DESARROLLO DE APLICACIONES CON JAVA 8

ORIENTADO A OBJETOS

—ECLIPSE ◆ NETBEANS—

Manuel Torres Remon

Perú - México - Colombia - Chile - Ecuador - España - Bolivia - Uruguay - Guatemala - Costa Rica

Desarrollo de aplicaciones con Java 8
Orientado a objetos
Autor: Manuel Torres Remon

© Derechos de autor registrados:
Empresa Editora Macro EIRL

© Derechos de edición, arte gráfico y diagramación reservados:
Empresa Editora Macro EIRL

Jefe de edición:
Magaly Ramon Quiroz

Diseño de portada:
Fernando Cavassa Repetto

Corrección de estilo:
Dina Dueñas Calderón

Diseño y diagramación:
Fernando Cavassa Repetto

Edición a cargo de:
© Empresa Editora Macro EIRL
Av. Paseo de la República N.° 5613, Miraflores, Lima, Perú

Teléfono: (511) 748 0560
E-mail: proyectoeditorial@editorialmacro.com
Página web: www.editorialmacro.com

Primera edición: Febrero 2018
Tiraje: 2000 ejemplares

Impresión:
Talleres gráficos de la Empresa Editora Macro EIRL
Jr. San Agustín N.° 612-624, Surquillo, Lima, Perú
Febrero 2018

ISBN N.° 978-612-304-552-4
Hecho el Depósito Legal en la Biblioteca Nacional del Perú N.° 2018-01252

Prohibida la reproducción parcial o total, por cualquier medio o método, de este libro sin previa autorización de la Empresa Editora Macro EIRL.

MANUEL TORRES REMON

Licenciado en Informática. Se ha dedicado a la consultoría y docencia de cursos de tecnología desde hace 17 años. A su vez, ha brindado diversos talleres de capacitación en las instituciones más importantes de Lima (Perú).

Realizó sus estudios en la Universidad Peruana del Norte, en la Universidad Alas Peruanas y en el Instituto de Educación Superior Manuel Arévalo Cáceres, en las cuales siguió las carreras de Ingeniería de Sistemas, Educación y Computación e Informática, respectivamente. En todas estas instituciones recibió una esmerada formación profesional que ha podido demostrar ampliamente a lo largo de su trayectoria docente.

Actualmente, se desempeña como docente de tecnología en instituciones de educación superior como Cibertec, Manuel Arévalo Cáceres y Unimáster de la Universidad Nacional de Ingeniería (UNI); en todas ellas imparte los cursos de Programación, Base de Datos y Análisis de Sistemas.

Dedicatoria

Este libro está dedicado con mucho cariño y aprecio a mis pequeñas Ángela Victoria y Fernanda Ximena, quienes son y seguirán siendo mi fuente de inspiración; y también a mi esposa Luz por comprenderme en todo lo que me propongo.

Agradecimientos

Todo libro escrito es un sacrificio que debe ser valorado, y ello solo se dará si usted aprecia este material tanto como lo aprecio yo mismo. Escuché alguna vez decir: «La lectura de un libro enriquece de conocimientos y empobrece la ignorancia». Esto es muy cierto, aunque hoy en día un libro impreso es difícil de leer. Son pocas las personas que cargan un libro en la mano, pues han sido reemplazados por los dispositivos móviles. Por eso, mi primer agradecimiento es para usted, amigo lector, por adquirir este material, que ha significado tiempo y esfuerzo para mí.

Asimismo, deseo agradecer a la gran familia Macro por confiar nuevamente en mi persona para el desarrollo de este libro: *Desarrollo de aplicaciones con Java 8*.

Índice

Introducción 11

■ Capítulo 1: Introducción al lenguaje Java 19

1.1 Generalidades 21
1.2 Lenguajes de programación 22
1.3 Java 23
1.4 Versiones Java 24
1.5 Orígenes del lenguaje Java 26
1.6 Características de la tecnología Java 27
1.7 Los programas en Java 28
1.8 La máquina virtual Java (JVM) 31
1.9 JDK 32
1.10 JRE 33
1.11 Aplicaciones Java 34
1.12 Instalando Java 35
1.12.1 Instalación de JDK 36
1.12.2 Instalación de NetBeans (versión 8.2) 38
1.12.3 Descarga de Eclipse (versión Mars 2) 42
1.13 El IDE NetBeans y su primera aplicación de plataforma 43
1.13.1 Crear una nueva aplicación 43
1.13.2 Añadir paquetes al proyecto 44
1.13.3 Añadir un formulario al proyecto 45
1.14 El IDE Eclipse Mars y su primera aplicación de plataforma 46
1.14.1 Crear una nueva aplicación 46
1.14.2 Añadir paquetes al proyecto 47
1.14.3 Añadir un formulario al proyecto 48
CASO DESARROLLADO 1: Monto de capital usando NetBeans 50
CASO DESARROLLADO 2: Venta de productos usando Eclipse Mars 2 52

■ Capítulo 2: Fundamentos de programación 57

2.1 Generalidades 59
2.2 Proceso de solución mediante una aplicación Java 59
2.2.1 Fase 1: Análisis del problema 59
2.2.2 Fase 2: Diseño del algoritmo 61
2.2.3 Fase 3: Codificación del problema 63
2.2.4 Fase 4: Ejecución del programa 66

2.3 Asociar el pseudocódigo al formulario en Java ... 66
2.4 Metodología de programación con Java ... 68
2.4.1 Programación estructurada ... 68
2.4.2 Programación modular ... 69
2.4.3 Programación orientada a objetos ... 70
2.5 Elementos que componen una aplicación Java ... 71
2.6 Los comentarios ... 72
2.7 Elementos léxicos en Java ... 74
2.7.1 Palabras reservadas ... 74
2.7.2 Identificadores ... 75
2.7.3 Literales ... 76
2.7.4 Operadores ... 77
2.7.5 Delimitadores ... 82
2.8 Los paquetes y las importaciones ... 84
2.8.1 java.lang ... 85
2.8.2 Java.awt ... 85
2.8.3 Javax.swing ... 85
2.8.4 java.io ... 85
2.8.5 Java.util ... 86
2.9 Tipos de datos ... 86
2.10 Las variables ... 87
2.10.1 Inicialización de variables ... 89
2.10.2 Conversiones de tipo ... 89
2.11 La clase **Integer** ... 91
2.12 La clase **String** ... 92
2.13 Manejo de la clase **JOptionPane** ... 97
2.14 La clase **Math** ... 102
2.14.1 Métodos que representan a **PI** y a **E** ... 102
2.14.2 Métodos de conversión entre grados y radianes ... 103
2.14.3 Métodos de la clase **Math** ... 103
CASO DESARROLLADO 1: Modelo de cotización bancaria (programación estructurada) ... 108
CASO DESARROLLADO 2: Modelo de cotización bancaria (programación modular) ... 110
CASO DESARROLLADO 3: Modelo de cotización bancaria (programación orientada a objetos) ... 114

■ Capítulo 3: Clase Swing ... 119

3.1 Generalidades ... 121
3.2 Clases **Swing** ... 121
3.3 La clase **JFrame** ... 124
3.4 La clase **JLabel** ... 125
3.5 La clase **JTextField** ... 126
3.6 La clase **JTextArea** ... 128
3.7 La Clase **JPasswordField** ... 131
3.8 La clase **JButton** ... 132

3.9 La clase **JCheckBox** 133
3.10 La clase **JRadioButton** 135
3.11 La clase **JComboBox** 136
3.12 La clase **DefaultListModel** 139
3.13 La clase **JList** 140
3.14 La clase **JMenuBar** 141
3.15 La clase **JMenuItem** 142
3.16 Clase **Desktop Pane** 143
CASO DESARROLLADO 1: Manejo de la clase **JList** 144
CASO DESARROLLADO 2: Manejo de la clase **JComboBox** 145
CASO DESARROLLADO 3: Manejo de la clase **JMenuBar** 146

■ Capítulo 4: Programación orientada a objetos 149

4.1 Generalidades 151
4.2 Conceptos iniciales de programación orientada a objetos 152
4.2.1 Abstracción 152
4.2.2 Encapsulamiento 152
4.2.3 Herencia 153
4.2.4 Polimorfismo 153
4.3 Paquetes Java 154
4.3.1 Creación de un paquete 155
4.3.2 Importación de paquetes 157
4.3.3 Alcance de los elementos contenidos en un paquete 157
4.4 Clases en Java 158
4.4.1 Estructura de una clase 158
4.4.2 Agregar una clase a un paquete en NetBeans 159
4.4.3 Atributos de una clase 160
4.4.4 Métodos en las clases 161
4.4.5 Métodos sin valor de retorno 162
4.4.6 Métodos con valor de retorno 163
4.5 Método constructor 164
4.6 Referencia **this** 165
4.7 Objetos en Java 166
4.7.1 Referenciar a los atributos de clase 167
4.7.2 Referenciar a los métodos de clase 168
4.8 Métodos **get** y **set** 168
4.8.1 Método **set** 168
4.8.2 Método **get** 170
4.8.3 Implementación de métodos **get** y **set** con NetBeans 171
CASO DESARROLLADO 1: Casa de cambio (con atributos públicos de clase) 173
CASO DESARROLLADO 2: Casa de cambio (con atributos privados de clase) 176
CASO DESARROLLADO 3: Casa de cambio (con métodos **get** y **set**) 179
CASO DESARROLLADO 4: Casa de cambio (con método constructor) 182

■ Capítulo 5: Estructura de secuencia 187
5.1 Generalidades 189
5.2 Estructura secuencial 189
5.3 Componentes de una estructura secuencial 190
5.3.1 Declaración 190
5.3.2 Asignación 190
5.3.3 Lectura u obtención de datos 190
5.3.4 Escritura de datos 191
CASO DESARROLLADO 1: Cambio de temperaturas 192
CASO DESARROLLADO 2: Conversión de medidas 195
CASO DESARROLLADO 3: Pago de empleados 198
CASO DESARROLLADO 4: Venta de productos 201

■ Capítulo 6: Estructura de selección 207
6.1 Generalidades 209
6.2 Operadores de relación 209
6.3 Operadores lógicos 211
6.4 Instrucción **if** simple 212
6.5 Instrucción **if** doble 213
6.6 Instrucción **if** doblemente enlazada 214
6.7 Sentencia **switch** 215
6.8 Implementar una condición lógica 216
CASO DESARROLLADO 1: Renta de autos (usando **if** doblemente enlazado) 220
CASO DESARROLLADO 2: Pago de trabajadores (usando **if** simple) 224
CASO DESARROLLADO 3: Control de hectáreas de algodón y maíz (usando **if** doble) 228
CASO DESARROLLADO 4: Control de consumo de agua (usando **if** doblemente enlazado) 232
CASO DESARROLLADO 5: Control de pensión de estudiantes (usando **switch**) 235

■ Capítulo 7: Estructura de repetición 241
7.1 Generalidades 243
7.2 Contadores 243
7.3 Acumuladores 244
7.4 Estructuras repetitivas 244
7.4.1 Estructura de repetición **for** 245
7.4.2 Estructura de repetición **while** 247
7.4.3 Estructura de repetición **do... while** 248
7.5 Variables y métodos de clase: modificador **static** 250
7.5.1 Variable de clase 250
7.5.2 Métodos estáticos 251
7.5.3 Inicializadores de variables de clase 252
CASO DESARROLLADO 1: Control de registro de participantes (usando contadores) 253
CASO DESARROLLADO 2: Registro de alquiler (usando acumuladores) 258

CASO DESARROLLADO 3: Registro de venta (usando **for**) 264
CASO DESARROLLADO 4: Control de pago (usando modificar **static**) 270

■ Capítulo 8: Lambda y manejo de excepciones 279

8.1 Generalidades 281
8.2 Expresión **Lambda** 281
8.3 Bloque **try-catch** 283
8.4 Cláusula **throw** 284
8.5 Bloque **finally** 286
CASO DESARROLLADO 1: Registro de libros 286
CASO DESARROLLADO 2: Control de pago de empleados usando **Lambda** 291

■ Capítulo 9: Arreglos 297

9.1 Generalidades 299
9.2 Arreglos 300
9.3 Arreglo unidimensional 301
9.4 Arreglo bidimensional 303
CASO DESARROLLADO 1: Listado de números básico 306
CASO DESARROLLADO 2: Listado de números usando clase 309
CASO DESARROLLADO 3: Certamen de belleza 314
CASO DESARROLLADO 4: Matriz de números enteros 319

■ Capítulo 10: Vector de objetos y ArrayList 327

10.1 Vector de objetos 329
10.2 Clase **ArrayList** 330
CASO DESARROLLADO 1: Mantenimiento de empleados (vector de objetos) 331
CASO DESARROLLADO 2: Mantenimiento de facturas (**ArrayList**) 339
CASO DESARROLLADO 3: Mantenimiento de productos (**ArrayList** y **Lambda**) 349

■ Capítulo 11: Archivos de texto 357

11.1 Configuración del JDK 359
11.2 Librerías a utilizar para el manejo de archivos 362
11.3 Clases y métodos para el manejo y control de archivos de texto 363
CASO DESARROLLADO 1: Mantenimiento de alumnos 365
CASO DESARROLLADO 2: Mantenimiento de productos 374

Referencias bibliográficas 383

CASO DESARROLLADO 3: Registro de venta (usando **for**) 264
CASO DESARROLLADO 4: Control de pago (usando modificar **static**) 270

■ Capítulo 8: Lambda y manejo de excepciones 279

8.1 Generalidades 281
8.2 Expresión **Lambda** 281
8.3 Bloque **try-catch** 283
8.4 Cláusula **throw** 284
8.5 Bloque **finally** 286
CASO DESARROLLADO 1: Registro de libros 286
CASO DESARROLLADO 2: Control de pago de empleados usando **Lambda** 291

■ Capítulo 9: Arreglos 297

9.1 Generalidades 299
9.2 Arreglos 300
9.3 Arreglo unidimensional 301
9.4 Arreglo bidimensional 303
CASO DESARROLLADO 1: Listado de números básico 306
CASO DESARROLLADO 2: Listado de números usando clase 309
CASO DESARROLLADO 3: Certamen de belleza 314
CASO DESARROLLADO 4: Matriz de números enteros 319

■ Capítulo 10: Vector de objetos y ArrayList 327

10.1 Vector de objetos 329
10.2 Clase **ArrayList** 330
CASO DESARROLLADO 1: Mantenimiento de empleados (vector de objetos) 331
CASO DESARROLLADO 2: Mantenimiento de facturas (**ArrayList**) 339
CASO DESARROLLADO 3: Mantenimiento de productos (**ArrayList** y **Lambda**) 349

■ Capítulo 11: Archivos de texto 357

11.1 Configuración del JDK 359
11.2 Librerías a utilizar para el manejo de archivos 362
11.3 Clases y métodos para el manejo y control de archivos de texto 363
CASO DESARROLLADO 1: Mantenimiento de alumnos 365
CASO DESARROLLADO 2: Mantenimiento de productos 374

Referencias bibliográficas 383

Introducción

Java es un poderoso lenguaje de programación que se utiliza para desarrollar aplicaciones computacionales en diversos dispositivos. Hasta hace un buen tiempo se pensaba que Java era un lenguaje complejo, pesado y hasta a veces aburrido. Nada más lejos de la realidad. Hoy en día, Java se ha convertido en uno de los lenguajes de programación de más uso en el mundo, gracias justamente a su legibilidad y simplicidad.

Ante tantos cambios en el veloz mundo en el que vivimos, las nuevas tecnologías se hacen cada vez más populares, como es el caso de Java, al punto que casi todo el mundo las reconoce o ha escuchado hablar de ellas de una forma u otra. Esto genera que las personas se adapten más rápido a dichas tecnologías, dando paso así a la necesidad de que estas evolucionen constantemente. De esta manera, es importante que estemos preparados para entender y programar código en Java. La tarea no es complicada, ya que en programación todo está basado en la lógica y en cuán ordenado se es al programar. Como se ve, no son muchos los requisitos, por lo que esta obra, *Desarrollo de aplicaciones con Java*, está pensado para usuarios iniciales, sin experiencia en programación, hasta desarrolladores en Java.

El libro se ha estructurado en once capítulos, cada uno con casos desarrollados. La forma secuencial de los capítulos ha sido diseñada para un mejor entendimiento de una aplicación Java. En otras palabras, se emplea una metodología práctica, didáctica y visual que facilita el aprendizaje del lector de una manera clara, sencilla y directa.

En el primer capítulo se presentan los conceptos básicos de programación, los cuales introducen al lector en la programación de aplicaciones en lenguaje Java. En el capítulo dos se habla técnicamente del lenguaje Java, desde los operadores hasta la forma de programación. En el capítulo tres se hace un reconocimiento de las principales clases que necesitamos para el desarrollo de nuestras aplicaciones. En el capítulo cuatro se revisan los componentes principales de la *programación orientada a objetos*; asimismo, se sugiere que este sea el punto de partida para el desarrollo de cualquier tipo de aplicación con Java. En los capítulos cinco, seis y siete se desarrollan las aplicaciones de las estructuras secuenciales, selectivas y repetitivas, integrándolas con la programación orientada a objetos. En el capítulo ocho se aprende a implementar el método Lambda y, además, se reconocen algunos aspectos de Java, como el manejo de las excepciones. En los capítulos nueve y diez se hace referencia al manejo de los arreglos en Java. Finalmente, en el capítulo once se examina el manejo de archivos con Java.

Introducción al lenguaje Java

CAPÍTULO

1

Competencias

- Reconoce los conceptos básicos del lenguaje Java.
- Identifica los componentes de la máquina virtual de Java.
- Configura e instala las aplicaciones Java en una computadora personal.
- Implementa aplicaciones básicas usando NetBeans y Eclipse Mars.

Contenido

1.1 Generalidades
1.2 Lenguajes de programación
1.3 Java
1.4 Versiones Java
1.5 Orígenes del lenguaje Java
1.6 Características de la tecnología Java
1.7 Los programas en Java
1.8 La máquina virtual Java (JVM)
1.9 JDK
1.10 JRE
1.11 Aplicaciones Java
1.12 Instalando Java
1.13 El IDE NetBeans y su primera aplicación de plataforma
1.14 El IDE Eclipse Mars y su primera aplicación de plataforma
- Caso desarrollado 1: Monto de capital usando NetBeans
- Caso desarrollado 2: Venta de productos usando Eclipse Mars 2

1.1 Generalidades

Los tiempos han cambiado gracias a la nueva era digital en la que se vive, en la cual los aparatos electrónicos han tomado mucho valor. Hoy se puede ver una nueva generación de equipos inteligentes (televisores, autos, lavadoras, etc.), capaces de hacer la vida de sus usuarios mucho más cómoda. Uno de los puntos importantes para esta revolución es la programación, ya que es percibida como un segundo lenguaje en la era digital. Así, se puede afirmar que en el futuro todos los aparatos electrónicos se manejarán mediante aplicaciones desarrolladas que permitirán tener una personalización de las actividades que estas realizan.

Para lograr estos objetivos es necesario desarrollar aplicaciones, y esto solo se logrará dominando un lenguaje de programación, lo cual —en realidad— no es una tarea difícil, pues es evidente en la actualidad el gran avance de la informática; no obstante, la gran mayoría de las personas aún está logrando adaptarse a ello. Así pues, en la era del desarrollo, desde temprana edad, ya se podrán crear aplicaciones, a medida, que solucionen algún proceso por más simple o complejo que este sea. Uno de los lenguajes de programación más conocidos es Java, pues se adapta mucho a los cambios y presenta una mejor performance al momento de aplicarse en aparatos electrónicos: televisores, relojes o aparatos móviles, como es el caso de Android.

