FUNDAMENTOS DE PROGRAMACIÓN EN LENGUAJE JAVA

FACULTAD DE INGENIERÍA

Juan D Castellanos
Fundación Universitaria
EDITORIAL

FUNDAMENTOS DE PROGRAMACIÓN EN LENGUAJE JAVA

Fundación Universitaria Juan de Castellanos
Tunja

Rodríguez Barrera, Claudia Hermencia; Delgado González, Iván Andrés; Bohada Jaime, John Alexander

Fundamentos de programación en lenguaje java
Editorial Fundación Universitaria Juan de Castellanos,
2019.

111 p. 17 x 24 cm

1. Lenguaje de programación en Java 2. Java (lenguaje de programación de computadores 3. Fundamentos de programación – Java
005.13
R696

Colección:
Facultad de Ingeniería

Primera edición:
Tunja, septiembre 2019

ISBN ELECTRÓNICO: 978-958-8966-30-4

Editorial:
Fundación Universitaria Juan de Castellanos
Sede Álvaro Castillo Dueñas.
Carrera 11 N° 11-44, Tunja (Boyacá-Colombia)
PBX: (8)7458676 Ext. 1128
editor@jdc.edu.co

Dirección editorial
Sandra Liliana Acuña González, Mg.

Corrección de estilo
Alfredo de Jesús Mendoza Escalante, Mg.
Fundación Universitaria Juan de Castellanos

Diagramación y Diseño de Carátula:
Aida Cecilia Barrera Torres

Impresión
Editorial JOTAMAR S.A.S.
Calle 57 No. 3 - 39.
Tunja - Boyacá - Colombia.

PRÓLOGO

Este libro es una oportunidad para iniciar en el proceso de aprendizaje de programación en el lenguaje Java, de una manera clara, divertida, concreta, pero, sobre todo, con una serie de bases pedagógicas y didácticas que le permitirán al lector, de manera práctica, realizar a su medida un entrenamiento apropiado para la consecución del conocimiento en uno de los lenguajes más utilizados en el desarrollo de aplicaciones.

De la misma forma, este libro a través de los diferentes ejercicios, le permite al lector adentrarse en cada uno de los códigos necesarios dentro del lenguaje, para que así sus proyectos sean debidamente estructurados a partir de las diferentes herramientas que el mismo lenguaje posee.

Quien lea este ejemplar, debe reconocer que tiene en sus manos una oportunidad para desarrollar y aplicar la programación, y es, a su vez, una apuesta para el mejoramiento de los conocimientos profesionales, ya que Java además de ser uno de los principales lenguajes de programación de la actualidad, posee también una alta flexibilidad en su desarrollo, haciendo que muchos de los problemas que se presentan en los diferentes campos se puedan solucionar con gran rapidez y efectividad.

Sabemos que el acelerado avance que tiene la tecnología en estos momentos debe hacer que quienes se interesan, desarrollan y programan con este lenguaje estén a la vanguardia de las necesidades y de los requerimientos propios de los diferentes usuarios que así lo determinen. Así mismo, el adentrarse en este lenguaje de programación, permite a los lectores superar la curiosidad por aprender y saber cada vez más, para que ellos mismos a través de la puesta en práctica de estos elementos comprendan que son capaces de llegar a un conocimiento y al desarrollo de sus habilidades más allá de lo que se imaginan.

Este libro surge como respuesta de la experiencia e investigación de sus autores, quienes llevan varios años dedicados a la docencia en áreas fundamentales como la algoritmia, programación, estructura de datos, modelos dinámicos e inteligencia artificial; campos que han sido la base y motivación para la construcción del presente compendio.

Contenido

1. ESTRUCTURA DE UN PROGRAMA EN JAVA

Un programa Java está compuesto por una o más clases. Es obligatorio que uno de los métodos de la clase principal sea el método ***main().*** Un método es un conjunto de instrucciones que realizan una o más acciones [1]. Java también permite que desde un programa se pueda hacer uso de otros programas escritos previamente, para hacer esto posible hacemos uso de la instrucción ***import*** seguida del nombre del programa (archivo) que queremos incluir en nuestro programa.

1.1 NOMBRE DEL PROGRAMA

Es el nombre que identifica al programa principal. Este nombre debe ser el mismo de la clase principal. Ejemplo: ProgramaAreas.java. En la clase principal debe estar el método main().

1.2 INCLUSIÓN DE OTROS ARCHIVOS

A través de la instrucción ***import*** se pueden incluir archivos que contienen clases y métodos previamente definidos. La línea siguiente permite que un programa java pueda utilizar las clases de entrada y salida.

```
Import java.io.*;
```

1.3 MÉTODO main()

Método encargado de inicializar la ejecución de un programa en java [2]. El argumento de ***main()*** es un arreglo de cadenas que recibe datos de la línea donde se ejecute el programa. Sin importar que no haya argumentos en la línea de ejecución es obligatorio especificar ***String[].*** Para este método, java reglamenta la siguiente sintaxis:

```
public static void main(String[] args)
        {
                declaraciones locales;
                instruccion_1;
                instruccion_2;
                instruccion_n;
        }
```

1.4 OTROS MÉTODOS

Son métodos adicionales al método main() que son definidos por el usuario dentro de una clase para realizar tareas específicas. Los métodos en java son subprogramas que pueden devolver un valor único, un conjunto de valores o no devolver valores y realizar una acción particular.

```
static double calcularArea()
{
                declaraciones locales;
                instruccion_1;
                instruccion_2;
                instruccion_n;
        }
```

1.5 COMENTARIOS

Son líneas que introducidas por los programadores pero que no son tenidas en cuenta al momento de compilar y ejecutar un programa. En java se pueden utilizar los símbolos siguientes para indicar un comentario:

```
//Comentario para una línea
/*Comentario
   Para
   Varias
   Líneas */
```

1.6 PUNTO Y COMA (;)

En java cada instrucción debe finalizar con un punto y coma (;) Es posible tener varias instrucciones en una sola línea e instrucciones de varias líneas.

```
Instruccion_n;
Instrucción_1; instrucción_2; instrucción_n;

System.out.println("Este ejemplo muestra como se puede
crear una instrucción java en varias líneas. Aquí el
triple de "+x+ "es "+ 3*x);
```

1.7 EJEMPLO DE UN PROGRAMA EN JAVA

```
import java.io.*;
public class Ejercicio1
{
      public static void main(String[]args)
      {
            Cuadro Cuadro1 = new Cuadro();
            System.out.println("Area ="+Cuadro1.
            calcularArea());
      }
}
class Cuadro
      {
            double alto = 5;
            double largo = 10;
            double calcularArea()
                  {
                        return alto * largo;
                  }
      }
```

1.8 PAQUETES (package)

Los paquetes son una agrupación de clases. Los paquetes están localizados en folders del disco que tienen el mismo nombre que el paquete. Para hacer uso de las clases de un paquete se utiliza la instrucción import, la cual sigue la siguiente sintaxis:

```
Import  nombrePaquete.nombreClase;
```

Ejemplo:

```
import java.io.*;          //Incorpora todas las clases del paquete java.io
Import java.util.Date;     //Incorpora la clase Date del paquete java.util
```

1.8.1 PAQUETE java.lang

Es un paquete especial que contiene todos los elementos básicos para la elaboración de un programa. Debido a su importancia el compilador

siempre lo incluye lo cual evita que tenga ser incorporado por el programador al programa.

1.9 DECLARACIÓN DE CLASES

Un programa en Java está organizado como una colección de clases. El programa debe tener al menos una clase, puede ser considerada como la clase principal, con el método main() y si es necesario otros métodos y variables[1]. Para declarar una clase en Java se hace con la siguiente sintaxis:

```
tipo_acceso class Nombre_Clase
```

tipo_acceso	Indica el tipo de acceso de la clase (Es opcional)
class	Palabra reservada de java para hacer la declaración de una clase.
Nombre_Clase	Nombre con el que se identifica una clase dentro de un programa.

```
//Ejemplo:  Programa con una clase única
public class Cuadrado {
      public static void main(String[]args)
            {
                  int numero = 3;
                  System.out.println("El cuadrado de
                  "+numero+" es "+numero*numero);
            }

}

//Ejemplo:  Declaración de clases adicionales y
Declaraciones dentro de una clase
public class Ejemplo
{
      public static void main(String[]args)
            {
                  //Creación de un objeto de la clase Suma
                  Suma Suma1=new Suma();
```

[1] JOYANES AGUILAR, Luis. Programación en Java 2. Pg 105.

```
                System.out.println("El total de la suma es
                "+Suma1.sumarValores());
            }
}
class Suma
    {
        int valor1, valor2; //Variables que se pueden
        utilizar en //cualquier método de la clase
        public int sumarValores()
            {
                int valor3, total; //Variables
                locales al método
                valor1 = 1000;      //y solo pueden
                utilizarse
                valor2=valor3=200;  // dentro de él
                total=valor1+valor2+valor3;
                return total;
            }
    }
```

1.10 PROGRAMACIÓN DEL MÉTODO main()

El método ***main()*** es el encargado de inicializar un programa. Todos los programas deben contener uno y sólo un método ***main()***, si se declaran dos o más métodos ***main()*** se generará un error así estos hayan sido creados en diferentes clases.

Sintaxis del método main()

```
public static void main(String[ ] args)
        {
                Instruccion_1;
                Instruccion_2;
                .....
                Instruccion_n;
        }
```

Encabezado: public static void main(String[ ] args)

El método ***main()*** tiene como argumento una lista o arreglo de cadenas que le dan la posibilidad de introducir cadenas de caracteres desde la línea de ejecución.

1.11 DEFINICIÓN DE OTROS MÉTODOS

Un programa java, además de tener un método main() desde el que se inicializa el programa, puede tener otros métodos definidos por el usuario según las necesidades que tenga para dar solución a un determinado problema. Estos métodos pueden estar definidos dentro de la clase principal del programa, si es que el programa es muy pequeño, pero para programas extensos, es recomendable que estos métodos se definan dentro de otras clases. Cada método pertenece a la clase en la cual es definido y se invoca desde esta misma clase.

1.12 ESTRUCTURA DE UN MÉTODO

```
tipo_de_retorno nombreMetodo(parámetros)
        {
                Variables internas del método;
                Instrucciones del método;
                return valor;
        }
```

tipo_de_retorno	*Corresponde al tipo de dato devuelto por el método.*
nombreMetodo	*odos los métodos deben llevar un nombre que los identifique. Se sugiere que este nombre identifique la tarea realizada por el método.*
parámetros	*Conjunto de parámetros que recibe el método.*

1.13 MÉTODOS QUE NO RETORNAN VALORES

No todos los métodos retornan valores para que sean tomados por otros métodos. Para poder hacer esto, es necesario anteponer al nombre del método la palabra reservada *void,* la cual indicará al compilador que el método que se está definiendo no retornará ningún tipo de valor (int, double, char, long, float, etc.).

El siguiente ejemplo muestra un programa donde se crea una nueva clase llamada Valores, en la cual se definen las variables a, b, y c de tipo int junto con el método *visualizarValores()* de tipo *void.* Este método permite visualizar los contenidos de las variables en un punto dado del programa. En la clase principal *OperadorAsigClase*, se define el método

main(), en el cual se crea una instancia de la clase Valores con el nombre *MiValor*. También se le asignan nuevos valores a las variables del nuevo objeto y podemos ver la forma como se hace el llamado al método *visualizarValores()*

```
public class OperadorAsigClase
{
      public static void main(String[]args)
      {
            Valores MiValor = new Valores();
            MiValor.visualizarValores(); //Llamado al método //
                                          visualizarValores()
            MiValor.a = MiValor.b;      //Asignación de valores
                                        entre //variables
            MiValor.c = MiValor.a;
            MiValor.visualizarValores();
            MiValor.a = MiValor.b = MiValor.c = 5000;
                                        //Asociatividad por
                                        //derecha
            MiValor.visualizarValores();
      }
}

class Valores
{
      int a = 1000;
      int b = 2000;
      int c = 3000;
      void visualizarValores()
      {
            System.out.println("El valor de a es "+a);
            System.out.println("El valor de b es "+b);
            System.out.println("El valor de c es "+c);
      }
}
```

Una vez ejecutado este programa la salida será la siguiente:

```
El valor de a es 1000
El valor de b es 2000
El valor de c es 3000
El valor de a es 2000
El valor de b es 2000
```

```
El valor de c es 2000
El valor de a es 5000
El valor de b es 5000
El valor de c es 5000
```

1.14 CREACIÓN Y EJECUCIÓN DE UN PROGRAMA EN JAVA

Java contempla los siguientes pasos para el proceso de creación y ejecución de un programa.

1. ***Escribir el programa(Código fuente):*** Para este paso se hace necesario tener un editor de texto, en el que el programador digitará y grabará las instrucciones que conforman el programa. Una vez escrito el programa se debe guardar con un nombre bajo el cual se identificará este programa. Los programas fuente deben tener el mismo nombre de la clase principal y se guardarán con la extensión .java (programa1.java)

2. ***Compilar el programa(Código fuente):*** El programa fuente escrito por el programador, no es entendido por la máquina, es por esto que se hace necesario traducir este programa a un formato entendible por la máquina. Java a través de su compilador convierte el código fuente en Byte Code, que es un código intermedio entre el código fuente y el lenguaje de máquina. Este Byte Code podrá ejecutarse en cualquier máquina independiente de la plataforma gracias al trabajo realizado por la Máquina Virtual de Java, la cual se encarga de adecuar este código a las diferentes plataformas sobre las que el programa puede ser ejecutado. Un programa compilado tendrá el mismo nombre del programa fuente pero con la extensión .class (programa1.class). Es de resaltar que si el programa hace referencia a clases de otros paquetes, estas son incluidas en el Byte Code durante el proceso de compilación.

3. ***Ejecución del programa(archivo.***class) El archivo .class es ejecutado por la Máquina Virtual de Java. Este proceso consta de tres etapas: primero el archivo .class es cargado en la memoria, luego se hace la verificación y finalmente se ejecutan cada una de las instrucciones contenidas el archivo .classs.

1.15 ERRORES EN JAVA

En los lenguajes de programación orientados a objetos el concepto de error se cambia por el de ***excepción,*** de esta manera cuando se diseña una aplicación en Java, uno de sus objetivos elementales es el de agregar el tratamiento de fallos o errores que se pueden producir durante su ejecución. "A estos errores o condiciones anormales que ocurren durante la ejecución de un segmento de código se les denomina ***excepciones"*** [3].

Son tres los tipos de errores que se pueden presentar al momento de compilar un programa en Java.

1. ***Errores de Sintáxis:*** Son aquellos que se presentan por no seguir las reglas propias del lenguaje. Entre estos se encuentran el mal uso de palabras reservadas, omisión de símbolos tales como el punto y coma al final de una instrucción, inserción de símbolos en lugares donde no deben ir, por ejemplo, la inclusión de un punto y coma al final de la definición de un método.

2. ***Errores lógicos:*** Son aquellos que se presentan por el hecho de hacer un análisis incompleto o mal hecho de un problema. Estos errores son difíciles de encontrar ya que el compilador no los detecta al momento de hacer la compilación y ejecución del programa. Entre los más frecuentes están: fórmulas de cálculo incorrectas y requerimientos mal definidos.

3. ***Errores de regresión:*** Son aquellos que se producen cuando se corrigen otros errores. Es importante que cuando se hagan correcciones sobre el programa fuente, se analicen los posibles efectos del cambio o cambios hechos para que el programa no llegue a presentar otros errores inesperados.

1.16 TIPOS DE DATOS EN JAVA

Se denominan tipos de datos al universo de valores que pueden ser asignados a una variable. Java posee ocho tipos de datos, los cuales son resumidos en la siguiente tabla:

Tipo de dato	Tamaño en bits	Rango	
		Valor mínimo permitido	Valor máximo permitido
Byte	8	-128	127
short	16	-32.768	32.767
Int	32	-2.147.483.648	2.147.843.647
Long	64	-263	263 - 1
Float	32	10^{-46}	10^{38}
double	64	10^{-324}	10^{308}
Char	16	'\0000'	'\FFFF'
Boolean	1	false	true

Como todos los lenguajes modernos de programación, Java soporta diferentes tipos de datos los cuales son utilizados para declarar variables y crear arreglos [4]

1.17 VARIABLES

Un programa en lenguaje Java, gestiona una serie de datos los cuales pueden ser fijos o variables, pueden estar también incorporados por defecto al programa o ser obtenidos en el momento que este es ejecutado. Estos datos se identifican a través de nombres simbólicos denominadas en general "variables o constantes". Estos nombres identifican espacios de memoria en los que se sitúan los datos antes de ser utilizados por el procesador en una operación elemental, tal como un cálculo aritmético, una comparación o el traslado de un dato de un lugar a otro [5]

Se denomina variable al nombre de una posición de memoria cuyo valor almacenado puede cambiar a lo largo del programa. Los nombres de las variables en java son dados por el programador, teniendo en cuenta que existen algunos símbolos del lenguaje que no pueden ser utilizados para tal fin, entre los que se encuentran los operadores matemáticos, los separadores (. , : ;), así como un conjunto de palabras reservadas del lenguaje que tienen significado propio para el compilador..

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

1.18 DECLARACIÓN DE UNA VARIABLE

Para declarar una variable es necesario especificar el tipo de datos que en ella serán almacenados y el nombre con que será reconocida dentro del programa. Su sintaxis es la siguiente:

```
Tipo_Dato  nombreVariable;
```

1.19 ÁMBITO DE UNA VARIABLE

Es indispensable declarar las variables antes de poder utilizarlas. En java las variables deben declararse ya sea dentro de una clase, al principio de un método o bloque de código o en el punto de utilización.

