el color a través del método *setColor* cuyo parámetro será un valor estático **BLUE** que debe escribirse precedido por su respectiva clase. El relleno se hace con el método *fillRect* y, para que rellene el rectángulo que ya se dibujó, todo lo que hay que hacer es enviar como parámetros los mismos que se enviaron en la construcción del rectángulo. Debe anotarse que, debido a que este libro se imprimió en una sola tinta, el color azul no aparece azul pero si el código se ejecuta, efectivamente aparecerá el color respectivo.

```
g.setColor(Color.BLUE);  
g.fillRect(50, 100, 100, 100);
```

De la misma manera, podemos llenar de color gris (**GRAY**) el rectángulo de esquinas redondeadas. Simplemente le enviamos los mismos parámetros cuando utilicemos el método *fillRoundRect* que es el método que se encarga de llenar la figura.

```
g.setColor(Color.GRAY);  
g.fillRoundRect(180, 100, 100, 100, 20, 20);
```

Para mantener el ejemplo, rellenamos de color negro el óvalo. Indicamos con *setColor* el color Color.BLACK y llenamos un óvalo que tiene los mismos parámetros que ya se especificaron cuando se construyó. Vale la pena anotar que si establecemos un color y luego usamos el método *drawOval*, se dibujará un óvalo (o cualquier figura que se quiera) con un contorno del color que hubiéremos especificado.

```
g.setColor(Color.BLACK);  
g.fillOval(150, 215, 100, 30);  
}  
}
```

El código completo del programa, sería el que se presenta a continuación.

```
package Ventanas;  
import javax.swing.*;  
import java.awt.*;
```

```
class Panel1 extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.setFont(new Font("TimesRoman", Font.PLAIN, 20));  
        g.setColor(Color.black);  
        g.drawString("Figuras Geométricas 2D", 30, 50);  
  
        g.drawRect(50, 100, 100, 100);  
        g.drawLine(300, 100, 400, 120);  
        g.drawRoundRect(180, 100, 100, 100, 20, 20);  
        g.drawOval(150, 215, 100, 30);  
  
        g.setColor(Color.BLUE);  
        g.fillRect(50, 100, 100, 100);  
  
        g.setColor(Color.GRAY);  
        g.fillRoundRect(180, 100, 100, 100, 20, 20);  
  
        g.setColor(Color.BLACK);  
        g.fillOval(150, 215, 100, 30);  
    }  
}  
  
class Frame1 extends JFrame {  
    public Frame1(int Ancho, int Altura, int Coordx, int Coordy, String  
    Titulo, boolean ReDim) {  
  
        setVisible(true);  
        setSize(Ancho, Altura);  
        setLocation(Coordx, Coordy);  
        setTitle(Titulo);  
        setResizable(ReDim);  
        Panel1 P = new Panel1();  
        add(P);  
    }  
}
```

```

public class Ventana1 {
    public static void main(String[] args) {
        Frame1 V1 = new Frame1(500,300,400,200,"Ventana de Prueba", true);
        V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

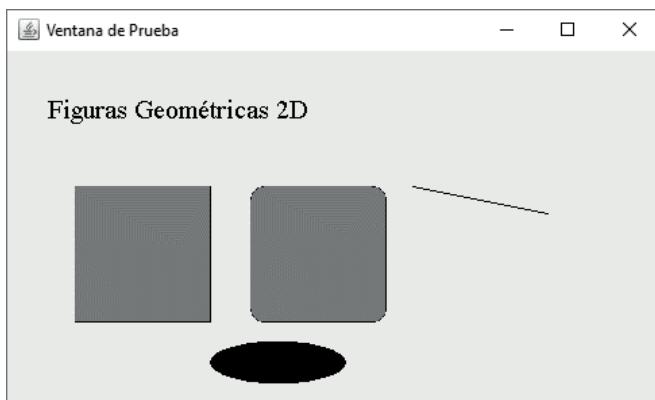
```

Consultar “Java API” (y dentro de ella, consultar la clase *Graphics*) podremos conocer un poco más el conjunto de métodos que provee Java. Algunos de ellos, tomados de esta consulta, se explican a continuación.

Método	Parámetros	Descripción
clearRect	<ul style="list-style-type: none"> <li>• Coordenadas x y y esq sup izq</li> <li>• Ancho y Alto</li> </ul>	Borra el rectángulo especificado rellenándolo con el color de fondo de la ventana donde está dibujado el rectángulo
draw3DRect	<ul style="list-style-type: none"> <li>• Coordenadas x y y</li> <li>• Ancho y alto</li> <li>• Profundidad (booleano)</li> </ul>	Dibuja un rectángulo en tres dimensiones
drawPolygon	<ul style="list-style-type: none"> <li>• Vector de coordenadas en x</li> <li>• Vecador de coordenadas en y</li> <li>• Cantidad de puntos</li> </ul>	Dibuja un polígono cerrado definido por las coordenadas x y y almacenadas en los respectivos vectores
drawPolyline	<ul style="list-style-type: none"> <li>• Vector de coordenadas en x</li> <li>• Vector de coordenadas en y</li> <li>• Cantidad de puntos</li> </ul>	Dibuja una secuencia de líneas conectadas definidas por las coordenadas x y y
fill3DRect	<ul style="list-style-type: none"> <li>• Coordenadas x y y de la esq sup izq</li> <li>• Ancho y altura del rectángulo</li> <li>• Valor booleano que describe la sensación 3D</li> </ul>	Rellena un rectángulo 3D con el color definido previamente

fillPolygon	<ul style="list-style-type: none"><li>• Vector de coordenadas en x</li><li>• Vector de coordenadas en y</li><li>• Número de puntos</li></ul>	Rellena un polígono cerrado según los vectores de coordenadas x y y
-------------	--	---

Al ejecutar el programa que se ha tomado como ejemplo, aparecerá en pantalla una ventana con la siguiente apariencia. Tenga en cuenta que el rectángulo de esquinas cuadradas está relleno de color azul. Por razones de impresión de este libro, aparece en un color diferente.



### 23.7 Modificando tipos de letras

Un recurso útil, en materia de utilidades gráficas, es el cambio, uso y adecuación de fuentes de letras para escribir dentro de las ventanas. Para ello el lenguaje de Programación Java nos brinda la posibilidad de hacer uso de las fuentes de que dispone el sistema operativo en el cual estemos trabajando. Tenga en cuenta que una *fuente* es, en palabras sencillas, un tipo de letra. Cuando usted trabaja en un procesador de palabra habrá notado que, en la parte superior, generalmente se despliega una lista desplegable con todos los tipos de letras disponibles. Cada uno de esos tipos de letras corresponde a una fuente diferente. De allí que algunos tipos de letras con los cuales posiblemente estemos familiarizados son *Times Roman*, *Arial*, *Console*, *Courier*, *Argelian*, *Calibri*, etc.

Además de definir el tipo de letra (o sea la fuente) que queremos utilizar en un título también se deben definir dos atributos más: la característica de la letra y el tamaño. La característica se refiere a la opción de utilizar una letra plana (*PLAIN*), letra resaltada o en negrilla (*BOLD*) o letra un poco

inclinada (*ITALICA*). El tamaño se especifica en puntos y, para exemplificarlo, construiremos un programa que muestre en un programa cuatro tipos de fuentes con características y tamaños diferentes. Nos vamos a valer de algunos métodos disponibles que facilitan la construcción de programas con una estética que sea agradable al usuario. El tipo de dato *Font* permite escoger una fuente con una característica y un tamaño específicos y asignarles un nombre como instancia de dicho tipo, de forma que lo podremos utilizar fácilmente de allí en adelante.

El método *setFont* permite establecer, como tipo de fuente, un determinado objeto tipo *Font*. El método *setColor* permite escoger el color para el tipo de letra. Es de anotar que si no se escoge un color, se asume por defecto que dicho color es negro. El método *drawString* permite escribir un texto dentro de una ventana especificando las coordenadas en *x* y *y* a partir de la cuales queremos que aparezca. El método *setForeground* permite especificar un color específico para todos los textos que se quieran escribir después de su utilización, sea cual fuere la fuente y las características de atributos y tamaño que se le hubieren conferido. Como ya sabemos, en este tipo de programas, nuestra atención se centrará en la adecuación de la clase que hemos llamado *Panel1* que es la que hereda métodos y atributos de *JPanel* que es una de las que provee los recursos gráficos del lenguaje de Programación Java. Seguidamente reescribimos el método *paintComponent* que requiere un parámetro de tipo *Graphics* para hacer efectivos sus métodos.

```
class Panel1 extends JPanel {  
    public void paintComponent(Graphics g) {
```

A continuación invocamos el método *paintComponent* de la superclase para que haga las adecuaciones del caso. Este método requiere un parámetro de tipo *Graphics*. En nuestro caso enviamos el parámetro *g* que es el nombre que le hemos dado a la instancia tipo *Graphics* que vamos a utilizar.

```
        super.paintComponent(g);
```

Ahora si podemos empezar a establecer las fuentes que queremos para desarrollar el ejemplo. Nótese que, inserto como parámetro del método *setFont*, podemos crear una instancia de tipo *Font* para especificar la fuente escogida y sus características. En este caso hemos escodigo en primera

instanciá la letra TimesRoman, resaltada (en negrilla) y de tamaño 10 puntos. Además hemos determinado que se muestre en pantalla de color negro. Seguidamente, a través del método drawString, establecemos el nombre de la letra como el texto que queremos que aparezca dentro de la ventana en las coordenadas (50, 30).

```
g.setFont(new Font("TimesRoman", Font.BOLD, 10));  
g.setColor(Color.black);  
g.drawString("Letra Times New Roman", 50, 30);
```

Continuando con el ejemplo, cambiamos de fuente. Esta vez seleccionamos la letra Arial, sin ningún efecto adicional y de tamaño 15 puntos. De la misma forma escogemos el color negro. Definimos el texto a mostrar en pantalla que corresponde al nombre de la fuente en la posición (50,70) o sea un poco mas abajo que el texto anterior.

```
g.setFont(new Font("Arial", Font.PLAIN, 15));  
g.setColor(Color.black);  
g.drawString("Letra Arial", 50, 70);
```

Seguidamente escogemos como tipo de letra la fuente Courier, un poco incilinada, tamaño 20 puntos y color negro. El texto que aparece en pantalla corresponde, como en los otros casos, al nombre de la fuente. Lo ubicamos en la posición relativa (50, 110).

```
g.setFont(new Font("Courier", Font.ITALIC, 20));  
g.setColor(Color.black);  
g.drawString("Letra Courier", 50, 110);
```

Por último, escogemos el tipo de fuente Verdana, resaltada, tamaño 25 puntos y color negro. El texto a mostrar en la posición (50, 150) corresponde al nombre de la fuente escogida. Con este último conjunto de instrucciones cerramos el programa.

```
g.setFont(new Font("Verdana", Font.BOLD, 25));  
g.setColor(Color.black);  
g.drawString("Letra Verdana", 50, 150);  
}  
}
```