Nuestro futuro está marcado por la programación, por eso nos vemos en la necesidad de aprender a desarrollar algo que nos permita dar solución a un determinado problema. Por ejemplo, en el Reino Unido se ha introducido la programación de *software* en el currículo de los colegios, la cual se desarrolla de forma obligatoria a partir de la educación primaria; este hecho se enfoca en tres fines que resulta importante analizar: el primero es para que los escolares sean capaces de entender el código fuente de cualquier programa; el segundo, para conocer los pilares fundamentales de la programación; y, finalmente, el tercero, para desarrollar sus propias aplicaciones enfocadas a sus necesidades. Esto significa que dentro de unos años se tendrá una nueva generación de programadores que poseerán una visión distinta con respecto a la programación; para ellos será como aplicar matemáticas al desarrollar una aplicación. Asimismo, Estados Unidos, mediante su organización Code.org, promueve un programa llamado la Hora del Código (del inglés *Hour of Code*), que incentiva a los estudiantes para que opten por un curso alternativo de desarrollo de aplicaciones. En el Perú, Microsoft promueve su campaña #YoPuedoProgramar, dirigida a estudiantes de primaria en un intento por difundir las posibilidades de programación informática con un programa llamado Una Hora de Código. Así pues, se puede advertir que todo lo dicho es un pequeño paso a la formación de futuros programadores.

Finalmente, se puede decir que el desarrollo del *software* es uno de los principales impulsores de la innovación en el mundo, lo cual se debe a la nueva generación de aparatos inteligentes y muchas cosas más que vendrán en un futuro no muy lejano. Todos los profesionales de distintas carreras deberán estar en un continuo proceso de búsqueda y aprendizaje de nuevos lenguajes de programación que se adapten a sus necesidades; esto se puede dar usando lenguajes como C#, PHP o Java, los cuales en la actualidad dominan el futuro de la programación. También se debe considerar que detrás de todos los lenguajes habidos y por haber se encuentran empresas muy influyentes como Google, Microsoft, Oracle, que siempre estarán presentes en la evolución de la programación.

1.2 Lenguajes de programación

Para entender el concepto de *lenguaje de programación*, se analizarán las palabras *lenguaje* y *programación* por separado:

- **Lenguaje.** Es un sistema de comunicación estructurado que permite la comunicación o expresión de una idea, usando algunos elementos como símbolos, señales, etc.
- **Programación.** Es implementar una aplicación bajo el proceso de diseñar, codificar, depurar y mantener el código fuente de un programa.

Entonces, se puede decir que un lenguaje de programación está diseñado para realizar procesos que pueden ser llevados a cabo por dispositivos electrónicos. Asimismo, está formado por un conjunto de caracteres especiales y reglas sintácticas o formatos que definen su estructura. Finalmente, se puede agregar que el proceso de escribir, probar, depurar, compilar y mantener el código fuente de un programa se llama *programación*.

Según TIOBE, empresa de *software* holandesa que se especializa en la evaluación y seguimiento de la calidad de los programas informáticos, en su última actualización de octubre del 2017, asegura que los lenguajes de programación mantienen un mismo puesto según su popularidad en estos últimos años, tal como se muestra en la siguiente tabla:

Tabla 1.1 Listado de popularidad de los lenguajes de programación.

Oct. 2017	Oct. 2016	Lenguaje de programación	Puntuación
1	1	Java	12.431 %
2	2	C	8.374 %
3	3	C++	5.007 %
4	4	C#	3.858 %
5	5	Python	3.803 %
6	6	JavaScript	3.010 %
7	7	PHP	2.790 %
8	8	Visual Basic .NET	2.735 %

Fuente: <https://www.tiobe.com/tiobe-index/>

Es evidente que Java es más popular que los otros lenguajes de programación; quizás este efecto se haya producido en estos últimos tiempos por la evolución de la programación móvil y Android. Si se refiere a la demanda de empleo, Java ocupa un gran segundo lugar, como se puede ver en la siguiente imagen:

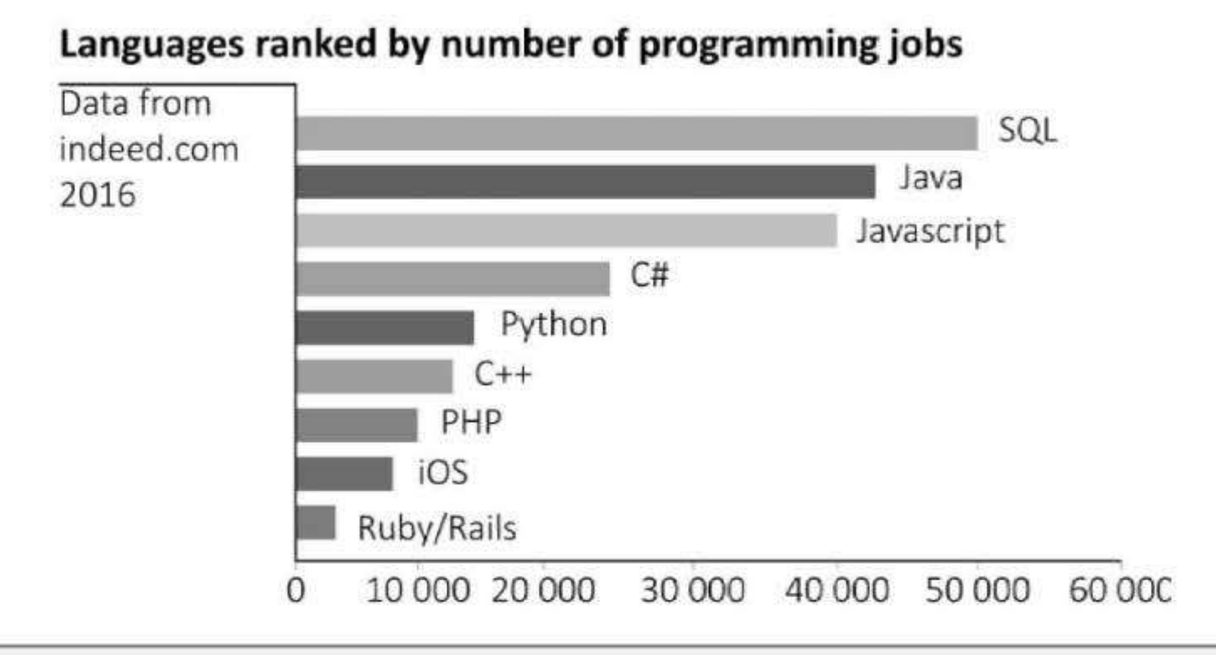


Figura 1.1 Listado de empleabilidad de los lenguajes de programación.
Fuente: <https://www.adslzone.net/app/uploads/2016/06/Programming-Languages-for-2016_graph.jpg>

1.3 Java

Con la evolución de la tecnología surgió un nuevo término en los usuarios, como lo es *Java*. Es muy probable que se haya escuchado o hasta se haya usado dicho lenguaje, y esto porque el código Java se encuentra en un gran número de dispositivos, especialmente móviles. Java es considerado un lenguaje de programación orientado a objetos, promovido inicialmente por la empresa Sun Microsystem, que luego fue comprada por Oracle en el año 2010. La idea principal de Java siempre fue que los desarrolladores implementaran el código una sola vez y que estos pudieran ejecutarlo en muchos dispositivos sin importar su plataforma de trabajo o, mejor dicho, su sistema operativo. Todo lo mencionado con respecto a Java se debe en gran parte a la máquina virtual de Java llamada JVM, la cual brinda la portabilidad necesaria al lenguaje, ya que hoy, para cada dispositivo, existe una máquina virtual que entiende, interpreta y compila todo el código Java de manera efectiva y rápida.

La empresa Sun Microsystems desarrolló en 1991 el lenguaje de programación Java; inicialmente sería usado por los dispositivos decodificadores de señales televisivas. El proyecto Java tuvo como primer nombre Oak, luego migró a Green para finalmente denominarse Java. Se toma como base del proyecto Java a la sintaxis C y C++, obteniendo un modelo de objetos simple y sencillo. Asimismo, es importante mencionar que la base fundamental de Java se debe principalmente a la programación orientada a objetos, a la multiplataforma, al soporte para trabajo en red, a ejecutar código en servidores remotos, teniendo el control de la seguridad y la facilidad de uso de su lenguaje.

En resumen:

Nombre del proyecto	Oak–Green–Java
Año de aparición	1995
Empresa original	Sun Microsystem
Empresa actual	Oracle (2010)
Licencia	GNU GPL
Logo	

1.4 Versiones Java

Las versiones que presenta Java a lo largo de su existencia se deben al incremento de clases y paquetes que componen su biblioteca. Desde la J2SE 1.4 todas las versiones superiores han sido reguladas por la Java Community Process, la cual promueve cambios oficiales en la plataforma Java.

Tabla 1.2 Listado de versiones y características de Java.

Versión	Fecha de lanzamiento	Aspectos importantes
JDK 1.0	Enero 1996	■ Reestructura el modelo de eventos AWT.
JDK 1.1	Febrero 1997	■ Incorpora clases internas. ■ Incorpora JavaBeans. ■ Incorpora JDBC (Java Database Connectivity) para la integración de bases de datos. ■ Incorpora RMI (Remote Method Invocation).
J2SE 1.2	Diciembre 1998	■ Incorpora la palabra reservada *keyword*. ■ Implementa la reflexión en la programación, es decir, modifica su estructura en alto nivel. ■ Incorpora la API gráfica Swing inicialmente con clases básicas. ■ Incorpora el compilador JIT a la máquina virtual de Java. ■ Incorpora colecciones.
J2SE 1.3	Mayo 2000	■ Incluye la máquina virtual de HotSpot JVM. ■ El RMI fue cambiado para que se basara en CORBA, de modo que las aplicaciones puedan correr en diferentes arquitecturas. ■ Incorpora el API JavaSound para gestión de audio en las aplicaciones. ■ Incorpora un conjunto de API Java Platform Debugger Architecture (JPDA).
J2SE 1.4	Febrero 2002	■ Incorpora la palabra reservada *assert*. ■ Incorpora expresiones regulares modeladas al estilo del lenguaje Perl. ■ Incorpora la encadenación de excepciones, la cual permite a una excepción encapsular a otra de bajo nivel. ■ Incorpora el API I/O para la lectura y escritura de imágenes en formatos como JPG o PNG. ■ Incorpora Parser XML integrado y procesador XSLT (JAXP). ■ Añade características de seguridad integrada y extensiones criptográficas como la JCE, la JSSE y la JAAS. ■ Incorpora el Java Web Start, disponible para ejecuciones desde un servidor web.

Versión	Fecha de lanzamiento	Aspectos importantes
J2SE 5.0	Septiembre 2004	■ Incorpora conversiones de tipos *type safety* en tiempo de compilación para colecciones y elimina la necesidad de la mayoría de conversión de tipos (*type casting*). ■ Incorpora metadatos o también llamados anotaciones. ■ Incorpora Autoboxing/unboxing — Conversiones automáticas entre tipos primitivos (como los *int*) y clases de envoltura primitivas (como *Integer*). ■ Incorpora enumeraciones como, por ejemplo, los días de la semana. ■ Actualiza el bucle for que es parecido al ForEach.
Java SE 6	Diciembre 2006	■ Incorpora un nuevo marco de trabajo y las API que hacen posible la integración de Java con otros lenguajes como PHP, Python, Ruby y JavaScript. ■ Incorpora el motor Rhino de Mozilla. ■ Incorpora un cliente completo de servicios web. ■ Incorpora soporte para servicios web como JAX-WS 2.0, JAXB 2.0, STAX y JAXP. ■ Incorpora mejoras en el rendimiento.
Java SE 7	Julio 2011	■ Incorpora soporte para XML dentro del propio lenguaje Java. ■ Incorpora soporte para *closures*. ■ Incorpora una nueva API para el manejo de días y fechas, la cual reemplaza a las antiguas clases Date y Calendar.
Java SE 8	Marzo 2014	■ Incorpora de forma completa la librería JavaFX mejorando la experiencia RIA. ■ Incorpora diferentes mejoras en seguridad. ■ Incorpora diferentes mejoras en concurrencia. ■ Mejora los procesos con el uso de funciones Lambda. ■ Mejora la integración de JavaScript. ■ Incorpora nuevas API para manejo de fechas y tiempo (*date–time*).
Java SE 9	Septiembre 2017	■ Mejora la clase Optional. ■ Mejora la API de Streams. ■ Incorpora la herramienta JShell. ■ Incorpora los Strings compactos, haciendo que los caracteres usen solo un byte en vez de dos. ■ Incorpora el JLink que es un sustituto más capaz que los *compact profiles*. ■ Se puede generar un único archivo *jar* con algunas clases para una o varias versiones de Java. ■ Se cambia el recolector de basura clásico por el G1 optimizado para un balance adecuado entre alto un rendimiento y una baja latencia.

1.5 Orígenes del lenguaje Java

A finales de los ochenta, la empresa Sun Microsystems promueve un proyecto de investigación dirigido por James Gosling con el objetivo de desarrollar un *software* para determinados aparatos electrónicos.

Sun Microsystems fue fundada en 1982 por Andreas Bechtolsheim, Vinod Khosla, Scott McNealy y Bill Joy como un proyecto inicial promovido por Bechtolsheim. Su objetivo era crear una estación de trabajo en red mucho más eficaz y, especialmente, de menor costo a las de la época; justamente, el nombre del proyecto es el que da el nombre a la compañía Stanford University Network (SUN).

Volviendo a la historia de Java, el equipo de Gosling buscaba crear un *software* que sea independiente de la arquitectura y que pueda ser ejecutado en otra plataforma sin problemas. Encontraron una base a sus estudios en los lenguajes C y C++, los cuales cumplían los requerimientos buscados. Como resultado de esta búsqueda, en 1990, Gosling empezó a diseñar un nuevo lenguaje de programación, llamado inicialmente Oak.

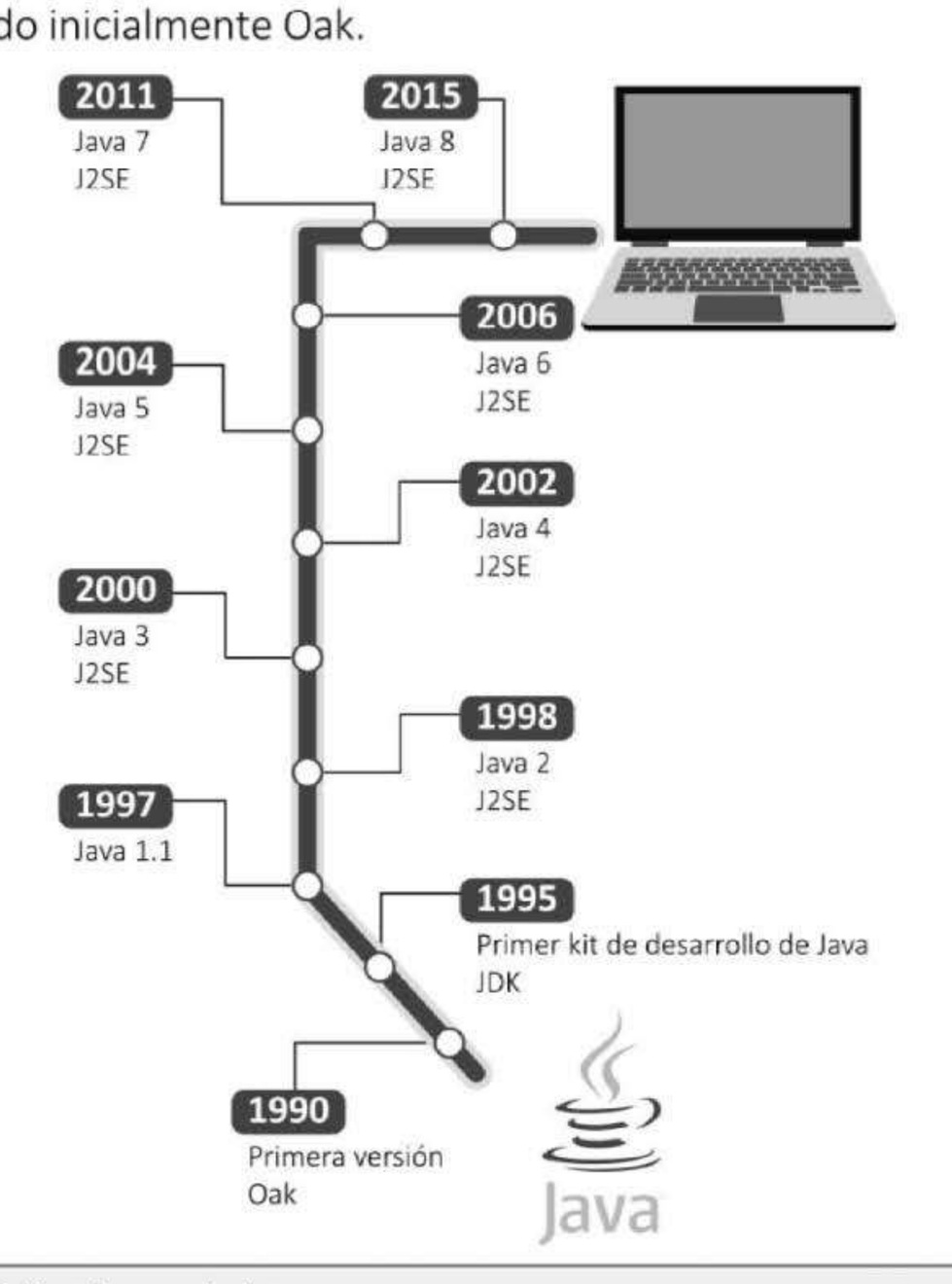


Figura 1.2 Versiones de Java.

En el año 1993, aparece en el mercado informático la Word Wide Web revolucionando a todo el proyecto, ya que ahora el equipo ponía sus ojos en la Internet debido a la diversidad de computadoras que se encuentran conectadas a la red. Es ahí que Sun Microsystem toma en serio el desarrollo de Java como un lenguaje de programación orientado a Internet. Para poder ejecutar aplicaciones Java, escribieron un Web Browser que llamaron HotJava, el cual tenía como finalidad soportar los applets Java, que eran aplicaciones Java incrustadas en un documento web.

Conforme se tenían avances considerables en el proyecto Java, se estaba publicando el navegador NetScape Navigator, el cual soportaría los applets Java. El resultado de esta fusión conllevó a un fuerte aumento de programadores Java en el mundo y, especialmente, despertó el interés de grandes empresas como IBM, SGI y la misma ORACLE, la cual termina haciendo una alianza con Sun para la compra de su tecnología Java.

Cabe destacar que la tecnología Java se encuentra respaldada por una gran comunidad de desarrolladores en el mundo, que trabajan activamente para mantener al lenguaje Java como uno de los lenguajes de programación más usados de la historia de la informática. Por otro lado, hay que mencionar que existe el Java Community Process establecido desde el año 1998, el cual nos permite estar informados sobre futuras versiones y características de la plataforma Java.

1.6 Características de la tecnología Java

Java no solo es un lenguaje de programación orientado a objetos, también presenta otras características que lo convierten en el lenguaje preferido por los programadores y por las empresas que, en la actualidad, optan por su uso. A continuación, se presenta una lista de estas características:

- **Orientada a objetos**

  Su tecnología se basa en la reutilización del código mediante métodos implementados en clases. Todo lo que se use en Java se encontrará encapsulado mediante clases que podrán ser usados a través del acceso a objetos.

- **Funcionalidad dinámica**

  Se podrá modificar el código Java sin necesidad de volverlo a crear. Es aquí donde se podrá añadir funcionalidad a las aplicaciones sin realizar cambios en el código inicial, solo será un tema de actualización mediante herencia de métodos o atributos.

- **Multiplataforma**

  Una aplicación desarrollada con Java se podrá ejecutar en diferentes sistemas operativos, pero esta tarea no la realiza el sistema operativo, sino la máquina virtual de Java. Así, el sistema operativo adopta a Java para aplicaciones en dispositivos móviles.