1.19.1 Variables declaradas dentro de una clase

Este tipo de variables recibe el nombre de ***variables miembro de la clase***. Su declaración se hace por dentro de las llaves **{}** que delimitan la clase y al mismo nivel de los métodos que pertenecen a dicha clase. Hecha la declaración de esta forma, estas variables estarán disponibles para todos los métodos de la clase. En el siguiente ejemplo podemos ver como los métodos calcularDefinitiva*(), calcularAcumulado() y calcularFaltante()* pueden hacer uso de las variables que han sido definidas dentro de la clase *Alumno*.

```
class Alumno
      {
            double Acumulado, Faltante, Definitiva;
            double nota1 = 3.5;
            double nota2 = 2.0;
```

```
        double nota3 = 1.5;
        double calcularDefinitiva()
                {
                        Definitiva = (nota1+nota2+nota3)/3;
                        return Definitiva;
                }
        double calcularAcumulado()
                {
                        Acumulado = Definitiva * 0.5;
                        return Acumulado;
                }
        double calcularFaltante()
                {
                        Faltante = (2.95 - Acumulado)*2;
                        return Faltante;
                }
}
```

1.19.2 Variables declaradas al principio de un método o bloque de código

Este tipo de variables recibe el nombre de variables locales y su ámbito o alcance está dado por el lugar donde fueron declaradas; si fueron declaradas al inicio de un bloque, serán reconocidas exclusivamente dentro de ese bloque y si fueron declaradas dentro de un método serán reconocidas exclusivamente dentro de ese método.

En el siguiente programa cuyo objetivo es calcular el valor de pago correspondiente que tiene que hacer un cliente a una cuota de un préstamo bancario, se puede observar la declaración de ***capital*** y ***cuotas*** como variables miembro de la clase ***Intereses,*** la declaración de ***intereses, pago*** y ***diasmora*** como variables locales del método ***calcularPago()*** e ***interesmora*** como variable local del bloque ***if*** perteneciente al método ***calcularPago().***

```
public class VariablesLocales
{
      public static void main(String[]args)
      {
            Intereses cliente1 = new Intereses();
            System.out.println("Valor a pagar "+cliente1.
            calcularPago());
```

```
        }
}
class Intereses
{
        double capital = 5000000; //Variable miembro de la clase
        int cuotas = 12;          //Variable miembro de la clase
        double calcularPago()
        {
                double interes = 0.015;  //Variable local al método
                double pago;             //Variable local al método
                int diasmora = 4;
                if (diasmora > 0)
                {
                        double interesmora = 0.03;  //Variable
                                                    local al bloque
                        System.out.println("Valor a pagar por
                        mora "+(diasmora*capital*interesmora)/30);
                }
                pago = (capital * interes)+(capital/cuotas);
                return pago;
        }
}
```

1.19.3 Variables declaradas en el punto donde se utilizan

Son también variables locales y son de gran utilidad en la construcción de bucles repetitivos, su alcance está limitado al bloque donde son declaradas.

En el siguiente ejemplo cuyo objetivo es convertir la distancia dada en kilómetros entre dos ciudades, se puede observar como en el método ***convertirMetros()*** se declara la variable ***metros*** justo en el punto donde es utilizada.

```
public class VariablesPunto
{
        public static void main(String[]args)
        {
                Kilometros tunjapaipa = new Kilometros();
                System.out.println("Distacia Tunja-Paipa en
                metros: "+tunjapaipa.convertirMetros());
        }
}
```

```
class Kilometros
{
      double km = 36.5;
      double convertirMetros()
      {
            double metros = km * 1000;  //Variable declarada
                                        en el punto //de
                                        utilización.
            return metros;
      }

}
```

1.20 MODIFICADORES PARA DECLARAR VARIABLES DE CLASE

Java permite los modificadores public, protected y private para las variables de clase. Las variables public tienen como ámbito cualquier clase del programa o paquete en que se encuentre. Las variables de tipo protected tienen como ámbito la clase en que fueron declaradas y en las clases derivadas de esta. Las variables de tipo private tienen como ámbito los métodos pertenecientes a la clase en que se declararon.

Las variables miembro de una clase que no tienen modificador son asumidas como variables públicas, es decir, su ámbito será todo el programa o paquete en que se encuentren. Para poder tener acceso a estas variables desde otra clase es necesario tener como argumento una referencia a un objeto de la clase en que la variable fue declarada.

En el siguiente programa cuyo objetivo es determinar el monto máximo a prestar a un cliente de un banco dependiendo de su salario, se puede ver cómo en los métodos mostrarCliente() y calcularMonto() de la clase Préstamo se pasa como argumento el objeto cliente1 de la clase Cliente; lo cual quiere decir que el método recibe como argumento un objeto de la clase Cliente con todas sus variables y métodos. Una vez recibidos estos objetos referencia, su utilización dentro del método respectivo se hace a través del operador punto. (`cliente1.cedula o cliente1.salariobasico)`

Para poder pasar objetos como argumentos hacia un determinado método, primero deben haber sido creadas sus respectivas instancias.

```
public class SistemaBancario
{
      public static void main(String[]args)
      {
          Cliente JuanPablo = new Cliente();
          Prestamo PrestamoConsumo = new Prestamo();
          System.out.println("JUAN PABLO");
          System.out.println("Salario Básico "+JuanPablo.
          salariobasico);
          System.out.println("Sueldo Neto "+JuanPablo.
          calcularSueldo());
          PrestamoConsumo.mostrarCliente(JuanPablo);
          System.out.println("Monto préstamo
          "+PrestamoConsumo.calcularMonto(JuanPablo));
      }
}
class Cliente
{
      double cedula = 7567340;
      double salariobasico = 2500000;
      double calcularSueldo()
      {
            double salud = salariobasico * 0.04;
            double pension = salariobasico * 0.03;
            double sindicato = salariobasico * 0.01;
            double sueldoneto = salariobasico-salud-
            pension-sindicato;
            return sueldoneto;
      }

}
class Prestamo
{
      double codigo = 1000;
      void mostrarCliente(Cliente cliente1)
      {
            System.out.println("Cédula: "+cliente1.cedula);
      }
      double calcularMonto(Cliente cliente1)
      {

            double monto = 2 * cliente1.salariobasico;
            return monto;
      }

}
```

1.21 CONSTANTES

Definimos una constante como un espacio de memoria que puede almacenar un tipo de dato pero cuyo valor no puede cambiar durante su existencia. Para hacer su definición se debe anteponer al tipo de datos que almacenará, la palabra *final*.

```
final tipo_datos nombre = VALOR;
```

```
class Longitud
{
	public double distanciaKm = 35.9;
	double convertirMetros()
	{
		final double FACTOR = 1000;
		return FACTOR * distanciaKm;
	}
}
```

2. OPERADORES E INSTRUCCIONES

2.1 OPERADORES DE ASIGNACIÓN

En java, la asignación de valores a una variable o de una variable a otra se hace a través de operadores de asignación, el operador de asignación más usado es el operador **=.** Mediante este operador, se le asigna a la variable de la izquierda el valor de la derecha [6].

```
Variable = valor;
Variable1 = Variable2;
Variable1 = Variable2 = Variable3;
```

En el siguiente ejemplo se pueden observar diferentes maneras de hacer uso del operador de asignación.

```
public class OperadorAsignacion
{
      public static void main(String[]args)
            {
            int operador1 = 1000;    //A operador1 se le
                                     asigna el valor//1000
            int operador2 = 2000;    //A operador2 se le
                                     asigna el valor//2000
            int operador3 = 3000;    //A operador3 se le
                                     asigna el valor//3000
            System.out.println("operador1 = "+operador1);
            System.out.println("operador2 = "+operador2);
            System.out.println("operador3 = "+operador3);
            operador1 = operador2;   //A operador1 se le
                                     asigna el valor
                                     //contenido en operador2
            operador3 = operador1;   //A operador3 se le
                                     asigna el valor
                                     //contenido en operador1
            System.out.println("operador1 = "+operador1);
            System.out.println("operador2 = "+operador2);
            System.out.println("operador3 = "+operador3);
```

```
            operador1 = operador2 = operador3 = 5000; // A
                                    operador1,
                                    //operador2 y operador3
                                    se le asigna
                                    //el valor 5000
            System.out.println("operador1 = "+operador1);
            System.out.println("operador2 = "+operador2);
            System.out.println("operador3 = "+operador3);
            }
}
```

En java es posible combinar el operador de asignación **=** con otros símbolos correspondientes a operadores aritméticos, esto permite obtener una simplificación de instrucciones en operaciones acumulativas. La siguiente tabla muestra la equivalencia de estas operaciones:

Operador	Sintaxis de la instrucción	Instrucción equivalente
+=	operador1 += operador2;	operador1 = operador1 + operador2;
-=	operador1 -= operador2;	operador1 = operador1 - operador2;
*=	operador1 *= operador2;	operador1 = operador1 * operador2;
/=	operador1 /= operador2;	operador1 = operador1 / operador2;
%=	operador1 %= operador2;	operador1 = operador1 % operador2;

2.2 OPERADORES ARITMÉTICOS

Los operadores aritméticos tienen la característica de ser operadores binarios, es decir requieren siempre de dos operandos para poderse ejecutar. Estos operadores permiten realizar las operaciones aritméticas usuales así:

Operador	Sintaxis de la instrucción	Operación Aritmética
+	operador1 = operador2 + operador3;	Suma
-	operador1 = operador2 - operador3;	Resta
*	operador1 = operador2 * operador3;	Multiplicación
/	operador1 = operador2 / operador3;	División
%	operador1 = operador2 % operador3;	Residuo de la división

2.3 OPERADORES DE IGUALDAD

Permiten la obtención de un valor lógico (true/false) a partir de la evaluación de dos expresiones.

Caso 1
expIzquierda == expDerecha;
Se obtiene como resultado **true** si *expIzquierda* es igual a *expDerecha*; en caso contrario el resultado será **false**.

Caso 2
expIzquierda != expDerecha;
Se obtiene como resultado **true** si *expIzquieda* es diferente de *expDerecha*; en caso contrario el resultado será **false**.

2.4 OPERADORES RELACIONALES

Permiten evaluar expresiones relacionales obteniendo un valor lógico (true/false) al hacer una comparación entre dos valores de una expresión.

Menor que
expIzquierda < expDerecha;
Se obtiene como resultado **true** si *expIzquierda* es menor que *expDerecha*; en caso contrario el resultado será **false**.

Mayor que
expIzquierda > expDerecha;
Se obtiene como resultado **true** si *expIzquierda* es mayor que *expDerecha*; en caso contrario el resultado será **false**.

Menor o igual que
expIzquierda <= expDerecha;
Se obtiene como resultado **true** si *expIzquierda* es menor o igual que *expDerecha*; en caso contrario el resultado será **false**.

Mayor o igual que
expIzquierda >= expDerecha;
Se obtiene como resultado **true** si *expIzquierda* es mayor o igual que *expDerecha*; en caso contrario el resultado será **false**.

2.5 OPERADORES LÓGICOS

Estos operadores se utilizan para la construcción de expresiones lógicas.

Operador Y (Conjunción)
expIzquierda **&&** expDerecha;
Se obtiene como resultado true si expIzquierda es verdadera y expDerecha es verdadera; en otros casos el resultado de su evaluación será false.

Operador O (Disyunción)
expIzquierda **||**expDerecha;
Se obtiene como resultado false si las dos expresiones (expIzquierda, expDerecha) son falsas; en los demás casos, el resultado de la evaluación será true.

Operador NO (Negación)
¡expresión;
Se obtiene como resultado el valor lógico (true/false) contrario al que posee la expresión.

2.6 OPERADORES INCREMENTALES

El operador ++ permite incrementar en uno el valor de una variable, mientras que el operador -- permite disminuir en uno el valor de la variable. La tabla siguiente muestra las forma como estos pueden ser utilizados:

Operador	Uso	Descripción
++	operador++;	Utiliza la variable y luego la incrementa.
	++operador;	Incrementa la variable y luego la utiliza.
--	operador--;	Utiliza la variable y luego la decrementa.
	--operador;	Decrementa la variable y luego la utiliza.

2.7 OPERADOR instanceof

El operador ***instanceof*** permite saber si un objeto pertenece o no a una clase dada. Es un operador binario cuya sintaxis de uso es la siguiente:

NombreObjeto instanceof NombreClase

Su uso está orientado hacia la evaluación de expresiones (verifica que el primer operando, que debe ser un objeto, pertenezca al segundo operando, que debe ser una clase) y da como resultado un valor true/ false.

En el siguiente ejemplo se puede observar su uso:

```
public class OperadorInstanceof
{
      public static void main(String[]args)
      {
            Muebles Escritorio = new Muebles();
            System.out.println(Escritorio.cuotaCredito());
            if(Escritorio instanceof Muebles)
            {
                  System.out.println("Escritorio es
                  instancia de la clase Muebles");
            }
            else
            {
                  System.out.println("Escritorio NO es
                  instancia de la clase Muebles");
            }
      }
}

class Muebles
{
      double valor = 300000;
      boolean credito = true;
      double cuotaCredito()
      {
            double valorcuota;
            final int CUOTAS = 12;
```

```
            valorcuota = valor/CUOTAS;
            return valorcuota;
      }
}
```

EJERCICIOS PROPUESTOS PARA ESTA UNIDAD

1. Cuál es el resultado final de las variables a,b,c,d,e,f,g y h al ejecutar el siguiente programa?

```
public class Operadores
{
      public static void main(String[]args)
      {
            Registro Registro1 = new Registro();
            Registro1.cambiarValores();
            Registro1.imprimirValores();
      }
}
class Registro
{
      int a,b,c;
      double d,e,f=10;
      char g,h='i';
      void cambiarValores()
      {
            a=b=c=300;
            d=5.5;
            e=28*d;
            f*=e;
            g='b';
            h++;
      }
      void imprimirValores()
      {
            System.out.println("a: "+a+" b: "+b+" c: "+c+"
            d: "+d+" e: "+e+" f: "+f+" g: "+g+" h: "+h);
      }
}
```

2. Si del método main() del anterior programa quitamos la instrucción `Registro1.cambiarValores()`;, cuál será el valor final para las variables a,b,c,d,e,f,g y h?

3. La administración de un edificio de apartamentos liquida la cuota mensual de administración de los copropietarios de la siguiente forma: Apartamentos tipo 1 cuota = 30000, tipo 2 = 40000, tipo 3 = 50000 y tipo 4 = 70000. Además para los apartamentos que tienen garaje, dicha cuota tiene un adicional de 20000. Elabore un programa en java, donde además de la clase principal, se tengan dos clases adicionales, una para Apartamentos y otra para Mensualidades. El programa debe permitir la liquidación de la cuota mensual de administración para un apartamento determinado, de un tipo dado y que posea garaje. La salida debe estar dada en dos líneas. Ejemplo:

```
CUOTA DE AMINISTRACION
APARTAMENTO: 302 VALOR: 60000
```

3. LECTURA DE DATOS DESDE EL TECLADO

Java ofrece una forma fácil de leer datos desde el teclado haciendo uso del objeto ***BufferedReader***. Este objeto cuenta con diferentes métodos que podemos utilizar dependiendo de la operación que se quiera realizar.

3.1 LECTURA DE CARACTERES

Para poder leer un caracter desde el teclado se hace necesario utilizar el método ***read(),*** cuyo objetivo es leer el caracter actual del buffer de entrada y su correspondiente sintaxis es:

```
variable = (char) System.in.read();
```

Para poder tener acceso al método ***read()*** es necesario incluir el paquete ***java.io***, lo cual demanda del programador incluir como primera línea del programa la instrucción `import java.io.*;` Este método hace que el compilador genere una excepción ***IOException*** que se hace necesario manejar adicionando la declaración ***throws*** para evitar la interrupción del programa. El siguiente ejemplo muestra la forma de uso del método ***read()*** así como el uso de ***throws***.

```
import java.io.*;
public class LecturaCaracteres
{
      public static void main(String[]args) throws
      IOException
      {

            Letras lectura1 = new Letras();
            lectura1.leerLetra();
            System.out.println("Caracter leido: "+lectura1.
            letra1);
      }
}
```

```
class Letras
{
      char letra1;
      void leerLetra() throws IOException
      {
                  System.out.println("Digite un caracter?");
                  letra1 = (char)System.in.read();
      }
}
```

3.2 LECTURA DE CADENAS

La mayoría de veces no solo necesitamos leer el carácter actual del buffer de entrada, sino que necesitamos leer toda una cadena de caracteres, tarea que no resultaría nada práctica si pretendiéramos hacerlo a través del método ***read()***. Para esta trabajo java cuenta con el método ***readLine()***, el cual permite crear un objeto tipo **String**(cadena) en el que se almacenan todos los caracteres de la línea leída. Además de ser necesario incluir el paquete java.io, el programador debe establecer el flujo de entrada, lo cual se hace colocando como primera línea del método ***main()*** la instrucción que contenga un objeto de la clase ***BufferedReader*** y la referencia a ***InputStreamReader.*** Para evitar que el programa realice acciones incontroladas es necesario el manejo de la excepción ***IOException*** adicionando el modificador ***throws IOException*** al método ***main().*** La línea leída debe ser almacenada mediante el método ***readLine()*** en una variable de tipo ***String***. El siguiente ejemplo ilustra la forma de hacer la lectura de una cadena de caracteres desde el teclado:

```
import java.io.*;
public class LecturaCadenas
{
      public static void main(String[]args)throws IOException
      {
            BufferedReader BufferedReader1 = new
            BufferedReader(new InputStreamReader(System.in));
             Cadenas Nombre = new Cadenas();
             Nombre.leerCadena(BufferedReader1);
             System.out.println("El nombre leído fue:
             "+Nombre.cadena1);
      }
}
class Cadenas
```

```
{
        String cadena1;
        void leerCadena(BufferedReader lector1)throws
        IOException
        {
                System.out.println("Digite su nombre: ");
                cadena1 = lector1.readLine();
        }
}
```

3.3 LECTURA DE DATOS NUMÉRICOS

Al igual que con los caracteres alfabéticos, los caracteres numéricos que se ingresan por teclado también son leídos como si fueran cadenas de tipo ***String***. Para poderlos almacenar en variables numéricas se hace necesario que una vez leídos se les haga la conversión en el programa.