El programa completo, incluyendo la definición de la clase Frame1 que hereda de la clase JFrame sus métodos y atributos, se presenta a continuación.

```
package Ventanas;  
import javax.swing.*;  
import java.awt.*;  
  
class Panel1 extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
  
        g.setFont(new Font("TimesRoman", Font.BOLD, 10));  
        g.setColor(Color.black);  
        g.drawString("Letra Times New Roman", 50, 30);  
  
        g.setFont(new Font("Arial", Font.PLAIN, 15));  
        g.setColor(Color.black);  
        g.drawString("Letra Arial", 50, 70);  
  
        g.setFont(new Font("Courier", Font.ITALIC, 20));  
        g.setColor(Color.black);  
        g.drawString("Letra Courier", 50, 110);  
  
        g.setFont(new Font("Verdana", Font.BOLD, 25));  
        g.setColor(Color.black);  
        g.drawString("Letra Verdana", 50, 150);  
    }  
}  
  
class Frame1 extends JFrame {  
    public Frame1(int Ancho, int Altura, int Coordx, int Coordy,  
String Titulo, boolean ReDim) {  
        setVisible(true);  
        setSize(Ancho, Altura);  
        setLocation(Coordx, Coordy);  
        setTitle(Titulo);  
        setResizable(ReDim);  
        Panel1 P = new Panel1();
```

```
        add(P);  
    }  
}  
  
public class Ventana1 {  
    public static void main(String[] args) {  
        Frame1 V1 = new Frame1(300,400,400,50,"Algunas fuentes ", true);  
        V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

Al momento de ejecutar el programa, el resultado será el siguiente:



Como se puede observar, aparecen en la ventana en ubicaciones diferentes en lo vertical, cuatro tipos de fuentes, descritas por su nombre, con tamaños y características para ejemplificar algunos de sus atributos.

### 23.8 Imágenes

Sin lugar a dudas, uno de los recursos que mas se pueden necesitar, cuando se trata de programas con interfaz gráfica, son precisamente las imágenes. Para ello debemos tener en cuenta que, en Java como todo está orientado a objetos, las imágenes son objetos que se pueden manejar a partir de sus atributos y los métodos que sobre ellos se pueden realizar.

El método *read()*, que pertenece a la clase *ImageIO* de la librería *java.imageio* permite referenciar y almacenar un objeto de tipo *File* que, en palabras simples, es una imagen representada en una foto en cualquiera

de los formatos comerciales. Java provee para tal fin un filtro bastante amplio de opciones y formatos posibles.

Por su parte, para dibujar una imagen requerimos del método *drawImage()* que se encuentra en la clase *Graphics* que pertenece al paquete *java.awt* y que permite “poner” la imagen dentro de la ventana en la cual necesitamos que aparezca. Para mostrar la manera de utilizar estos métodos, dentro de un programa en Java, realizaremos un ejemplo que muestre en pantalla una ventana dentro de la cual aparezca una foto del autor de este libro.

Al igual que en el ejemplo anterior, describiremos los cambios sustanciales que tendrá la clase *Panel1* que hereda métodos y atributos de la clase *JPanel* que, a su vez, es la que contiene los recursos para satisfacer las necesidades gráficas del programa que se quiere realizar.

```
class Panel1 extends JPanel {
```

Para el fin propuesto (desplegar en una ventana una foto del autor) declaramos de manera encapsulada (*private*) una instancia de la clase *Image* (o sea un objeto tipo *Image*) al cual llamaremos *Foto*.

```
    private Image Foto;
```

A continuación, reescribimos el método *paintComponent* cuyo parámetro, al cual hemos llamado *g*, es de tipo *Graphics*. De la misma manera, lo primero que hacemos es llamar a la superclase para que adecúe el parámetro de tipo *Graphics* *g* y poder empezar a utilizar los recursos gráficos requeridos.

```
    public void paintComponent(Graphics g) {
```

```
        super.paintComponent(g);
```

Establecemos un tipo de fuente, característica sin efectos, tamaño 20 puntos y color negro. En la parte superior de la ventana aparecerá el título “Foto del Autor”.

```
        g.setFont(new Font("Courier", Font.PLAIN, 20));  
        g.setColor(Color.black);  
        g.drawString("Foto del Autor", 30, 50);
```

Seguidamente declaramos un objeto de tipo *File* al cual hemos llamado *miFoto* y lo creamos a través del método constructor asociándolo con la ruta en la cual se encuentra la foto (o imagen) que queremos mostrar.

```
File miFoto = new File("src/Ventanas/FotoOmar.jpg");
```

A continuación le indicamos a Java que intente leer el archivo *miFoto* (que fue asociado con una imagen de formato *jpg* en una ruta específica). El método *ImageIO.read* “lanza” una excepción. Esto significa que, cuando intenta encontrar la imagen, avisa en caso de que no la encuentre.

Por lo tanto, nos corresponde escribir el código que le indique qué debe hacer en caso de que no la haya podido encontrar. La instrucción *try* le indica que intente encontrarla, la instrucción *catch(IOException e)* le indica qué hacer en caso de que no la encuentre. En nuestro caso, si no encuentra la imagen entonces le indicamos que muestre un título en pantalla anunciándolo.

```
try {
    Foto = ImageIO.read(miFoto);
}
catch(IOException e) {
    System.out.println("La foto no se encontró");
}
```

Ahora bien, si la encontró, podrá ejecutar la orden que permite poner la imagen sobre la transparencia (*Panel1*) en unas coordenadas específicas y tal como se presenta a continuación. El último parámetro por ahora será null pero mas adelante nos ocuparemos de explicarlo con mas detalle.

```
g.drawImage(Foto, 20, 20, null);
}
```

El programa completo se presenta a continuación incluyendo la definición de las demás clases.

```
package Ventanas;
import javax.swing.*;
import java.awt.*;
import javax.imageio.*;
```

```
import java.io.*;  
  
class Panel1 extends JPanel {  
    private Image Foto;  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
  
        g.setFont(new Font("Courier", Font.PLAIN, 20));  
        g.setColor(Color.black);  
        g.drawString("Figuras Geométricas 2D", 30, 50);  
  
        File miFoto = new File("src/Ventanas/FotoOmar.jpg");  
        try {  
            Foto = ImageIO.read(miFoto);  
        }  
        catch(IOException e) {  
            System.out.println("La foto no se encontró");  
        }  
  
        g.drawImage(Foto, 20, 20, null);  
    }  
}  
  
class Frame1 extends JFrame {  
    public Frame1(int Ancho, int Altura, int Coordx, int Coordy,  
                 String Titulo, boolean ReDim) {  
        setVisible(true);  
        setSize(Ancho, Altura);  
        setLocation(Coordx, Coordy);  
        setTitle(Titulo);  
        setResizable(ReDim);  
        Panel1 P = new Panel1();  
        add(P);  
    }  
}  
  
public class Ventana1 {  
    public static void main(String[] args) {  
  
        Frame1 VI = new Frame1(700,800,400,200,"Ventana de Prueba", true);  
    }  
}
```

```
V1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}  
}
```

Al ejecutar este código, se mostraría la siguiente ventana si se cuenta con la foto que aparece en ella.



El desarrollo de un programa gráfico implica el uso de una gran cantidad de recursos que permiten que éste sea mucho más estético de manera que lo que presente en pantalla sea más entendible para el usuario, razón de ser todo esfuerzo en programación.

### 23.9 Ejercicios Propuestos

- Construir un programa que muestre en pantalla una foto suya con los principales datos de su hoja de vida de una forma presentable y estética
- Construir un programa que presente tres diapositivas mostrando en cada una el origen del nombre de los días incluyendo imágenes y texto
- Construir un programa que permita mostrar en pantalla una casa dibujada con figuras geométricas
- Construir un programa que permita mostrar en pantalla su nombre completo en 10 tipos de letras, con colores, tamaños y ubicaciones diferentes
- Construir un programa que permita mostrar en pantalla su nombre completo de forma que cada letra aparezca en un tipo de fuente diferente, en un tamaño diferente y con un color diferente

Lección

# 24 Eventos del mouse



## 24.1 Concepto General

La Real Academia Española define la palabra “evento”, en una de sus opciones, como “Eventualidad, hecho imprevisto, o que puede acaecer”. Efectivamente en la POO un evento es algo que puede suceder o no y que, depende, de la voluntad del usuario. Cuando usted hace uso de un programa de uso masivo, como por ejemplo un navegador, son muchas las acciones que usted podría realizar a su voluntad bien sea por el uso del mouse o bien por el manejo de la ventana de trabajo dependiendo de sus propias necesidades. En esencia, un evento es una acción que inicia el usuario y que forma parte de un conjunto de acciones que el mismo usuario puede activar en cualquier momento. Los recursos gráficos de usuario como ventanas, botones, cajas de texto, etc. no solamente permiten que los programas tengan un estilo mucho más estético y, por lo tanto, más entendible. En la medida en que la interfaz de un programa sea bastante entendible, en esa misma medida se puede pensar en que se reduzcan los errores que el usuario pudiera cometer. Estos recursos gráficos tienen sentido toda vez que se puedan utilizar para que el usuario interactúe con el programa.

## 24.2 Definición

Para la programación de eventos a través de recursos gráficos de usuario deben responderse tres preguntas que facilitarán la comprensión del concepto y el desarrollo de las aplicaciones: a) ¿Qué evento del usuario inicia la acción? b) ¿Qué control de usuario desencadena la acción? y finalmente c) ¿Qué parte de la aplicación recibe la acción? Responder estas tres preguntas permitirá que el programador escriba el código en consonancia con sus respuestas y, por lo tanto, hará que el programa se comporte tal como lo ha planeado. El evento del usuario que inicia la acción se conoce como “Objeto Evento”, el control de usuario que desencadena la acción se conoce como “Objeto Fuente” y la parte de la aplicación que recibe la acción se conoce como “Objeto Oyente”. Luego para programar eventos deben tenerse en cuenta estas tres partes para que el programa quede construido tal como se planeó y diseño.

Para ejemplificar esto un poco mas claro, supongamos que usted se encuentra en un programa en el cual, al hacer click con el mouse sobre un botón, se cambia de color el fondo de la ventana donde está trabajando. Entonces, de acuerdo a lo dicho, las respuestas a las tres preguntas serían las siguientes:

- a) ¿Qué evento del usuario inicia la acción? → Hacer click
- b) ¿Qué control de usuario desencadena la acción? → Un botón
- c) ¿Qué parte de la aplicación recibe la acción? → La ventana de trabajo

Ahora bien, en términos puramente del lenguaje de programación Java, para recurrir al Objeto Evento debe incluirse el paquete `java.awt.event` y la Clase `EvenObject` es una clase de la que descienden el resto de clases que desencadenan eventos como por ejemplo las clases `ActionEvent` y `WindowEvent`. La clase `ActionEvent` controla los eventos del mouse y la clase `WindowEvent` controla los eventos de ventanas.

¿Cómo se ven las preguntas a responder desde la perspectiva de Java? muy sencillo. Para el ejemplo citado, como Objeto Fuente puede designarse una instancia de la clase `JButton` ya que es el control de usuario que desencadena una acción. La instancia de esta clase requiere que se implemente el método `addActionListener` que correspondería al Objeto Oyente ya que es la parte que recibe la acción a ejecutar luego de que se ha hecho click sobre el botón. Normalmente al Objeto Oyente también se conoce técnicamente como Objeto Listener. En el caso del ejemplo, el Objeto Oyente (Listener) sería una ventana, es decir, una instancia de la clase `JPanel`. Para la correcta ejecución de la acción, luego de haber hecho click con el mouse, se requiere implementar la interfaz `ActionListener` que es la que hace efectiva la acción a partir del Objeto Fuente. Esta interfaz requiere que se implemente el método `actionPerformed` al cual se le envía como parámetro un Objeto Evento.

En términos resumidos, el Objeto Fuente desencadena la acción, el Objeto Oyente (Listener) recibe la acción. Cuando el usuario activa el Objeto Fuente, entonces se activa el Objeto Evento que va hacia el Objeto Oyente. El método que recibe el Objeto Evento es el método `actionPerformed` y es el que ejecuta lo que haya adentro de este método, que correspondería a la acción concreta a realizar una vez se haya producido el evento que, para el ejemplo, sería hacer click.

### 24.3 Ejemplo

Vamos a construir un ejemplo que haga más sencillo toda esta teoría que, por momentos pareciera tornarse confusa, sin serlo obviamente. Ya verá que es bastante sencillo cuando se implemente. Vamos a construir un programa que presente una ventana con cuatro botones. El 1<sup>er</sup> botón mostrará dentro de la ventana el nombre del autor de este libro cuando se haga click sobre él. El 2º botón mostrará la imagen de un bombillo encendido cuando se haga click sobre él. El 3<sup>er</sup> botón mostrará la imagen de un bombillo apagado cuando se haga click sobre él. El 4º botón borrará el contenido de la ventana dejándola en su estado inicial, es decir, solamente con los botones de control.

Para que el programa sea un poco más entendible, los botones se rotularán así: el 1<sup>er</sup> botón tendrá el rótulo “Nombre”, el 2º botón tendrá el rótulo “Bombillo On”, el 3<sup>er</sup> botón tendrá el rótulo “Bombillo Off” y el 4º botón tendrá el rótulo “Limpiar”. Lo primero que vamos a construir es la ventana que sirva como marco general para ubicar los botones y para poner a funcionar el programa tal como se ha descrito. En primer lugar vamos a incluir el paquete con el cual estamos trabajando que, para este caso, corresponde al nombre de una carpeta que se ha llamado *PruebaEventos2*. Seguidamente, y debido a que vamos a crear una ventana, importamos el paquete que lo posibilita, es decir, el paquete *JFrame*.