- **Open source**

  Las librerías nativas de Java están disponibles libremente para los desarrolladores; esto implica que lo podrán leer, corregir, actualizar y, especialmente, mejorar, haciendo que Java se mantenga siempre actualizado y disponible a todo tipo de usuario.

- **Lenguaje robusto**

  Java desarrolla aplicaciones para cualquier plataforma de trabajo, pues su lenguaje es considerado como potente y robusto entre otros lenguajes; aquí es posible implementar aplicaciones usando hilos o frameworks como Hibernate, Spring, Struts, JSF, etc.

- **Seguridad**

  La seguridad es un tema importante para todo tipo de aplicación; por ello, Java continuamente libera actualizaciones que corrigen o previenen cualquier tipo de inseguridad, ya sea a nivel de aplicación o de redes.

- **Soporte y documentación – Comunidad Java**

  Gracias a las múltiples comunidades que existen en el planeta, se puede tener acceso a una documentación completa sobre la funcionalidad Java; en ese mismo sentido, Oracle pone a nuestra disposición toda la documentación sobre nuevas clases, métodos y componentes del lenguaje Java.

1.7 Los programas en Java

Normalmente, cuando se escucha la frase «programar en Java», se piensa en digitar un código y ejecutarlo en un determinado dispositivo, pero detrás de ello ocurren muchas actividades que probablemente se pueden suponer; no obstante, será necesario especificarlas para entender la cultura que maneja Java y así poder hablar con propiedad la próxima vez que se escuche hablar de programación en Java. Seguidamente, se describirán las actividades que ocurren para ejecutar un programa implementado en Java.

1. **Edición.** Esta actividad es conocida con el nombre *generación del script* o *código fuente*, la cual se puede ejecutar en aplicaciones que incluyan un editor de texto como Eclipse, NetBeans, etc. Hay que tener en cuenta que para tener un script se deben respetar las reglas Java como sintaxis, palabras reservadas, variables, asignaciones, objetos de clase, entre otras. La extensión del archivo que contiene el script Java es .java.

Tabla 1.3 Principales editores de código para Java.

Eclipse **Versión: Mars2**	Eclipse MARS	**Ventajas**: ■ Gestiona de manera profesional los proyectos. ■ Presenta un gran soporte para *plugins*. ■ Presenta opciones de personalización sobre su entorno. ■ Ofrece herramientas para repositorios y servidores. **Desventajas**: ■ Tiene un mayor consumo de recursos en *hardware*.
NetBeans **Versión: 8.2**	NetBeans	**Ventajas:** ■ Presenta una eficiente gestión sobre los proyectos. ■ Identifica y soluciona problemas de código comunes. ■ Presenta una librería visual amigable al usuario. ■ Presenta herramientas de desarrollo integrado. **Desventajas:** ■ Genera código basura al usar los controles *arrastrar* y *soltar*. ■ Las librerías y complementos hacen un poco pesada la aplicación.
Oracle JDeveloper **Versión: 12c**	ORACLE JDEVELOPER	**Ventajas:** ■ Genera código de forma semiautomática. ■ Tiene un editor UML. ■ Posee un creador de interfaces visuales. **Desventajas:** ■ Es lenta y presenta un consumo mayor de *hardware*.

Todas las plataformas de desarrollo contienen un editor de código, en el cual se podrá digitar el código Java, de modo que se genere el código fuente de la aplicación.

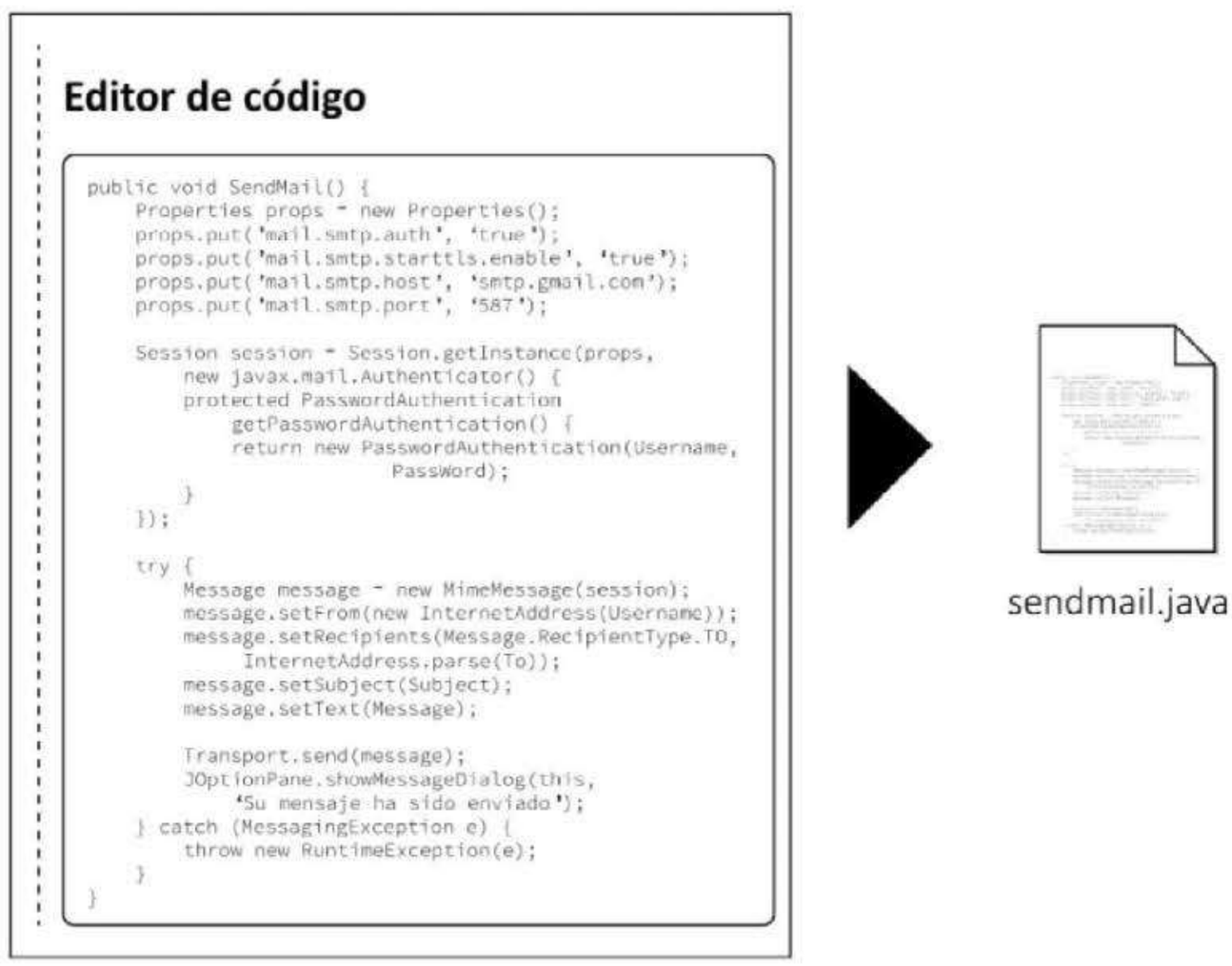


Figura 1.3 Editor de código.

2. **Compilación.** Esta actividad genera un nuevo archivo de extensión .class, el cual contiene un código llamado ByteCode ('código de bytes'). Este código es considerado como un código intermedio, cuyo objetivo es hacer que el dispositivo entienda el código Java.

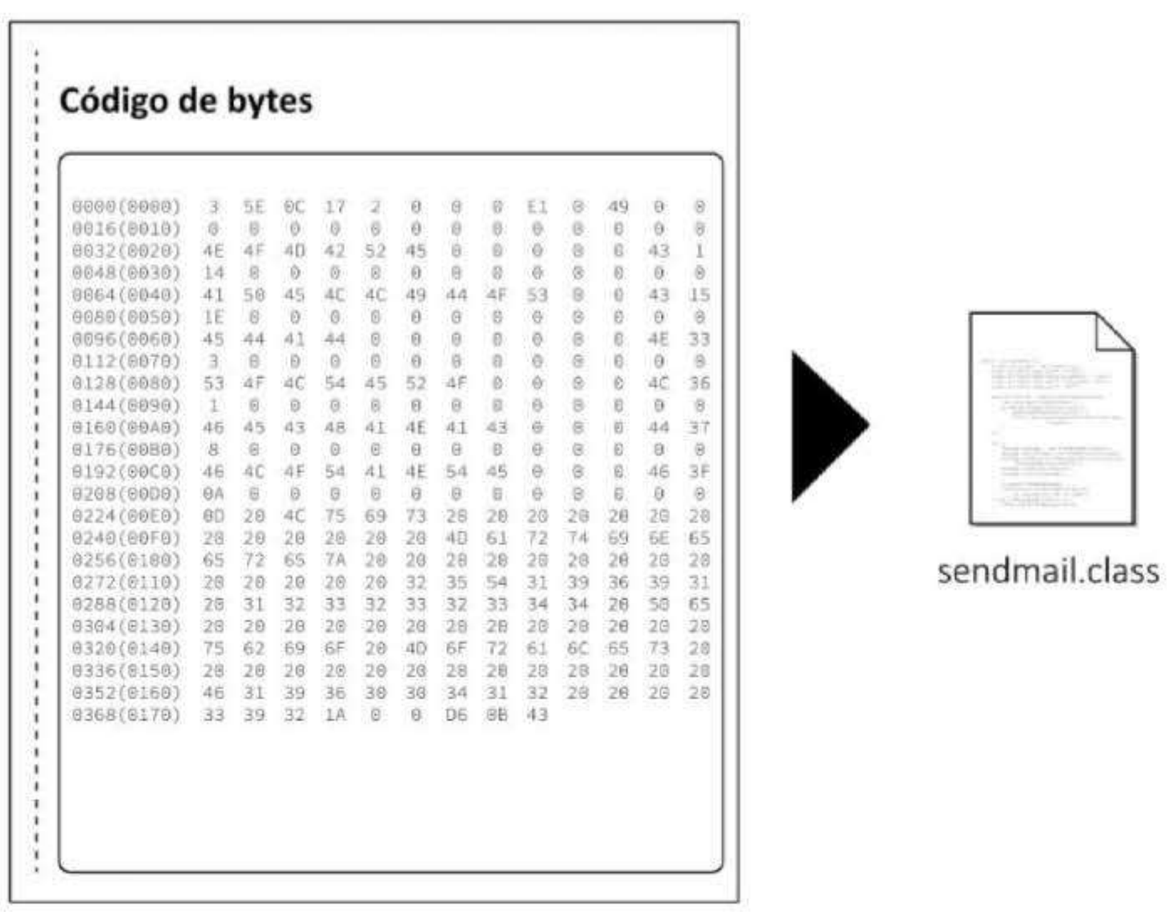


Figura 1.4 Código de bytes.

3. **Carga.** Esta actividad envía el código de bytes generado a la memoria principal del dispositivo por medio del cargador de clases obtenido desde la unidad donde se guardaron los archivos java y class.

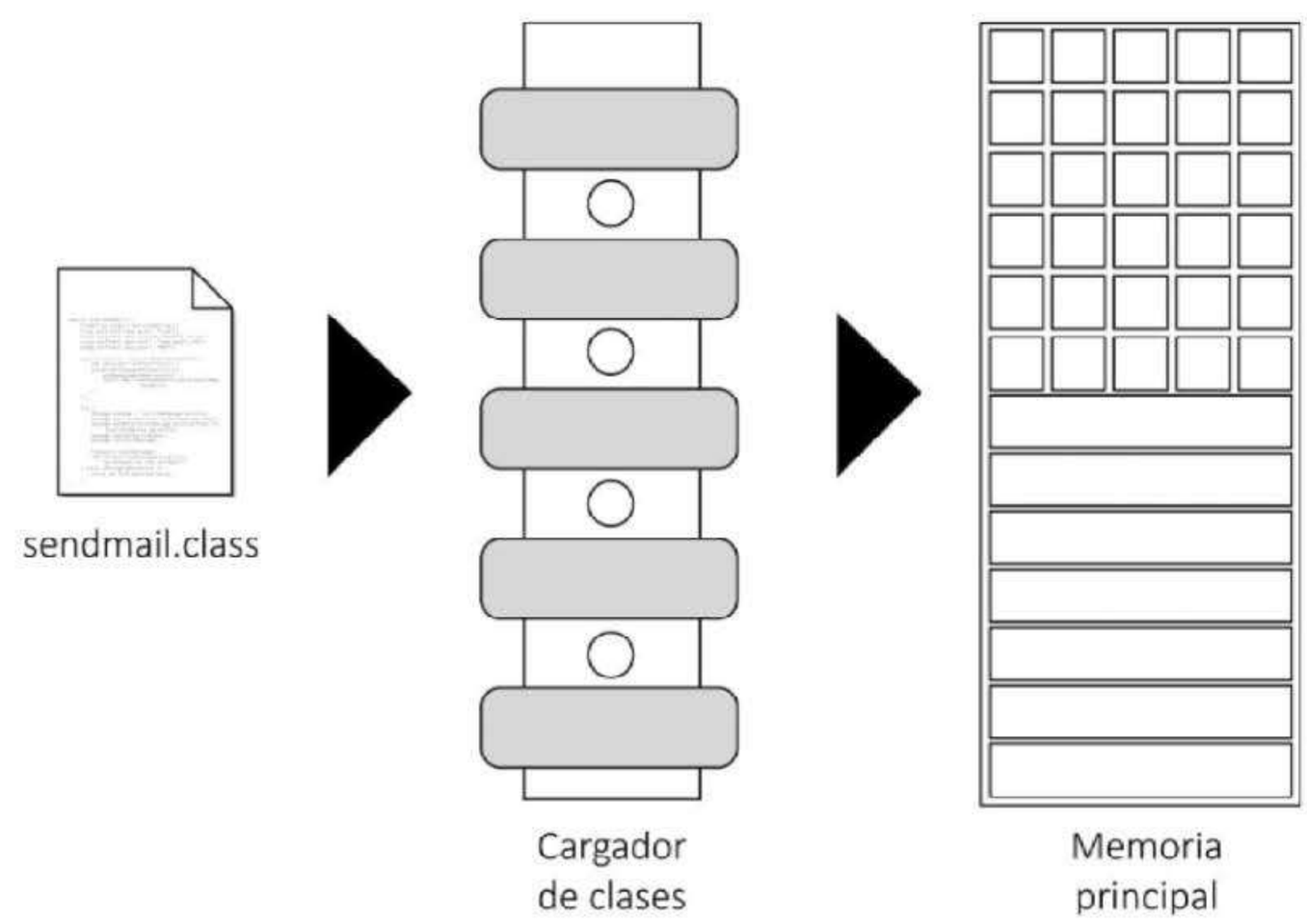


Figura 1.5 Cargador de clases.

Las clases son enviadas al cargador de clases, el cual apila cada clase para luego ser enviada a la memoria del computador central, y así agilizar su respectivo proceso.

4. **Interpretación.** Esta actividad verifica que el código de byte sea válido y lo pueda convertir en un código entendible por el sistema operativo donde se va a ejecutar la aplicación o programa.

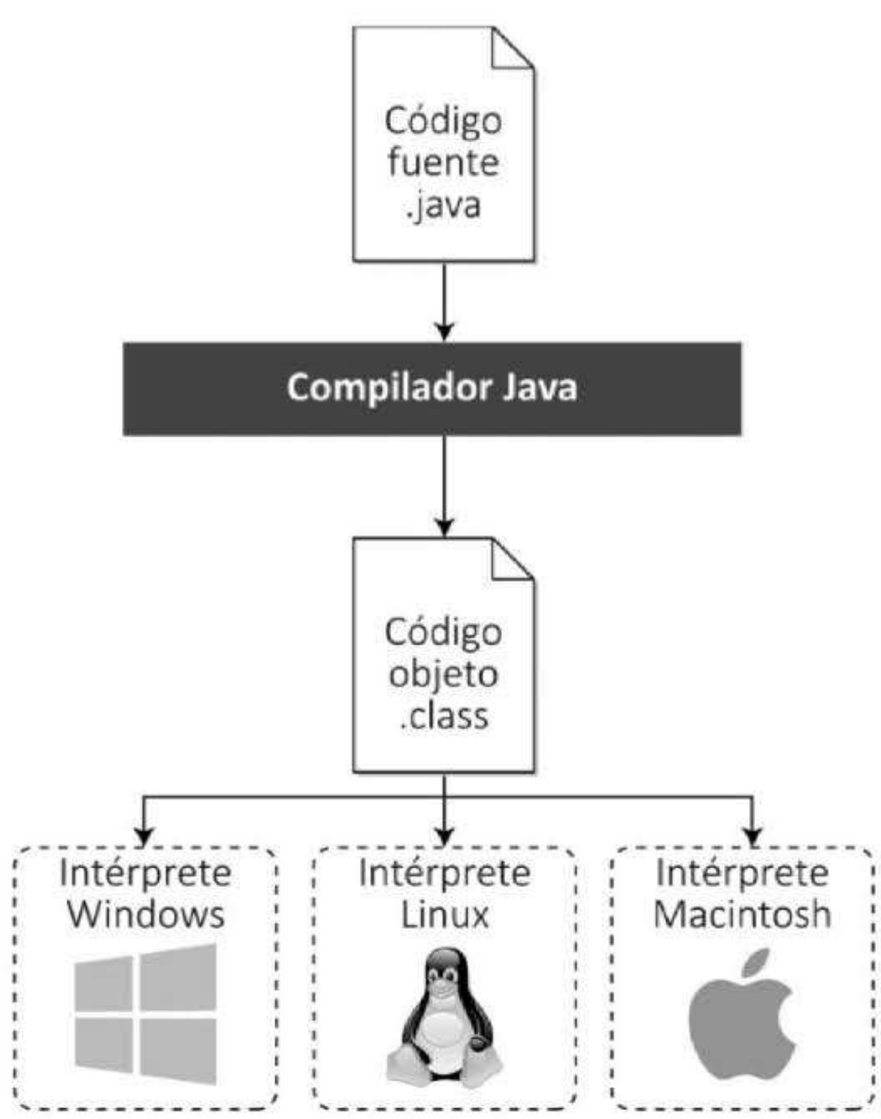


Figura 1.6 Proceso de interpretación de código en diferentes sistemas operativos.

5. **Ejecución.** Esta actividad envía el código de bytes depurado a la máquina virtual Java (JVM) para su ejecución y visualización en el entorno que el usuario crea conveniente; de aquí se desprende el término de *portabilidad*, ya que dependiendo de la JVM se podrá ejecutar la aplicación de una u otra plataforma.

   En resumen, cuando se crea un archivo con extensión Java, un compilador convierte dicho archivo en un código de bytes y lo almacena en un archivo de tipo class; este código es enviado al intérprete, el cual lo convierte en algo entendible por los sistemas operativos de cualquier dispositivo que lo soporte.