3.3.1 Lectura de números enteros

Para convertir una cadena de caracteres numéricos en un número entero (***int***) es necesarioo:

- Convertir la cadena en un objeto de la clase envoltorio ***Integer***, a través de la función miembro estática ***valueOf***.
- Convertir el objeto de la clase envoltorio ***Integer*** en una variable de tipo ***int***, a través de la función ***intValue()***

El siguiente ejemplo cuyo objetivo es registrar y posteriormente visualizar el número de registro de un taxi dentro de una empresa de transporte, ilustra la forma de leer una cadena y convertirla en un dato de tipo entero:

```
mport java.io.*;
public class LeerEnterosTeclado
{
        public static void main(String[]args)throws IOException
        {
                BufferedReader BufferedReader1 = new
                BufferedReader(new InputStreamReader (System.in));
                 Taxis taxi1 = new Taxis();
                 taxi1.registrarTaxi(BufferedReader1);
                 System.out.println("Número interno del taxi:
                 "+taxi1.numInterno);
```

```
        }
}

class Taxis
{
        int numInterno;
        void registrarTaxi(BufferedReader Lectura)throws
        IOException
        {
                System.out.println("Registre el número interno
                del taxi: " );
                /* Conversión de una cadena leída desde el
                teclado en un dato de tipo int */
                numInterno = Integer.valueOf(Lectura.readLine()).
                intValue();
        }
}
```

3.3.2 Lectura de números con decimales

Para convertir una cadena de caracteres numéricos en un número con decimales (double) es necesario:

- Convertir la cadena en un objeto de la clase envoltorio ***Double***, a través de la función miembro estática ***valueOf***.
- Convertir el objeto de la clase envoltorio ***Double*** en una variable de tipo ***double***, a través de la función ***doubleValue()***

El siguiente ejemplo cuyo objetivo es registrar y posteriormente visualizar el valor de la cuota correspondiente al impuesto de rodamiento para un bus dentro de una empresa de transporte, ilustra la forma de leer una cadena y convertirla en un dato de tipo double:

```
import java.io.*;
public class LeerDoublesTeclado
{
        public static void main(String[]args)throws IOException
        {
                BufferedReader BufferedReader1 = new BufferedReader
                (new InputStreamReader(System.in));
```

```
            Buses Bus1 = new Buses();
            Bus1.leerImpuestoRodamiento(BufferedReader1);
            System.out.println("Valor rodamiento:
            $"+Bus1.impuestoRodamiento);
      }
}

class Buses
{
      double impuestoRodamiento;
      void leerImpuestoRodamiento(BufferedReader Lectura)
      throws IOException
      {
            System.out.println("Valor impuesto de
            rodamiento:");
            /* Conversión de una cadena leída desde el
            teclado en un dato de tipo doublé */
            impuestoRodamiento =
            Double.valueOf(Lectura.readLine()).doubleValue();
      }
}
```

3.4 CONVERSIÓN DE CADENAS EN BOOLEANOS

También se hace necesario que cadenas leídas desde teclado (Ejemplo: "SI"/"NO") deban ser almacenadas en variables de tipo ***boolean*** con valores true/false. Esta conversión se puede programar mediante un bloque ***if-else*** en la que también se utilizan los métodos ***toUpperCase()*** y ***equals(cadena)***.

El método ***toUpperCase()*** convierte una cadena de caracteres a su equivalente en caracteres alfabéticas en mayúscula.

Con el método ***equals(cadena)*** es posible comparar el contenido de una cadena***(String).***

A continuación se presenta un ejemplo en el que a través de un bloque ***if-else*** se le asignan valores a una variable de tipo ***boolean***, a partir de una cadena de caracteres leída desde el teclado. Además, este ejemplo también permite visualizar el uso de los métodos ***toUpperCase()*** y ***equals(cadena).***

```
import java.io.*;
public class LeeBooleanosTeclado
{
	public static void main(String[]args)throws IOException
	{
		BufferedReader BufferedReader1 = new BufferedReader
		(new InputStreamReader(System.in));
		Casas Casa1 = new Casas();
		Casa1.leerCasas(BufferedReader1);
		if(Casa1.garaje==true)
				System.out.println("La casa tiene
					garaje!");
		else
			System.out.println("La casa no tiene
			garaje!");
	}
}
class Casas
{
	boolean garaje;
	String auxgaraje;
	void leerCasas(BufferedReader Lectura)throws IOException
	{
		System.out.println("Garaje?  SI/NO");
		auxgaraje = Lectura.readLine();
		auxgaraje = auxgaraje.toUpperCase();
		if(auxgaraje.equals("SI"))
			garaje = true;
		else
			garaje = false;
	}
}
```

3.5 EJERCICIO DE APLICACIÓN *

El siguiente ejemplo ilustra de forma íntegra los ítems tratados en los anteriores numerales. El objetivo de este programa es calcular el valor a pagar por concepto de administración por parte de un propietario de apartamento. Además se muestran los resultados obtenidos al ejecutar el programa con datos correspondientes al número del apartamento, clase, garaje y valor total.

```
import java.io.*;
public class AdministracionEdificio
{
	public static void main(String[]args)throws IOException
	{
	BufferedReader BufferedReader1 = new BufferedReader(new
	InputStreamReader(System.in));
	Apartamentos Apto1 = new Apartamentos();
	Mensualidades Mayo = new Mensualidades();
	Apto1.leerDatosApartamento(BufferedReader1);
	System.out.println("CUOTA DE AMINISTRACION");
	System.out.println("APARTAMENTO: "+Apto1.numero +" TIPO:
	"+Apto1.tipo+ " GARAJE: "+Apto1.auxgaraje +
	" VALOR: " + Mayo.calcularadministracion(Apto1));
	}
}
class Apartamentos
{
	double numero;
	int tipo;
	boolean garaje;
	String auxgaraje;
	void leerDatosApartamento(BufferedReader entrada) throws
	NumberFormatException, IOException
	{
		System.out.println("DIGITE NUMERO DEL
		APARTAMENTO: ");
		System.out.flush();
		numero = Double.valueOf(entrada.readLine()).
		doubleValue();
		System.out.println("DIGITE EL TIPO DE
		APARTAMENTO: (1,2,3,4)");
		System.out.flush();
		tipo = Integer.valueOf(entrada.readLine()).
		intValue();
		System.out.println("EL APARTAMENTO TIENE
		GARAJE?  SI/NO ");
		System.out.flush();
		auxgaraje = entrada.readLine();
		auxgaraje = auxgaraje.toUpperCase();  //El método
		toUpperCase Convierte una cadena a MAYUSCULAS
		if (auxgaraje.equals("SI"))  //El método equals()
		pemite evaluar el contenido de una cadena
```

```
                garaje = true;
            else
                garaje = false;
        }

}
class Mensualidades
{
        double calcularadministracion(Apartamentos Apartamento1)
        {
            double valcuota;
            if(Apartamento1.tipo == 1)
                valcuota = 30000;
            else if(Apartamento1.tipo == 2)
                valcuota = 40000;
            else if(Apartamento1.tipo == 3)
                valcuota = 50000;
            else
                valcuota = 70000;
            if(Apartamento1.garaje==false)
                return valcuota;
            else
                return valcuota + 20000;
        }
}
```

```
DIGITE NUMERO DEL APARTAMENTO:
304
DIGITE EL TIPO DE APARTAMENTO: (1,2,3,4)
4
EL APARTAMENTO TIENE GARAJE?  SI/NO
SI
          CUOTA DE ADMINISTRACIÓN
  APARTAMENTO: 304.0 TIPO: 4 GARAJE: SI
              VALOR: 90000.0
```

Datos ingresados y salida después de la ejecución del programa

4. INSTRUCCIONES DE CONTROL

4.1 CONDICIONALES

4.1.1 Instrucción if

Una vez evaluada una condición determina que instrucción o instrucciones deben ejecutarse, dependiendo del resultado obtenido en la evaluación, el cual debe ser un booleano true/false. La instrucción ***if*** tiene diferentes estructuras sintácticas para su uso.

Estructura 1: Se usa cuando posterior a la evaluación de la condición se ejecuta una única instrucción.

```
if(condicion)
        Instuccion1;
```

Estructura 2: Se usa cuando posterior a la evaluación de la condición se ejecuta un único bloque de instruccioness.

```
if(condicion)
{
        Instruccion1;
        Instruccion2;
        Instruccion3;
        ……
        Instruccionn;
}
```

Estructura 3: Se usa cuando posterior a la evaluación de la condición y dependiendo del resultado de esta se ejecuta una u otra instrucción.

```
if(condicion)
        Instuccion1;
else
        Instruccion2;
```

Estructura 4: Se usa cuando posterior a la evaluación de la condición es posible ejecutar dos bloques de instrucciones.

```
if(condicion)
{
        Instuccion1;
        Instruccion2;
        ……..
        Instruccionn;
}
else
{
       Instruccion3;
        Instrucción4;
        ……..
        Instruccionq;
}
```

Estructura 5: Se usa cuando posterior a la evaluación de varias condiciones es posible ejecutar más de dos instrucciones por separado.

```
if(condicion1)
        Instruccion1;
else if(condicion2)
        Instruccion2;
else if(condicion3)
        Instruccion3;
else
        instruccionn;
```

Estructura 6: Se usa cuando posterior a la evaluación de varias condiciones es posible ejecutar más de dos bloques de instrucciones.

```
if(condicion1)
{
        Instruccion1;
        Instruccion2;
        ……..
        Instruccionn;
}
else if(condicion2)
{
        Instruccion3;
        Instruccion4;
        ……..
        Instruccionm;
}
else
{
        Instruccion5;
        Instruccion6;
        ……..
        Instruccionp;
}
```

Estructura 7: Una vez evaluada una condición a través de una instrucción ***if*(condicion)** es posible que una de las alternativas **(true/false)** a seguir sea la evaluación de otra condición.

```
if(condicion1)
{
        if(condicion2)
                Instruccion1;
        Instruccion2;
}
```

4.1.2 Operador ?

Es un operador que interactúa con tres operandos: Una **condición** que se evalúa, **Instruccion_t** que se ejecuta en caso que el operador retorne true y una **instruccion_f** que se ejecuta en caso que el operador retorne false.

```
condicion ? instruccion_t : instruccion_f;
```

Este operador se convierte en una forma de simplificar un operador if, cosa que podemos observar a continuación:

```
if (auxgaraje.equals("SI"))
      garaje = true;
else
      garaje = false;;
```

es equivalente a:

```
garaje = auxgaraje.equals("SI")? true : false;
```

4.1.3 Operador switch

El operador ***switch*** permite la simplificación de un operador ***if*** en el que se tenga que seguir la ejecución de diferentes bloques de instrucciones* una vez evaluada una condición.

```
switch(lectura)
{
        case 1 :
                instruccion1;
                instruccion2;
                break;
        case 2:
                instruccion3;
                instruccion4;
                break;
        default:
                instruccion5;
                instruccion6;
}
```

* También aplica para cuando se quiera seguir una única instrucción para cada caso.

La sentencia ***break*** que nos sirve para interrumpir la ejecución de un bloque iterativo en cualquier momento y el lenguaje Java permite esta posibilidad, no es muy recomendable ya que hace difícil la comprensión del código [7]. Dentro de la estructura de un ***swich*** la instrucción ***break*** permite separar cada uno de los casos, es decir, que una vez sea cumplida

la condición en un caso, no se siguen revisando los demás. La opción ***default*** se utiliza para ejecutar una serie de instrucciones una vez no se haya cumplido ningún caso.

El ejemplo que se da a continuación utiliza el operador ***switch*** para la construcción de una sencilla calculadora con las cuatro operaciones básicas (suma, resta, multiplicación y división).

```
import java.io.*;

public class Menu
{
	public static void main(String[]args)throws IOException
	{
		BufferedReader BufferedReader1 = new BufferedReader
		(new InputStreamReader(System.in));
		Calculadora Calculadora1 = new Calculadora();
		Calculadora1.leerTerminos(BufferedReader1);
		Calculadora1.hacerOperacion();
	}
}

class Calculadora
{
	double operando1;
	double operando2;
	char operador;
	double resultado;
	void leerTerminos(BufferedReader Lectura)throws
	IOException
	{
		System.out.println("--CALCULADORA--");
		System.out.print("Digite el primer término: ");
		System.out.flush();
		operando1 = Double.valueOf(Lectura.readLine()).
		doubleValue();
		System.out.println();
		System.out.print("Digite el segundo término: ");
		System.out.flush();
		operando2 = Double.valueOf(Lectura.readLine()).
		doubleValue();
		System.out.println();
		System.out println("=========================");
		System.out.println("SUMA              +       ");
```

```
            System.out.println("RESTA              -        ");
            System.out.println("MULTIPLICACION     *        ");
            System.out.println("DIVISION           /        ");
            System.out.println("==========================");
            System.out.print("  OPCION:  ");
            System.out.flush();
            operador = (char)System.in.read();
      }

      void hacerOperacion()
      {
            switch(operador)
            {
            case '+':
                  resultado = operando1 + operando2;
                  break;
            case '-':
                  resultado = operando1 - operando2;
                  break;
            case '*':
                  resultado = operando1 * operando2;
                  break;
            case '/':
                  resultado = operando1 / operando2;
                  break;
            default:
                  System.out.println("OPERACION NO VALIDA!");
            }
            System.out.println("RESULTADO = "+ resultado);
      }
}
```

4.2 BUCLES REPETITIVOS

En java, así como en otros lenguajes de programación, los bucles repetitivos permiten que un bloque de instrucciones sea ejecutado mientras se cumpla una condición.

4.2.1 Bucle for

En el bucle ***for*** se ejecutan una o varias instrucciones mientras se cumpla una condición que ha sido predeterminada dentro de su estructura. La estructura general de un bucle ***for*** es la siguiente:

```
for(inicializacion;condicion;incremento)
{
        instruccion_1;
        instruccion_2;
        ……..
        instruccion_n;
}
```

La inicialización corresponde al valor inicial que tomará la variable encargada de llevar el control del bucle. La condición es la expresión que se evaluará en cada iteración y determina el momento en que el bucle debe terminar. El incremento determina la variación de la variable de control cada vez que se ejecuta o itera el bucle.

El siguiente ejemplo imprime en una línea los números del 1 al 10.

```
public class Buclefor
{
      public static void main(String[]args)
      {
            for(int i = 1;i<=10;i++)
            {
                  System.out.print(i+" ");
            }
      }
}
```

Así como es posible que la variación sea de incremento, también se puede hacer que esta vaya en decremento:

```
public class BucleforDecremento
{
      public static void main(String[]argas)
      {
            int i;
            for(i=10;i>0;i--)
                  System.out.print(i+" ");
      }
}
```

4.2.1.1 Uso de la instrucción break en un blucle for

Java también permite utilizar la instrucción ***break*** para devolver el control del programa a la línea siguiente a un bucle ***for***. De esta forma se garantiza que no existan bucles infinitos. El siguiente ejemplo en el que el programa efectúa la lectura de caracteres desde el teclado y que termina cuando pulsamos el carácter 'z', ilustra el uso de la instrucción ***break***:

```
import java.io.*;
public class LeerCaracteres
{
      public static void main(String[]args)throws IOException
      {
            Entrada Entrada1 = new Entrada();
            Entrada1.leerdesdeteclado();
            System.out.println("FIN DE LA LECTURA!");
      }
}

class Entrada
{
      char caracter;
      void leerdesdeteclado()throws IOException
      {
            for(;;)
            {
                  System.out.println("Digite un caracter!");
                  caracter = (char)System.in.read();
                  if(caracter == 'z')
                        break;
            }
      }
}
```

4.2.2 Bucle while

En el bucle ***while*** las iteraciones se repiten mientras se cumpla una condición. La estructura general de un bucle ***while*** es la siguiente:

```
while(condicion)
{
        instruccion_1;
        instruccion_2;
        ……..
        instruccion_n;
}
```

Su construcción es más sencilla que la del bucle ***for***. La condición puede ser cualquier expresión o valor que pueda ser evaluado a través de un valor booleano. En el siguiente ejemplo se hace uso de un bucle ***while*** para construir una diagonal con los números desde el 10 al 1.

```
public class Diagonal
{
      public static void main(String[]args)
      {
            int i = 10;
            String espacio ="";
            while(i>0)
            {
                  System.out.println(espacio+ i);
                  espacio = espacio+" ";
                  i--;
            }
      }
}
```

4.2.3 Bucle do-while

A diferencia del bucle ***while***, el bucle ***do-while*** no hace la evaluación de la condición al inicio del bucle, sino que lo hace al final de este. La estructura general del bucle ***do-while*** es la siguiente:

```
do
{
        instruccion_1;
        instruccion_2;
        ……..
        instruccion_n;
}while(condicion);
```

El bucle ***do-while*** se repite hasta que la condición se hace falsa. Se puede observar su uso en el siguiente ejemplo, en el que una vez leído un número entero, se hace la sumatoria de cada uno de los números enteros desde el 1 hasta el número leído.

```
import java.io.*;
public class Ejemplodowhile
{
      public static void main(String[]args)throws IOException
      {
            BufferedReader BufferedReader1 = new BufferedReader
            (new InputStreamReader(System.in));
            Sumatoria Sumatoria1 = new Sumatoria();
            Sumatoria1.leerNumTerminos(BufferedReader1);
            Sumatoria1.hacerSumatoria();
            System.out.println("La sumatoria es:
            "+Sumatoria1.hacerSumatoria());
      }
}

class Sumatoria
{
      int total = 0;
      int numero;
      void leerNumTerminos(BufferedReader Lectura)throws
      IOException
      {
            System.out.println("Terminos de la sumatoria: ");
            System.out.flush();
            numero = Integer.valueOf(Lectura.readLine()).
            intValue();
      }
      int hacerSumatoria()
      {
            do
            {
                  total = total + numero;
                  numero--;
            }while(numero>0);
            return total;
      }
}
```

4.2.4 Uso de bucles anidados

En muchas ocasiones el programador debe incluir un bucle dentro de otro bucle. A este tipo de operación se conoce como anidación de bucles. Cuando esto se presenta, el compilador primero ejecuta el bucle interno, luego si sigue cumpliéndose la condición del bucle externo, entonces se vuelve a ejecutar el bucle interno hasta que su condición sea válida. El programa siguiente, que imprime la tabla que se presenta a continuación, contiene un bucle ***for*** anidado dentro de otro bucle ***for***:

```
1 2 3 4 5 6 7
2 3 4 5 6 7
3 4 5 6 7
4 5 6 7
5 6 7
6 7
7
```

```
import java.io.*;
public class MediaMatriz
{
      public static void main(String[]args)throws IOException
      {
            BufferedReader BufferedReader1 = new BufferedReader
            (new InputStreamReader(System.in));
            Tablaimpresa tablaimpresa1 = new
            Tablaimpresa();
            tablaimpresa1.leerNumero(BufferedReader1);
            tablaimpresa1.imprimirtabla();
      }
}

class Tablaimpresa
{
      int n;
      void leerNumero(BufferedReader Lectura)throws IOException
      {
            System.out.println("Digite un número entero: ");
            System.out.flush();
            n = Integer.valueOf(Lectura.readLine()).
            intValue();
      }
      void imprimirtabla()
      {
```

```
            int i, j;
            for(i=1;i<=n;i++)
            {
                  for(j=i;j<=n;j++)
                  {
                        System.out.print(j+" ");
                  }
                  System.out.println();
            }
      }
}
```

4.2.5 Instrucciones de salto

Java también proporciona instrucciones que le permiten transferir el control de un punto a otro del programa. Estas instrucciones son: ***break, continue y return***.