```
package PruebaEventos2;  
import javax.swing.JFrame;
```

A continuación declararemos la clase principal a la cual se le ha puesto por nombre *VentanaSola*, como para indicar que este programa todo lo que hace es poner una ventana en la pantalla sin nada adicional. Esta clase contiene su respectivo método principal (*main*).

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

En esta clase decaramos una instancia de la clase *MarcoBotones* que hemos llamado *mimarco* y que describirá las características generales mínimas de la ventana.

```
MarcoBotones mimarco = new MarcoBotones();
```

Indicamos que la ventana sea visible y establecemos la acción a seguir una vez se cierre la ventana que, en este caso, corresponde a cerrar el programa.

```
mimarco.setVisible(true);  
mimarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Utilizando la llave de cierre, cerramos el método principal (main) y la clase VentanaSola.

```
}
```

A continuación, debido a que declaramos una instancia de la clase *MarcoBotones* (que hemos llamado *mimarco* y que contendrá los parámetros generales mínimos de una ventana), declaramos dicha clase que, para poder implementar una ventana, deberá heredar de la clase JFrame.

```
class MarcoBotones extends JFrame{
```

Dentro de esta clase declararemos el respectivo constructor (método que tiene el mismo nombre de la clase) y allí consignamos el título general de la ventana, la coordenada (x,y) que corresponderá a la esquina superior izquierda de dicha ventana y, finalmente, el ancho y el alto de dicha ventana.

```
public MarcoBotones() {  
    setTitle("Prueba de Botones y Eventos");  
    setBounds(700,300,500,300);  
}
```

```
}
```

De acuerdo a lo planteado, el código completo de la ventana general se presenta a continuación. Para facilitar su comprensión, se han adicionado líneas de comentarios.

```
// Inclusión de paquete (carpeta donde se almacena el programa)  
package PruebaEventos2;
```

```
// Inclusión de paquete JFrame para podere implementar ventanas
```

```
import javax.swing.JFrame;

// Clase que implementa una ventana sola. Esta es la clase principal
public class VentanaSola {

    // Método principal de la clase Ventana Sola
    public static void main(String[] args) {

        // Instancia de la clase MarcoBotones que es la ventana en sí
        MarcoBotones mimarco = new MarcoBotones();

        // Se indica que se visibilice la ventana
        mimarco.setVisible(true);

        // Se indica que se cierre el programa apena cierran la ventana
        mimarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    }      // Fin del método principal (main) de la clase VentanaSola
          // Fin de la clase VentanaSola (clase principal)

// Declaración de la clase MarcoBotones (la ventana en sí)
// pues esta es la clase que hereda de JFrame para implementarla
class MarcoBotones extends JFrame{

    // Método constructor de la clase MarcoBotones
    public MarcoBotones() {

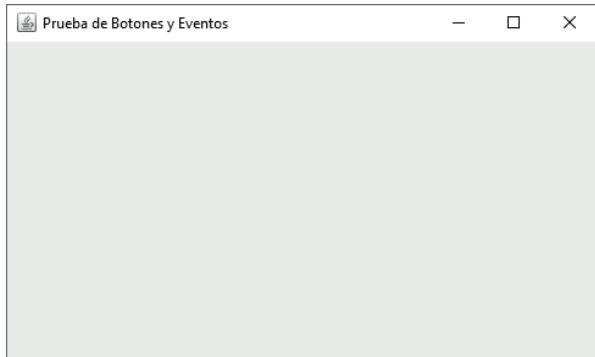
        // Título principal de la ventana (va en la parte superior)
        setTitle("Prueba de Botones y Eventos");

        // Coordenadas y tamaño de la ventana
        setBounds(700,300,500,300);

    }      // Fin del método constructor MarcoBotones()
          // Fin de la clase MarcoBotones

}
```

Al ejecutar este programa, aparecerá en pantalla lo siguiente:



Que efectivamente corresponde a una ventana general sobre la cual empezaremos a construir el resto del programa. En términos simples, hasta aquí no hemos comenzado a hacer el programa como tal pero ya construimos una ventana sobre la cual podemos hacerlo y eso ya es un pequeño avance. Ahora sí, vamos a comenzar a construir los botones y el resto del programa según las indicaciones que se han establecido. Debido a que vamos a trabajar con botones, acciones, eventos del mouse, títulos dentro de la ventana e imágenes, entonces debemos incluir los paquetes respectivos que posibiliten el uso de los métodos que corresponden. La tabla que sigue a continuación explica los paquetes que se requieren.

<b>Nombre del Paquete Java</b>	<b>Descripción</b>
Java.awt	Contiene todas la clases para crear interfaces gráficas de usuario y manipular gráficos e imágenes
Java.awt.event	Provee interfaces y clases para entenderse con diferentes tipos de eventos que tienen que ver con componentes awt
Javax.swing	Facilita un conjunto de componentes fáciles de manipular que, normalmente, funcionan iguales en cualquier tipo de plataforma. Incluye recursos para interfaz gráfica de usuario como cajas de texto, botones desplegables, botones de radio, botones de chequeo, tablas, etc.
Java.awt.Font	La clase Font incluye las fuentes de letras, caracteres y glifos que se utilizan para representar el texto de una manera visible

	tanto en gráficos como en componentes de diversa índole
Javax.swing.ImageIcon	La clase ImageIcon permite manejar, desplejar e intervenir imágenes bien sea en su tamaño natural o modificado o también convirtiéndolo en ícono
Javax.swing.JFrame	La clase JFrame permite manipular las características y operaciones básicas con las ventanas (conocidas también como frames) respetando sus propiedades de funcionamiento y operación básica
Javax.swing.JLabel	La clase JLabel permite manipular cadenas de texto que deben desplegarse sobre gráficos y componentes propios de Java

De esta forma, lo primero que vamos a hacer es importar todos estos paquetes dado que el enunciado del ejercicio implica operaciones que los requieran tal como ya se explicó. Además de ello se incluye el paquete (carpeta) en donde, personalmente, se han almacenado el código y los recursos requeridos.

```
package graficos;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import java.awt.Font;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
```

Ahora vamos a declarar la clase principal PruebaEventos. Recuerde que se reconoce que esta es la clase principal dado que es la única que contiene un método llamado *main*. Esta clase contendrá los elementos simples que vimos al inicio, es decir, declaración una instancia de la clase MarcoBotones, indicación de que se haga visible dicha instancia y descripción de acción a seguir apenas se cierre la ventana (cerrar también el programa).

```
public class PruebaEventos {  
  
    public static void main(String[] args) {  
        MarcoBotones mimarco = new MarcoBotones();  
        mimarco.setVisible(true);  
        mimarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

Declaramos la clase *MarcoBotones* que es la que hereda a la clase JFrame y que va a contener los parámetros generales mínimos de una ventana. De la misma forma codificamos el método constructor de la clase *MarcoBotones*.

```
class MarcoBotones extends JFrame{
```

```
    public MarcoBotones() {  
        setTitle("Prueba de Botones y Eventos");  
        setBounds(700,300,500,300);
```

En este punto se van a adicionar tres instrucciones nuevas en comparación con el código que solo creaba la ventana sin botones y nada. El método setLocationRelativeTo permite establecer la posición de la ventana relativa a un componente que se ha indicado como parámetro. Este componente también puede ser igual al valor figurativo **null** caso en el cual el posicionamiento se hará en el centro de la pantalla.

```
        setLocationRelativeTo(null);
```

Seguidamente vamos a declarar una instancia de la clase LaminaBotones (que aún no hemos construido) y que será la que contiene los botones de acuerdo a las especificaciones. Recuerde que en Java se construye una ventana y en ella se sobrepone una lámina que es la que contiene los controles que necesitamos. De esta forma una misma ventana puede servir para interfaces que incluyan diferentes controles gráficos de usuario.

```
LaminaBotones milamina = new LaminaBotones();
```

Una vez declarada la lámina (que aquí hemos llamado milamina) procedemos a adicionarla a adherirla (o adicionarla) a la ventana en mención.

*add(milamina);*

Finalmente cerramos la llave del método constructor y la llave de la clase respectiva.

```
}
```

Ahora vamos a construir la lámina que necesitamos para atender las necesidades del enunciado. Para ello decalramos una clase que llamaremos LaminaBotones (recuerde que el nombre de una clase puede ser cualquiera) y que hereda de la clase JPanel pues es esta la que contiene los recursos gráficos de usuario que posibilitan construir láminas que cumplan con especificaciones de usuario. Además la clase LaminaBotones implementa la interfaz ActionListener para que todo pueda funcionar como se espera desde el momento en que el usuario haga click sobre alguno de los botones.