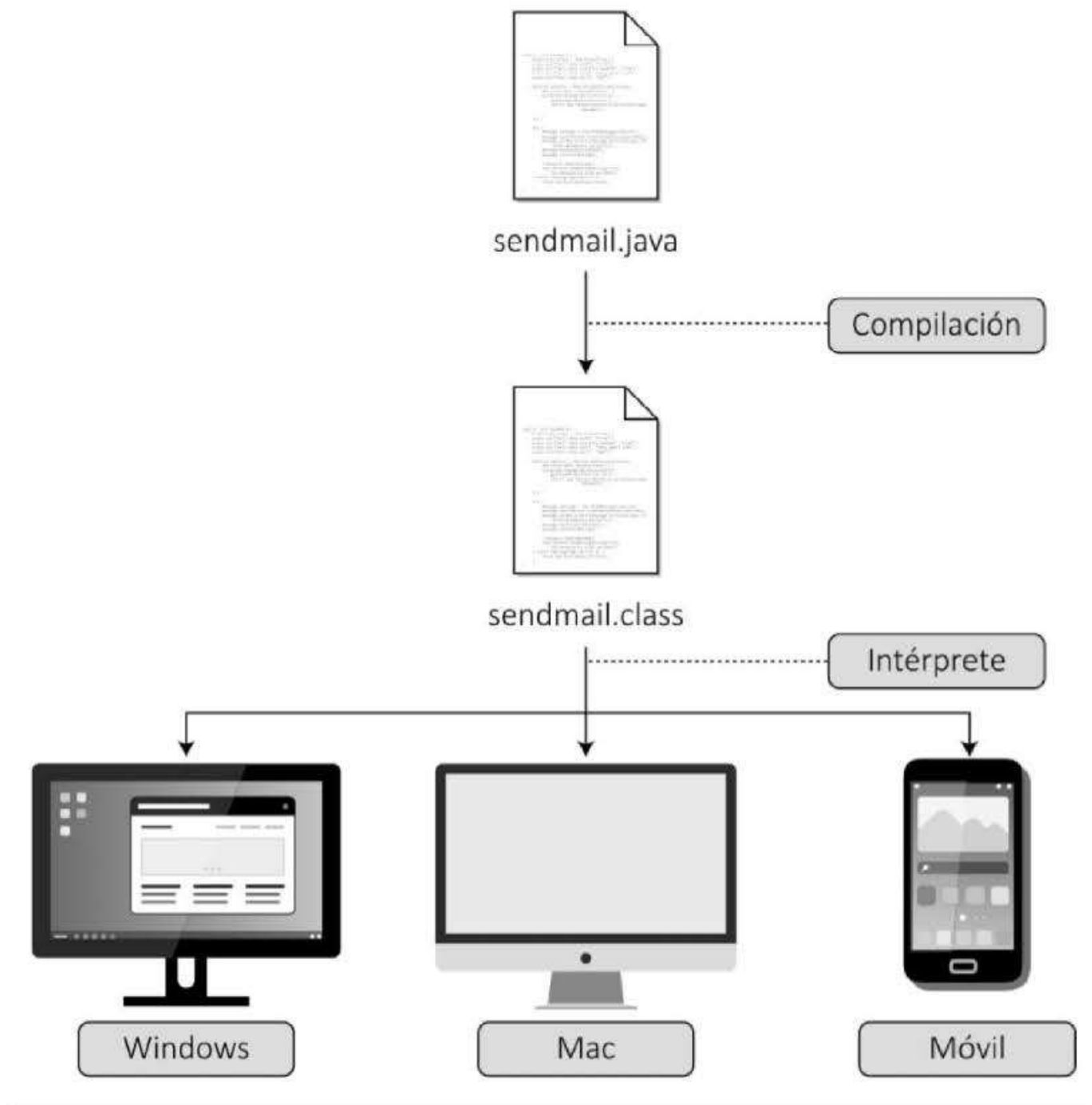


Figura 1.7 Actividades para crear un programa Java.

1.8 La máquina virtual Java (JVM)

Es un entorno donde se ejecutan todos los programas Java, normalmente la JVM es nativa del sistema operativo y viene cargada por defecto. Su característica principal es que se puede ejecutar en una plataforma específica; además, es capaz de interpretar y ejecutar código de bytes que luego será visualizado en el dispositivo.

Asimismo, la JVM garantiza la portabilidad de las aplicaciones Java, ya que define un procesador virtual sobre cualquier dispositivo que lo soporte y especifica qué instrucciones se ejecutarán en este dispositivo. En la figura 1.7 se pueden ver algunas de las características que presenta la JVM.

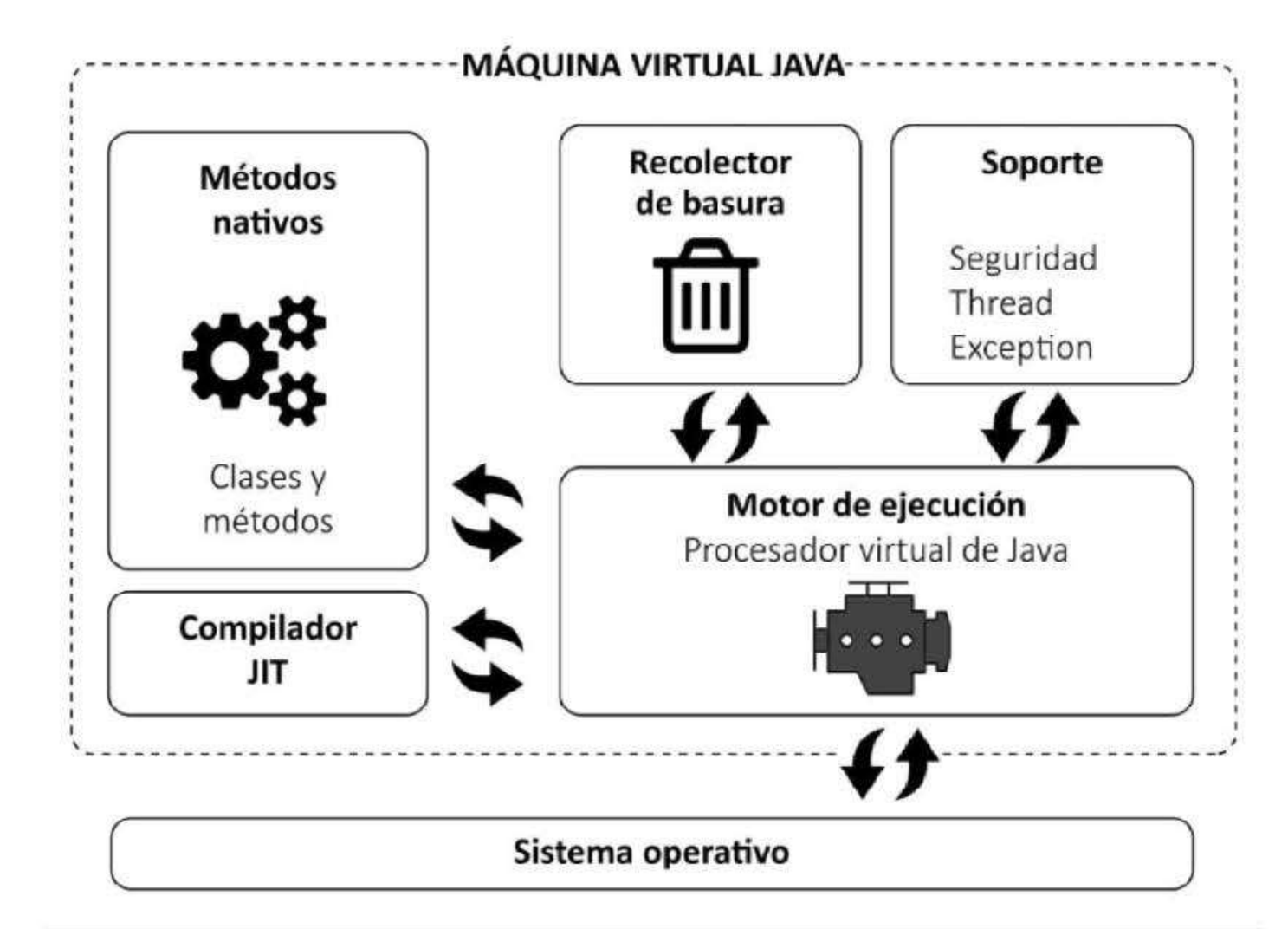


Figura 1.8 Estructura de la JVM.

Se les llama *métodos nativos* a aquellos procedimientos que fueron escritos en otros lenguajes y que son usados en Java; la mayoría de estos fueron heredados de C y C++. Las clases y sus métodos son almacenados de manera lógica en forma de pilas dentro de un área llamada Stream de clases de la máquina virtual; estas clases contienen la siguiente información:

- Un identificador de clase, el cual contiene la dirección del paquete, que incluye la clase más el nombre de la clase, como, por ejemplo, javax.swing.Jlabel.
- Un identificador de la clase padre.
- Información que identifica si es una clase o una interface.
- Una lista de identificadores de interfaces.

El *recolector de basura* ayuda a una mejor gestión de la memoria; primero identifica qué objetos son necesarios para la aplicación y elimina los que no son necesarios. La *línea de soporte* menciona principalmente a tres componentes: la seguridad, los hilos o trabajo en concurrencia y el manejo de excepciones. Toda esta información es enviada al *motor de ejecución* de la JVM para ser entendida por el *sistema operativo* del dispositivo donde se ejecutará la aplicación Java.

1.9 JDK

El Java Development Kit (JDK) o 'kit de desarrollo Java' contiene un conjunto de herramientas y librerías que son de vital importancia en la creación de una aplicación Java. Antes de instalar un entorno de desarrollo como Eclipse, NetBeans o JDeveloper, se tiene que instalar un JDK en su sistema operativo.

Es importante mencionar que algunos entornos de trabajo tienen incluido el JDK, pero se recomienda que se instale de manera independiente. Existe un JDK para cada tipo de sistema operativo, esto se debe gracias a la distribución hecha por Oracle (<http://www.oracle.com/technetwork/es/java/javase/downloads/index.html>). A partir de la versión

JDK 1.2 se le comenzó a llamar Java 2; en la actualidad, se presenta la distribución del JDK 1.8, es decir, Java 8.

Algunas de las utilidades que presentan, son las siguientes:

- **Javac.** Es el compilador de Java que está encargado de convertir el código fuente de Java en código entendible por los sistemas operativos (bytecode).
- **Java.** Es el intérprete de Java, desde aquí el sistema operativo entenderá nuestra aplicación.
- **Javadoc.** Es el encargado de generar la documentación de todas las clases en una aplicación Java.
- **Javap.** El comando javap desensambla un fichero class. Su salida depende de las opciones utilizadas. Si no se utilizan opciones, javap imprime los campos y los métodos públicos de la clase pasada. Javap muestra la salida en stdout.
- **JRE.** Es considerado el archivo ejecutable de Java.

1.10 JRE

El Java Runtime Environment (JRE) actúa como un intermediario entre el sistema operativo donde reside la aplicación y Java. EL JDK incluye toda la funcionalidad que ofrece el JRE para la implementación de una aplicación Java.

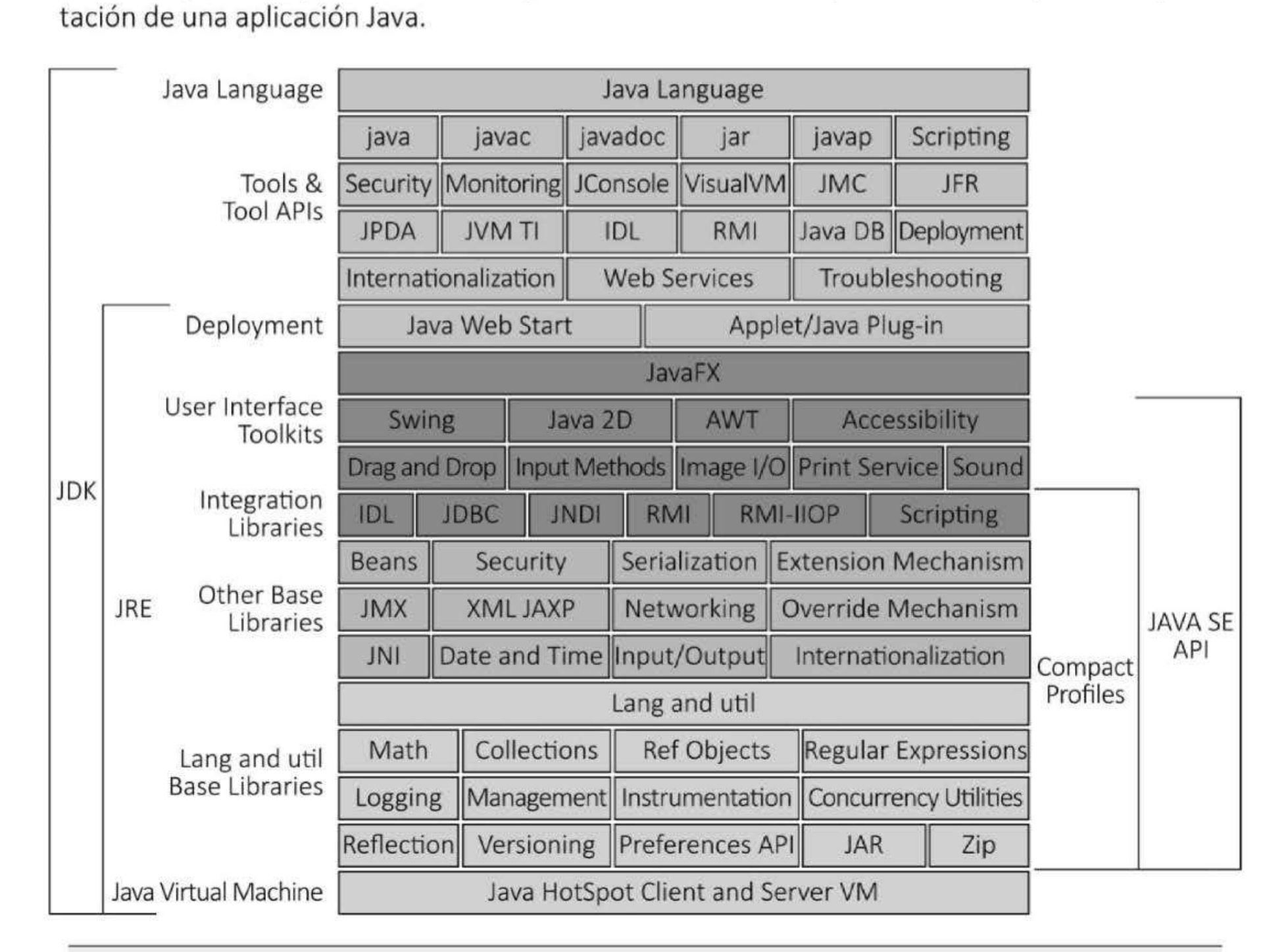


Figura 1.9 Lista de capacidades que ofrece el JDK y JRE.
Fuente: <http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/2167990.jpg>

1.11 Aplicaciones Java

Java es un lenguaje de programación multiplataforma, por la variedad de procesadores donde se puede ejecutar la aplicación, y multisistema, porque asimismo se pueden ejecutar en muchos sistemas operativos. Es así como Java permite construir diversos tipos de aplicaciones que se listan a continuación:

a) **Aplicaciones de consola.** Son aplicaciones que se ejecutan en ambientes del tipo DOS o un terminal de Linux; la idea principal es crear una aplicación básica sin enfoque visual.

Figura 1.10 Aplicación de consola.

b) **Aplicaciones de escritorio.** Son aplicaciones que se presentan en forma de ventanas; también son llamadas *aplicaciones de plataforma*, como, por ejemplo, la aplicación de *control de venta de productos*.

Figura 1.11 Aplicación de escritorio.

c) **Aplicaciones tipo applets.** Son aplicaciones que se ejecutan bajo la plataforma de una página web; solo se debe tener en cuenta que el navegador web soporte applets e incluya la máquina virtual de Java.

Figura 1.12 Applet de Java.

d) **Aplicaciones tipo servlet.** Son aplicaciones que se ejecutan directamente en un servidor web y que, al ser invocado el servidor, se encargará de compilar y ejecutar el servlet, mostrando así la información al cliente mediante un código HTML.

e) **Aplicaciones JSP.** Son aplicaciones embebidas que combinan código HTML y código Java en una sola aplicación; cuando se realiza una petición de ejecución al servidor, este compila y lo convierte a servlet para mostrarle el resultado al cliente.

f) **Aplicaciones Beans y Java Beans.** Son aplicaciones de tipo servlet especializadas en la tecnología Cliente-Servidor, es decir, se podrá desarrollar aplicaciones con conexión a base de datos.

g) **Aplicaciones móviles.** Son aplicaciones que se ejecutan bajo un sistema operativo móvil como Android.

Figura 1.13 Aplicación Java vista desde un dispositivo móvil.

1.12 Instalando Java

Para instalar la máquina virtual de Java primero hay que considerar algunos términos de referencia como JRE o JDK. El Java Runtime Environment (JRE) es un conjunto de utilidades de Java que nos permitirá configurar e instalar la JVM; sin embargo, no permite el desarrollo completo de una aplicación Java. Asimismo, el Java Development Kit (JDK), el cual contiene el JRE y la JVM necesarios para implementar las aplicaciones en Java, generalmente se usa como intermediario entre el sistema operativo y Java.

Para iniciar con las aplicaciones con Java, se puede usar un Framework que contenga todo lo que necesita una aplicación, como el editor de lenguaje, compilador Java, JVM, librerías, etc. A continuación, se presentan los logotipos de los más conocidos: NetBeans, Oracle y Eclipse.

Figura 1.14 Frameworks de Java.

1.12.1 Instalación de JDK

Es requisito indispensable tener el JDK instalado, ya que este incluye la máquina virtual de Java.

Descarga

Desde la página oficial de Oracle, se descarga el JDK.

Java SE Development Kit 8u144

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.
Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.89 MB	jdk-8u144-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.83 MB	jdk-8u144-linux-arm64-vfp-hflt.tar.gz
Linux x86	164.65 MB	jdk-8u144-linux-i586.rpm
Linux x86	179.44 MB	jdk-8u144-linux-i586.tar.gz
Linux x64	162.1 MB	jdk-8u144-linux-x64.rpm
Linux x64	176.92 MB	jdk-8u144-linux-x64.tar.gz
Mac OS X	226.6 MB	jdk-8u144-macosx-x64.dmg
Solaris SPARC 64-bit	139.87 MB	jdk-8u144-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.18 MB	jdk-8u144-solaris-sparcv9.tar.gz
Solaris x64	140.51 MB	jdk-8u144-solaris-x64.tar.Z
Solaris x64	96.99 MB	jdk-8u144-solaris-x64.tar.gz
Windows x86	190.94 MB	jdk-8u144-windows-i586.exe
Windows x64	197.78 MB	jdk-8u144-windows-x64.exe

Figura 1.15 Listado de JDK para diferentes sistemas operativos.
Fuente: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Proceso de instalación

- Ubique el archivo JDK descargado.