4.2.5.1 Instrucción break

El uso de la instrucción ***break*** tiene especial importancia en la separación de casos dentro de una instrucción ***switch***, en la que break permite separar cada uno de sus casos (Ver apartado 4.1.3) y como instrucción para forzar la salida de un bucle (Ver apartado 4.2.1.1) .

El siguiente ejemplo ilustra el uso de la instrucción break para forzar el rompimiento de un bucle ***for***. El bucle ***for*** perteneciente al método *imprimirOriginal( )* se ejecuta en su totalidad dando como resultado la impresión consecutiva de los números del 1 hasta el 50. El bucle ***for*** perteneciente al método *romperBucle( )*puede ser interrumpido al digitar un nuevo número que corresponde al valor en el cual se rompe el bucle y devuelve el control a la línea siguiente al bucle, aun cuando el límite superior del bucle sigue siendo 50.

```
import java.io.*;
public class InterrumpeBucles
{
      public static void main(String[]args) throws IOException
      {
            BufferedReader BufferedReader1 = new BufferedReader
            (new InputStreamReader(System.in));
            Ciclo CicloUno = new Ciclo();
```

```
            CicloUno.imprimirOriginal();
            CicloUno.romperBucle(BufferedReader1);
      }
}

class Ciclo
{
      int i,rompe;
      void imprimirOriginal()
      {
            for(i=1;i<=50;i++)
            {
                  System.out.println(i);
            }
            System.out.println("Fin del bucle");
      }
      void romperBucle(BufferedReader Lectura)throws IOException
      {
            System.out.println("Introduzca el número de
            rompimiento! ");rompe = Integer.valueOf
            (Lectura.readLine()).intValue();
            for(i=1;i<=50;i++)
            {
                  if(i==rompe+1)break;
                  System.out.println(i);
            }
            System.out.println("Fin del bucle");
      }
}
```

EJERCICIOS PROPUESTOS PARA ESTA UNIDAD

1. Elaborar un programa que permita leer un número entero **x** y lo eleve a la **y** potencia (x^y). Ejemplo: 3^4 = 3*3*3*3= 81

2. Elaborar un programa que genere la tabla de multiplicar para cualquiera de los números enteros entre 1 y 9. Debe contener la clase Tabla, que a su vez tendrá la variable número de tipo entero; para almacenar el número de la tabla que se quiere construir, y los métodos leerTabla( ) y construirTabla( ).

3. Elaborar un programa que genere las tablas de multiplicar para los números pares entre 1 y 20.

4. Cuál es la salida por pantalla del siguiente programa:

```
{public class Diagonal
{
      public static void main(String[]args)
      {
            int i = 10;
            String espacio ="";
            while(i>0)
            {
                  System.out.print(espacio+ i);
                  espacio = espacio+" ";
                  i--;
            }
      }
}
```

5. Elabore un programa en el que con los números del 0 al 9 y haciendo uso de un bucle while genere una figura en forma de V.

6. Haciendo uso de un bucle do-while, elabore un programa que permita calcula el factorial de un número.

7. Elabore un programa que calcule la suma de los múltiplos de tres comprendidos entre 0 y 100.

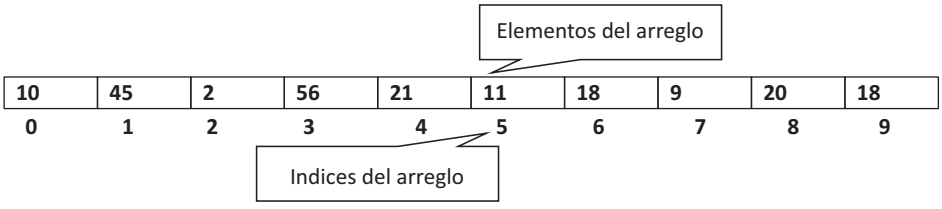
8. Elabore un programa que permita calcular la suma de los números primos comprendidos entre 0 y 100.

5. ARREGLOS

Un arreglo es una estructura de almacenamiento de un número determinado de variables primitivas o referencias a objetos, todas del mismo tipo y que cada elemento está identificado por uno o varios índices. Con los arreglos se pueden construir vectores, matrices y demás estructuras repetitivas de datos; las existencias de estas estructuras han sido de gran utilidad, solo limitadas por el hecho de que su dimensión debe estar previamente declarada y no puede ser ampliado en tiempo de ejecución [5]. Un arreglo lo podemos definir básicamente como un conjunto de datos o variables del mismo tipo referenciadas bajo un mismo nombre. Para poder acceder a cada uno de los elementos que conforman este conjunto es indispensable hacer uso de una variable de tipo entero que hace las veces de índice.

El siguiente ejemplo ilustra la definición de un arreglo y sus componentes:

Arreglo ***Numeros***



En la ilustración anterior se pueden identificar las tres características esenciales que debe contener todo arreglo:

- **Nombre**: El nombre bajo el cual se identifican cada uno de los elementos del arreglo. Para este caso ***Numeros***.

- **Elementos**: Cada una de las variables individuales agrupadas bajo el nombre ***Numeros***. Note que para este caso es un arreglo de tipo entero, por lo cual todos sus elementos son de tipo entero.

- **Índice:** Representa la posición que ocupa un elemento dentro del arreglo. Es importante no llegar a confundir la posición con el contenido dentro del arreglo; en arreglo ***Numeros*** se observa que en la posición 0 contiene 10, la posición 1 contiene 45, la posición 2 contiene 2, la posición 3 contiene 56 y así sucesivamente.

5.1 DECLARACIÓN DE UN ARREGLO

La declaración de un arreglo se hace especificando el tipo de datos que agrupará el arreglo seguido por los símbolos [ ] y el nombre del arreglo. En el ejemplo siguiente se crean dos arreglos de tipo enter:

```
int conjunto[];
int [] lista;;
```

5.2 INICIALIZACIÓN DE UN ARREGLO

La instrucción anterior permite la declaración de un arreglo, pero para poder inicializarlo, es necesario recurrir al operador new:

```
int conjunto[] = new int[10];;
```

Esta instrucción inicializa un arreglo de tipo entero compuesto por 10 elementos. Es importante puntualizar que los índices de un arreglo se inician en 0 y no en 1. Para este ejemplo los índices del arreglo van desde 0 hasta 9 y no desde 1 hasta 10.

Cuando se inicializa un arreglo de tipo entero, cada uno de sus elementos contendrá como valor predefinido 0.

5.3. REFERENCIACIÓN DE ELEMENTOS

Para poder hacer referencia a cada uno de los elementos de un arreglo es necesario utilizar el índice que representa su posición dentro del arreglo.

```
void imprimirVector()
      {
            for(int i = 9;i>=0;i--)
            {
                  System.out.print(conjunto[i]+" ");
            }
      }
```

El método anterior permite la impresión del contenido de cada uno de los elementos del arreglo ***conjunto***. Tenga en cuenta que el índice máximo del arreglo es 9 y no 10. Si se pretendiera tener acceso a *conjunto [10]* se generaría un error al momento de ejecutar el programa.

Java permite conocer la longitud de un arreglo a través de la propiedad length del arreglo, cuya sintaxis es:

NombreArreglo.***length***

Un ejemplo de su utilización es el siguiente:

```
void conocerLongitud()
	{
		for(int i=0;i<conjunto.length;i++)
		{
			System.out.print(conjunto[i]+" " );
		}
	}
```

Este método, aunque en forma ascendente, también hace la impresión del contenido de cada uno de los elementos del arreglo ***conjunto***.

El siguiente ejemplo crea un arreglo de 10 elementos de tipo entero y lo llena con los números del 1 al 10 a través de un bucle for. También con un bucle for, se imprime el contenido de cada uno de los elementos del arreglo ***Lista***. En esta impresión es posible ver el comportamiento de la variable que hace de índice así como su contenido.

```
import java.io.*;
public class Arreglos
{
	public static void main(String[]args)throws IOException
	{
		Vectores vector1 = new Vectores();
		vector1.llenarVector();
		vector1.imprimirVector();
		System.out.println();
		System.out.println("Finalizó el programa!");
	}
}
```

```
class Vectores
{
	int Lista[] = new int[10];
	void llenarVector()
	{
		for(int i = 0;i<10;i++)
		{
			Lista[i]=i+1;
		}
	}
	void imprimirVector()
	{
		for(int i = 0;i<Lista.length;i++)
		{
			System.out.println("Lista ["+i+"] =
			"+Lista[i]+" ");
		}
	}
}
```

Al ejecutar el programa, su salida por consola es la siguiente:

```
Lista [0] = 1
Lista [1] = 2
Lista [2] = 3
Lista [3] = 4
Lista [4] = 5
Lista [5] = 6
Lista [6] = 7
Lista [7] = 8
Lista [8] = 9
Lista [9] = 10

¡Finalizó el programa!
```

5.4 INICIALIZACIÓN DE ARREGLOS INDETERMINADOS

En muchas oportunidades, debido a que se hace dispendioso conocer la longitud de un arreglo, se hace necesario definir e inicializar arreglos de tamaño indeterminado. Java calcula automáticamente el tamaño del arreglo dependiendo de los valores que lo conforman. Considere el siguiente ejemplo en el que se desea construir un arreglo para almacenar códigos tipo carácter para diferentes mensajes correspondientes a fallos que se presentan en un sistema:

```
public class CodigosFalla
{

      public static void main(String[]args)
      {
            char letras[] = {'a', 'b','c','d'};
            for(int i = 0;i<letras.length;i++)
                  System.out.println(letras[i]);
      }
}
```

Se puede observar que la definición y el llenado del arreglo se hacen en una misma línea de código y sin utilizar la palabra reservada ***new***.

5.5 COPIAR UN ARREGLO EN OTRO

Es posible que necesitemos manipular los elementos contenidos en un arreglo pero conservando una copia de sus valores originales. En este caso lo más lógico es crear un arreglo del mismo tipo y tamaño que el original y crear un algoritmo que nos permita pasar uno a uno los elementos contenidos en el vector inicial hacia un vector destino, con lo cual, tendríamos lo siguiente:

```
import java.io.*;
public class CopiaArreglos
{
      public static void main(String[]args)
      {
            Pares pares1 = new Pares();
            System.out.println("Arreglos originales!");
            pares1.imprimirListas();
            pares1.trasladarLista();
            System.out.println("Arreglos modificados!");
            pares1.imprimirListas();

      }
}

class Pares
{
      int lista1 [] ={2,4,6,8,10};
      int lista2 [] = new int[5];
      void trasladarLista()
      {
```

```
            int i;
            for(i=0;i<lista1.length;i++)
                  lista2[i]=lista1[i];
      }
      void imprimirListas()
      {
            System.out.println();
            for(int i=0;i<lista1.length;i++)
                  System.out.println("Lista1 ["+i+"] =
                  "+lista1[i]);
            System.out.println();
            for(int i=0;i<lista2.length;i++)
                  System.out.println("Lista2 ["+i+"] =
                  "+lista2[i]);
      }
}
```

Pero teniendo en cuenta que estamos incursionando en la Programación Orientada a Objetos, donde una de sus principios es la reutilización, es bueno que hagamos uso del paquete del sistema, en donde se nos ofrece un método para copiar un arreglo y cuya sintaxis general es la siguiente:

```
System.arraycopy(arregloOri,iniarregloOri,arregloDest,iniarregloDest,numelementosCopiar);
```

Donde ***arregloOri*** corresponde al nombre del arreglo origen, ***iniarregloOri*** corresponde a la posición del arreglo origen desde la que se inicia la copia, ***arregloDest*** corresponde al nombre del arreglo en el que se hará la copia*, ***iniarregloDest*** corresponde a la posición del arreglo destino en que se inicia la copia y ***numelementosCopiar*** corresponde al número de elementos del arreglo origen que serán copiados en el arreglo destino.

* Es importante tener en cuenta que si se quiere hacer una copia total del arreglo origen, los dos arreglos deben ser del mismo tipo y el arreglo destino debe tener como mínimo la misma cantidad de elementos que el arreglo origen.

Aplicando esta sintaxis, el programa anterior queda transformado de la siguiente forma:

```
import java.io.*;
public class CopiaArreglo1
{
	public static void main(String[]args)
	{
		OtrosPares otrosPares1 = new OtrosPares();
		//IMPRIMIR ARREGLOS INICIALES
		otrosPares1.imprimirLista();
		otrosPares1.trasladarLista();
		System.out.println();
		//IMPRIMIR ARREGLOS DESPUES DE LA COPIA
		otrosPares1.imprimirLista();
	}
}
class OtrosPares
{
	int lista1[] = {12,14,16,18,20};
	int lista2[] = new int[5];
	void trasladarLista()
	{
		System.arraycopy(lista1,0,lista2,0,lista1.length);
	}
	void imprimirLista()
	{
		for(int i=0;i<lista1.length;i++)
			System.out.println("lista1 ["+i+"] =
			"+lista1[i]);
		System.out.println();
		for(int i=0;i<lista2.length;i++)
			System.out.println("lista2 ["+i+"] =
			"+lista2[i]);
	}
}
```

Con lo cual la salida por consola estaría dada de la siguiente forma:

Impresión de los arreglos antes de hacer la copia de lista1 en lista2:

```
lista1 [0] = 12                    lista2 [0] = 0
lista1 [1] = 14                    lista2 [1] = 0
lista1 [2] = 16                    lista2 [2] = 0
lista1 [3] = 18                    lista2 [3] = 0
lista1 [4] = 20                    lista2 [4] = 0
```

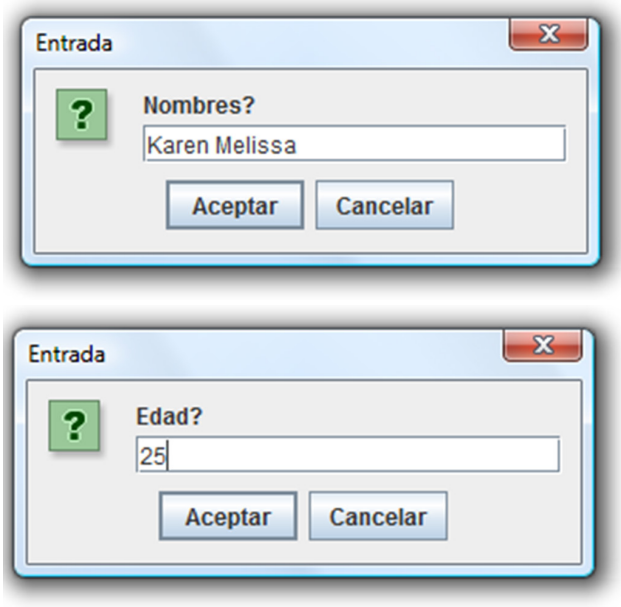
y la impresión de los arreglos luego de copiar lista1 en lista2:

```
lista1 [0] = 12                    lista2 [0] = 12
lista1 [1] = 14                    lista2 [1] = 14
lista1 [2] = 16                    lista2 [2] = 16
lista1 [3] = 18                    lista2 [3] = 18
lista1 [4] = 20                    lista2 [4] = 20
```

En java el método **arraycopy( )** ofrece una forma rápida y fácil para copiar un arreglo de cualquier tipo en otro, pero es importante tener en cuenta que este método nos puede generar las siguientes excepciones:

`NullPointerException:`	Cuando alguno de los arreglos es nulo, es decir, no ha sido inicializado.
`ArrayStoreException:`	Cuando se intenta copiar en un arreglo de diferente tipo.
`IndexOutOfBoundsException:`	Cuando si intenta copiar fuera del área reservada para el arreglo.