```
class LaminaBotones extends JPanel implements ActionListener {
```

Como podría suponerse, lo primero que vamos a hacer es declarar los botones. Para ello vamos a tener en cuenta que cada botón es un objeto diferente. Cuando me refiero a objeto no lo hago en términos de la POO sino en términos prácticos, en el mundo físico, que también tiene su analogía en este paradigma de programación. Como a cada botón se le ha especificado el rótulo que debe tener entonces procedemos a crear cuatro instancias de la clase JButton siguiendo las indicaciones establecidas inicialmente. Recuerde que Java es “case sensitive”, es decir, las mayúsculas representan caracteres diferentes de las minúsculas. La instancia para los botones de “Nombre”, “Bombillo On”, “Bombillo Off” y “Limpiar” se han llamado *botonnom*, *botonBombilloOn*, *botonBombilloOff* y *botonBorrar* respectivamente. Recuerde igualmente que estos nombres de instancias obedecen a los gustos y preferencias del programador.

```
JButton botonnom = new JButton("Nombre");  
JButton botonBombilloOn = new JButton("Bombillo On");
```

```
JButton botonBombilloOff = new JButton("Bombillo Off");
JButton botonBorrar = new JButton("Limpiar");
```

A continuación vamos a codificar el método constructor de la clase *LaminaBotones* (que también deberá llamarse *LaminaBotones*). En este método constructor todo lo que vamos a hacer es llamar a un método que hemos llamado *LaminaAux* y que contendrá la esencia de la descripción de los botones tal como se había solicitado. Tal vez usted se pregunte ¿por qué no codificar las características de los botones de una vez en este método? La respuesta es muy sencilla, el conjunto de instrucciones del método *LaminaAux* lo requeriremos más adelante y si dejamos dicho código dentro del método constructor no será fácil invocarlo pues recuerde que un método constructor se invoca de manera automática cuando se crea una instancia de clase.

```
public LaminaBotones() {
    LaminaAux();
}
```

El método *LaminaAux()* contendrá las especificaciones de funcionamiento de los diferentes botones.

```
public void LaminaAux() {
```

Debemos tener claro que este método es donde se va a definir, detalladamente, las características de la lámina que se va a sobreponer en la ventana que ya construimos. En primer lugar vamos a desactivar un diseño específico para la ventana que necesitamos (debido a que Java tiene algunos diseños predefinidos). Para esto nos valemos del método *setLayout* y le enviamos el parámetro **null**.

```
setLayout(null);
```

Seguidamente especificamos la ubicación de los botones y el tamaño respectivo. Para ello nos valemos del método *setBounds* que requiere en sus dos primeros parámetros las coordenadas X y Y y en sus otros dos parámetros el ancho y el alto del botón. Como son cuatro botones, entonces cada uno lo caracterizamos independientemente acudiendo a su respectivo nombre de instancia y verificando que todos comiencen en el pixel relativo 10 para que queden alineados verticalmente.

```
botonnom.setBounds(10,60,120,30);  
botonBombilloOn.setBounds(10,100,120,30);  
botonBombilloOff.setBounds(10,140,120,30);  
botonBorrar.setBounds(10,180,120,30);
```

Ahora adicionamos estos botones a la lámina que hemos construido.

```
add(botonnom);  
add(botonBombilloOn);  
add(botonBombilloOff);  
add(botonBorrar);
```

Como cada botón va a desencadenar una acción, entonces lo “activamos” de manera que se pueda detectar cuándo el usuario hace click sobre alguno de los botones. El parámetro this indica que cuando se haga click sobre un botón, será la misma lámina la que reciba la acción a realizar. Esto lo deberemos hacer con cada uno de los botones. Es apenas natural pensar que botón que no se “active”, no ejecutará nada así aparezca en pantalla dentro de la ventana.

```
botonnom.addActionListener(this);  
botonBombilloOn.addActionListener(this);  
botonBombilloOff.addActionListener(this);  
botonBorrar.addActionListener(this);
```

Con esto cerramos, usando la llave respectiva, el método *LaminaAux()* que es el que contiene en la realidad el diseño con los botones tal como se había especificado.

```
}
```

Ahora viene una parte muy interesante. Como la clase LaminaBotones implementó la interfaz ActionListener, entonces debe tener OBLIGATORIAMENTE un método llamado actionPerformed que es el método en donde se indica qué se debe hacer cuando se haga click en cada botón. Si no lo ha notado le comento que aún no se ha indicado lo que se debe hacer en cada botón. Para implementar esta interfaz requerimos un objeto tipo *ActionEvent* que es el que detecta las acciones del mouse. Recuerde que si fuera una acción con ventanas entonces debería ser *WindowEvent* pero ese es otro tema.

```
public void actionPerformed(ActionEvent e) {
```

Lo primero que vamos a hacer es declarar una instancia de tipo *Object* (como quien dice un objeto general) que permita detectar en dónde fue que el usuario hizo click para poder proceder en consecuencia. La detección de la fuente que se activa por la acción del usuario se hace usando el método *getSource()*.

```
Object botonpulsado = e.getSource();
```

Ahora vamos a evaluar cada uno de los botones para establecer la acción a realizar en caso de que el usuario haga click sobre uno de ellos. Como la instancia *botonpulsado* (que es de tipo *Object*) almacena la fuente de la cual viene la acción (a través del método *getSource()* por intermedio del componente *e* de tipo *ActionEvent*) entonces es esta instancia la que debemos evaluar para saber cuál fue el botón sobre el cual se hizo click.

En primer lugar vamos a preguntar si se hizo click con el mouse sobre el botón cuyo rótulo era “Nombre”. En términos de Java, eso lo preguntamos comparando el contenido de la instancia *botonpulsado* con el contenido de la instancia *botonnom*.

```
if(botonpulsado == botonnom)  
{
```

En caso de que la respuesta de este condicional sea Verdadero entonces removemos todo lo que esté dibujado en ese momento en la ventana donde todo está funcionando. Esta medida se toma porque nunca sabemos el orden en que el usuario hace click, es decir, no podemos predecir si el usuario primero va a hacer click sobre el 4º botón y luego sobre el 2º botón o si hace click sobre el 3<sup>er</sup> botón y luego sobre el 1<sup>er</sup> botón. Entonces en el caso de que el usuario haga click en el 1er botón, borramos todo lo que haya en la ventana previamente. Si no hay nada, entonces la ventana queda como estaba. Para hacerlo, nos valemos del método *removeAll()* asociado al objeto **this** (es decir, a la misma ventana en donde estamos trabajando).

```
this.removeAll();
```

Seguidamente establecemos como color de fondo de la ventana el color gris encendido. Esta acción la realizamos debido a que debemos hacer

alguna acción que afecte a la ventana completa dado que es el recurso sobre el cual se va a realizar la acción.

```
setBackground(Color.lightGray);
```

A continuación adicionamos a la lámina el valor returnedo de un método que permite escribir el nombre del autor de este libro y que aún no hemos construido.

```
add(verNombre());
```

Luego, repintamos la lámina sobre la ventana, de manera que ahora visualice el nombre escrito sobre la ventana.

```
this.repaint();
```

Por último, volvemos a llamar al método *LaminaAux()* para que se vuelvan a poner, sobre la ventana, los botones que originalmente se requieren y no se cambie la estética de dicha ventana. Con esto finalizamos el conjunto de acciones a realizar en el caso de que se haya hecho click sobre el botón rotulado como “Nombre”.

```
LaminaAux();
{}
```

Ahora, en el caso de que el usuario haya hecho click sobre el botón rotulado como “Bombillo On” (o lo que significa lo mismo, en el caso de que *botonpulsado* sea igual a *botonBombilloOn*) procedemos de manera similar.

```
if(botonpulsado == botonBombilloOn)
{
```

En primer lugar, removemos todo lo que haya en la ventana donde estamos trabajando para que lo que allí aparezca no interfiera con lo que se va a visualizar sobre ella.

```
this.removeAll();
```

Después, establecemos como color de fondo de la ventana de trabajo el color gris encendido. Esta acción la hacemos para activar el recurso sobre el cual se realiza la acción.

```
setBackground(Color.lightGray);
```

A continuación, adicionamos a la lámina el resultado que retorna el método *verBombilloOn()* que mostrará la imagen de un bombillo encendido y que aún no hemos construido.

```
add(verBombilloOn());
```

Por último, indicamos que se repinte todos sobre la lámina, es decir, que aparezca la imagen del bombillo encendido y volvemos a llamar al método *LaminaAux()* para que, de nuevo, visualice los botones en su estado original, tal como los diseñamos desde el principio. Cerramos con la llave respectiva este bloque de instrucciones.

```
this.repaint();  
LaminaAux();  
}
```

Como tercera opción, en el caso de que se haya hecho click sobre el botón cuyo rótulo era “Bombillo Off” (o lo que significa lo mismo, en el caso de que el contenido de *botonpulsado* sea igual a *botonBombilloOff*) entonces procedemos de manera similar.

```
if(botonpulsado == botonBombilloOff)  
{
```

Removemos lo que haya en la ventana donde estamos trabajando para lo cual nos valemos del método *removeAll()* aplicado sobre el objeto **this** que hace referencia a dicha ventana.

```
this.removeAll();
```

Establecemos, como color de fondo, el gris encendido para afectar la ventana que es el recurso receptor de la acción a realizar.

```
setBackground(Color.lightGray);
```

Seguidamente adicionamos a la lámina el resultado que retorna el método *verBombilloOff()* que es un método que muestra un bombillo apagado y que aún no hemos construido.

```
add(verBombilloOff());
```

A continuación, en la misma ventana (**this**), indicamos que se repinte el resultado para que se muestre la acción de haber hecho click sobre este botón.

```
this.repaint();
```

Por último, hacemos un llamado al método *LaminaAux()* que es la que visualiza los botones tal como se diseñaron desde el principio y cerramos este bloque de instrucciones con la llave respectiva.

```
LaminaAux();  
}
```

Finalmente, si el usuario ha hecho click en el botón rotulado como “Limpiar” y que aquí, caprichosamente, hemos llamado *botonBorrar*, entonces procederemos de manera similar. Recuerde que el objetivo de este botón es que se borre cualquier acción que se haya ejecutado sobre la ventana y que ésta quede como inicialmente se diseñó, es decir, con los botones y nada más.

```
if(botonpulsado == botonBorrar)  
{
```

Removemos cualquier acción que se haya realizado sobre la ventana haciendo referencia a la ventana de trabajo a través de la instancia **this**.

```
this.removeAll();
```

Establecemos el gris encendido como color de fondo para afectar el recurso que recibe la acción.

```
setBackground(Color.lightGray);
```

Indicamos que se repinte la acción respectiva ejecutada sobre la ventana y luego volvemos a invocar el método *LaminaAux()* para que los botones queden en el lugar en donde estaban. De esta manera cerramos las llaves correspondientes al bloque de instrucciones y al método *ActionPerformed*.

```
        this.repaint();
        LaminaAux();
    }
}
```

Ahora, dentro de esta misma clase *LaminaBotones*, construimos los métodos que faltan y que son el método que visualiza el nombre del autor de este libro, el método que muestra una imagen de un bombillo encendido y el método que muestra una imagen de un bombillo apagado. Comenzamos, entonces, a codificar el método *verNombre* que retorna un valor tipo *JLabel*. Este método permitirá configurar el nombre completo del autor. Para ello inicialmente declaramos el nombre del método *verNombre* con su característica *private* (lo cual significa que está encapsulado) y con el tipo de valor a retornar que es una instancia de la clase *JLabel*.