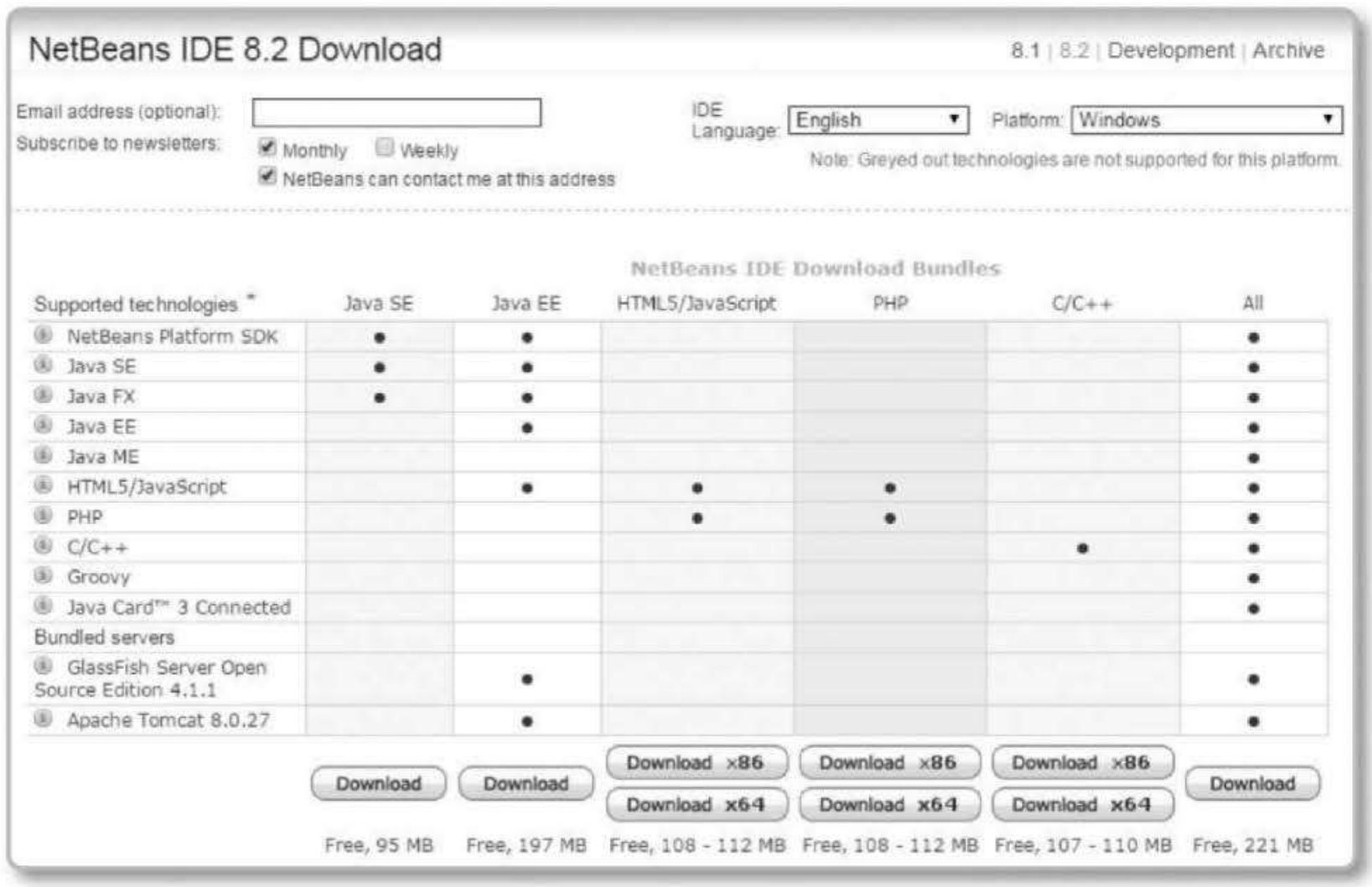
- Inicie la instalación ejecutando el archivo **jdk-8u144-windows-x64**.

- En estos primeros pasos, solo seleccionará el botón **Next**.

- Debe definir la carpeta de destino del JDK; se podría modificar seleccionando el botón **Cambiar...**, pero se recomienda conservar la carpeta por defecto **Program Files\Java**. Inicie la instalación seleccionando el botón **Siguiente**.

- A continuación, se muestra el avance de la instalación.

1.12.2 Instalación de NetBeans (versión 8.2)



NetBeans IDE 8.2 Download

8.1 | 8.2 | Development | Archive

Email address (optional):
Subscribe to newsletters: Monthly Weekly
NetBeans can contact me at this address

IDE Language: English Platform: Windows

Note: Greyed out technologies are not supported for this platform.

NetBeans IDE Download Bundles

Supported technologies *	Java SE	Java EE	HTML5/JavaScript	PHP	C/C++	All
NetBeans Platform SDK	•	•				•
Java SE	•	•				•
Java FX	•	•				•
Java EE		•				•
Java ME						•
HTML5/JavaScript		•	•	•		•
PHP			•	•		•
C/C++					•	•
Groovy						•
Java Card™ 3 Connected						•
Bundled servers						
GlassFish Server Open Source Edition 4.1.1		•				•
Apache Tomcat 8.0.27		•				•
	Download	Download	Download x86 / Download x64	Download x86 / Download x64	Download x86 / Download x64	Download
	Free, 95 MB	Free, 197 MB	Free, 108 - 112 MB	Free, 108 - 112 MB	Free, 107 - 110 MB	Free, 221 MB

Figura 1.16 Zona de descarga NetBeans 8.2.
Fuente: <https://netbeans.org/downloads/>

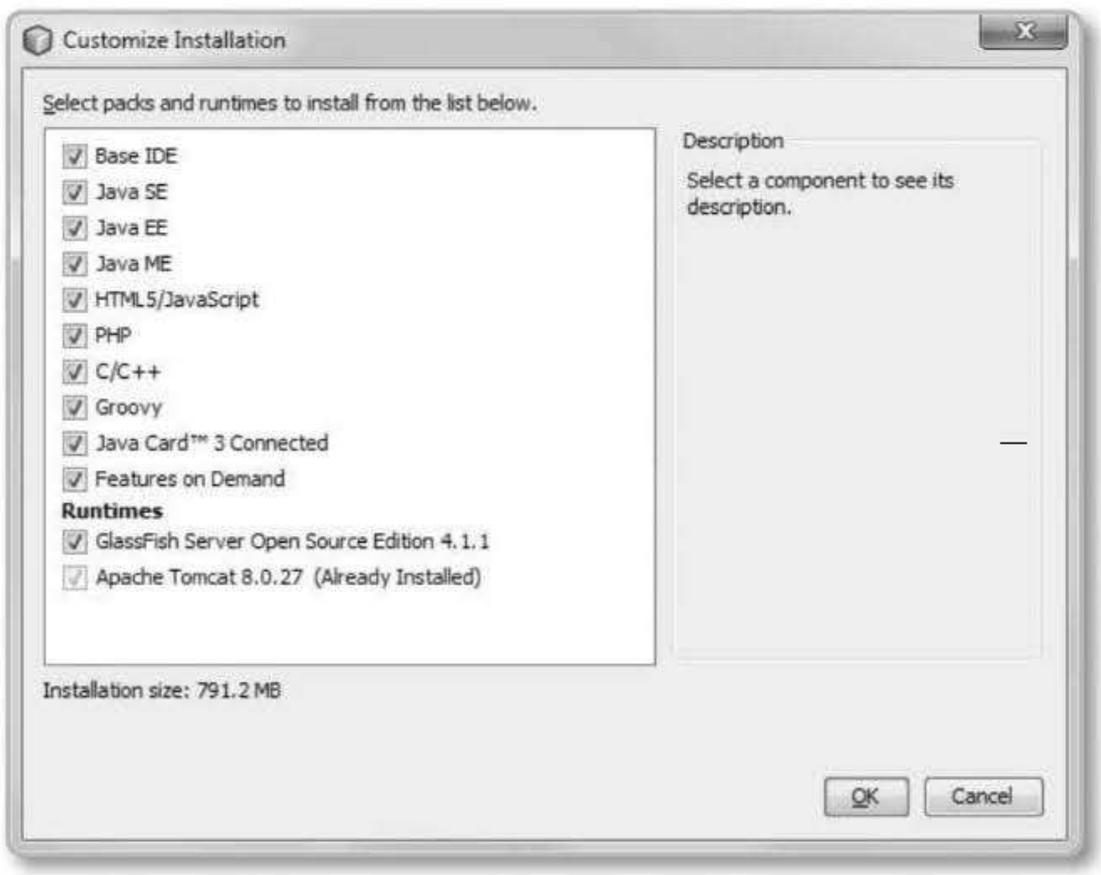
Es un entorno de desarrollo integrado (IDE), enfocado al desarrollo de aplicaciones, que usa el lenguaje de programación Java; pero, asimismo, permite el uso de otros lenguajes bajo el mismo entorno. Su característica principal radica en la distribución gratuita sin restricciones de uso.

Fue la empresa Sun Microsystems la que funda el proyecto de código abierto llamado NetBeans a mediados del año 2000. NetBeans soporta el desarrollo de todos los tipos de aplicación Java como la J2SE, aplicaciones web, aplicaciones EJB y aplicaciones móviles.

Para iniciar con la instalación, se debe descargar la versión completa del IDE, el cual tiene un peso de 221 MB. Una vez realizada la descarga, se seguirán los siguientes pasos:

- Al iniciar la instalación, se presenta la pantalla de bienvenida en la cual observará una lista de aplicaciones y el tamaño necesario para el proceso de instalación. Para continuar con la instalación, con valores predeterminados, seleccione el botón **Next**.

Por otro lado, puede seleccionar algunas aplicaciones particulares de la lista, para esto seleccione el botón **Customize...**, el cual mostrará la siguiente ventana de personalización de la instalación:



- La siguiente ventana muestra la aceptación de la licencia NetBeans, para lo cual solo debe activar el *check* **I accept the terms in the license agreement** y seleccionar el botón **Next**.

- La siguiente ventana muestra dónde se instalará la aplicación, pero es importante que tome principal interés en **JDKTM for the NetBeans IDE**, ya que según el paquete de instalación también está instalando la versión 1.8.0_91 del JDK.

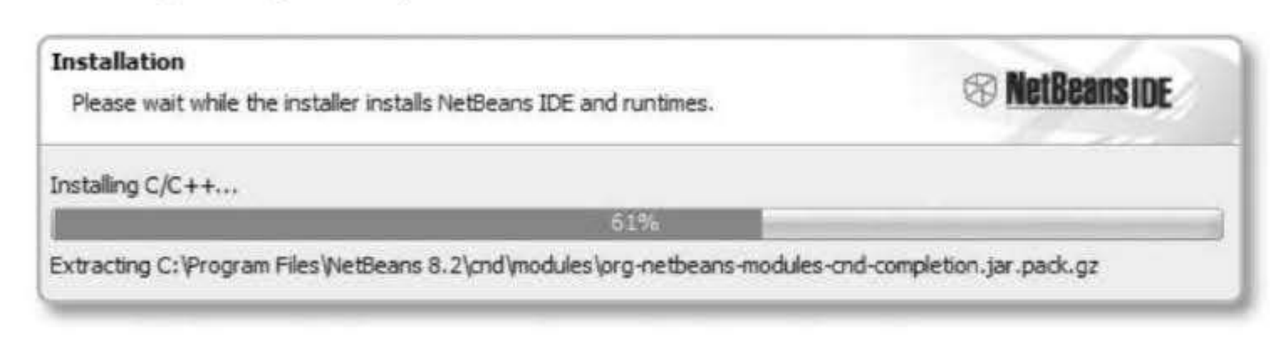
En el caso que la sección **JDKTM for the NetBeans IDE** se encontrara vacía, debe cancelar la instalación del NetBeans e instalar el JDK independientemente a partir de la siguiente URL:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

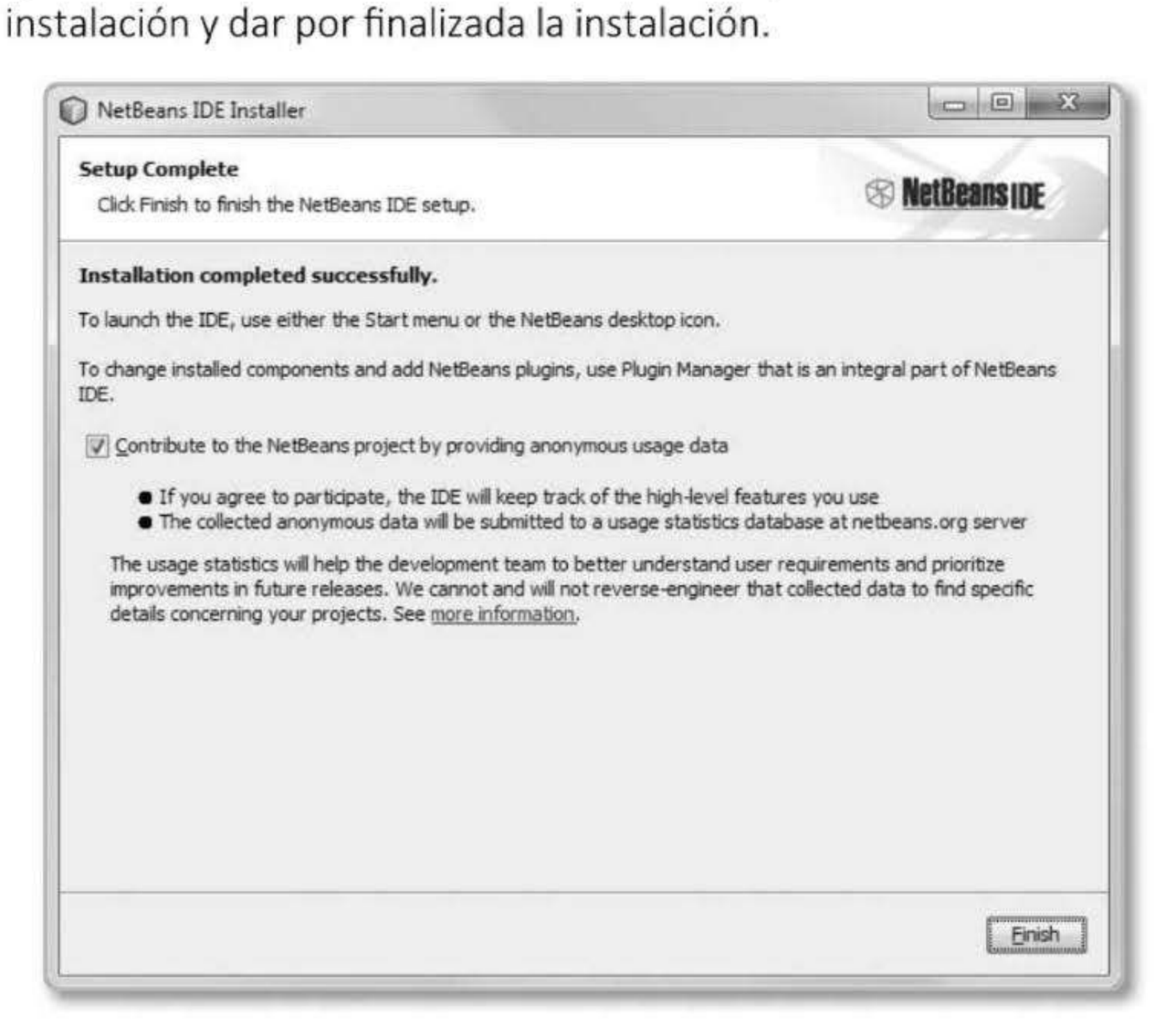
Para realizar la descarga, primero active la opción **Accept License Agreement**; luego, a partir del sistema operativo con que cuente, seleccione un paquete de instalación de JDK; para nuestro caso, seleccione el archivo perteneciente a Windows x64 (64 bits).

- A continuación, aparecerá una ventana de resumen con los datos de la ubicación de la instalación y el tamaño que ocupará en el disco. Se recomienda que active el *check* correspondiente a **Check for Updates**, el cual realiza una actualización de forma automática del producto NetBeans cada vez que se encuentre conectado a Internet. Haga clic en el botón **Install** para iniciar con la instalación del NetBeans.

- Ahora, solo tiene que esperar que NetBeans se instale en su unidad de disco.



- Finalmente, se muestra la ventana de instalación completa. Seleccione el botón **Finish** para salir de la instalación y dar por finalizada la instalación.



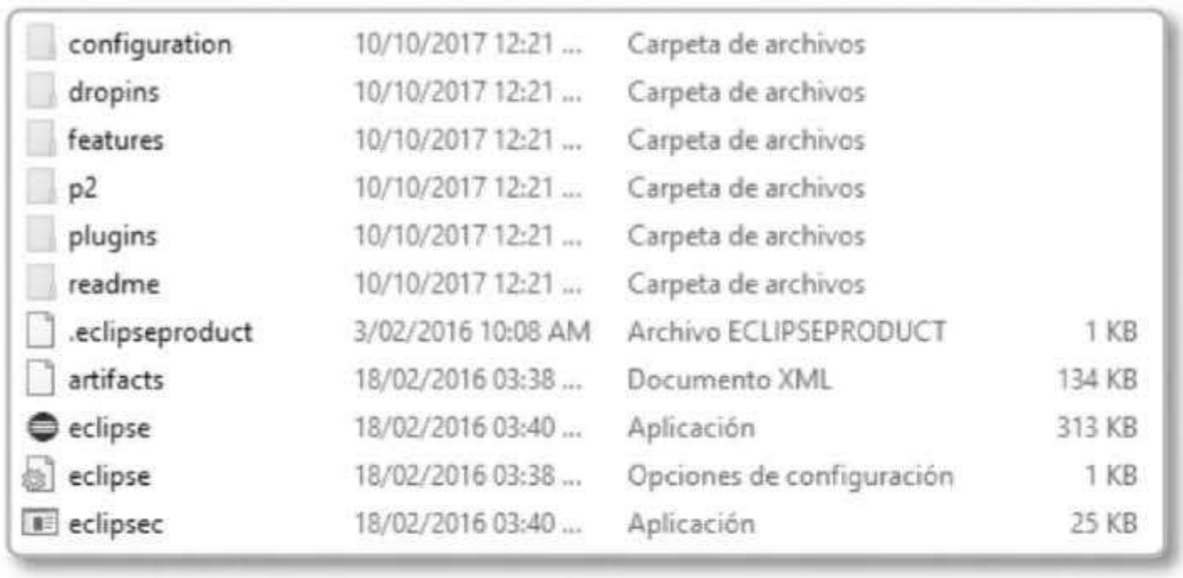
1.12.3 **Descarga de Eclipse (versión Mars 2)**

Figura 1.17 Zona de descarga Eclipse.
Fuente: <http://www.eclipse.org/downloads>

Eclipse es un IDE de Java, fue desarrollado originalmente por la empresa IBM como una evolución de su aplicación VisualAge para Java. Eclipse trabaja bajo licencia de *software* libre; por lo tanto, su distribución es gratuita y es una buena alternativa para el desarrollado de aplicaciones Java.

En la actualidad el IDE de Eclipse se encuentra patrocinado por la Fundación Eclipse, la cual es una organización totalmente independiente, sin ánimo de lucro, que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Esta aplicación no necesita instalación; por tanto, una vez descargada, se debe descomprimir su archivo. Para iniciar Eclipse, solo bastará con dar doble clic sobre el ícono de la aplicación eclipse.



configuration	10/10/2017 12:21 ...	Carpeta de archivos	
dropins	10/10/2017 12:21 ...	Carpeta de archivos	
features	10/10/2017 12:21 ...	Carpeta de archivos	
p2	10/10/2017 12:21 ...	Carpeta de archivos	
plugins	10/10/2017 12:21 ...	Carpeta de archivos	
readme	10/10/2017 12:21 ...	Carpeta de archivos	
.eclipseproduct	3/02/2016 10:08 AM	Archivo ECLIPSEPRODUCT	1 KB
artifacts	18/02/2016 03:38 ...	Documento XML	134 KB
eclipse	18/02/2016 03:40 ...	Aplicación	313 KB
eclipse	18/02/2016 03:38 ...	Opciones de configuración	1 KB
eclipsec	18/02/2016 03:40 ...	Aplicación	25 KB

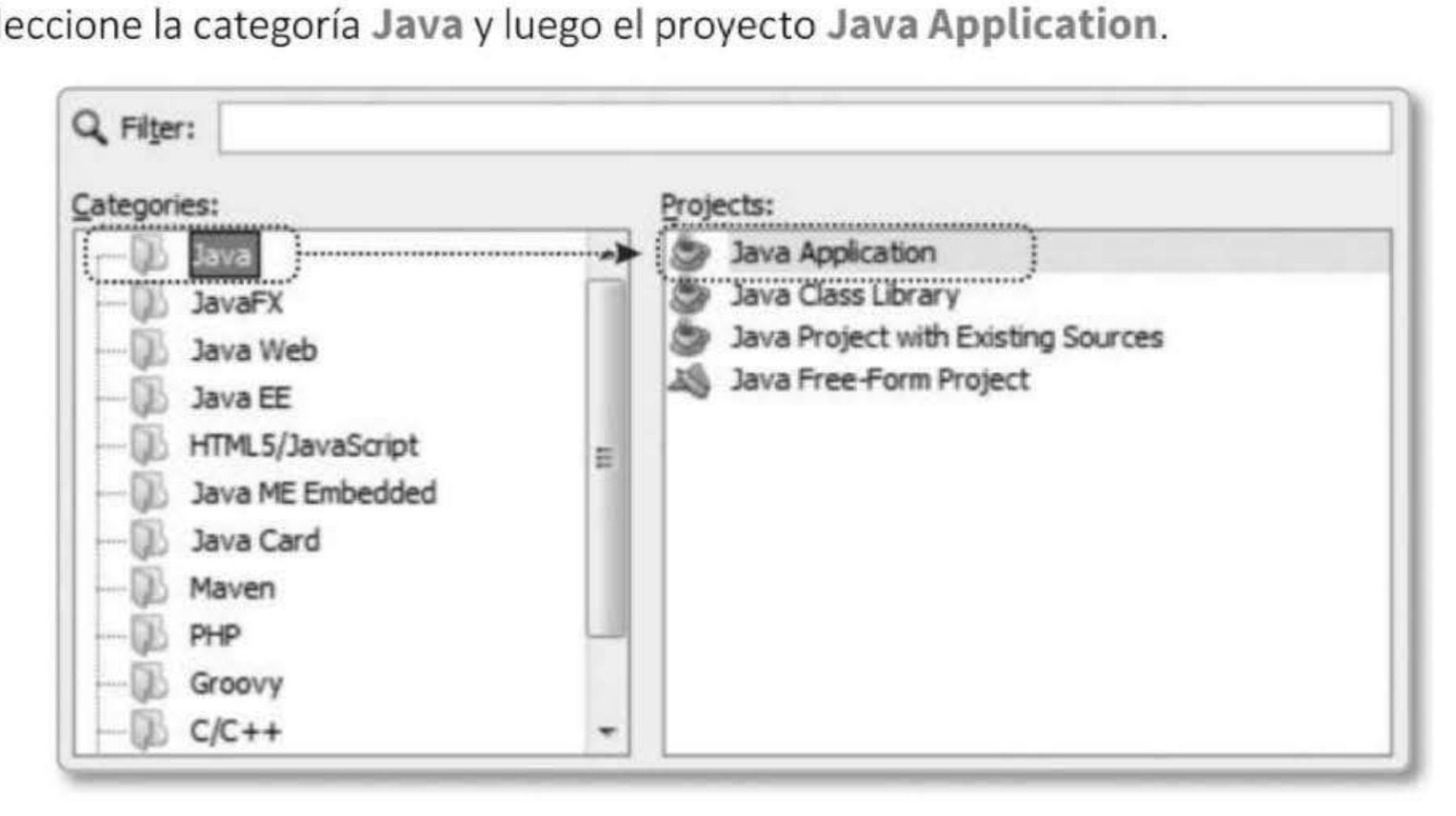
Figura 1.18 Contenido de la carpeta Eclipse.