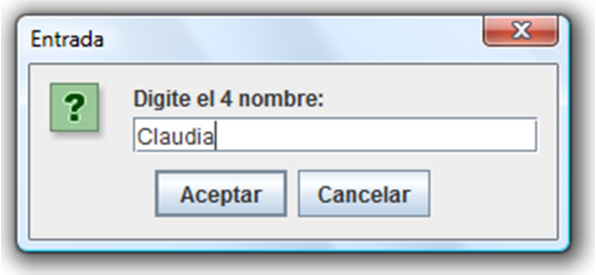
5.6 CAJAS DE DIALOGO PARA INGRESAR DATOS



Ejemplo de programa para ingresar datos a un arreglo:

```
import javax.swing.JOptionPane;

public class ArreglosChar
{
	public static void main(String[]args)
	{
		FallaSistema fs1 = new FallaSistema();
		fs1.llenarVector();
		fs1.imprimirVector();
	}
}

class FallaSistema
{
	String Falla[] = new String[10];
	void llenarVector()
	{
		for(int i = 0;i<Falla.length;i++)
		{
			Falla[i]=JOptionPane.showInputDialog
			("Digite el "+i+" nombre: ");
		}
	}
	void imprimirVector()
	{
		for(int i = 0;i<Falla.length;i++)
		{
			System.out.print(Falla[i]+" ");
		}
	}
}
```

Donde la caja de diálogo a través de la cual se ingresan los datos desde el teclado al arreglo es la siguiente:



Y cuya salida por consola una vez ejecutado el programa puede ser similar a la siguiente:

```
María José Paula Hernán Claudia Nicolás Rocío Juan Manuel
Fernanda
```

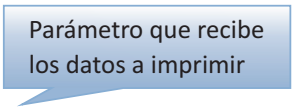
5.7. ARREGLOS COMO PARÁMETROS

En java es posible pasar un arreglo completo (con todos sus elementos) a un método. Una vez en el método sus elementos podrán ser utilizados para realizar diferentes operaciones. A continuación se ilustra mediante un ejemplo su uso; se implementa el método ***imprimirDatos()*** que recibe como parámetro un arreglo de tipo entero e imprime cada uno de sus valores.

```
package invertirarreglo;

public class Main {
        public static void main(String[] args) {
                Arreglo miArreglo = new Arreglo();
                miArreglo.leerDatos();
                miArreglo.imprimirDatos(miArreglo.A);
                miArreglo.invertirDatos();
                miArreglo.imprimirDatos(miArreglo.B);
        }
}

package invertirarreglo;
import javax.swing.JOptionPane;

public class Arreglo {
        int longitudArreglo = Integer.parseInt(JOptionPane.
        showInputDialog("Digite la longitud del arrego: "));
        int A[] = new int[longitudArreglo];
        int B[] = new int[longitudArreglo];

void leerDatos() {
        for (int i = 0; i < A.length; i++) {
                A[i] = Integer.parseInt(JOptionPane.showInputDialog
                ("Digite el valor de A[" + i + "] : "));
        }
}
```

```
void imprimirDatos(int[] Impresion) {
        for (int i = 0; i < Impresion.length; i++) {
                System.out.println("Valor[" + i + "] = " + Impresion[i]);
        }
}

void invertirDatos() {
        for (int i = (A.length - 1); i >= 0; i--) {
                        B[(A.length - 1) - i] = A[i];
                }
        }
}
```

En el ejemplo anterior, podemos observar que el método imprimirDatos(), recibe como parámetro un arreglo de tipo entero. A través de este parámetro el método recibe los datos del arreglo que se desea imprimir.




```
void imprimirDatos(int[] Impresion)
```

5.8 ORDENAR UN ARREGLO

Es importante que una vez ingresada una cantidad de elementos en un arreglo los podamos ordenar y visualizar. Podemos hacerlo desarrollando nuestro propio algoritmo o haciendo uso el método **Arrays.sort( )** contenido en el paquete *java.util.Arrays* y cuya sintaxis es la siguiente:

```
Arrays.sort(nombreArreglo);
```

A continuación se presenta el programa anterior, modificado para hacer el ordenamiento de un arreglo a través del método **Arrays.sort( )**.

```
import java.util.Arrays;
import javax.swing.JOptionPane;

public class ArreglosChar
{
```

```
	public static void main(String[]args)
	{
		FallaSistema fs1 = new FallaSistema();
		fs1.llenarVector();
		fs1.imprimirVector();
	}
}

class FallaSistema
{
	String Falla[] = new String[10];
	void llenarVector()
	{
		for(int i = 0;i<Falla.length;i++)
		{
			Falla[i]=JOptionPane.showInputDialog
			("Digite el "+i+" nombre: ");
		}
	}
	void imprimirVector()
	{
		Arrays.sort(Falla);
		for(int i = 0;i<Falla.length;i++)
		{
			System.out.print(Falla[i]+" ");
		}
	}
}
```

5.9 ARREGLOS MULTIDIMENSIONALES

Un arreglo multidimensional es un arreglo de arreglos y se caracterizan por necesitar más de un índice para acceder a sus elementos, los más usados son las **matrices** o **tablas**, en los que el acceso se hace a través de dos índices: uno que nos ubica en la fila y otro que nos ubica en la columna[]. Para hacer referencia a un objeto se hace dando el nombre de la matriz y la posición de intersección fila-columna, cuya forma de referencia es la siguiente:

```
Matriz[fila][columna]
```

La siguiente figura ilustra una matriz de 3 filas por 3 columnas con sus respectivos índices.

Matriz M:

M[0][1]	M[0][1]	M[0][2]
M[1][0]	M[1][1]	M[1][2]
M[2][0]	M[2][1]	M[2][2]

5.9.1 DECLARACIÓN DE UNA MATRIZ

La forma de declarar una matriz es similar a la de declarar un arreglo, pero con la diferencia que se adicionan un par de corchetes más en los que se indica el número de columnas que contendrá dicha matriz.

```
int M[][] = new int[3][3];
```

5.9.2 INICIALIZACIÓN DE UNA MATRIZ

De la misma forma que es posible inicializar arreglos, también se pueden inicializar las matrices

```
Int M[][] =
{
        {1,2,3},
        {4,5,6},
        {7,8,9}
};
```

Una vez inicializada la matriz, el acceso a cada uno de sus elementos se hace a través del nombre de la matriz y los correspondientes índices del elemento. Por ejemplo la línea de código ***System.out.println(M[0][2]0;***, arrojará como resultado por consola el número 3, que es el valor almacenado en la posición (0,2) de la matriz **M**.

El siguiente programa muestra detalladamente la forma como se crea, inicializa y se tiene acceso a los elementos de una matriz. El método **imprimirMatriz()** utiliza los índices **i** (para manejar el acceso entre filas) y **j** (para manejar el acceso entre columnas).

```
public class Multidimensionales
{
	public static void main(String[]args)
	{
		Matrices matriz1 = new Matrices();
		matriz1.imprimirMatriz();
	}
}
class Matrices
{
	int i,j,k;
	int M[][]=
		{
				{1,2,3},
				{4,5,6},
				{7,8,9}
		};
	void imprimirMatriz()
	{
		for(i=0;i<3;i++)
		{
			for(j=0;j<3;j++)
			{
				System.out.print(M[i][j]+" ");
			}
			System.out.println();
		}
	}
}
```

Al ejecutar el programa, su salida por consola será:

```
1 2 3
4 5 6
7 8 9
```

También es posible acceder a cada una de las posiciones de memoria de una matriz para asignar valores, tarea que también se puede hacer a través de dos bucles que manejen sus respectivos índices. El siguiente ejemplo crea una matriz **Tablas[9][10]** para almacenar las tablas de multiplicación desde el 1 hasta el 9.

```
public class MatTablas
{
	public static void main(String[]args)
	{
		Multiplicacion multiplicacion1 = new
		Multiplicacion();
		multiplicacion1.inicializarMatriz();
		multiplicacion1.imprimirMatriz();
	}

}


class Multiplicacion
{
	int i,j,k;
	int MatrizMultiplicacion[][];
	void inicializarMatriz()
	{
		MatrizMultiplicacion = new int [9][10];
		for(i=0;i<9;i++)
		{
			for(j=0;j<10;j++)
			{
				MatrizMultiplicacion[i][j]=
				(i+1)*(j+1);
			}
		}
	}
	void imprimirMatriz()
	{
		for(i=0;i<9;i++)
		{
			for(j=0;j<10;j++)
			{
				System.out.print
				(MatrizMultiplicacion[i][j]+" ");
			}
			System.out.println();
		}
	}
}
```

5.9.3 MATRICES IRREGULARES

No siempre los arreglos bidimensionales son regulares, algunas veces es necesario definir un arreglo bidimensional donde no todos sus arreglos son del mismo tamaño, tal es el caso que se presenta en la siguiente figura.

A	B		
A	B	C	D
A	B	C	
A			

Esto es posible gracias a que cuando se reserva memoria para una matriz, solo es necesario especificar la cantidad de memoria para la primera dimensión. Posteriormente es posible especificar la cantidad de memoria requerida para las demás dimensiones. A continuación se presenta un programa en el que el tamaño de la segunda dimensión no está determinado al momento de definir la primera dimensión.

```
public class MatrizIndeterminada
{
      public static void main(String[]args)
      {
            Indeterminada indeterminada1 = new
            Indeterminada();
            indeterminada1.completarMatriz();
            indeterminada1.llenarMatriz();
            indeterminada1.imprimirMatriz();
      }
}

class Indeterminada
{
      char M[][] = new char[4][];
      void completarMatriz()
      {
            M[0]=new char[2];
            M[1]=new char[4];
            M[2]=new char[3];
            M[3]=new char[1];
      }
```

```
        void llenarMatriz()
        {
                for(int i=0;i<4;i++)
                {
                        for(int j=0;j<M[i].length;j++)
                        {
                                if(j==0)
                                        M[i][j]= 'A';
                                else if(j==1)
                                        M[i][j]='B';
                                else if(j==2)
                                        M[i][j]='C';
                                else
                                        M[i][j]='D';
                        }
                }
        }
        void imprimirMatriz()
        {
                for(int i=0;i<4;i++)
                {
                        for(int j=0;j<M[i].length;j++)
                        {
                                System.out.print(M[i][j]+" ");
                        }
                        System.out.println();
                }
        }
}
```

La salida generada luego de ejecutar el programa será:

A B
A B C D
A B C
A

Quizá sea creamos que es poca e incluso nula la aplicabilidad de este tipo de matrices. No obstante, su uso está expandido a muchas situaciones, tal es el caso de las matrices dispersas, en las que sabemos que no se hace uso de todos sus elementos.

Una de las formas más comunes de darle uso a la información hoy en día es presentándola en matrices conformadas por filas y columnas en las cuales se introducen, editan, procesan y visualizan datos. A través de ellas es posible presentar desde informes sencillos de gastos hasta cálculos financieros y análisis matemáticos y estadísticos muy complejos.

Entre las aplicaciones típicas de las matrices están los presupuestos, registro de ingresos, proyecciones, gastos, planificaciones de producción, juegos y hojas electrónicas entre muchos otros.

5.9.4 ARREGLOS TRIDIMENSIONALES

En muchas ocasiones es necesario declarar arreglos de más de dos dimensiones y entre los que más uso tienen están los arreglos tridimensionales, los cuales deben ser accedidos mediante tres índices. El siguiente programa crea un arreglo tridimensional y le asigna a cada elemento un valor correspondiente a la suma de los tres índices que permiten su acceso.

```
public class MatrizTridimensional
{
	public static void main(String[]args)
	{
		Cubo cubo1 = new Cubo();
		cubo1.llenarNum();
		cubo1.imprimirCubo();
	}

}
class Cubo
{
	int numsumpro[][][]=new int [3][3][3];
	void llenarNum()
	{
		for(int i=0;i<numsumpro.length;i++)
		{
			for(int j=0;j<3;j++)
			{
				for(int k=0;k<3;k++)
				{
					numsumpro[i][j][k]=i+j+k;
				}
			}
		}
```

```
        }
        void imprimirCubo()
        {
                for(int i=0;i<numsumpro.length;i++)
                {
                        for(int j=0;j<3;j++)
                        {
                                for(int k=0;k<3;k++)
                                {
                                        System.out.print(numsumpro[i]
                                        [j][k]+ "   ");
                                }
                                System.out.println();
                        }
                        System.out.println();
                }
        }
}
```

EJERCICIOS PROPUESTOS PARA ESTA UNIDAD

1. Cree un programa que contenga un arreglo que permita almacenar los siguientes números: 23, 24, 25, 26, 67, 90, 67, 32. Liste por consola el contenido del arreglo.
2. Elabore un programa que lea desde teclado y a través de una caja de diálogo los datos para un arreglo de tipo entero y tamaño 10, luego copie los datos de este arreglo en otro arreglo del mismo tipo y tamaño e imprímalos.
3. Elaborar un programa que inicialice dos arreglos te tipo entero y tamaño 7. El primero contiene los números desde el 1 hasta el 7 y el segundo contiene los números desde el 8 al 14. Una vez inicializados los arreglos copie los contenidos de las posiciones 0,1 y 2 del primer arreglo a partir de la segunda posición del segundo arreglo. (Haga uso del método ***arraycopy()***).
4. Cuál es la salida por consola al ejecutar el siguiente programa?
5. Elabore un programa que genere la siguiente salida.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

6. ARRAYS DINÁMICOS

Un array dinámico (arreglo dinámico), es un arreglo de elementos que crece o decrece dinámicamente conforme se adicionan o eliminan elementos. Por lo general esta clase es suministrada como librería estándar por muchos lenguajes de programación, entre ellos java.

Si creamos un arreglo normal de 10 posiciones, no será posible para este arreglo adicionar una nueva posición una vez el arreglo haya sido utilizado en su totalidad. Por lo tanto, si queremos agregar más elementos, tendríamos que crear un arreglo de mayor tamaño y copiar allí inicialmente los elementos que tenemos en nuestro arreglo inicial y posteriormente agregar los nuevos elementos. En la figura siguiente podemos ver de forma gráfica el proceso de adicionar más elementos a un arreglo de tipo entero y de tamaño 10 al cual deseamos agregarle 5 posiciones más.

Arreglo A

1	2	3	4	5	6	7	8	9	10

Arreglo B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Copiando el Arreglo A en el Arreglo B:

Arreglo B

1	2	3	4	5	6	7	8	9	10	0	0	0	0	0

Se tendrían 5 posiciones disponibles para agregar nuevos elementos. Para el caso del Arreglo B, estas posiciones disponibles están llenas con el valor 0 ya que el Arreglo B es de tipo entero.

Al pasar del ejemplo gráfico a un ejemplo codificado en lenguaje de programación java quedaría así:

```
//Clase ArregloFijo

public class ArregloFijo {
   int ArregloA[]=new int[10];
   int ArregloB[]=new int[15];

   public void llenarArreglo()
   {
      for(int i=0;i<ArregloA.length;i++)
      {
         ArregloA[i]=i+1;
      }
   }

   public void adicionarElemento(int elemento, int posicion)
   {
      ArregloA[posicion]=elemento;
   }

   public void imprimirArreglo()
   {
      for(int i=0;i<ArregloA.length;i++)
      {
         System.out.print(ArregloA[i]+" ");
      }
   }

}
```

La clase **ArregloFijo** contiene dos arreglos de tipo entero y longitud fija: **ArregloA[]** y **Arreglo[B]** y los métodos **llenarArreglo()** en el cual se asignan los valores del uno al diez al **ArregloA[]**, el método **adicionarElemento(int elemento, int posición)** que recibe como parámetros un elemento y una posición con los que se le asigna al **ArregloA** en la posición recibida el elemento **recibido** y el método **imprimirArreglo[]** el cual haciendo un recorrido a través del arreglo imprime cada uno de sus elementos.

A continuación en la clase Main se crea el objeto ArregloFijoA de tipo ArregloFijo y de inmediato se procede a llamar al método llenarArreglo con el que se asignan valores a cada una de las posiciones del arreglo, luego se imprime, se le adiciona un valor a la posición cero del arreglo y se vuelve a imprimir el arreglo una vez hecho el cambio en arreglo.

```
/Clase Main

public class Main {

   public static void main(String[] args) {
     ArregloFijo A = new ArregloFijo();
     A.llenarArreglo();
     A.imprimirArreglo();
     A.adicionarElemento(15,0);
     System.out.println("Después de hacer un cambio: ");
     A.imprimirArreglo();
   }

}
```

Si miramos en la Clase Main la línea que aparece en negrita (**A.adicionarElemento(15,0);**), esta línea de código funcionará siempre y cuando no agreguemos un valor para posiciones mayores a las dadas en el arreglo inicial. El método *adicioanrElemento()* de la clase ***ArregloFijo*** contiene dos parámetros de tipo entero: el primero equivale al valor numérico que se quiere incorporar en el vector y el segundo hace referencia a la posición en la cual se ingresará dicho valor (Para nuestro caso no puede ser superior a 9 que es el índice máximo del arreglo).