```
private JLabel verNombre() {
```

Se declara una instancia de tipo *Jlabel* (que es un texto con atributos gráficos) valiéndonos de su constructor.

```
JLabel ejemplo = new JLabel();
```

A continuación, invocamos el método *setText* para definir el texto que debe aparecer que corresponde al nombre del autor de este libro.

```
ejemplo.setText("Omar Iván Trejos Buriticá");
```

Seguidamente establecemos la ubicación del texto que se quiere mostrar dentro de la ventana de trabajo y valiéndonos del método *setBounds()*.

```
ejemplo.setBounds(200, 10, 230, 100);
```

Luego definimos el tipo de letra a utilizar acudiendo al método constructor de la clase *Font*.

```
ejemplo.setFont(new Font("Arial", 0, 18));
```

Por último, retornamos la instancia de la clase *JLabel* que se ha configurado con las características del texto que se quiere mostrar y con el texto en sí. Igualmente cerramos la llave de este método *verNombre()*.

```
return ejemplo;  
}
```

De la misma manera, debemos codificar el método *verBombilloOn* cuyo objetivo es configurar una imagen de un bombillo encendido. Para hacerlo, ya hemos ubicado en la carpeta de trabajo respectiva (*graficos*) una imagen con extensión jpg. Lo primero que hacemos es declarar el método *verBombilloOn* con restricción **private** y que retorna un valor tipo *JLabel*.

```
private JLabel verBombilloOn() {
```

A continuación, declaramos una instancia de la clase *JLabel* valiéndonos de su método constructor. Como parámetro en este método invocamos el constructor de la clase *ImageIcon* y, a su vez, enviamos como parámetro el nombre de la imagen jpg que queremos visualizar.

```
JLabel ejemplo = new JLabel(new ImageIcon  
("src/graficos/BombilloOn.jpg"));
```

Establecemos el tamaño de la imagen acudiendo al método *setBounds()* que nos exige las coordenadas X y Y de ubicación en la ventana respectiva y el ancho y altura de la imagen.

```
ejemplo.setBounds(110, 30, 400, 200);
```

Finalmente, en este método, retornamos la instancia de la clase *JLabel* que contiene las características y contenido de la imagen a visualizar. Al final se cierra con la llave respectiva.

```
return ejemplo;  
}
```

Finalmente, en este programa, se codifica el método *verBombilloOff()* que visualiza la imagen de un bombillo apagado. Como ha de suponerse, las características de este método son exactamente iguales al método anterior lo cual hace suponer que podríamos construir una sola función que reciba una ruta tipo String y que retorne la imagen respectiva, siempre y cuando

se cuente con el respectivo archivo ubicado en la carpeta requerida. Para facilitar su comprensión se ha comentado este método. Es muy similar al método explicado anteriormente en detalle.

```
// Declaración del método verBombilloOff()
private JLabel verBombilloOff() {

// Declaración de la instancia tipo JLabel y relación con la imagen a
mostrar
JLabel ejemplo = new JLabel(new ImageIcon("src/graficos/BombilloOff.jpg"));

// Definición de los parámetros de ubicación y tamaño de la imagen
ejemplo.setBounds(110, 30, 400, 200);

// Retorno de la instancia que contiene las características y contenido de
// la imagen a mostrar
return ejemplo;
} // Fin del método verBombilloOff()

} // Fin de la clase LaminaBotones
```

A continuación, se presenta el código completo debidamente comentado para facilitar, aún más, la comprensión del mismo.

```
//INICIO DEL PROGRAMA JAVA

// Inclusión de paquete local (carpeta)
package graficos;
// Inclusión de paquetes complementarios
// Clases para crear interfaces gráficas de usuario
import java.awt.*;
// Clases e interfaces para eventos de componentes awt
import java.awt.event.*;
// Componentes de recursos gráficos
import javax.swing.*;
// Clase con fuentes de letras, caracteres y glifos
import java.awt.Font;
// Clase para manejar imágenes
import javax.swing.ImageIcon;
```

```
// Clase para manipular ventanas
import javax.swing.JFrame;
// Clase para manipular cadenas de textos y otros recursos
import javax.swing.JLabel;

// Clase principal del programa
public class PruebaEventos {

    // Método principal (main)
    public static void main(String[] args) {
        // Declaración de la ventana general
        MarcoBotones mimarco = new MarcoBotones();
        // Visibilización de la ventana general
        mimarco.setVisible(true);
        // Indicación de cierre de programa cuando se cierre la ventana
        mimarco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }           // Fin del método principal
}           // Fin de la clase principal

// Clase que define la ventana principal
class MarcoBotones extends JFrame{

    // Método constructor
    public MarcoBotones() {

        // Título de la ventana
        setTitle("Prueba de Botones y Eventos");
        // Ubicación y tamaño de la ventana
        setBounds(700,300,500,300);
        // Desactivación de ubicación relativa
        setLocationRelativeTo(null);
        // Declaración de lámina para sobreponer
        LaminaBotones milamina = new LaminaBotones();
        // Sobreposición de la lámina en la ventana
        add(milamina);
    }           // Fin método constructor
}           // Fin de la clase que define la ventana principal

// Clase que configura la lámina según requerimientos del programa
class LaminaBotones extends JPanel implements ActionListener {
```

```
// Declaración de cada uno de los botones
 JButton botonnom = new JButton("Nombre");
 JButton botonBombilloOn = new JButton("Bombillo On");
 JButton botonBombilloOff = new JButton("Bombillo Off");
 JButton botonBorrar = new JButton("Limpiar");

// Método constructor de la lámina
public LaminaBotones() {

    // Invocación al método que configura estéticamente la ventana
    LaminaAux();
}

// Método que configura estéticamente la ventana
public void LaminaAux() {

    // Desactivación de cualquier formato predefinido
    setLayout(null);

    // Ubicación y tamaño de cada botón
    botonnom.setBounds(10,60,120,30);
    botonBombilloOn.setBounds(10,100,120,30);
    botonBombilloOff.setBounds(10,140,120,30);
    botonBorrar.setBounds(10,180,120,30);

    // Adición de los botones a la lámina
    add(botonnom);
    add(botonBombilloOn);
    add(botonBombilloOff);
    add(botonBorrar);

    // Activación de los botones
    botonnom.addActionListener(this);
    botonBombilloOn.addActionListener(this);
    botonBombilloOff.addActionListener(this);
    botonBorrar.addActionListener(this);

}      // Fin del método que configura estéticamente la lámina
```

```
// Método que ejecuta las acciones de los eventos por mouse
public void actionPerformed(ActionEvent e) {

    // Declaración de un objeto general y captura de la fuente
    Object botonpulsado = e.getSource();

    // Si se hizo click sobre el botón rotulado con “Nombre”
    if(botonpulsado == botonnombro)
    {
        // Borrar todo lo que haya en la ventana activa
        this.removeAll();
        // Establecer color de fondo para la ventana
        setBackground(Color.lightGray);
        // Visualizar nombre del autor de este libro
        add(verNombre());
        // Volver a pintar la lámina
        this.repaint();
        // Volver a poner los botones
        LaminaAux();
    }

    // Si se hizo click sobre botón rotulado con “Bombillo On”
    if(botonpulsado == botonBombilloOn)
    {
        // Borrar todo lo que haya en la ventana activa
        this.removeAll();
        // Establecer color de fondo para la ventana
        setBackground(Color.lightGray);
        // Visualizar imagen de un bombillo encendido
        add(verBombilloOn());
        // Volver a pintar la lámina
        this.repaint();
        // Volver a poner los botones
        LaminaAux();
    }

    // Si se hizo click sobre botón rotulado “Bombillo Off”
    if(botonpulsado == botonBombilloOff)
    {
        // Borrar todo lo que haya en la ventana activa
```

```
    this.removeAll();
    // Establecer color de fondo para la ventana
    setBackground(Color.lightGray);
    // Visualizar imagen de un bombillo apagado
    add(verBombilloOff());
    // Volver a pintar la lámina
    this.repaint();
    // Volver a poner los botones
    LaminaAux();
}

// Si se hizo click sobre el botón rotulado con "Limpiar"
if(botonpulsado == botonBorrar)
{
    // Borrar todo lo que haya en la ventana activa
    this.removeAll();
    // Establecer coor de fondo para la ventana
    setBackground(Color.lightGray);
    // Volver a pintar la lámina
    this.repaint();
    // Volver a poner los botones
    LaminaAux();
}
} // Fin método que ejecuta acciones eventos mouse

// Método para establecer el nombre del autor
private JLabel verNombre() {

    // Declaración instancia tipo JLabel
    JLabel ejemplo = new JLabel();
    // Asignación de un texto (nombre del autor)
    ejemplo.setText("Omar Iván Trejos Buriticá");
    // Establecer una ubicación dentro del Frame
    ejemplo.setBounds(200, 10, 230, 100);
    // Le asignamos un tipo de fuente
    ejemplo.setFont(new Font("Arial", 0, 18));
    // Se retorna el objeto debidamente configurado
    return ejemplo;
} // Fin método para establecer nombre del autor
```

```
// Método que agrega imagen de bombillo encendido
private JLabel verBombilloOn() {

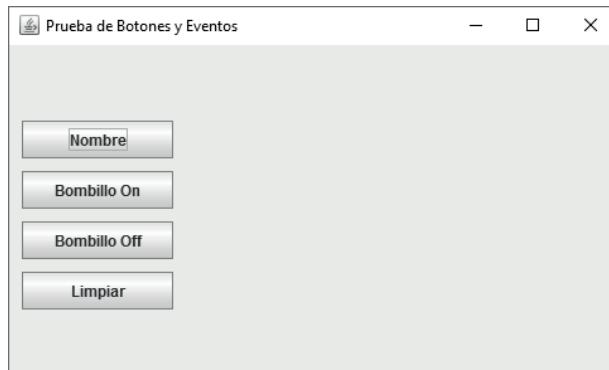
    // Se declara instancia tipo JLabel y asignación de imagen
    JLabel ejemplo = new JLabel(new ImageIcon("src/gráficos/BombilloOn.jpg"));
    // Se establece ubicación de la imagen dentro del Frame
    ejemplo.setBounds(110, 30, 400, 200);
    // Se retorna el objeto debidamente configurado
    return ejemplo;
}      // Fin método para agregar imagen de bombillo encendido

// Método que agrega una imagen de bombillo apagado
private JLabel verBombilloOff() {