1.13 El IDE NetBeans y su primera aplicación de plataforma

Se realizará una revisión de la pantalla inicial que presenta el IDE NetBeans y, además, se examinará cómo se crea una nueva aplicación de plataforma Java.

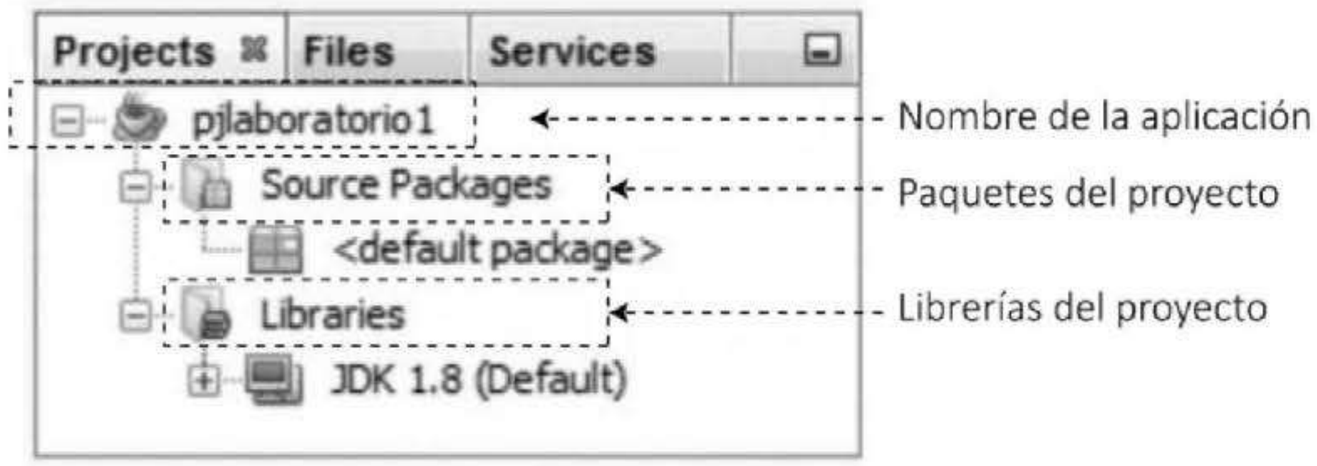
1.13.1 Crear una nueva aplicación

- Elija **File > New Project**.
- Seleccione la categoría **Java** y luego el proyecto **Java Application**.



- Luego asigne un nombre al proyecto, como, por ejemplo, **pjLaboratorio1**. Todas las aplicaciones, por defecto, se almacenan en la carpeta **NetBeansProjects** ubicada en Mis Documentos; si desea modificar la ubicación, seleccione el botón **Browse...** Por otro lado, desactive el *check* **Create Main Class**.

- La ventana de proyectos debe mostrarse inicialmente de la siguiente manera:

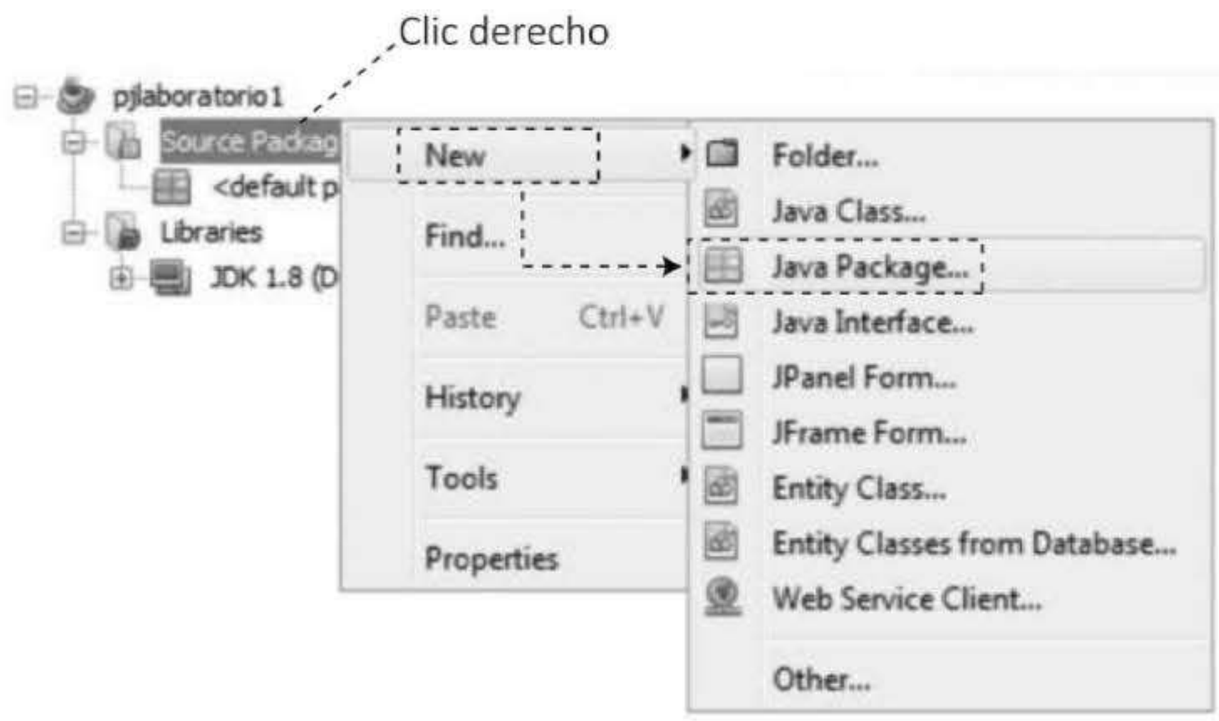


Los paquetes del proyecto permiten mantener un orden dentro de la aplicación, pues se podrían crear paquetes para el conjunto de clases, formularios, imágenes que necesite la aplicación. Las librerías del proyecto muestran una lista de librerías necesarias para el desarrollo de la aplicación.

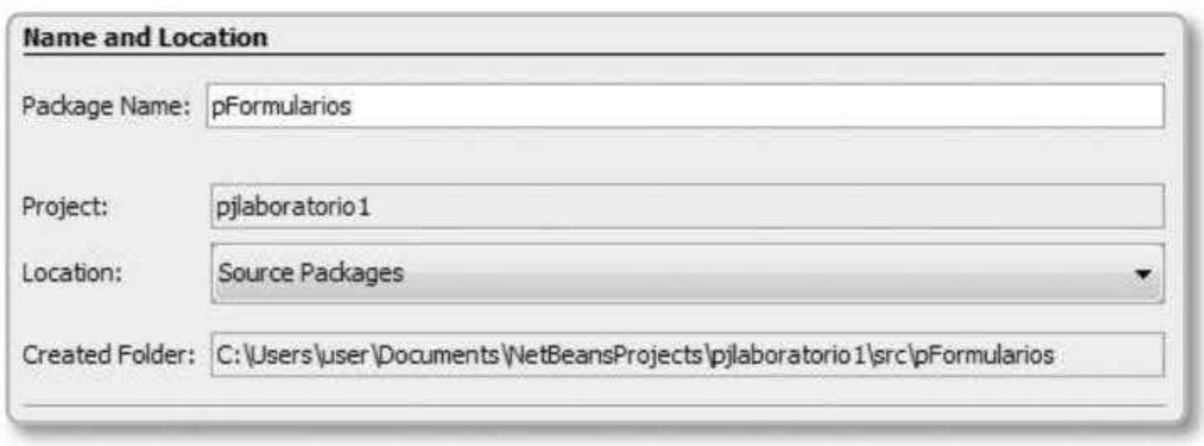
1.13.2 Añadir paquetes al proyecto

Los paquetes generan un determinado orden dentro del proyecto, por eso es importante añadir paquetes a la aplicación. Para ello se realizan los siguientes pasos:

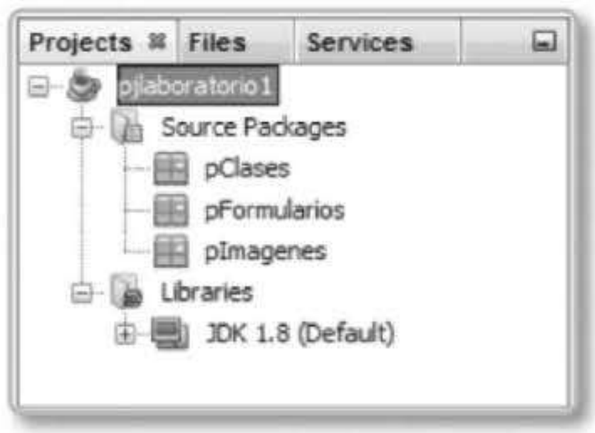
- Haga clic derecho sobre **Source Package**, luego seleccione **New > Java Package**.



- Asigne un nombre al paquete, como, por ejemplo, **pFormularios**, tal como se muestra en la siguiente imagen:



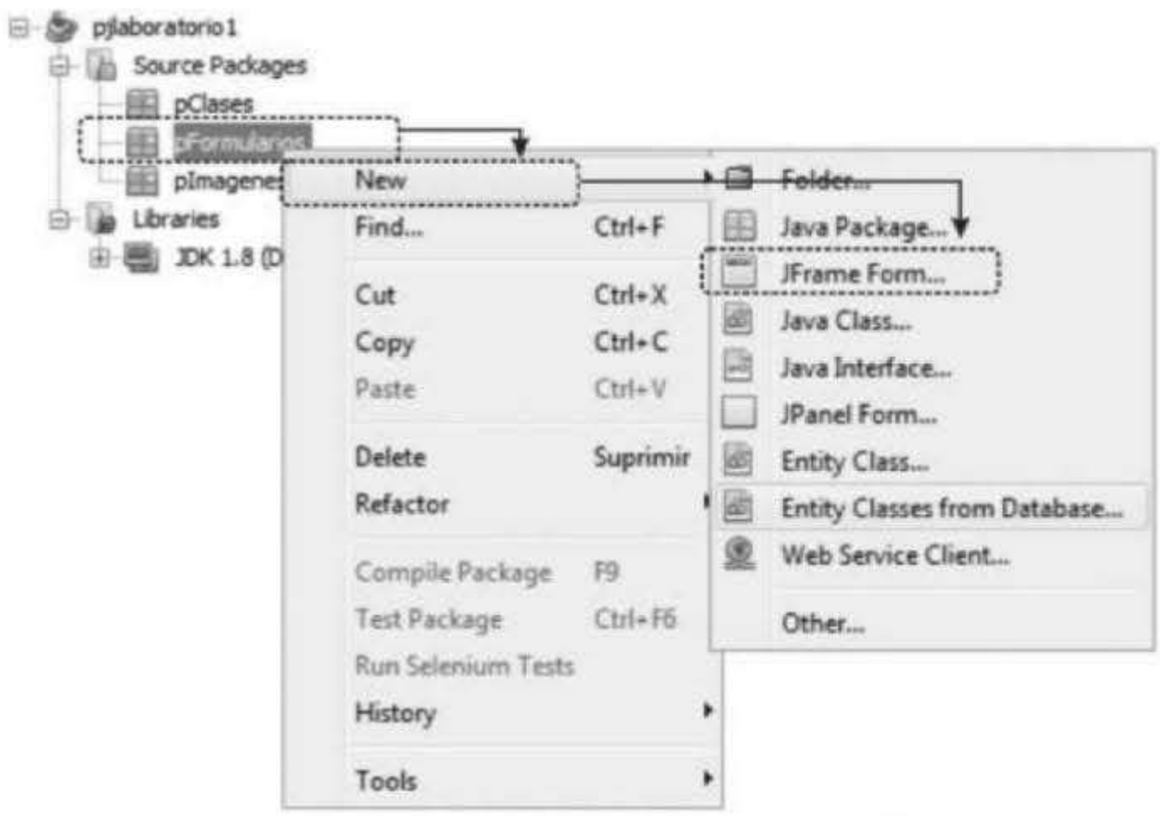
- De la misma forma, puede agregar paquetes como **pClases** y **pImagenes**, véase la siguiente imagen:



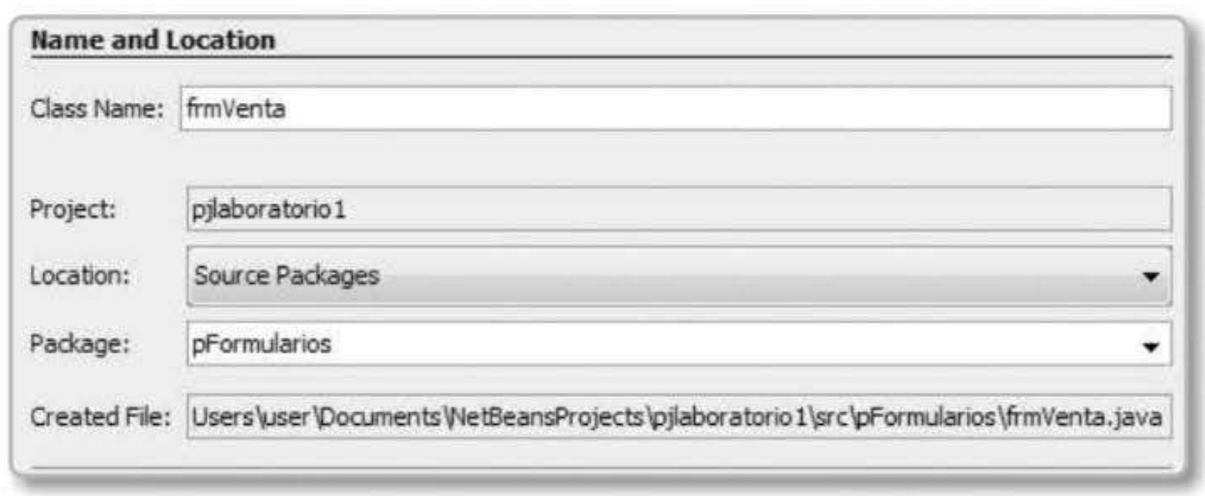
1.13.3 Añadir un formulario al proyecto

Los formularios representan la forma visual de una aplicación en Java. Por lo tanto, es importante añadir un formulario al proyecto; hay que tener en cuenta, no obstante, que deben añadirse en un paquete determinado. Para añadir un formulario, se realizan los siguientes pasos:

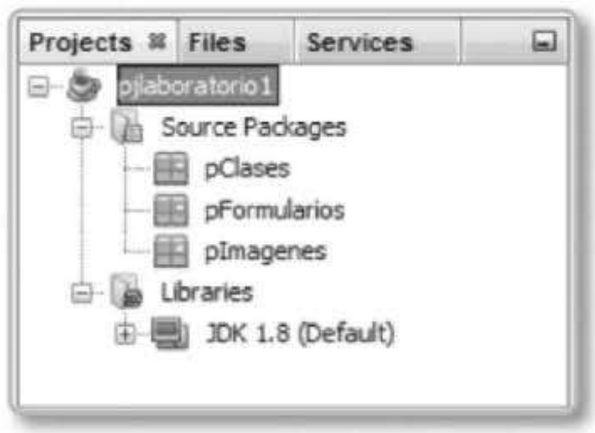
- Haga clic derecho sobre el paquete **pFormularios**, luego seleccione **New > JFrame Form...**



- Asigne un nombre al formulario, como, por ejemplo, **frmVenta**, tal como se observa en la siguiente imagen:



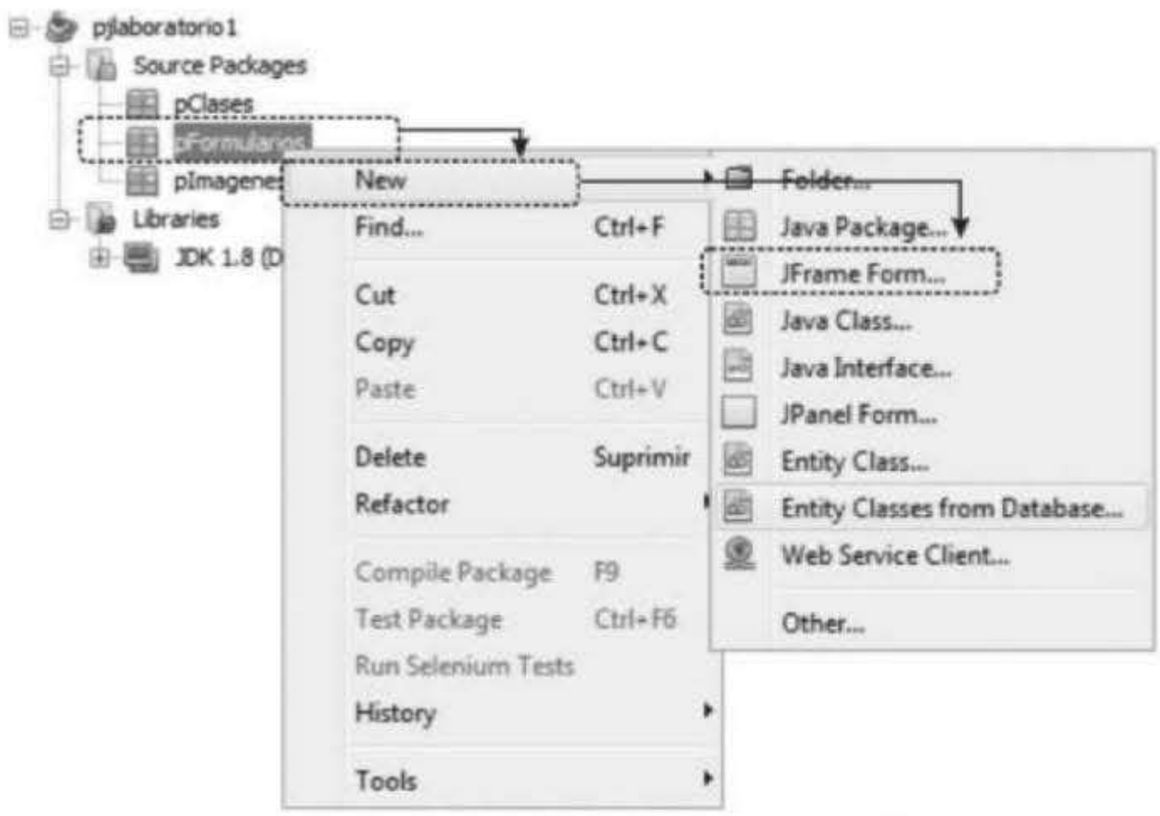
- De la misma forma, puede agregar paquetes como **pClases** y **pImagenes**, véase la siguiente imagen:



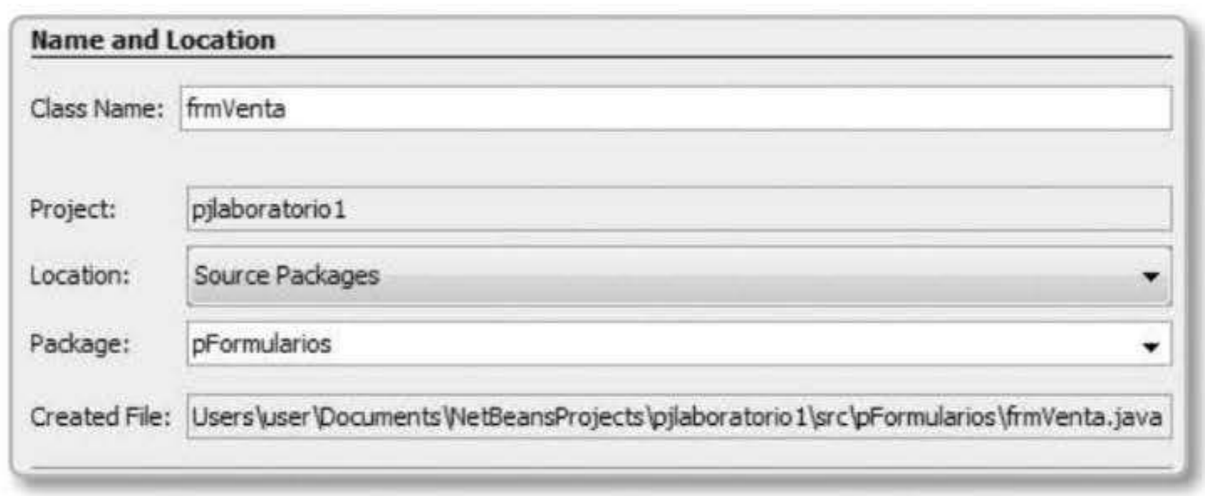
1.13.3 Añadir un formulario al proyecto

Los formularios representan la forma visual de una aplicación en Java. Por lo tanto, es importante añadir un formulario al proyecto; hay que tener en cuenta, no obstante, que deben añadirse en un paquete determinado. Para añadir un formulario, se realizan los siguientes pasos:

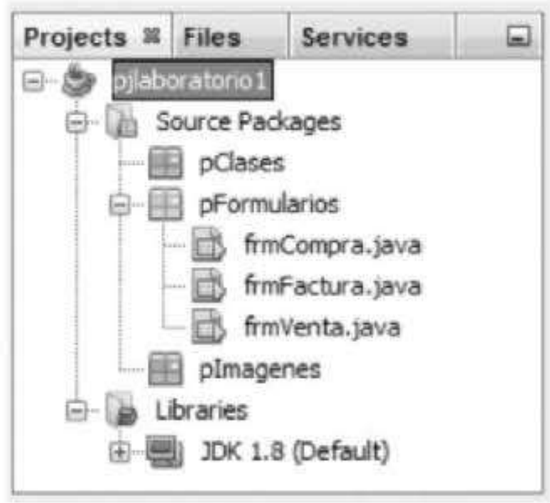
- Haga clic derecho sobre el paquete **pFormularios**, luego seleccione **New > JFrame Form...**



- Asigne un nombre al formulario, como, por ejemplo, **frmVenta**, tal como se observa en la siguiente imagen:



- De la misma manera, podría agregar los formularios **frmCompra** y **frmFactura** al paquete **pFormularios**. La ventana de proyectos debe mostrarse de la siguiente manera:

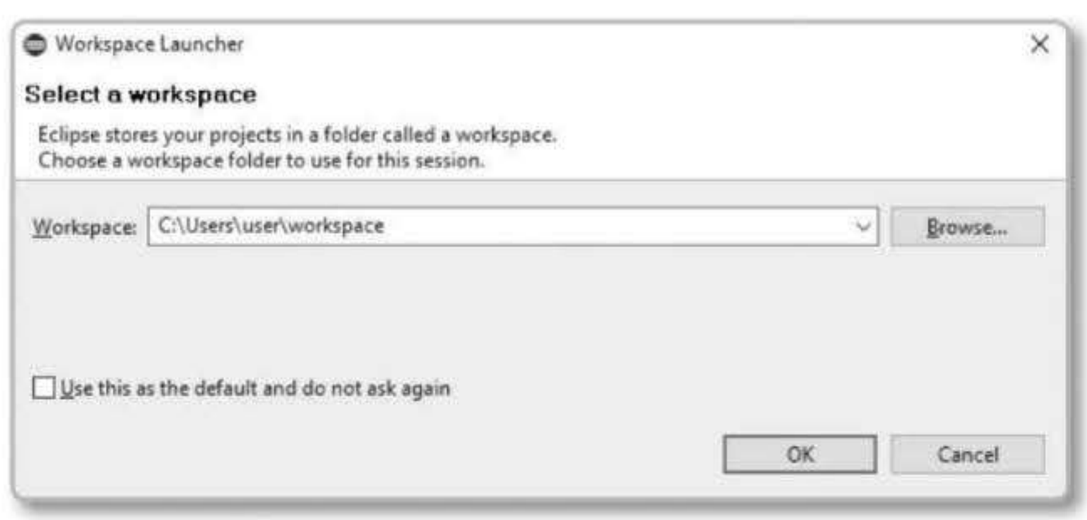


1.14 El IDE Eclipse Mars y su primera aplicación de plataforma

Eclipse Mars es un IDE para el lenguaje de programación Java, incluye un editor de texto para crear nuestro código fuente, realiza marca de errores en el código y autocompleta funciones, clases o variables. La versión Eclipse Mars es 2 y presenta algunas nuevas funcionalidades frente a sus predecesores.

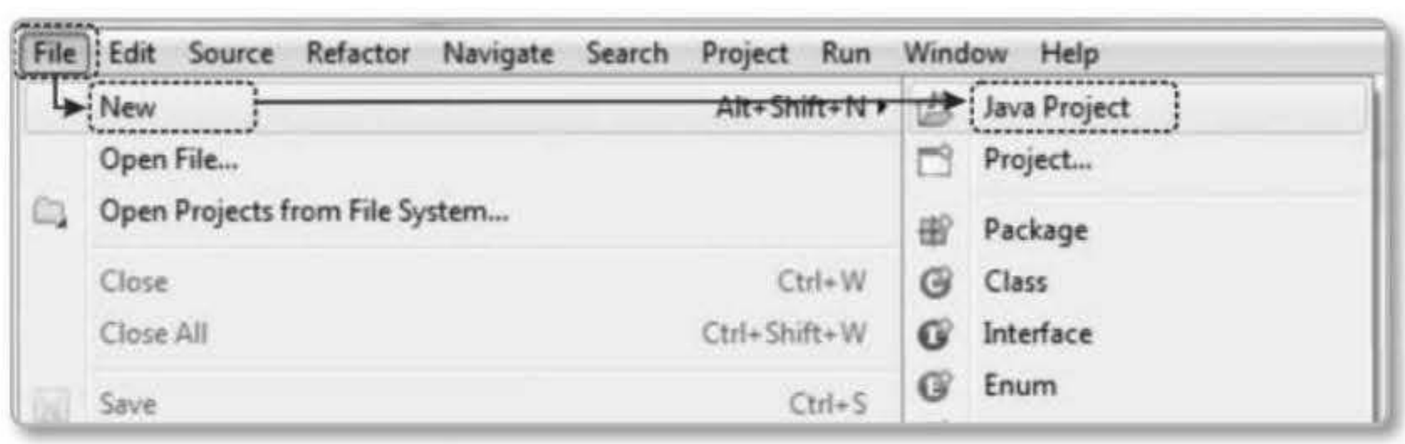
1.14.1 Crear una nueva aplicación

- Al iniciar, Eclipse Mars solicita seleccionar un espacio de trabajo para nuestras aplicaciones:

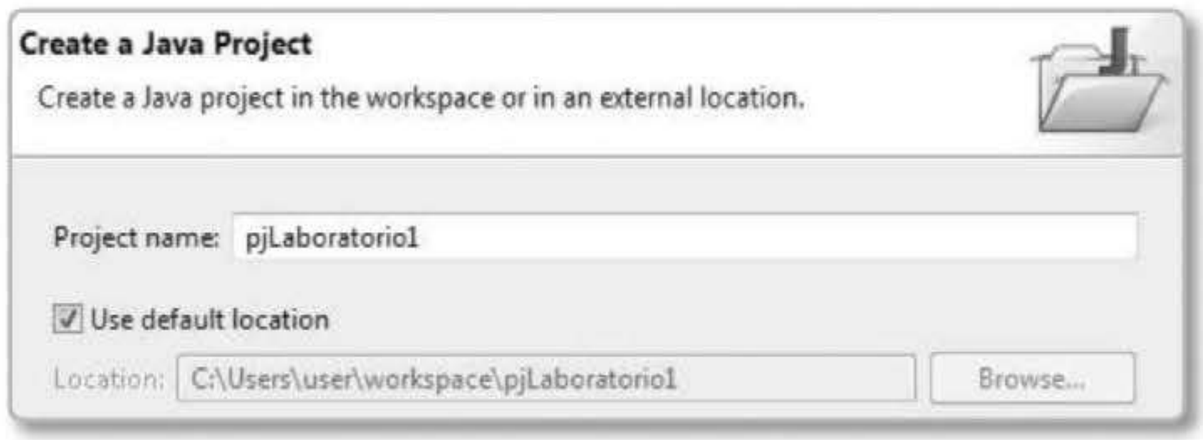


Por defecto, se toma como ubicación la carpeta **WorkSpace**, pero puede modificarla seleccionando el botón **Browse...**

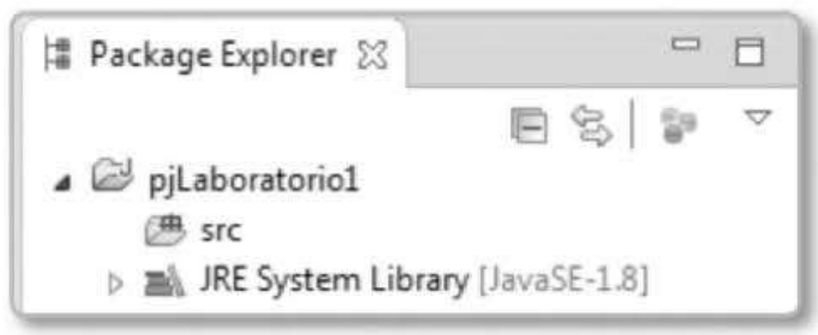
- Luego seleccione **File > New > Java Project**.



- A continuación, asigne un nombre al proyecto, como, por ejemplo, **pjLaboratorio1**; los demás valores mostrados debe mantenerlos por defecto; por lo tanto, solo haga clic en el botón **Finish** de la ventana.



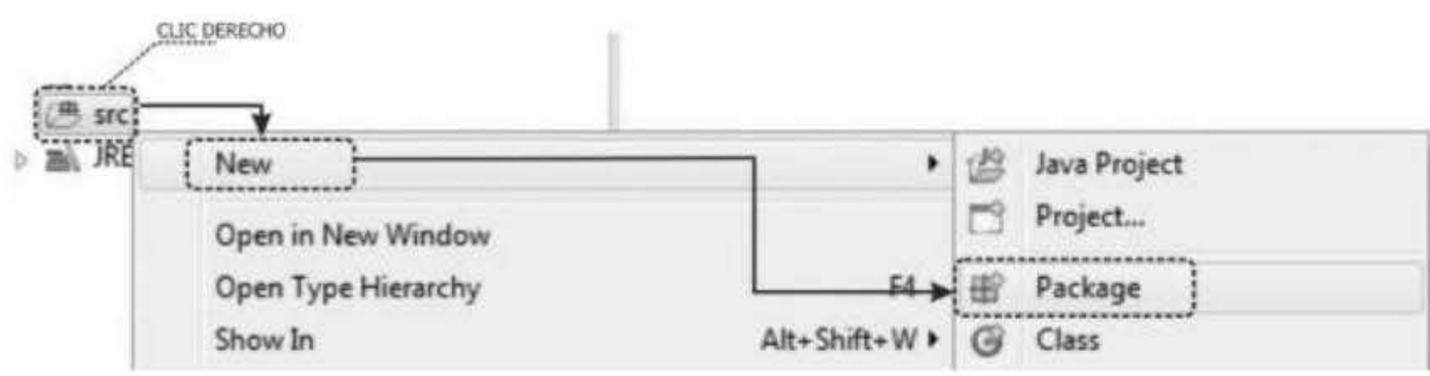
- La ventana exploradora de paquetes inicialmente se muestra de la siguiente manera:



1.14.2 Añadir paquetes al proyecto

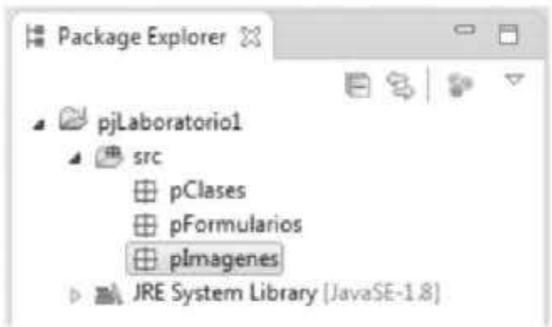
Los paquetes generan un determinado orden dentro del proyecto; por eso, es importante añadir paquetes a la aplicación. Para ello se realizan los siguientes pasos:

- Haga clic derecho sobre el ícono **src**. Luego seleccione **New > Package**.



- Asigne un nombre al paquete, como, por ejemplo, **pFormularios**, tal como se muestra en la siguiente imagen:

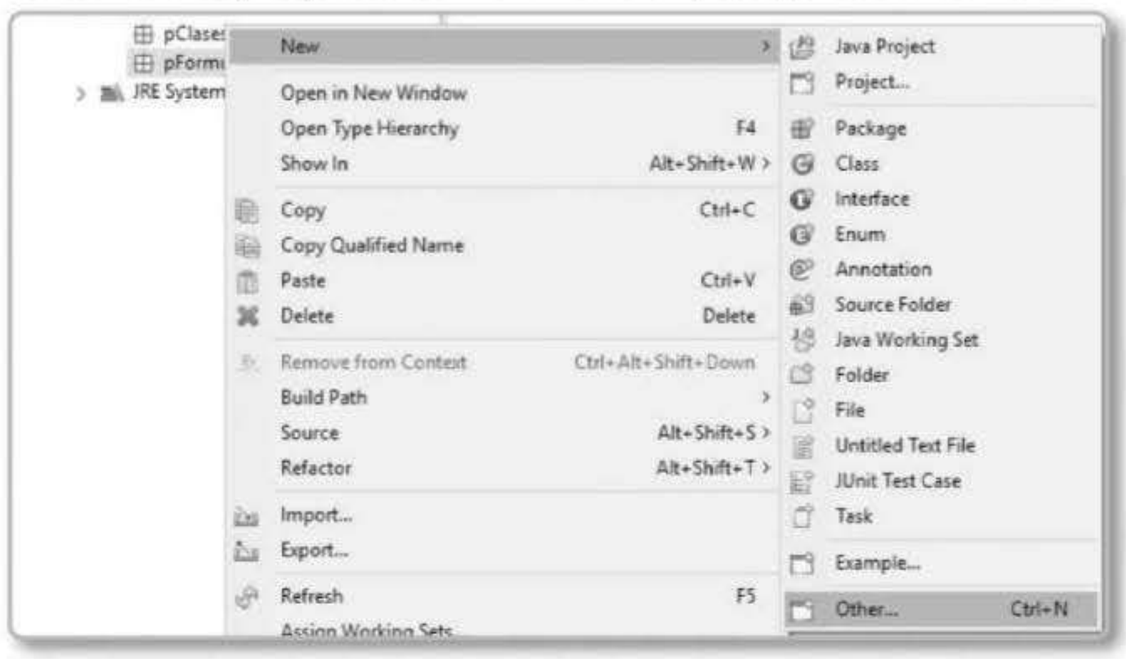
- De la misma forma, se puede agregar paquetes como **pClases** y **pImagenes**, tal como se observa en la siguiente imagen:



1.14.3 Añadir un formulario al proyecto

Para añadir formularios en un proyecto Eclipse Mars, se necesita de la instalación de WindowBuilder. No obstante, la versión Mars descargada ya cuenta con la categoría **Window Builder**. Para su instalación se realizan los siguientes pasos:

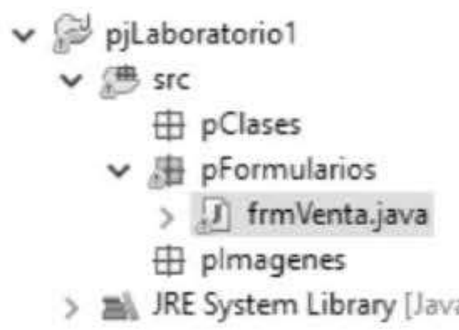
- Haga clic derecho sobre el paquete **pFormularios**, luego seleccione **New > Other...**



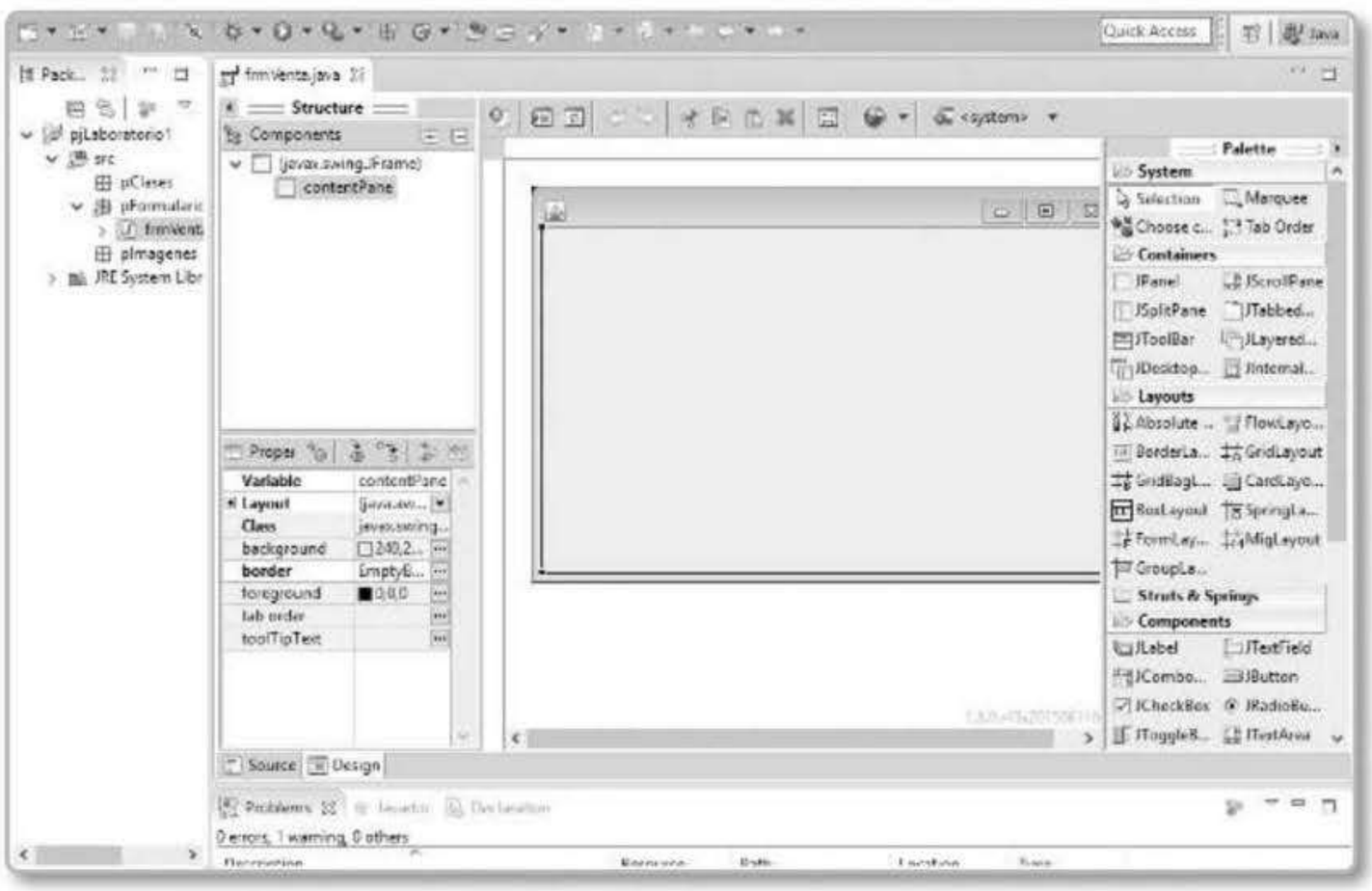
- Seleccione la categoría **WindowBuilder**, aperture la carpeta **Swing Designer** y elija **JFrame**, tal como se muestra en la siguiente imagen:

- Luego asigne un nombre al JFrame, como, por ejemplo, **frmVenta**, y haga clic en el botón **Finish**.