Ejecutando el programa anterior, la salida para el programa sería:

```
run:
1 2 3 4 5 6 7 8 9 10
Después de hacer un cambio:
15 2 3 4 5 6 7 8 9 10 BUILD SUCCESSFUL (total time: 2 seconds).
```

Pero si lo ejecutásemos llamando al método adicionarElemento de la siguiente forma: **A.adicionarElemento(15,10);**

La salida nos producirá será:

```
run:
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
1 2 3 4 5 6 7 8 9 10        at ArregloFijo.adicionarElemento(ArregloFijo.java:16)
          at Main.main(Main.java:8)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

Esto debido a que no es posible adicionar nuevos elementos (en este caso la posición 10) a un arreglo que se ha definido de forma fija. Por lo tanto como ya se había mencionado anteriormente, la solución será crear o utilizar un arreglo del mismo tipo y de mayor tamaño para poder así adicionar un nuevo elemento.

Haciendo las siguientes modificaciones al programa inicial:

```
//Clase Arreglo Fijo:

public class ArregloFijo {
   int ArregloA[]=new int[10];
   int ArregloB[]=new int[15];

   public void llenarArreglo()
   {
      for(int i=0;i<ArregloA.length;i++)
      {
         ArregloA[i]=i+1;
      }
   }

   public void adicionarElemento(int elemento, int posicion)
   {
      ArregloA[posicion]=elemento;
   }

   public void imprimirArreglo(int Arreglo[])
   {
      for(int i=0;i<Arreglo.length;i++)
      {
         System.out.print(Arreglo[i]+" ");
      }
   }

   public void copiarArreglo()
   {
      for (int i=0;i<ArregloA.length;i++)
      {
         ArregloB[i]=ArregloA[i];
      }
   }
```

///Clase Main:

```
public class Main {

   public static void main(String[] args) {
     ArregloFijo A = new ArregloFijo();
     A.llenarArreglo();
     A.imprimirArreglo(A.ArregloA);
     A.adicionarElemento(11,0);
     System.out.println();
     System.out.println("Después de hacer un cambio: ");
     A.imprimirArreglo(A.ArregloA);
     A.copiarArreglo();
     System.out.println();
     System.out.println("Impresión del arreglo B: ");
     A.imprimirArreglo(A.ArregloB);

   }

}
```

Y ejecutando nuevamente el programa la salida será:

```
run:
1 2 3 4 5 6 7 8 9 10
Después de hacer un cambio:
11 2 3 4 5 6 7 8 9 10
Impresión del arreglo B:
11 2 3 4 5 6 7 8 9 10 0 0 0 0 0 BUILD SUCCESSFUL (total time: 1 second)
```

Ahora se modificará el programa nuevamente de tal forma que se pueda adicionar un Nuevo elemento al ArregloB que es donde hemos copiado todos los elementos del ArregloA:

//Clase ArregloFijo:

```
public class ArregloFijo {
   int ArregloA[]=new int[10];
   int ArregloB[]=new int[15];
```

```
   public void llenarArreglo()
   {
      for(int i=0;i<ArregloA.length;i++)
      {
         ArregloA[i]=i+1;
      }
   }

   public void adicionarElemento(int elemento, int posicion,int Arreglo[])
   {
      Arreglo[posicion]=elemento;
   }

   public void imprimirArreglo(int Arreglo[])
   {
      for(int i=0;i<Arreglo.length;i++)
      {
         System.out.print(Arreglo[i]+" ");
      }
   }

   public void copiarArreglo()
   {
      for (int i=0;i<ArregloA.length;i++)
      {
         ArregloB[i]=ArregloA[i];
      }
   }

}

//Clase Main:

public class Main {

   public static void main(String[] args) {
      ArregloFijo A = new ArregloFijo();
      A.llenarArreglo();
      A.imprimirArreglo(A.ArregloA);
      A.adicionarElemento(11,0,A.ArregloA);
      System.out.println();
```

```
      System.out.println("Después de hacer un cambio: ");
      A.imprimirArreglo(A.ArregloA);
      A.copiarArreglo();
      System.out.println();
      System.out.println("Impresión del arreglo B: ");
      A.imprimirArreglo(A.ArregloB);
      A.adicionarElemento(12, 10, A.ArregloB);
      System.out.println();
      System.out.println("Impresión del arreglo B: una vez se ha modificado");
      A.imprimirArreglo(A.ArregloB);
    }

}
```

Y ejecutando nuevamente el programa la salida será:

```
run:
1 2 3 4 5 6 7 8 9 10
Después de hacer un cambio:
11 2 3 4 5 6 7 8 9 10
Impresión del arreglo B:
11 2 3 4 5 6 7 8 9 10 0 0 0 0 0
Impresión del arreglo B: una vez se ha modificado
11 2 3 4 5 6 7 8 9 10 12 0 0 0 0 BUILD SUCCESSFUL (total time: 1 second)
```

Como podemos ver, esto es costoso en términos de trabajo del programador y llegará el momento (para nuestro caso cuando el ArregloB tenga ocupadas sus 15 posiciones) en que tendremos que hacer modificaciones nuevamente al programa.

6.1 LA CLASE ARRAYLIST

En Java, ArrayList es una clase que permite almacenar de forma dinámica diferentes tipos de datos u objetos sin necesidad de declarar el tamaño al momento de su creación como sí se hace cuando creamos un Array. Su aplicación es muy variada, pero sobre todo se usan para el manejo de estructuras dinámicas como Colas, Pilas, Arboles y Grafos.

6.2 PRINCIPALES MÉTODOS DE LA CLASE ARRAYLIST

Método add: Permite añadir un nuevo elemento al ArrayList. Su sintáxis es la siguiente:

```
nombreArrayList.add("Elemento nuevo");
```

Adiciona un nuevo elemento al final del ArrayList.

```
nombreArrayList.add(posición,"Elemento nuevo");
```

Adiciona un nuevo elemento en una posición dada del ArraList.

Método size: Permite obtener el tamaño del ArrayList en un momento dado.

```
nombreArrayList.size();
```

Método get: Permite conocer el elemento que se encuentra en una posición dada del ArrayLis

```
nombreArrayList.get(posición);
```

Método contains: Permite conocer si existe dentro del ArrayList un elemento dado y que es pasado como parámetro del método.

```
nombreArrayList.contains("Elemento");
```

Método indexOf: Permite conocer la posición de la primera ocurrencia del elemento que es pasado como parámetro.

```
nombreArrayList.indexOf("Elemento");
```

Método lastIndexOf: Permite conocer la posición de la última ocurrencia del elemento que es pasado como parámetro.

```
nombreArrayList.lastIndexOf("Elemento");
```

Método remove: Permite eliminar elementos del ArrayList.

```
nombreArrayList.remove("Elemento");
```

Elimina del ArrayList el elemento que fue pasado como parámetro.

```
nombreArrayList.remove(posición);
```

Elimina del ArrayList el elemento cuya posición es pasada como parámetro.

Método clear: Elimina todos los elementos de un ArrayList dado.

```
nombreArrayList.clear();
```

Método isEmpty: Este método nos permite saber si un ArrayList está vacío o no. Si está vacío devuelve true o sino false.

```
nombreArrayList.isEmpty();
```

7. OBJETOS Y CLASES

Este capítulo pretende brindarle al estudiante una introducción a los principales conceptos de la **POO** (Programación Orientada a Objetos), la cual presenta un enfoque muy diferente de la Programación Estructurada que se maneja en otros lenguajes de programación. Para aquellos que han venido trabajando en el paradigma de la programación estructurada, el cambio no será fácil, pero es importante que sepan que si quieren ser competitivos en el mundo de Java, es indispensable tener muy claros los conceptos de la **POO**.

7.1 CONCEPTOS

7.1.1 CLASE

Una clase es un conjunto de datos y métodos que operan sobre estos datos. "En su forma más sencilla, una clase se define con la palabra reservada ***class,*** seguida del nombre elegido para la clase y un bloque de definición, que se delimita haciendo uso de las llaves {}, estas llaves son muy importantes ya que dentro de ellas van los detalles descriptivos de la clase" [8]. Su definición se hace de la siguiente forma:

```
[public] class NombreClase
{
        tipodato nombreDato1;
        tipodato nombreDato2;
        tipodato nombreDaton;
        tipoRetorno nombreMetodo(argumento1, argumenton)
        {
                Instrucciones del método;
        }
}
```

La palabra ***public*** es opcional; si no se antepone a la palabra ***class***, de todas formas el compilador asume que la clase es pública y tendrá visibilidad para las demás clases del mismo paquete.

La palabra ***class*** es una palabra reservada que se utiliza siempre que se quiera definir una clase.

NombreClase corresponde al nombre bajo el cual agruparemos variables (datos) y métodos). Todas las variables y métodos deben declararse dentro del bloque de la clase { … }.

No siempre es necesario que una clase tenga incorporado uno o varios métodos. Los métodos contienen las instrucciones (operaciones) que se pueden realizar para una clase.

7.1.2 OBJETO

Luego de que una clase ha sido definida, un programa puede contener una ***instancia*** de la clase, denominada ***objeto de la clase.*** Un objeto se crea con el operador **new** aplicado a un constructor de la clase [9]. De esta manera un objeto o instancia es un ejemplo concreto de una clase. Por ejemplo si tenemos la clase Estudiante, un objeto o instancia de esta clase será Roberto, el cual estará conformado por las variables y métodos que componen la clase. La clase será un tipo de datos y el objeto una variable en particular.

```
NombreClase Objeto;
```

Por ejemplo:

Estudiante Roberto;

Cuando se habla de objetos es importante tener en cuenta tres características indispensables: identificación, funcionalidad y estado.

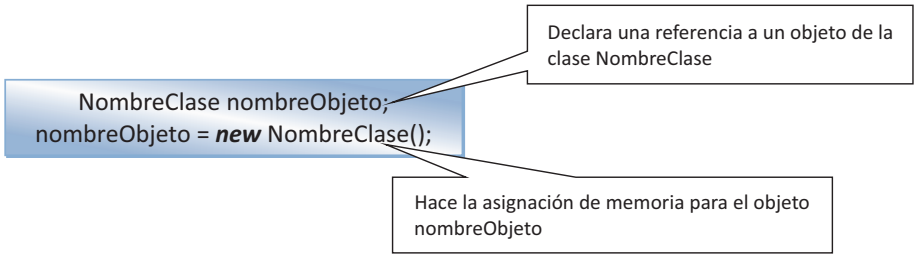
Identificación: característica que hace que un objeto sea único e identificable aunque se encuentre entre otros que tengan similar funcionalidad y estado.

Conducta: característica que determina cual es la misión del objeto y que métodos le pueden involucrar.

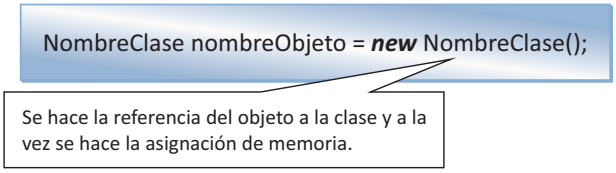
Estado: comportamiento que adquiere un método al aplicarle un método.

Para crear un objeto es necesario declarar una variable del tipo de la clase y posteriormente hacer la asignación de memoria para el objeto a través del operador new. Esta asignación de memoria se hace dinámicamente, es decir, se hace al momento de ejecutar el programa. Esta declaración se puede hacer de las siguientes formas:

Línea a línea



Combinada



A continuación se indica cómo crear la instancia apartamento1 de la clase Apartamento de las dos formas:

Línea a línea:

```
Apartamento apartamento1;
apartamento1 = new Apartamento();
```

Combinada:

```
Apartamento apartamento1 = new Apartamento();
```

7.1.3 MÉTODOS

Los métodos son las acciones que pueden ejecutar sobre una determinada clase y por ende en cada una de sus instancias (objetos). La estructura general de un método es la siguiente:

```
tipo_de_retorno nombreMetodo(parámetros)
        {
                Variables internas del método;
                Instrucciones del método;
                return valor;
        }
```

Algunos métodos no retornan ningún valor, por lo cual deben declararse de tipo ***void***. En cuanto al nombre del método, este puede ser cualquier expresión válida para java y, por buenas prácticas de programación se recomienda que este debe iniciar con una letra minúscula y si está formado por dos o más palabras, deben ir unidas y sus letras iniciales en mayúscula.

Los parámetros son variables que reciben los valores necesarios para que el método realice sus acciones. Estos valores son enviados por otras variables llamadas argumentos y que vienen desde el lugar del programa donde se haga el llamado al método. Tanto argumentos de llamada como parámetros del método deben ser del mismo tipo. La lista de parámetros está conformada por una serie de parejas separadas por comas, donde el primer miembro de cada pareja corresponde al tipo de datos que puede recibir el parámetro y el segundo es el nombre del parámetro.

```
tipoRetorno nombreMétodo(tipo nombreParámetro_1, tipo nombreParámetro_2,……..,
        tipo nombreParámetro_n){
        cuerpoDelMétodo;
}
```

Aunque algunos métodos no requieren parámetros para su funcionamiento, si queremos generalizar un método, es necesario que dentro de su estructura incluyamos parámetros que le permitan recibir datos desde el exterior. El siguiente ejemplo muestra de forma corriente como un método devuelve el valor promedio de 3 números:

```
double promedioNumeros()
        {
                return (5 + 3 + 6)/3;
        }
```

Observando de manera rápida es posible comprobar que el método anterior devuelve el valor promedio entre los números 5, 3 y 6. Pero también podemos ver que su uso está restringido a los valores que se han suministrado en el interior del método.

Si se modifica el método como se ilustra a continuación, éste se hace más útil, ya que estará en capacidad de recibir valores enviados desde diferentes puntos en los cuales sea llamado.

```
double calcularPromedio(double a, double b, double c)
        {
                return (a+b+c)/3;
        }
```

A continuación se ilustra un ejemplo de una *clase* en Java en el que se crea un arreglo para contener tres números digitados desde el teclado y se les calcula el promedio a través del método **calcularPromedio( )**, el cual contiene tres parámetros de tipo **double**.

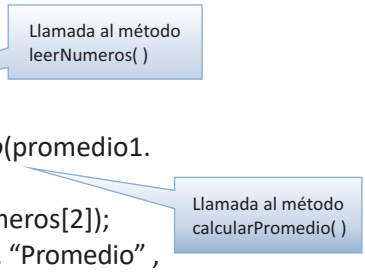
```
import javax.swing.JOptionPane;

public class Promedio
        {
          double listaNumeros [] = new double[3];
          void leerNumeros()
              {
                  for(int i = 0;i<3;i++)
                     {
                        listaNumeros[i]=Double.valueOf(JOptionPane.showInput
                        (“Digite el “     +        (i+1) + “ número: “)).doubleValue();
                     }
              }
        double calcularPromedio(double a, double b, double c)
          {
             return (a+b+c)/3;
         }
        }
```

Los métodos de una clase deben estar contenidos dentro del cuerpo de la clase.

7.2 LLAMADA A UN MÉTODO

Una llamada a un método permite enviarle valores llamados argumentos para que el método pueda ejecutarse de manera generalizada. El siguiente ejemplo ilustra como enviar al método calcularPromedio( ) tres argumentos con valores de tipo doble contenidos en el arreglo listaNumeros[ ]].

```
import javax.swing.JOptionPane;

public class Main
  {
     public static void main(String[] args)
          {
             Promedio promedio1 = new Promedio();
             promedio1.leerNumeros();
             double resultado = promedio1.calcularPromedio(promedio1.
             listaNumeros[0],
             promedio1.listaNumeros[1], promedio1.listaNumeros[2]);
             JOptionPane.showMessageDialog(null, resultado , "Promedio" ,
             JOptionPane.INFORMATION_MESSAGE);
       }
}
```



7.3 CONSTRUCTORES

Los constructores son métodos especiales que tienen el mismo nombre que la clase a la que pertenecen y permiten inicializar un objeto una vez creado sin necesidad de hacer llamado al constructor o a cualquier otro método que permita asignar valores de inicialización al objeto. En el programa siguiente, el constructor Empleado( ) permite inicializar el objeto empleado1 con los valores: 0 para la ***cedula***, "Nombre empleado" para ***nombre*** y "Apellido empleado" para ***apellido***.

```
import javax.swing.JOptionPane;

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

```
            Empleado empleado1 = new Empleado();
            JOptionPane.showMessageDialog(null, empleado1.cedula,
            "Empleados!",
            JOptionPane.INFORMATION_MESSAGE);
            JOptionPane.showMessageDialog(null, empleado1.apellido,
            "Empleados!",
            JOptionPane.INFORMATION_MESSAGE);
        }
}

public class Empleado
{
   String cedula;
   String nombre;
   String apellido;
   Empleado()
        {
                cedula = "0";
                nombre = "Nombre empleado";
                apellido = "Apellido empleado";
        }
}
```

Constructor Empleado( )

Si no hemos definido ningún constructor dentro de la clase, cuando se haga el llamado este a través de la instrucción new, se estará haciendo uso del constructor por defecto, que es aquel que aunque no esté definido explícitamente dentro del programa inicializa el objeto con los datos por defecto correspondientes a cada tipo.

El programa siguiente no define de forma explícita un constructor pero puede visualizar los datos por defecto para cada variable de la instancia creada.

7.4 SOBRECARGA DE MÉTODOS

Cuando dentro de una misma clase existen dos o más métodos con el mismo nombre aunque con diferentes tipos y números de parámetros se dice que existe sobre carga de métodos. Cuando se hace el llamado a un método sobrecargado, el compilador se vale del tipo y número de parámetros para saber a cuál de los métodos sobrecargados debe llamar.

```
public class Operacion {
   int numero1;
   int numero2;
   int numero3;
   char signo;
   int hacerOperacion(int a, int b, char s){
      numero1 = a;
      numero2 = b;
      signo = s;
      if(signo=='+') return a+b;
      if(signo=='-') return a-b;
      if(signo=='*') return a*b;
      if(signo=='/')return a/b;
      else
         System.out.println("Operación no válida!");
         return 0;
   }
   int hacerOperacion(int a,int b, int c){
      numero1 = a;
      numero2 = b;
      numero3 = c;
      return numero1+numero2+numero3;
   }
   int hacerOperacion(int a, int b){
      numero1 = a;
      numero2 = b;
      return numero1 * numero2;
   }
   int hacerOperacion(int base){
      numero1 = base;
      return numero1 * numero1;
   }
}
```

En este ejemplo el método hacerOperacion( ) está sobrecargado, de tal forma que el compilador puede hallar el cuadrado de un número, realizar una operación aritmética entre dos números, hacer la suma de tres números y hacer la multiplicación de dos números, haciendo la llamada de similar forma desde el método main( ) pero cambiando los argumentos que hacen parte de la llamada.

```
public class Main {

   public static void main(String[] args) {
     Operacion operacion1 = new Operacion();
     System.out.print("El cuadrado 10 es: ");
     System.out.println(operacion1.hacerOperacion(10));
     System.out.print("10 - 2 : ");
     System.out.println(operacion1.hacerOperacion(10, 2, '-'));
     System.out.print("1 + 2 + 3 : ");
     System.out.println(operacion1.hacerOperacion(1, 2, 3));

System.out.print("5 * 4 : ");
     System.out.println(operacion1.hacerOperacion(5, 4));
   }

}
```

Llama al método hacerOperacion(int

Llama al método hacerOperacion(int a, int b, char s)

Llama al método hacerOperacion(int a, int b, int c)

Llama al método hacerOperacion(int a, int b)

7.5 SOBRECARGA DE CONSTRUCTORES

De la misma forma cómo es posible tener en una clase sobrecarga de métodos, también es posible tener sobrecarga de constructores.