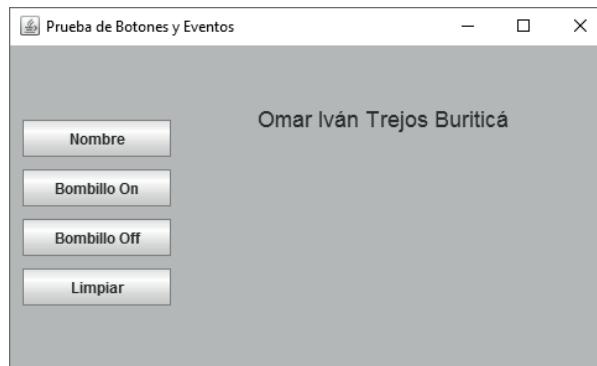
    // Se crea el objeto JLabel
    JLabel ejemplo = new JLabel(new ImageIcon("src/gráficos/BombilloOff.jpg"));
    // Se establece ubicación de la imagen dentro del Frame
    ejemplo.setBounds(110, 30, 400, 200);
    // Se retorna el objeto debidamente configurado
    return ejemplo;
}      // Fin método para agregar imagen de bombillo apagado
}          // Fin de la clase LaminaBotones
```

## 24.4 Ejecución del programa

Al ejecutar el código anterior, aparecerá en pantalla, la ventana con los cuatro botones tal como lo indicaban las especificaciones iniciales.



Si se hace click sobre el primer botón, es decir, el botón rotulado como “Nombre” entonces aparecerá el nombre del autor de este libro dentro del frame en la posición y con las características que se le han indicado.



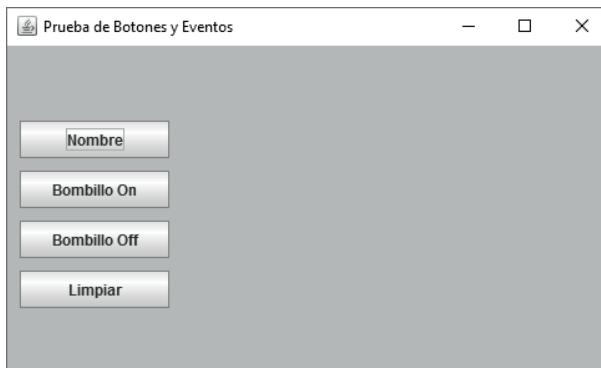
Si se hace click sobre el segundo botón, es decir, el botón rotulado como “Bombillo On” aparecerá la imagen de un bombillo encendido dentro del frame.



Si se hace click sobre el tercer botón, es decir, el botón rotulado como “Bombillo Off” aparecerá dentro del frame la imagen de un bombillo apagado.



Finalmente, si se hace click sobre el cuarto botón, es decir, el botón rotulado como “Limpiar” entonces el frame quedará como estaba inicialmente al empezar el programa.



Es de anotar que en cualquier orden en que se haga click sobre los botones, el resultado será el mismo. Es de anotar que, debido a que el presente libro se ha impreso a una sola tinta, la diferencia entre la imagen del bombillo encendido y la imagen del bombillo apagado es poca pero, al ejecutar el programa con las imágenes apropiadas, se notará que el bombillo encendido tiene color amarillo y el bombillo apagado no tiene ningún color.

## 24.5 Ejercicios Propuestos

- a) Construir un programa en Java que presente un frame con dos botones. Cuando hagan click en el primer botón, deberá aparecer el nombre del autor del programa. Cuando hagan click en el segundo botón deberá desaparecer dicho nombre dejando el frame como originalmente estaba.
- b) Construir un programa en Java que visualice un frame con tres botones. Cuando hagan click en el primer botón deberá aparecer la imagen de una persona con actitud seria. Cuando hagan click en el segundo botón deberá aparecer la imagen de esa misma persona pero sonriendo. Cuando hagan click en el tercer botón deberá aparecer la imagen de esa misma persona pero con actitud enojada.
- c) Construir un programa en Java que visualice un frame dos botones. Cuando hagan click en el primer botón deberá aparecer la nómina (listado de jugadores) de su equipo de futbol favorito. Cuando hagan click en el segundo botón deberá aparecer la foto su equipo de futbol favorito.
- d) Construir un programa en Java que visualice un frame con tres botones. Cuando hagan click en el primer botón deberá aparecer la imagen de un niño en el extremo izquierdo del frame. Cuando hagan click en el segundo botón deberá aparecer esa misma imagen en el centro del frame y cuando hagan click en el tercer botón deberá aparecer esa misma imagen en el extremo derecho del frame.
- e) Construir un programa en Java que visualice un frame con cuatro botones. Cuando hagan click en el primer botón deberá aparecer su nombre completo. Cuando hagan click en el segundo botón deberá aparecer su fecha de nacimiento debajo de su nombre. Cuando hagan click en el tercer botón deberá aparecer la ciudad donde nació debajo de la fecha de nacimiento (que a su vez deberá estar debajo del nombre completo). Cuando hagan click en el cuarto botón deberá aparecer una foto suya debajo de los datos anteriores.

Lección

# 25 Eventos de ventana



## 25.1 Concepto general

Se definen como “eventos de ventana” todas aquellas operaciones que pueden suceder, en un momento dado, con alguna ventana que, en algún sentido, se encuentre disponible dentro de una aplicación o programa Java. Recordemos que una ventana es un objeto que tiene las características de posibilitar la existencia, ubicación, activación, acción y ejecución de controles gráficos de usuario como botones, cajas de texto, botones de chequeo, botones de radio, etc.

Dentro del lenguaje Java se consideran como eventos de ventana los siguientes:

- Activación de ventana. Cuando, en una aplicación, existen varias ventanas de trabajo solamente una es la ventana activa, es decir, la ventana que responde a nuestros requerimientos de usuario. En el momento en que se hace click en otra ventana, en ese instante se desactiva la ventana en la que se estaba trabajando y se activa esa otra ventana. El evento de activación de ventana se presenta cuando, efectivamente, activamos una ventana en medio de varias ventanas de trabajo.
- Cierre de ventana con programa activo. Este evento se presenta cuando se cierra una ventana de la aplicación pero el programa sigue activo pues así lo hemos determinado a través del código como lo veremos mas adelante.
- Cierre de ventana con cierre de programa. Este evento se presenta cuando al cerrar una ventana se ha adecuado el código para que se cierre completamente la aplicación. Luego con este evento le indicamos al programa qué se debe hacer cuando así sea.
- Desactivación de ventana. En el caso de que la aplicación tenga varias ventanas (por lo menos dos) de trabajo, recordemos que solamente una es la ventana activa. En el momento en que se hace click sobre otra ventana (que no es la ventana activa), en ese instante se activa la nueva ventana y se desactiva la ventana en la que veníamos trabajando.
- Minimización de ventana. Este evento se presenta cuando hacemos click en el botón de minimizar que tiene la ventana, en su formato estándar, en el extremo superior derecho. En el momento en que sucede este evento, un botón de restauración aparece en la barra de tareas.

- Restauración de ventana. Este evento sucede cuando hacemos click en el botón de restauración que se encuentra en la barra de tareas luego de que una ventana ha sido minimizada.
- Apertura de ventana. Este evento se presenta cuando se inicia una aplicación, por primera vez, y ésta despliega en pantalla una ventana de trabajo.

Debido a las características de funcionamiento de los eventos para ventanas, hacer click en el botón que permite maximizar una ventana y que normalmente, en el formato estándar, se encuentra en la esquina superior derecha de una ventana (en la mitad de los tres botones que allí se encuentran), no se considera un evento como tal. Para comprender un poco más la naturaleza de los eventos tengamos en cuenta las operaciones que podrían considerarse contrarias unas a otras, según las definiciones que se han explicado, siempre y cuando se trate (para cada definición) de la misma ventana:

- *Activar una ventana* es el evento contrario de *desactivar una ventana*
- *Abrir una ventana* es el evento contrario de *cerrar la ventana* bien sea dejando el programa activo o, bien, desactivando también el programa
- *Minimizar una ventana* es el evento contrario de *restaurar una ventana*

## 25.2 Implementación en Java

Los eventos de ventanas que se han explicado corresponden a métodos que ya tiene definido el lenguaje de programación Java y frente a los cuales, lo que nos corresponde es escribir el código de lo que se debe hacer en cada uno de ellos. Para trabajar con eventos de ventana, además de escribir el código correspondiente que nos permita crear ventanas también debe implementarse la interfaz *WindowListener* que permite “escuchar” o registrar lo que suceda con las ventanas a nivel de “eventos de ventana”. Para que esta interfaz sea efectiva, el lenguaje de programación Java exige la implementación de siete métodos predefinidos (en su nombre) que corresponden a cada uno de los eventos establecidos como eventos de ventana. Es de anotar que en todos estos métodos debe pasarse un parámetro de tipo *WindowEvent*. La tabla siguiente describe el nombre de cada método y su equivalente:

Método Java	Evento	Descripción según Java API
windowActivated	Activación de ventana	Método que se invoca cuando la ventana pasa a ser la ventana activa

windowClosed	Cierre de ventana sin cierre de aplicación	Método que se invoca cuando se cierra una ventana pero la aplicación sigue activa
windowClosing	Cierre de ventana con cierre de aplicación	Método que se invoca cuando se cierra una ventana y también se cierra la aplicación
windowDeactivated	Desactivación de ventana	Método que se invoca cuando la ventana activa ya no es mas la ventana activa
windowDeiconified	Restauración de ventana	Método que se invoca cuando una ventana pasa de estar minimizada a su estado normal
windowIconified	Minimización de vntana	Método que se invoca cuando una ventana pasa de estar en su estado normal a estar minimizada
windowOpened	Apertura de ventana	Método que se invoca la primera vez que una ventana se hace visible

### 25.3 Código de Ejemplo

Ahora vamos a mirar cómo se implementan y se usan estos métodos para el control de los eventos de ventana. Vamos a construir un programa que genere dos ventanas de trabajo y que vaya contando y relatando los diferentes eventos que van sucediendo con las ventanas a través de avisos pertinentes que aparezcan en la consola. Lo que primero haremos será recordar la forma como se genera una ventana sencilla en pantalla. Para ello el siguiente código viene explicado a través de los comentarios que tiene insertos en él.

```
// Inclusión del paquete con el cual ha trabajado el autor de este libro
package VentanaEventos1;
// Inclusión del paquete javax.swing que contiene algunos
// recursos gráficos de usuario
import javax.swing.*;
// Inclusión del paquete java.awt.evento que posibilita el control
// de eventos en Java
import java.awt.event.*;

// Clase principal que explica los eventos de ventana
public class VentanaEventos {
    // Método principal de la clase principal
    public static void main(String[] args) {

    }
}
```

```
// Definición de una instancia tipo Ventana
Ventana Frame1 = new Ventana();

// Al cerrar la ventana, se cierra la aplicación
Frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

}      // Fin del método principal de la clase principal
      // Fin de la clase principal

// Definición de la clase Ventana
class Ventana extends JFrame {

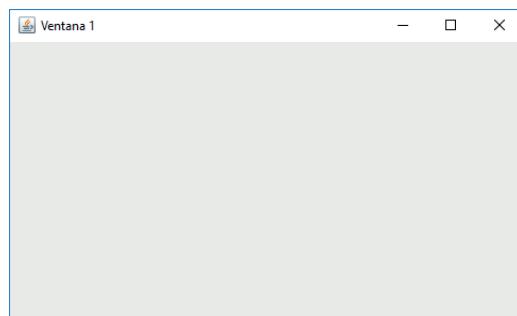
    // Método constructor de la clase Ventana
    public Ventana() {
        // Definición del título de la ventana
        setTitle("Ventana 1");

        // Ubicación y tamaño de la ventana
        setBounds(200,200,500,300);

        // Visibilización de la ventana
        setVisible(true);

    }      // Fin del método constructor Ventana
}      // Fin de la clase Ventana
```

Al ejecutar este programa, aparecerá en pantalla, una ventana similar a la que se muestra a continuación.



Ahora sí vamos a desarrollar el código que permite cumplir con el objetivo que se ha planteado. Sobre esta base vamos a construir el programa Java

que se solicita en el enunciado. Para ello vamos a ir explicando cada una de las instrucciones que hemos utilizado. Tal como lo acabamos de ver, lo primero que debemos hacer es incluir la carpeta de trabajo (que en el caso personas se llama *VentanaEventos1*) y los paquetes Java que posibilitan el uso de recursos gráficos de usuario para que el programa supla los requerimientos explicados.

```
package VentanaEventos1;
```

```
import javax.swing.*;
```

```
import java.awt.event.*;
```

A continuación creamos la clase principal. Recuerde que esta clase se considera como la clase principal porque es la única, en todo el programa Java, que contiene un método llamado main.

```
public class VentanaEventos {
```

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

En este método, vamos a declarar una variable de tipo entero que es la que llevará el proceso de conteo de las ventanas que se quieran abrir. Debe admitirse que si bien esta no es la forma más apropiada (pues debería crearse un método aparte para ello) por ahora lo utilizaremos así para agilizar la explicación. Esta variable se inicializa con el valor 0 indicando que aún no se ha creado ninguna ventana. Seguidamente creamos la ventana acudiendo a la clase Ventana que hereda sus características de JFrame.

```
    int n=0;
```

```
    Ventana Frame1 = new Ventana(++n);
```

Una vez creada la ventana, se le indica a Java que cuando esta ventana se cierre, entonces se cerrará toda la aplicación. Esto se indica con la constante JFrame.EXIT\_ON\_CLOSE.

```
Frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Para efectos del enunciado que estamos resolviendo, vamos a crear una segunda ventana. Para ello creamos otra instancia de la clase Ventana solo

que esta vez incrementamos la variable n antes de su creación (el operador ++ está antes de la variable) y de esta forma, la ventana debe aparecer en pantalla con el número 2 a diferencia de la primera ventana que debe aparecer con el número 1.

```
Ventana Frame2 = new Ventana(++n);
```

Para la creación de esta segunda ventana se le indica a Java que, en el caso de que esta ventana sea cerrada, la aplicación continúa, es decir, no finaliza la aplicación con el cierre de esta ventana a diferencia de la primera. Para indicar esto acudimos a la constante JFrame.DISPOSE\_ON\_CLOSE. Seguidamente finalizamos cerrando con llaves tanto el método principal como la clase principal.

```
Frame2.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
}
}
```

Ahora vamos a construir la clase Ventana que es la que hereda sus características de JFrame que es quien permite construir ventanas tipo Windows.

```
class Ventana extends JFrame {
```

Lo primero que haremos será codificar el método constructor de la clase Ventana que, por ser su método constructor, deberá tener el mismo nombre de la clase. A este método le hemos incorporado, como parámetro, una variable de tipo entero con el ánimo de ir contando y registrando cada ventana que se vaya abriendo con un número diferente.

```
public Ventana(int n) {
```

Definimos el título de la ventana adecuándolo para que, en su parte superior, aparezca el ordinal correspondiente. Esto significa que cuando se muestre la primera ventana entonces aparecerá “1a. Ventana”, cuando sea la segunda ventana entonces aparecerá “2a. Ventana” y así sucesivamente.

```
setTitle(n+"a. Ventana");
```

Definimos la ubicación y tamaño de la ventana. En este caso utilizamos la misma variable entera, que lleva la cuenta de las ventanas, para ubicar cada ventana en una coordenada X (columna) diferente. De esta forma las ventanas aparecerán alineadas en la misma fila pero en columnas diferentes. Debe admitirse que esta versión de la ubicación es útil para un máximo de 3 ventanas pues el resultado de la multiplicación superaría la dimensión horizontal de la pantalla. De acuerdo a esto, cuando el contenido de la variable *n* sea igual a 1 entonces el producto  $(n-1)*600$  será igual a  $0*600$  lo cual es igual a 0, por lo tanto la primera ventana aparecerá arrinconada hacia la izquierda. Si el contenido de *n* es igual a 2 entonces la expresión  $(n-1)*600$  será igual a  $1*600$  y la ventana aparecerá a partir de la columna 600 de la pantalla. Por las dimensiones de ancho, es fácil entender por qué solo sería posible ubicar tres ventanas en la pantalla.

```
setBounds((n-1)*600,200,500,300);
```

De la misma forma se indica que se haga visible la ventana en la pantalla.

```
setVisible(true);
```

A continuación, y con el ánimo de poder detectar los eventos que sucedan con cada ventana, creamos una instancia de la clase *OyenteVentana* (que aún no hemos construido) pero cuyo objetivo es “escuchar” lo que suceda con cada ventana construida.

```
OyenteVentana VI = new OyenteVentana();
```

Esta instancia “oidora” de los eventos de la ventana, lo activamos adicionándola como un “oyente de ventana” y eso lo hacemos a través del método *addWindowListener* cuyo parámetro precisamente, es el objeto recién creado. Después de esto finalizamos con las respectivas llaves tanto el método constructor como la clase *Ventana* que hereda de *JFrame*.

```
    addWindowListener(VI);  
}  
}
```

Como el objetivo de la clase *OyenteVentana* es “escuchar” lo que suceda con cada ventana, es decir, que sirva de oyente de los eventos de ventana entonces se requiere implementar la interface *WindowListener*.

***class OyenteVentana implements WindowListener {***

En esta clase lo primero que haremos será crear una variable de tipo entero que vaya contando la cantidad de eventos que sucedan con las ventanas que se involucren en este programa. Se ha declarado static para que el valor almacenado por esta variable, en cualquier momento, no se pierda y siempre continúe incrementando a partir del último valor.

***static int Contador=0;***

Por razones propias del lenguaje de programación Java, se requiere que la clase que implemente la interface *WindowListener* sea el que, a su vez, contenga los métodos que registran los siete eventos que considera Java en relación con las ventanas y que se explicaron al inicio de esta lección. Debe anotarse que la detección de estos eventos es automática. Solo tienen que suceder y nada más. Recuerde que todos estos métodos de eventos de ventana predefinidos por Java requieren que se les invoque enviando como parámetro un objeto tipo *WindowEvent*. El primer método que codificaremos será el que sucede cuando una ventana se activa. En este método todo lo que haremos es incrementar el contenido de la variable que va contando cada evento de ventana que suceda y luego mostramos en consola el número del evento y el tipo de evento que acaba de suceder. Recuerde que el carácter \n hace que el cursor pase a la siguiente línea.

***public void windowActivated(WindowEvent e) {***

***Contador++;  
System.out.println(Contador + "a. Acción \n Activar Ventana \n");  
}***

El segundo evento que codificaremos será el que sucede cuando se cierra una ventana sin que eso implique que se finalice la aplicación. En este método registramos el incremento de la variable que cuenta la cantidad de eventos y mostramos en pantalla el evento que acaba de suceder.

***public void windowClosed(WindowEvent e) {***

***Contador++;  
System.out.println(Contador + "a. Acción \n Cerrar Ventana\n");  
}***

El tercer método corresponde al evento de ventana que sucede cuando se cierra una ventana con lo cual también se cierra la aplicación. Se muestra en pantalla el aviso respectivo indicando el número de evento y la acción sucedida.

```
public void windowClosing(WindowEvent e) {  
  
Contador++;  
System.out.println(Contador + "a. Acción \n Cierre de Ventana\n");  
}
```

El cuarto método corresponde al evento que sucede cuando se desactiva una ventana, en el caso en que existan varias ventanas en pantalla. Iguamente en este método, contamos la cantidad de eventos a través de la variable Contador y mostramos en pantalla el aviso respectivo.

```
public void windowDeactivated(WindowEvent e) {  
  
Contador++;  
System.out.println(Contador + "a. Acción \n Desactivar Ventana\n");  
}
```

El quinto evento a codificar corresponde al método que registra cuando se restaura una ventana después de que esta ha sido minimizada. Se cuenta el evento y se presenta en pantalla el aviso respectivo.

```
public void windowDeiconified(WindowEvent e) {  
  
Contador++;  
System.out.println(Contador + "a. Acción \n Restaurar Ventana\n");  
}
```

El sexto método a codificar corresponde al evento que sucede cuando se minimiza una ventana y, normalmente, pasa a la barra de tareas. Se cuenta el evento y presenta en pantalla el resultado de dicho conteo junto con la descripción breve del evento sucedido.

```
public void windowIconified(WindowEvent e) {  
  
Contador++;
```

```
System.out.println(Contador + "a. Acción \n Minimizar Ventana\n");  
}
```

El séptimo (y último) método corresponde al evento que detecta la apertura de una ventana por primera vez. En este caso, al igual que en los demás y en cumplimiento de los requerimientos del programa, se cuenta el evento con la variable respectiva y se muestra en pantalla con el aviso pertinente.

```
public void windowOpened(WindowEvent e) {  
  
    Contador++;  
    System.out.println(Contador + "a. Acción \n Abrir Ventana\n");  
}
```

Finalmente se cierra la clase OyenteVentana que implementa la interface *WindowListener*.

```
}
```