En el ejemplo siguiente, la clase **Operacion** tiene el constructor **Operacion( )** sobrecargado en tres versiones diferentes.

```
public class Operacion {
   int n1, n2, n3;

public Operacion(int a) {
      n1 = n2 = n3 = a;
   }

   public Operacion(int a, int b) {
      n1 = n2 = n3 = (a * b);
   }

   public Operacion(int a, int b, int c) {
      n1 = n2 = n3 = (a * b * c);
   }
   public  void visualizarDatos(){
      System.out.println(n1+" "+n2+" "+n3);
   }
}
```

Desde el método Main( ) se hace el llamado a las diferentes versiones del constructor cambiando el número de parámetros que se pasan como argumentos.

```
public static void main(String[] args) {
    Operacion operacion1, operacion2, operacion3;
    operacion1 = new Operacion(1);
    operacion1.visualizarDatos();
    operacion2 = new Operacion(1,2);
    operacion2.visualizarDatos();
    operacion3 = new Operacion(1,2,3);
    operacion3.visualizarDatos();
  }
}
```

Es muy importante recordar que una vez definido uno o varios constructores para una clase (sobre carga) el programa ya no hace llamado al constructor por defecto.

7.6 PASO DE OBJETOS COMO PARÁMETROS

De la misma forma cómo es posible pasar datos simples como parámetros a un método en java, también es posible pasar objetos. En el ejemplo siguiente la clase **Estudiante** contiene el método **compararNacimiento( )**, que tiene como parámetro el objeto **estudianteX** de tipo Estudiante y a través del cual son enviados al método los datos pertenecientes a un objeto de su clase. Este método retorna ***true*** si los dos objetos del tipo **Estudiante** (el objeto que hace el llamado y el objeto que hace de argumento) son iguales y ***false*** si son diferentes.

```
public class Estudiante {
  int yearNacimiento;
  int mesNacimiento;
  int diaNacimiento;
  //Constructor Estudiante
  Estudiante(int a,int b, int c){
     yearNacimiento = a;
     mesNacimiento = b;
     diaNacimiento = c;
  }
```

```
   /*Método que devuelve verdadero si el objeto recibido como parámetro
    * es igual al objeto con el que se le hace el llamado
    */
   boolean comparaNacimiento(Estudiante estudianteX){
      if(estudianteX.yearNacimiento==yearNacimiento &&
            estudianteX.mesNacimiento==mesNacimiento &&
            estudianteX.diaNacimiento==diaNacimiento){
         return true;
      }
      else
         return false;
   }
}

public class Main {
   public static void main(String[] args) {
      Estudiante estudiante1 = new Estudiante(1990,10,5);
      Estudiante estudiante2 = new Estudiante(1990,10,5);
      Estudiante estudiante3 = new Estudiante(1991,1,1);
      Estudiante estudiante4 = new Estudiante(1991,1,1);
      System.out.println("estudiante1 es igual a estudiante2 ? ");
       System.out.println("Respuesta: "+estudiante1.comparaNacimiento(estudi
ante2));
      System.out.println("estudiante3 es igual a estudiante4 ? ");
       System.out.println("Respuesta: "+estudiante3.comparaNacimiento(estudi
ante4));
   }
}
```

A continuación se presenta un programa en el que se hace la comparación de dos matrices de 3 x 3 en donde una es pasada como parámetro al método **compararMallas( )**. La clase Malla está compuesta por la variable **matrizA[ ][ ]** de tipo entero.

```
public class Malla {

   int matrizA[][] = new int[3][3];
```

Inicialmente, el constructor **Malla( )** asigna el valor -1 a cada uno de los elementos **i,j** de la matriz.

```
Malla() {
    for (int i = 0; i < 3; i++) {
       for (int j = 0; j < 3; j++) {
          matrizA[i][j] = -1;
       }
    }
 }
```

El método **imprimirMalla ( )** permite hacer la impresión de cada uno de los elementos **i,j** de la matriz.

```
 void imprimirMalla() {
    for (int i = 0; i < 3; i++) {
       for (int j = 0; j < 3; j++) {
          System.out.print(matrizA[i][j] + " ");
       }
       System.out.println();
    }
 }
```

El método **llenarMalla(int a)** recibe el parámetro a de tipo entero y lo asigna a cada uno de los elementos **i,j** de la matriz.

```
 void llenarMalla(int a) {
    for (int i = 0; i < 3; i++) {
       for (int j = 0; j < 3; j++) {
          matrizA[i][j] = a;
       }
    }
 }
```

El método **compararMallas(Malla mallaX)** recibe como parámetro el objeto **mallaX** del tipo Malla y lo compara elemento a elemento con el objeto que hace el llamado; en este caso la **matrizA[i][j]**. En este método se utiliza la variable bandera de tipo entero y que hace las veces de un acumulador para saber si todos los elementos de las dos matrices comparadas son iguales.

```
 boolean compararMallas(Malla mallaX){
    int bandera=0;
    for(int i=0;i<3;i++){
       for(int j=0;j<3;j++){
```

```
            if(matrizA[i][j]==mallaX.matrizA[i][j]){
               bandera+=1;
            }
         }
      }
      if(bandera==9)
         return true;
      else
         return false;
   }
}
```

El método **main( )** de este programa crea dos objetos **malla1** y **malla2** de tipo **Malla**, les asigna el valor -1 a través del constructor **Malla( )**, imprime el valor de las dos matrices, luego les asigna a cada uno de sus elementos **i,j** el valor 5 llamando al método **llenarMalla( )** y finalmente imprime si son o no iguales. Para este caso como las dos matrices tienen en cada uno de sus elementos **i,j** el valor 5 la salida impresa será ***true***.

```
public class Main {
   public static void main(String[] args) {
      Malla malla1 = new Malla();
      Malla malla2 = new Malla();
      malla1.imprimirMalla();
      malla1.llenarMalla(5);
      malla2.llenarMalla(5);
      malla1.imprimirMalla();
      System.out.println(malla1.compararMallas(malla2));
   }
}
```

Pruebe cambiando el valor 5 que se envía como argumento en la línea donde el objeto **malla2** hace el llamado al método **llenarMalla( )** y podrá ver que la salida pasará de ser true a false.

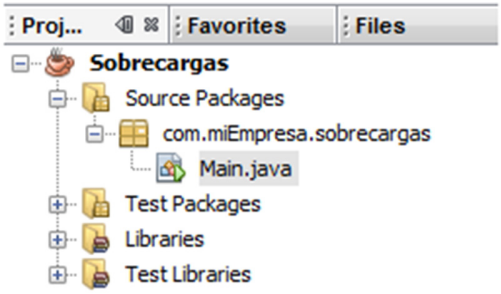
7.7 PAQUETES

Java le permite al programador agrupar en una colección llamada paquete las clases que éste considere, de tal forma que su trabajo sea más claro y organizado. Es muy similar a lo que se puede hacer con los archivos

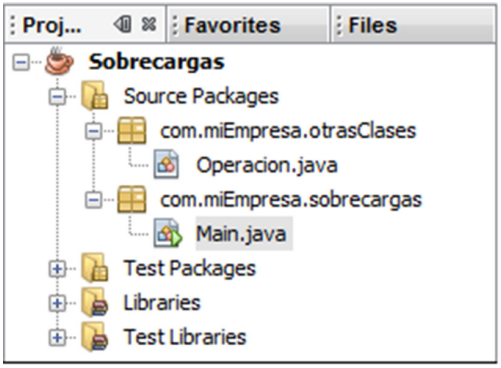
que guardamos en un computador; los organizamos por categorías y los agrupamos en carpetas según esta clasificación.

La compañía Sun Microsystems, creadora de Java, teniendo en cuenta que el nombre del dominio de todas las empresas es único sugiere que el nombre del paquete raíz donde almacenaremos nuestras clases corresponda con el nombre inverso de dicho dominio. Por ejemplo miEmpresa.com corresponde con com.miEmpresa. Además podemos crear subpaquetes según la necesidad que tengamos para un claro y manejable trabajo.

La figura siguiente, que ilustra la ventana de proyectos en NetBeans 6.8 muestra cómo queda inicialmente el árbol de paquetes para el proyecto Java de nombre Sobrecargas. En el paquete com.miEmpresa queda ubicada la clase Main de este proyecto.



De la misma forma como se tiene el paquete com.miEmpresa.sobrecargas, es posible crear otros paquetes donde es posible agrupar otras clases según la necesidad del programador.



Para poder hacer uso de las clases que se encuentran en otros paquetes es necesario importarlas desde la cabecera de la clase que las utilizará. Esto se hace a través de la instrucción **import,** con la cual podremos importar una clase específica del paquete o todas las clases que se encuentran dentro de este. Es importante recordar que el esquema general de la instrucción import es el siguiente:

```
Import nombrePaquete.nombreClase;
```

En la cabecera de la clase desde la que estamos haciendo la llamada a otras clases, también aparece la instrucción **package.nombrePaquete**;. Esta instrucción identifica el paquete al cual pertenece la clase actual.

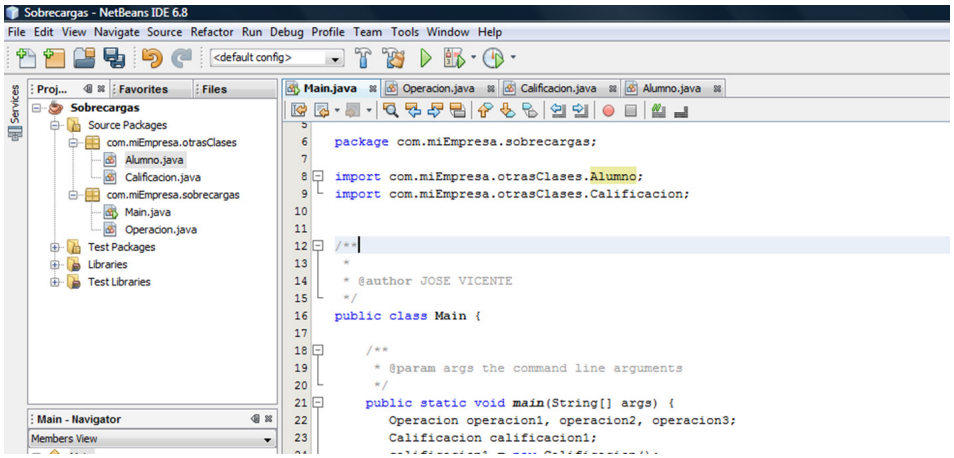
```
package com.miEmpresa.sobrecargas;

import com.miEmpresa.otrasClases.*;
```

Importa todas las clases contenidas en el paquete com.miEmpresa.otrasClases

```
package com.miEmpresa.sobrecargas;

import com.miEmpresa.otrasClases.Alumno;
import com.miEmpresa.otrasClases.Calificacion;
```

Importan las clases Alumno y Calificacion contenidas en el paquete com.miEmpresa.otrasClases

La figura siguiente permite ver la estructura del proyecto **Sobrecargas**, en el que existen dos paquetes de clases y desde la clase **Main** se importan las clases **Alumno** y **Calificación**.



7.8 ACCESO A DATOS DE UNA CLASE DE OTRO PAQUETE

Para poder tener acceso a datos de una clase perteneciente a otro paquete, sus métodos deben ser declarados públicos, lo mismo que aquellos datos a los que deseemos acceder de forma directa. De lo contrario su acceso no será permitido, esto debido a la capacidad de encapsulamiento que tiene Java para proteger sus clases.

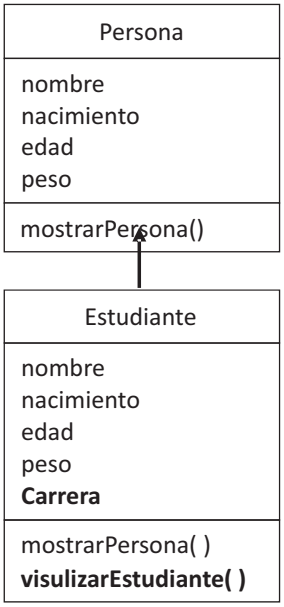
8. HERENCIA

8.1 CONCEPTO

Java permite la herencia simple de clases, básicamente lo que es que una clase copie de otra su interfaz y su comportamiento y que agregue un nuevo comportamiento en forma de código, así mismo la herencia de clases posibilita que los miembros públicos y protegidos de una clase A, sean públicos y protegidos respectivamente, en cualquier clase B descendiente de A. de ahí que decimos que los miembros públicos y protegidos de una clase se dice que son accesibles desde las clases que derivan de ella [10].

La herencia es la capacidad que tienen los lenguajes de programación orientada a objetos para crear una o varias clases nuevas a partir de una clase existente. Las clases nuevas heredan todas las características y métodos de la clase existente, pero además pueden tener otras características y métodos que las convierten en clases más especializadas. La clase base, o sea aquella que sirve como soporte para la creación de una nueva clase recibe el nombre de **Superclase** y, las que se deriven de ella serán las **Subclases.**

A continuación, se muestra un ejemplo en el que se ilustra el uso y ventaja de la herencia. Se parte de una superclase llamada **Persona**, que está compuesta por las variables *nombre, nacimiento, edad y peso* y el método *mostrarPersona( )*. Por herencia se crea la subclase **Estudiante**, que además de acceder a los datos y método de la superclase **Persona**, tiene la variable *carrera* y el método *visualizarEstudiante( )*.



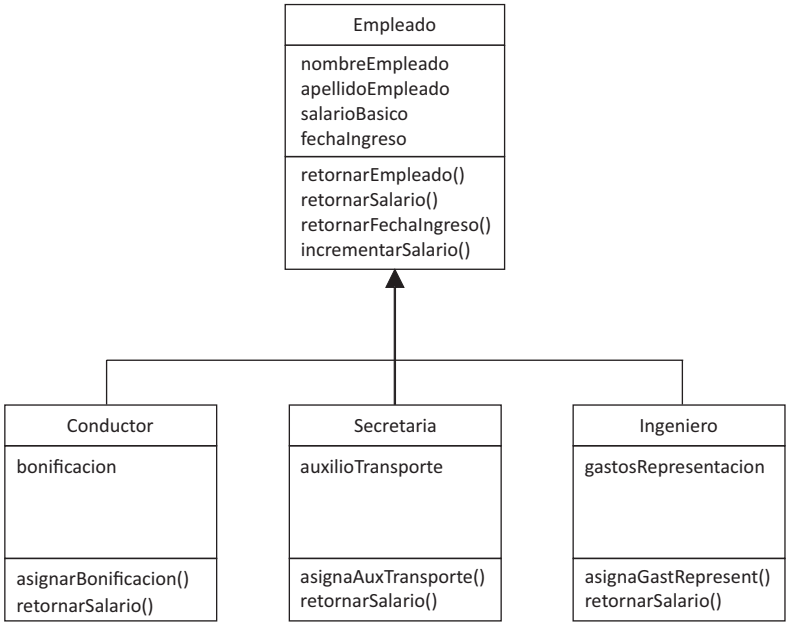
Su codificación en java está dada en el siguiente programa:

```
public class Persona {
   String nombre;
   int nacimiento;
   int edad;
   int peso;
   Persona(String nombre,int nacimiento, int edad, int peso){
      this.nombre = nombre;
      this.nacimiento = nacimiento;
      this.edad = edad;
      this.peso = peso;
   }
   Persona(){
      nombre = “”;
      nacimiento = -1;
      edad = -1;
      peso = -1;
   }
   public void mostrarPersona(){
      System.out.println(“      “+”Nombre:       “+nombre);
      System.out.println(“      “+”Año nacimiento: “+nacimiento);
```

```
      System.out.println("      "+"Edad:        "+edad);
      System.out.println("      "+"Peso:        "+peso);
   }
}

public class Estudiante extends Persona {
   String carrera;
   Estudiante(String nombre, int nacimiento,int edad, int peso, String carrera){
      this.nombre = nombre;
      this.nacimiento = nacimiento;
      this.edad = edad;
      this.peso = peso;
      this.carrera = carrera;
   }
   public void visualizarEstudiante(){
      System.out.println("      "+"Nombre:          "+nombre);
      System.out.println("      "+"Carrera:         "+carrera);
   }
}

public class Main {
   public static void main(String[] args) {
      Persona jaime = new Persona("Jaime Gutiérrez", 1980, 30, 65);
      Estudiante manuel = new Estudiante("Manuel Romero",1990 , 20, 70,
      "Ing Sistemas");
      System.out.println("Datos de Jaime: ");
      jaime.mostrarPersona();
      System.out.println("Datos de Manuel como persona: ");
      manuel.mostrarPersona();
      System.out.println("Datos de Manuel como estudiante: ");
      manuel.visualizarEstudiante();
   }
}
```

La principal ventaja de la herencia es la reutilización de código.

Las relaciones de Herencia entre las clases conllevan a la existencia de una jerarquía de clases. En el ejemplo siguiente la clase Empleado hace el papel de Superclase y las clases Conductor, Secretaria e Ingeniero son Subclases derivadas de la clase Empleado. Es común encontrar jerarquías de clases en múltiples niveles o capas; para el ejemplo que nos ocupa, es una jerarquía de dos niveles.



El código para la implementación del anterior modelo es el siguiente:

La clase Empleado, que en este programa toma el papel de superclase, contiene los datos básicos de un empleado como: nombre, apellido, salario básico y fecha de ingreso. La fecha de ingreso es un dato de tipo Date, que debe ser manejado con soporte de los paquetes ***java.util.Date*** y ***java.util.GregorianCalendar***.

De la clase Empleado hacen parte los métodos: **retornarEmpleado( )** que devuelve el nombre y apellido de un empleado, **retornarSalario( )** que devuelve el salario básico del empleado, **retornarFechaIngreso( )** que devuelve la fecha de ingreso del empleado e **incrementarSalario( )** que incrementa en un porcentaje el salario básico del empleado.

```
package com.usoherencia.principal;
import java.util.Date;
import java.util.GregorianCalendar;

public class Empleado {
   private String nombreEmpleado;
   private String apellidoEmpleado;
```

```
   private double salarioBasico;
   Date fechaIngreso;
   public Empleado(String nombre, String apellido, double salario,
                           int anno, int mes, int dia){
      nombreEmpleado = nombre;
      apellidoEmpleado = apellido;
      salarioBasico = salario;
      GregorianCalendar miCalendario = new GregorianCalendar(anno, mes-1, dia);
      //GregorianCalendar toma enero como 0
      fechaIngreso = miCalendario.getTime();
   }
   public String retornarEmpleado(){
      String datosEmpleado = nombreEmpleado + " " + apellidoEmpleado;
      return datosEmpleado;
   }
   public double retornarSalario(){
      return salarioBasico;
   }
   public Date retornarFechaIngreso(){
      return fechaIngreso;
   }
   public void incrementarSalario(double porcentajeIncremento){
      double incremento;
      incremento = salarioBasico*porcentajeIncremento/100;
      salarioBasico = salarioBasico+incremento;
   }
}

public class Conductor extends Empleado {
   private double bonificacion;
   //Constructor de la clase
   public Conductor(String nombre,String apellido,double salario,
         int anno, int mes, int dia){
      //Llamada al constructor de la superclase
      super(nombre,apellido,salario, anno,mes,dia);
   }
   //Asignarle valor a la variable propia
   public void asignarBonificacion(double bonificacion){
      this.bonificacion = bonificacion;
   }
   public double retornarSalario(){
      double salarioNeto;
```

```
      //se hace el llamado al método retornarSalario() de la superclase
      salarioNeto = super.retornarSalario()+ this.bonificacion;
      return salarioNeto;
   }
}

public class Secretaria extends Empleado {
   private double auxTransporte;
   private String dependencia;

   public Secretaria(String nombre, String apellido, double salario,
         int anno, int mes, int dia) {
      //Se hace le llamado al constructor de la superclase
      super(nombre, apellido, salario, anno, mes, dia);
      dependencia = "Talento Humano";
   }
   public void asignarAuxTransporte(double auxilio){
      auxTransporte = auxilio;
   }
   public double retornarSalario(){
      double salarioNeto;
      //Se hace llamado al método retornarSalario() de la superclase
      salarioNeto = super.retornarSalario()+auxTransporte;
      return salarioNeto;
   }
}

public class Ingeniero extends Empleado{
   double gastosRepresentacion;
   public Ingeniero(String nombre, String apellido, double salario,
         int anno, int mes, int dia){
      super(nombre, apellido, salario, anno, mes, dia);
   }
   public void asignarGastosRepresentacion(double salBasico){
      gastosRepresentacion = salBasico * 0.9;
   }
   public double retornarSalario(){
      double salarioNeto;
      salarioNeto = super.retornarSalario()+gastosRepresentacion;
      return salarioNeto;
   }
}
```

En la primera parte es posible encontrar la clase Empleado, de la cual se derivan las subclases Conductor, Secretaria e Ingeniero. La clase Empleado, como clase padre tiene las variables y métodos que le serán heredados a las clases Conducto, Secretaria e Ingeniero, las cuales a su vez tienen variables y métodos propios.

En Java para hacer que una clase herede los atributos y métodos de una superclase o clase padre, se hace uso de la instrucción extends la cual se coloca entre el nombre de la clase que se esté creando y el nombre de la clase de la que se va a heredar.

```
public class Ingeniero extends Empleado;
```

9. PROGRAMACIÓN GRÁFICA

9.1 SWING

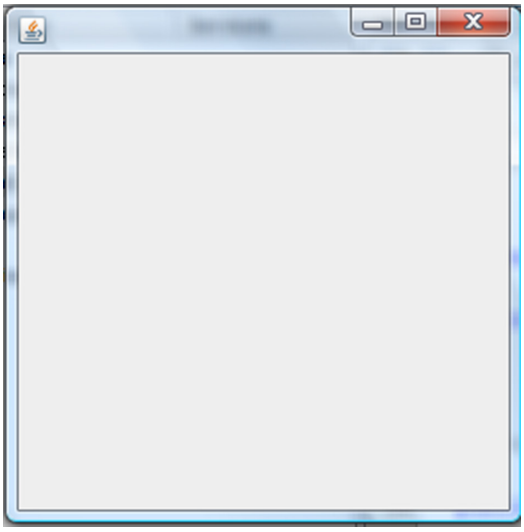
Swing es un conjunto robusto de herramientas creadas por ***Sun Microsystems*** para la creación de interfaces gráficas en java. Básicamente swing es un grupo de clases que proporciona alternativas poderosas y flexibles a los componentes estándar de los ***AWT*** (Kits de Herramientas de Ventana Abstracta); por ejemplo, además de los componentes que siempre utiliza como las etiquetas, botones, casillas de verificación; ***Swing*** proporciona otros componentes importantes como paneles desplazables, paneles de fichas, árboles y tablas los cuales permiten a los programadores un mejor trabajo en el desarrollo de una aplicación [11].

9.2 CREACIÓN DE UN FRAME

Un frame es un componente Swing capaz de contener a otros componentes (botones, etiquetas, cajas de texto, etc). Para crear un Frame es necesario hacer uso de la clase JFrame contenida en el paquete *javax.swing* mediante la instrucción extends. Una vez esté definido el uso del JFrame es necesario establecer un tamaño del Frame a través del constructor. Finalmente se crea el objeto desde el método main() de la clase principal y se le establecen las acciones que este ejecutará; en este caso, visualizarse y terminar el programa cuando se presione el botón de cierre del Frame. A continuación se muestra el código y la salida gráfica que se produce al ejecutar el programa.

```
package framesimple;
import javax.swing.JFrame;

public class FrameVacio extends JFrame {
        public FrameVacio(){
            setSize(300,300);
          }
        }

package framesimple;
import javax.swing.JFrame;
```

```
public class Main {
        public static void main(String[] args) {
            FrameVacio fv1 = new FrameVacio();
            fv1.setVisible(true);
            fv1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          }
        }
```



9.3 ASIGNAR TITULO A UN FRAME

El método setTitle() permite asignarle un título a la barra de títulos de un frame. Su sintaxis es:

```
nombreFrame.setTitle("Título del frame");
```

9.4 LOCALIZACIÓN DE UN FRAME

Al momento de crear un frame, este queda posicionado en las coordenadas (0,0) de la pantalla del computador. Para ubicar un frame ubicado en una posición diferente, se utiliza el método **setLocation()**, cuya sintaxis es la siguiente:

```
nombreComponente.setLocation(posición_x, posición_y);
```

Los valores para posición_x y posición_y deben ser enteros y corresponden al número de pixeles desde el borde izquierdo y número de pixeles desde el borde superior de la pantalla respectivamente en donde su ubicará la esquina superior izquierda del frame. Para hacer que el frame creado en el programa anterior se ubique en la posición (300,300) se debe escribir la siguiente instrucción una vez creado el objeto:

fv1.setLocation(300,300);

9.5 MOVER Y CAMBIAR EL TAMAÑO DE UN FRAME

A través del método **setBounds()** es posible cambiar la posición y tamaño de un componente Swing. Su sintaxis es:

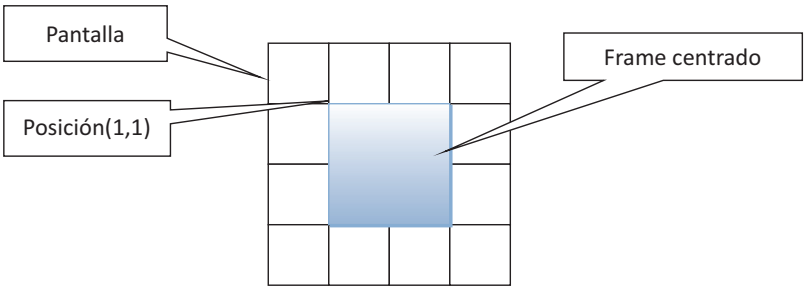
```
nombreComponente.setBounds(posición_x, posición_y, ancho, alto);
```

Los argumentos posición_x, posición_y, ancho y alto deben ser enteros y corresponden a la distancia en pixeles desde el borde izquierdo, borde derecho, ancho del componente y alto del componente respectivamente. Los valores ancho o alto serán modificados automáticamente si cualquiera de los dos está por debajo del mínimo especificado previamente mediante el método **setMinimumSize()**. Es importante tener en cuenta que si no especifican las coordenadas de posicionamiento y el tamaño del frame, este se posicionara en las coordenadas (0,0) y tendrá un tamaño de 0 x 0 pixeles. También se debe tener en cuenta que aplicaciones software de calidad deben ser capaces de adaptarse a la configuración de pantalla de los usuarios al momento de visualizar formularios con diferentes tipos de información; esto requiere de los programadores la agilidad para capturar la resolución de video de los usuarios y ajustar según estas la posición y tamaño de los formularios de captura y/o visualización de datos.

9.6 CENTRAR UN FRAME

Para centrar un frame en la pantalla de un usuario determinado, es necesario partir por obtener la configuración de la resolución en pixeles de la pantalla del usuario y posteriormente establecer un frame que ocupe una cuarta parte del total de la pantalla. A continuación se ilustra gráficamente el problema planteado; se parte de la suposición de tener

una pantalla de 4 x 4 pixeles en la que se centra un frame de 2 x 2 pixeles, la esquina superior izquierda del frame debe ser ubicada en la posición (1,1) de la pantalla.



Observando la figura anterior podemos observar que el ancho del frame que se pretende centrar en la pantalla corresponde a la mitad del ancho de la pantalla, el alto del frame corresponde a la mitad del alto de la pantalla y la posición de la esquina superior izquierda en la que se ubicará el frame para que quede centrado en la pantalla corresponde a la cuarta parte del ancho de la pantalla para la posición **x** y a la cuarta parte del alto de la pantalla para la posición **y**.

Para poder obtener la resolución en pixeles a la cual está configurada la pantalla del usuario es necesario recurrir al sistema operativo de su computador. En java es posible hacer esto gracias a la clase **Toolkit** la cual cuenta con el método **getScreenSize()** para retornar un objeto de tipo ***Dimension*** con el tamaño en pixeles **(ancho, alto)** de la pantalla del computador.

A continuación se adjunta el código en java de un programa que centra un frame en la pantalla de un computador:

```
import java.awt.Dimension;
import java.awt.Toolkit;
import javax.swing.JFrame;
public class FrameCentrado extends JFrame{
        int ancho, alto, posicion_x, posicion_y;
        Toolkit tk1 = Toolkit.getDefaultToolkit();
        Dimension pantallaActual = tk1.getScreenSize();
        void darTamanno(){
                ancho = pantallaActual.width/2;
```

```
                alto = pantallaActual.height/2;
                setSize(ancho,alto);
                }
        void centrarFrame(){
                posicion_x = pantallaActual.width/4;
                posicion_y = pantallaActual.height/4;
                setLocation(posicion_x,posicion_y);
                }
          }
public class Main {
        public static void main(String[] args) {
                FrameCentrado fc1 = new FrameCentrado();
                fc1.darTamanno();
                fc1.centrarFrame();
                fc1.setVisible(true);
                }
          }
```

9.7 DIFERENCIA ENTRE UN FRAME Y UNA VENTANA

```
package javentanas;

public class Main {
          public static void main(String[] args) {
                Marco marco1 = new Marco();
                Ventana ventana1 = new Ventana();
                marco1.setVisible(true);
                ventana1.setVisible(true);
             }
}
```

CREACIÓN DE LA CLASE Marco

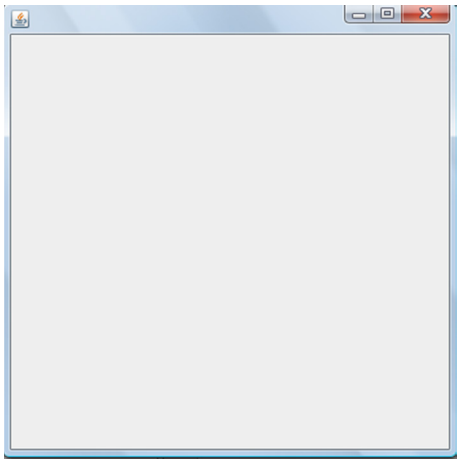
```
          package javentanas;

          import javax.swing.JFrame;

          /**
           *
           * @author USUARIO
```

```
 */
public class Marco extends JFrame {
   Marco(){
      setSize(400,400);
      setLocation(100,100);
   }
}
```

CREACIÓN DE LA CLASE Ventana

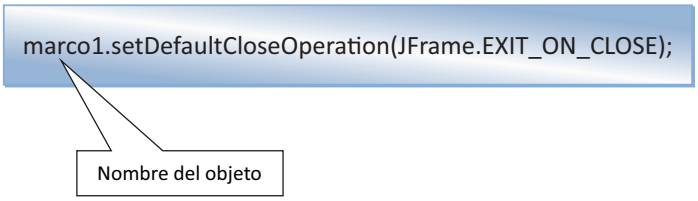
```
package javentanas;

import javax.swing.JWindow;

/**
 *
 * @author USUARIO
 */
public class Ventana extends JWindow{
   Ventana(){
      setSize(400,400);
      setLocation(600,100);
   }
}
```

LA SALIDA SERÁ:



9.8 SALIDA DE LA APLICACIÓN DESDE UN FRAME

Un objeto de la clase JFrame posee entre otras propiedades: un ícono, una barra de título y los botones de control estándares para cualquier ventana (minimizar, maximizar y cerrar). Haciendo uso del botón cerrar de un objeto de la clase JFrame es posible finalizar con el funcionamiento de la aplicación que se esté ejecutando; para esto se hace utilizando la siguiente línea de código:



Mediante esta línea de código, se establece la acción que seguirá el programa cuando el usuario de click sobre el botón cerrar del objeto JFrame; para este caso EXIT_ON_CLOSE, finaliza la ejecución de la aplicación en curso.

Es importante tener en cuenta que la acción por defecto para los JFrame es simplemente ocultar el JFrame sobre el que se efectuó la acción y no terminar la aplicación, sin embargo, en el programa del ejemplo, la instrucción puede considerarse como lógica, pues se está cerrando un único JFrame.

CONCLUSIONES

Con la elaboración del presente libro se ha podido crear una guía detallada para los estudiantes y demás personas que hacen su ingreso al mundo de la programación, el libro abarca temas fundamentales como conceptos básicos, estructura de un programa en java, tipos de datos, estructuras fundamentales, programación orientada a objetos, herencia y manejo de interfaz gráfica.

El trabajo de investigación y creación que se ha llevado a cabo ha tenido como foco la capacidad de asimilación y aprendizaje autónomo del estudiante. Cada capítulo hace una introducción precisa de la temática que involucra, luego lleva al estudiante al desarrollo de la misma a través de ejercicios sencillos y reales, que son explicados en detalle y finalmente plantea una serie de ejercicios para ser resueltos con los que se garantiza una autoevaluación de lo aprendido.

Se quiso crear un libro fácil de asimilar, con ejemplos que involucran el planteamiento de problemas, una explicación gráfica de las temáticas, explicación de las estructuras usadas para dar solución al problema y la inclusión de comentarios en la codificación con el fin de facilitar al estudiante la comprensión de los contenidos.

Este libro se convierte en pieza fundamental para crear nuevas obras que abarquen temas más avanzados y que requieran una mayor dedicación y esfuerzo por parte de estudiantes y docentes que estén haciendo su ingreso al mundo de la programación orientada a objetos.

Finalmente, agradecemos a la editorial de la Universidad, por tener dentro de su misión, políticas que apoyen y motiven a los docentes en estos compromisos bibliográficos; ya que son ellos quienes más cerca están de los estudiantes, conocen sus intereses y necesidades, y por ende son quienes deben liderar esta clase de trabajo con el fin de producir obras que ayuden a mejorar el aprendizaje y a la vez, ayuden al posicionamiento y reconocimiento de la Universidad y sus docentes tanto nacional e internacionalmente.

BIBLIOGRAFÍA

[1] F. J. Ceballos, Java2: lenguaje y aplicaciones, Madrid: RAMA, 2006, pp 27.

[2] H. A. Flórez, Programación orientada a objetos usando java. Bogotá, Col: Ecoe Ediciones, 2012, pp 6.

[3] A. Casanova, F. Marqués, Empezar a programar usando java, Valencia, Esp: Editorial de la Universidad de Valencia, 2012, pp 29, 30, 403.

[4] H. Schildt, Java: Manual de referencia", México, D.F: McGraw-Hill Interamericana, 2009, pp 33-37.

[5] F. J. Moldes, Manual imprescindible Java 8, España: ANAYA Multimedia, 2014, pp 37,38,39.

[6] J. Sánchez, T. Huecas, B. Fernández, Programación en JAVA, 3a ed. Madrid, España: McGraw-Hill, 2009, pp. 15,20,43.

[7] J. Arnedo, D. Riera Terrén, J. Brínquez, E. García, Programación orientada a objectes, Editorial UOC, 2007, pp 173

[8] J. Sánchez, B. Fernández, Java 2: Iniciación y referencia, 2a ed. Madrid, España: McGraw-Hill, 2005, pp. 30-35.

[9] L. Joyanes, I. Zohonero, Estructuras de datos en Java, Madrid, España: McGraw-Hill, 2008, pp.

[10] J. Vélez, A. Peña, and P. Cortázar, Diseñar y programar, todo es empezar. Una introducción a la programación Orientada a Objetos usando UML y Java, España: Dykinson, 2011, pp 52, 53.

[11] H. Schildt, Fundamentos de Java, 3ª ed. México: McGraw-Hill-interamericana editores, 2007, pp 530.

Juan Ð Castellanos
Fundación Universitaria