A continuación se presenta, de nuevo, el programa completo pero esta vez con líneas de comentario que le permitirán entenderlo de forma más directa y sencilla sobre el mismo código. Recuerde que una línea de comentario se inicia con los caracteres // y no requieren finalización. Si son varias líneas de comentarios entonces se inician con /\* y se finalizan con \*/.

```
// Se incluye la carpeta de trabajo  
package VentanaEventos1;  
  
// Se importa el paquete que proporciona uso de diversos controles  
// gráficos de usuario  
import javax.swing.*;  
  
// Se importa el paquete que permite la implementación de eventos  
import java.awt.event.*;  
  
// Se define la clase principal  
public class VentanaEventos {  
  
    // Se define el método principal de la clase principal  
    public static void main(String[] args) {  
  
        // Se inicializa variable para contar ventanas en 1
```

```
int n=1;

// Se declara un objeto tipo Ventana
Ventana Frame1 = new Ventana(n++);

// Se indica que cuando esta ventana se cierre
// también se cierra la aplicación
Frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Se declara otro objeto tipo Ventana
Ventana Frame2 = new Ventana(n++);

// Se indica que cuando esta ventana se cierre
// no se cierra la aplicación

Frame2.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

}

}      // Fin del método principal
       // Fin de la clase principal

// Se define la clase Ventana que hereda de JFrame
class Ventana extends JFrame {

    // Se define el constructor de la clase Ventana
    public Ventana(int n) {

        // Se le coloca un título a la ventana
        setTitle(n+"a. Ventana");

        // Se establece ubicación y tamaño de la ventana
        setBounds((n-1)*600,200,500,300);

        // Se indica que se visibilice la ventana
        setVisible(true);

        // Se declara un objeto que oye los eventos de ventana
        OyenteVentana V1 = new OyenteVentana();

        // Se activa el proceso para escuchar los eventos
        addWindowListener(V1);
    }
}
```

```
        }      // Fin del método constructor Ventana
    }      // Fin de la clase Ventana

// Se define la clase que implementa WindowListener
class OyenteVentana implements WindowListener {

// Se declara una variable para contar los eventos
static int Contador=0;

// Se define el método que detecta el evento que activa
// una ventana
public void windowActivated(WindowEvent e) {

// Se incrementa en 1 la variable que cuenta eventos
Contador++;

// Se muestra en consola el aviso respectivo
System.out.println(Contador + "a. Acción \n Activar Ventana \n");

        }      // Fin método que activa una ventana por primera vez

// Se define el método que detecta el evento de cierre de una
// ventana sin que eso implique el cierre de la aplicación
public void windowClosed(WindowEvent e) {

// Se incrementa en 1 la variable que cuenta eventos
Contador++;

// Se muestra en consola el aviso respectivo
System.out.println(Contador + "a. Acción \n Cerrar Ventana\n");

        }      // Fin método que detecto cierre de ventana
        // sin cierre de aplicación

// Se define el método que detecta el evento de cierre de una
// ventana y que al tiempo cierra la aplicación
public void windowClosing(WindowEvent e) {

// Se incrementa en 1 la variable que cuenta eventos
```

```
Contador++;  
  
// Se muestra en consola el aviso respectivo  
System.out.println(Contador + "a. Acción \n Cierre de Ventana\n");  
  
}  
    // Fin método que detecta cierre de ventana  
    // con cierre de aplicación  
  
// Se define el método que detecta la desactivación de  
// una ventana cuando otra ventana pasa a ser la ventana activa  
public void windowDeactivated(WindowEvent e) {  
  
    // Se incrementa en 1 la variable que cuenta eventos  
    Contador++;  
  
    // Se muestra en consola el aviso respectivo  
    System.out.println(Contador + "a. Acción \n Desactivar Ventana\n");  
  
}  
    // Fin método que detecta desactivación de una ventana  
  
// Se define el método que detecta la restauración de una ventana  
public void windowDeiconified(WindowEvent e) {  
  
    // Se incrementa en 1 la variable que cuenta eventos  
    Contador++;  
  
    // Se muestra en consola el aviso respectivo  
    System.out.println(Contador + "a. Acción \n Restaurar Ventana\n");  
  
}  
    // Fin método que detecta restauración de una ventana  
  
// Se define el método que detecta la minimización de una ventana  
public void windowIconified(WindowEvent e) {  
  
    // Se incrementa en 1 la variable que cuenta eventos  
    Contador++;  
  
    // Se muestra en consola el aviso respectivo  
    System.out.println(Contador + "a. Acción \n Minimizar Ventana\n");
```

```
        } // Fin método que detecta minimización de ventana

// Se define el método que detecta el evento que abre una
// ventana por primera vez
public void windowOpened(WindowEvent e) {

// Se incrementa en 1 la variable que cuenta eventos
Contador++;

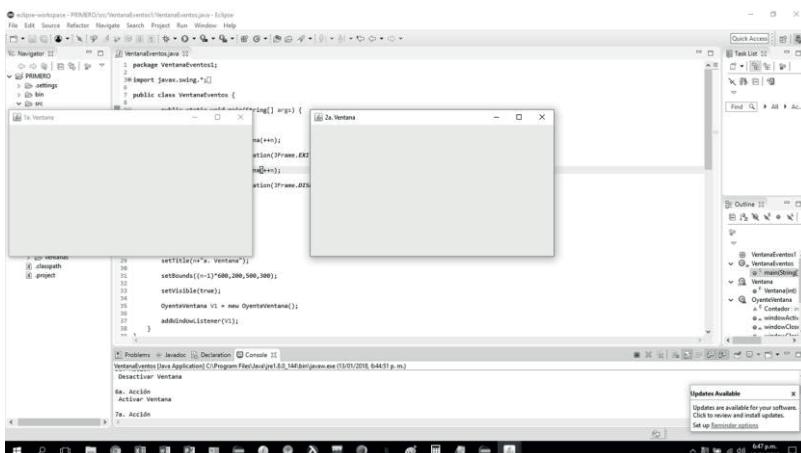
// Se muestra en consola el aviso respectivo
System.out.println(Contador + "a. Acción \n Abrir Ventana\n");

        } // Fin método que detecta apertura de ventana 1a vez
    } // Fin de la clase que implementa WindowListener
```

Ahora vamos a verificar el resultado en la consola de la ejecución del programa.

## 25.4 Ejecución del Programa Ejemplo

Cuando ejecutamos este programa, inmediatamente aparecen en pantalla dos ventanas. Una identificada en su parte superior como “1a. Ventana”, la otra identificada como “2a. Ventana”. La 2<sup>a</sup> ventana aparece un poco más oscura que la 1<sup>a</sup> ventana, la cual aparece un poco más gris. Esto indica que la 2<sup>a</sup> ventana es la ventana activa.



Mientras tanto, con solo ejecutar el programa, aparecen en consola los siguientes avisos:

1a. Acción  
Abrir Ventana

2a. Acción  
Desactivar Ventana

3a. Acción  
Activar Ventana

4a. Acción  
Abrir Ventana

5a. Acción  
Desactivar Ventana

Si hacemos click en la parte superior de la 1<sup>a</sup> ventana, esta se torna un poco más oscura y la 2<sup>a</sup> ventana se torna de color gris, indicando que ahora la ventana activa es la 1<sup>a</sup> ventana. Al mismo tiempo, en la consola aparecen los siguientes avisos.

6a. Acción  
Activar Ventana

7a. Acción  
Desactivar Ventana

Si ahora hacemos click en el botón de minimizar la ventana 1 y luego la restauramos, aparecerán (adicionalmente) en consola los siguientes avisos:

8a. Acción  
Desactivar Ventana

9a. Acción  
Activar Ventana

10a. Acción  
Desactivar Ventana

11a. Acción  
Restaurar Ventana

12a. Acción  
Activar Ventana

Si, ahora, cerramos la 2<sup>a</sup> ventana, entonces queda la ventana 1 activa y aparecerán en consola los siguientes avisos.

13a. Acción  
Desactivar Ventana

14a. Acción  
Activar Ventana

15a. Acción  
Cierre de Ventana

16a. Acción  
Desactivar Ventana

17a. Acción  
Activar Ventana

Finalmente si hacemos click en el botón para cerrar la ventana 1, entonces desaparecerá la ventana, se cerrará la aplicación y en consola aparecerán los siguientes avisos.

19a. Acción  
Cierre de Ventana

Como usted puede observar, todos los eventos que sucedan con las ventanas son detectados por los métodos que tiene predefinidos el lenguaje de programación Java. Tenga en cuenta que los eventos de ventana son diferentes a eventos de mouse. Los de ventana tienen que ver con *WindowListener* y los de mouse tienen que ver con *ActionListener*.

## 25.5 Comentario adicional

Debido a que la implementación de la interfaz *WindowListener* implica la codificación de los siete métodos que se ocupan de los eventos de ventana predefinidos, y considerando que esa tarea podría resultar un poco agotadora y por momentos innecesaria pues no siempre se requiere tener control sobre todos los métodos, el lenguaje de programación Java provee un mecanismo para que se haga un poco mas sencillo el proceso de codificación. En vez de codificar la clase que implementa *WindowsListener* (y que obliga a tener presente todos los métodos de

eventos de ventana) se puede cambiar por heredar la clase *WindowAdapter*. Con este simple recurso, dentro de la clase declarada, se pueden codificar solamente los métodos de eventos de ventana que sean necesario dentro del contexto de los requerimientos de un programa. Situación equivalente sucede cuando se hereda la clase *KeyAdapter* o *MouseAdapter*, clases que proveen los métodos para control de eventos con el teclado y con el mouse respectivamente. Llevado al código Java, esto implicaría que en vez de la línea de código

```
class OyenteVentana implements WindowListener {
```

se podría escribir la línea de código

```
class OyenteVentana extends WindowAdapter {
```

y con ese simple cambio es suficiente para que no se tengan que escribir todos los métodos asociados a los eventos de ventana sino que se pueden codificar solamente aquellos que se necesiten. Tenga en cuenta que cuando se implementa la interfaz *WindowListener* y no se codifican todos los siete métodos correspondientes, se genera un error en Java. En cambio, cuando hereda la clase *WindowAdapter*, no es necesario que aparezcan los siete métodos sino solamente los que se requieran programar.

## 25.5 Ejercicios

- Construir un programa en Java que active una ventana y que al cerrarse muestre en consola su nombre completo
- Construir un programa en Java que active una ventana y que, cuando la cierre, muestre en consola la cantidad de veces que usted lo minimizó y lo restauró con contadores diferentes
- Construir un programa en Java que active una ventana y que, a medida que la minimice y la restaure, vaya contando la cantidad de veces que lo vaya haciendo
- Construir un programa en Java que active una ventana y que cuando se minimice muestre en consola el año de su nacimiento y, por cada vez que la restaure o la vuelva a minimizar, vaya incrementando en 1 ese valor progresivamente
- Construir un programa en Java que active dos ventanas y que cuando se active la 1<sup>a</sup> ventana muestre en consola su nombre completo y cuando active la 2<sup>a</sup> ventana muestre en consola su fecha de nacimiento

## BIBLIOGRAFÍA

- Bloch, J. (2018). Effective Java. Reading (Massachusetts) - USA: Addison Wesley Professional.
- Burt, B. (2017). Beginning programming with Java for Dummies. New York (USA): For Dummies Editorial.
- Deitel, P., & Deitel, H. (2017). Java - Cómo programar. New York (USA): Pearson Education.
- Horstman, C. (2018). Core Java - Fundamentals. Bergen County (New Jersey) - USA: Prentice Hall .
- Lowe, D. (2017). Java All in One for Dummies. New York (USA): For Dummies Editorial.
- Murach, J. (2017). Java Programming. Lancaster (UK): Mike Murach & Associates.
- Oracle, O. W. (2018). Go Java. (Oracle, Productor) Recuperado en julio 24 2018, de <https://go.java/index.html?intcmp=gojava-banner-java-com>
- Schildt, H. (2017). Java - A begginers guide. New York (USA): McGraw Hill Education.

En la actualidad, el enfoque de programación orientado a objetos ha permitido que la lógica se aproxime más a la realidad, es decir, que la interpretación que se haga desde los modelos computacionales se parezca más al mundo que nos rodea y es eso lo que ha llevado a que sea este paradigma el favorito en la programación moderna. Este libro ofrece un curso completo de programación en Java tanto las explicaciones como los ejemplos a los cuales se acude son completamente originales tomados de experiencias conjuntas de los autores de forma que se puedan enriquecer tanto el proceso de aprendizaje como la utilización y aplicación de la POO en toda su plenitud.

El texto tiene de especial que se han refinado los conceptos básicos para que se pueda desarrollar todo un corpus que lo lleve a usted desde lo más simple hasta los conceptos que distinguen la misma POO tales como herencia, polimorfismo, encapsulamiento y otros. Si no ha entrado en el fascinante mundo de la POO entonces podrá hacerlo lección por lección de forma secuencial. Si ya ha tenido alguna experiencia de programación con Java entonces podrá pasar a la segunda parte del libro y comenzar a conocer el uso y manejo de conjuntos de datos y si ya tiene una experiencia notoria en POO entonces la tercera parte podrá enriquecer su conocimiento de forma que le permitan hacer suyo este maravilloso paradigma de programación moderna.