- Finalmente, el aspecto de la ventana exploradora de paquetes debe mostrarse de la siguiente manera:



- Asimismo, el entorno debe mostrarse como sigue:

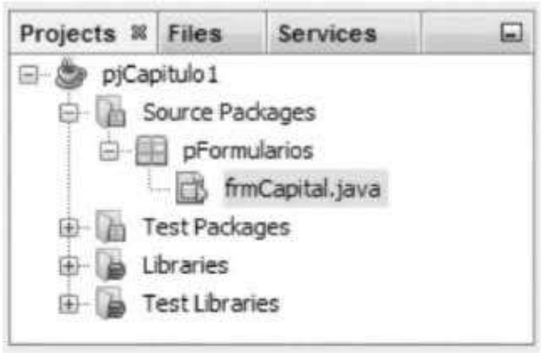


CASO DESARROLLADO 1: **Monto de capital usando NetBeans**

Tres socios de una cooperativa han decidido reunir un capital de dinero basándose en el aporte de cada uno de ellos. Se requiere implementar una aplicación en Java que determine el monto total del capital reunido entre todos los socios y que, además, precise qué tanto por ciento le corresponde a cada uno.

Solución:

1. Seleccione **File > New Project**. Luego elija la categoría **Java** y, enseguida, el proyecto **Java Application**.
2. Asigne el nombre **pjCapitulo1** al proyecto y desactive el *check* **Create Main Class**.
3. Agregue el paquete **pFormularios**.
4. Agregue un **JFrame Form** dentro del paquete **pFormularios** y asígnele el nombre **frmCapital**.



5. Diseñe la siguiente interfaz y asigne nombres a los controles, tal como se muestra en la imagen:

6. Agregue código a los botones; para esto, desde la vista de diseño del NetBeans, haga doble clic sobre los botones.

```
package pFormularios;
public class frmCapital extends javax.swing.JFrame {

    public frmCapital() {
        initComponents();
    }

    private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt) {
        //Recuperando información del formulario
        double monto1=Double.parseDouble(txtMonto1.getText());
        double monto2=Double.parseDouble(txtMonto2.getText());
        double monto3=Double.parseDouble(txtMonto3.getText());

        //Calculando el monto capital
        double capital = monto1+monto2+monto3;

        //Calculando el porcentaje por monto
        double porcentaje1 = (monto1*100) /capital;
        double porcentaje2 = (monto2*100) /capital;
        double porcentaje3 = (monto3*100) /capital;

        //Imprimir los resultados
        lblCapital.setText('$'+String.format('%.2f',capital));
        lblPor1.setText(String.format('%.2f',porcentaje1)+'%');
        lblPor2.setText(String.format('%.2f',porcentaje2)+'%');
        lblPor3.setText(String.format('%.2f',porcentaje3)+'%');
    }

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtMonto1.setText('');
        txtMonto2.setText('');
        txtMonto3.setText('');

        lblCapital.setText('');
        lblPor1.setText('');
        lblPor2.setText('');
        lblPor3.setText('');

        txtMonto1.requestFocus();
    }
}
```

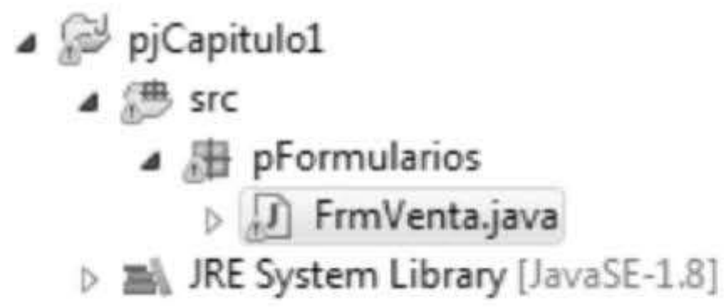
7. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**; ahora pruebe la aplicación con los valores que se muestran en la siguiente imagen:

CASO DESARROLLADO 2: **Venta de productos usando Eclipse Mars 2**

Una tienda de venta de productos al por mayor se encuentra promocionando un nuevo producto, para lo cual hace una oferta de descuento del 12% sobre el monto total de productos adquiridos. Se desea implementar una aplicación en Java que le permita al usuario comprador determinar cuánto debe cancelar por la compra de ciertos productos a un determinado precio; los productos deberán ser ingresados a la aplicación y esta debe devolver el monto subtotal, el descuento y el neto a cancelar en la caja de la tienda.

Solución:

1. Seleccione **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjCapitulo1**.
3. Agregue el paquete **pFormularios**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmVenta**.



5. Diseñe la siguiente interfaz y asigne nombres a los controles, tal como se muestra en la imagen:

6. Agregue código a los botones; para esto, desde la vista de diseño del Eclipse NEON, haga doble clic sobre los botones.

```
package pFormularios;

import java.awt.BorderLayout;
import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JLabel;
import javax.swing.JTextField;
import java.awt.Font;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class frmVenta extends JFrame implements ActionListener {

    private JPanel contentPane;
    private JLabel lblVentaDeProductos;
    private JLabel lblPrecioDelProducto;
    private JLabel lblCantidad;
    private JTextField txtPrecio;
    private JTextField txtCantidad;
    private JButton btnCalcular;
    private JButton btnLimpiar;
    private JLabel lblSubtotal;
    private JLabel lblDescuento;
    private JLabel lblNetoAPagar;
    private JLabel lblSubtotal_1;
    private JLabel lblDescuento_1;
    private JLabel lblNeto;

    /**
     * Launch the application.
     */

    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    frmVenta frame = new frmVenta();
                    frame.setVisible(true);
                }
                catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

public frmVenta() {
        setTitle('Venta de Productos - Eclipse Mars 2');
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setBounds(100, 100, 504, 244);
        contentPane = new JPanel();
        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
        setContentPane(contentPane);
        contentPane.setLayout(null);
```

```
        lblVentaDeProductos = new JLabel("VENTA DE PRODUCTOS - PROMOCIÓN");
        lblVentaDeProductos.setFont(new Font("Tahoma", Font.BOLD, 14));
        lblVentaDeProductos.setBounds(28, 22, 324, 25);
        contentPane.add(lblVentaDeProductos);

        lblPrecioDelProducto = new JLabel("PRECIO DEL PRODUCTO");
        lblPrecioDelProducto.setBounds(30, 58, 136, 14);
        contentPane.add(lblPrecioDelProducto);

        lblCantidad = new JLabel("CANTIDAD");
        lblCantidad.setBounds(180, 58, 79, 14);
        contentPane.add(lblCantidad);

        txtPrecio = new JTextField();
        txtPrecio.setBounds(28, 83, 138, 25);
        contentPane.add(txtPrecio);
        txtPrecio.setColumns(10);

        txtCantidad = new JTextField();
        txtCantidad.setBounds(180, 83, 79, 25);
        contentPane.add(txtCantidad);
        txtCantidad.setColumns(10);

        btnCalcular = new JButton("CALCULAR");
        btnCalcular.addActionListener(this);
        btnCalcular.setBounds(269, 83, 112, 24);
        contentPane.add(btnCalcular);

        btnLimpiar = new JButton("LIMPIAR");
        btnLimpiar.addActionListener(this);
        btnLimpiar.setBounds(385, 83, 89, 24);
        contentPane.add(btnLimpiar);

        lblSubtotal = new JLabel("SUBTOTAL");
        lblSubtotal.setBounds(28, 142, 79, 14);
        contentPane.add(lblSubtotal);

        lblDescuento = new JLabel("DESCUENTO");
        lblDescuento.setBounds(28, 170, 89, 14);
        contentPane.add(lblDescuento);

        lblNetoAPagar = new JLabel("NETO A PAGAR");
        lblNetoAPagar.setBounds(317, 142, 89, 14);
        contentPane.add(lblNetoAPagar);

        lblSubtotal_1 = new JLabel("$426.00");
        lblSubtotal_1.setBounds(123, 142, 79, 14);
        contentPane.add(lblSubtotal_1);

        lblDescuento_1 = new JLabel("$51.12");
        lblDescuento_1.setBounds(123, 170, 79, 14);
        contentPane.add(lblDescuento_1);

        lblNeto = new JLabel("$374.88");
        lblNeto.setFont(new Font("Tahoma", Font.PLAIN, 18));
        lblNeto.setBounds(327, 167, 89, 14);
        contentPane.add(lblNeto);
    }
```

```
        public void actionPerformed(ActionEvent arg0) {
            if (arg0.getSource() -- btnLimpiar) {
                actionPerformedBtnLimpiar(arg0);
            }
            if (arg0.getSource() -- btnCalcular) {
                actionPerformedBtnCalcular(arg0);
            }
        }

        protected void actionPerformedBtnCalcular(ActionEvent arg0) {
            //Capturando los valores del formulario
            double precio - Double.parseDouble(txtPrecio.getText());
            int cantidad - Integer.parseInt(txtCantidad.getText());

            //Calculando el subtotal
            double subtotal - precio * cantidad;

            //Calculando el descuento
            double descuento - subtotal * 0.12;

            //Calculando el neto a pagar
            double neto - subtotal - descuento;

    //Imprimir los resultados
            lblSubtotal.setText('$'+String.format('%.2f',subtotal));
            lblDescuento.setText('$'+String.format('%.2f',descuento));
            lblNeto.setText('$'+String.format('%.2f',neto));
        }
        protected void actionPerformedBtnLimpiar(ActionEvent arg0) {
            txtCantidad.setText('');
            txtPrecio.setText('');

            lblSubtotal.setText('');
            lblDescuento.setText('');
            lblNeto.setText('');

            txtPrecio.requestFocus();
        }
    }
```

7. Ejecute la aplicación. Una vez colocado el código en los botones, haga clic derecho sobre el **formulario > Run As > Java Application**; ahora pruebe la aplicación con los valores que se muestran en la siguiente imagen:

Fundamentos de programación

CAPÍTULO

2

Competencias

- Reconoce los conceptos básicos de programación estructurada.
- Identifica los elementos que componen una aplicación Java.

Contenido

2.1 Generalidades

2.2 Proceso de solución mediante una aplicación Java

2.3 Asociar el pseudocódigo al formulario en Java

2.4 Metodología de programación con Java

2.5 Elementos que componen una aplicación Java

2.6 Los comentarios

2.7 Elementos léxicos en Java

2.8 Los paquetes y las importaciones

2.9 Tipos de datos

2.10 Las variables

2.11 La clase **Integer**

2.12 La clase **String**

2.13 Manejo de la clase **JOptionPane**

2.14 La clase **Math**

- Caso desarrollado 1: Modelo de cotización bancaria (programación estructurada)
- Caso desarrollado 2: Modelo de cotización bancaria (programación modular)
- Caso desarrollado 3: Modelo de cotización bancaria (programación orientada a objetos)

2.1 Generalidades

Actualmente vivimos en una sociedad informática gracias a muchos factores, uno de los más importantes es la evolución de la Internet. Ello ha permitido que muchas personas usen aplicaciones o programas como Facebook, WhatsApp, etc., que han sido desarrollados justamente bajo un lenguaje de programación.

Java es un lenguaje de programación que nos permitirá implementar aplicaciones que den solución a un determinado problema o necesidad. Así, en la actualidad, se cuenta con miles de aplicaciones para todo tipo de usuario, desde un reloj digital en un dispositivo móvil hasta aplicaciones que permiten controlar el inventario de activos en una empresa.

Este capítulo se enfocará en reconocer los elementos que componen una aplicación Java. Se iniciará con una breve explicación sobre algoritmos y, al final, se implementarán aplicaciones usando como IDE a NetBeans.

2.2 Proceso de solución mediante una aplicación Java

La solución a un determinado problema se puede dar de diferentes formas y en diferentes escenarios; cuando el problema es conocido, se puede solucionar directamente en el lenguaje de programación, pero siempre se recomienda realizar un análisis del mismo.

Asimismo, en Java sucede algo particular: contiene muchas reglas que es importante respetar. Estas reglas, como, por ejemplo, la sensibilidad de mayúsculas y minúsculas que tiene Java o el uso de métodos que contienen sus librerías, se irán conociendo en el transcurso de este material.

Entonces, a modo de resumen, para solucionar un determinado problema se deben seguir las fases que se proponen en la tabla 2.1.

Tabla 2.1 Fases para solucionar un problema.

Actividades	Fases
Analizar el problema de tal forma que se entienda cuál es el objetivo del mismo.	Análisis del problema
Diseñar el algoritmo de solución al problema, de tal manera que satisfaga la necesidad del problema.	Diseño del algoritmo
Codificar el problema en un determinado editor de código Java.	Codificación del problema
Ejecutar el programa de tal manera que refleje el resultado esperado.	Ejecución del programa
Verificar si el resultado es el esperado y modificar si fuera necesario.	Verificación y depuración

2.2.1 Fase 1: Análisis del problema

Esta fase tiene por objetivo reconocer claramente a dónde se quiere llegar con el problema, además de identificar los elementos necesarios para la solución. Para ello se requiere de tres capacidades que hay que reconocer y que en el futuro será muy sencillo detectar. Así, una vez entendido el problema, se deben responder las preguntas de la tabla 2.2.

Tabla 2.2 Proceso de entrada y salida.

Preguntas	Valores de solución
¿Qué datos se necesitan para la solución del problema?	Aquí se deben listar los datos que se necesitan para la solución del problema; por lo menos debe haber uno y para esta etapa debe ser un valor atómico, es decir, no debe contener más de un valor. Comúnmente se le llama ENTRADA.
¿Cómo se usan dichos datos o cuál será la fórmula que se empleará en la solución?	Aquí se pueden emplear los conocimientos previos que se tienen acerca de los cálculos matemáticos, estadísticos o financieros, los cuales serán necesarios como parte de la solución al problema, ya que se usarán fórmulas o funciones para buscar una solución adecuada al problema. Comúnmente se le llama PROCESO.
¿Cuál es el resultado?	Aquí se debe determinar cuál será el resultado emitido por las fórmulas haciendo uso de los datos, es decir, se debe lograr una integración entre la primera y segunda pregunta. Comúnmente se le llama SALIDA.

A continuación, se presentan algunos casos para entender mejor cómo se debe analizar un problema antes de codificarlo en Java.

> **Problema 1.** Por enviar un paquete de 5 kg de peso a una población que está a 60 km de distancia, una empresa de transporte cobró $ 9.00. ¿Cuánto costará enviar un paquete de 15 kg a 200 km de distancia?

- Analizando el problema, se encuentran los siguientes datos:

Peso del paquete (kg)	Distancia de la población (km)	Costo de transporte ($)
5	60	9.00
15	200	¿?

- Seguidamente, se organizan los valores encontrados de tal forma que satisfaga el patrón de valores entrada-proceso-salida.

ENTRADA	Peso del paquete (peso), distancia de la población (distancia)

PROCESO	$\text{costo} = 9 \times \frac{15}{5} \times \frac{200}{60}$
	El costo del transporte es directamente proporcional al peso del paquete; es decir, si por 5 kg cobran $ 9.00, se puede decir que por 15 kg el costo de transporte debe ser mayor; finalmente, el costo aumenta en proporción al peso del paquete.
	De la misma manera se debe comparar el costo con la distancia; es decir, si por 60 km cobran $ 9.00, por 200 km cobrarán más; por lo tanto, el costo también es directamente proporcional a la distancia, ya que a mayor distancia mayor será el costo de transporte.
	Finalmente, se puede concluir con la siguiente fórmula de solución al problema:
	$\text{costo} = 9 \times \frac{\text{peso}}{5} \times \frac{\text{distancia}}{60}$
	El código Java se muestra de la siguiente manera:
	`double costo;`
	`costo = 9 * (peso * 5) * (distancia/60);`
	Nótese en la fórmula que para poder hallar el costo se necesita saber cuál es el peso y la distancia; por lo tanto, es válido decir que **peso** y **distancia** son datos de entrada, mientras que **costo** representa el valor de salida.

SALIDA	Costo de transporte (costo)

Problema 2. Doce obreros, trabajando 8 horas diarias, terminan un trabajo en 25 días. ¿Cuánto tardarán en hacer ese mismo trabajo 5 obreros trabajando 10 horas diarias?

- Analizando el problema, se encuentran los siguientes datos:

Cantidad de obreros	Horas de trabajo	Días trabajados
12	8	25
5	10	¿?

- Seguidamente, se organizan los valores encontrados de tal forma que satisfaga el patrón de valores entrada-proceso-salida.

ENTRADA	Cantidad de obreros (obreros), horas de trabajo (horas)

PROCESO	$\text{Días} = 25 \times \frac{12}{5} \times \frac{8}{10}$
	
	Los días trabajados son inversamente proporcionales a la cantidad de obreros; es decir, si por 25 días se necesitan 12 obreros, se puede decir que por 5 obreros se necesitará más días; finalmente, los días trabajados aumentarán si la cantidad de obreros disminuye.
	
	De la misma manera se deben comparar las horas diarias de trabajo con los días trabajados; es decir, si trabajando 8 horas diarias se demoran 25 días, entonces se puede decir que por 10 horas diarias de trabajo se demorarán menos días; por lo tanto, las horas de trabajo también son inversamente proporcionales a los días trabajados.
	
	Finalmente, se puede concluir con la siguiente fórmula de solución al problema:
	
	$\text{Días} = 25 \times \frac{12}{\text{obreros}} \times \frac{8}{\text{horas}}$
	
	El código Java se muestra de la siguiente manera:
	
	`int dias;`
	`dias-25*(12/obreros) * (8/horas);`
	
	Nótese en la fórmula que para poder hallar los días se necesita saber la cantidad de obreros y las horas; por lo tanto, es válido decir que **obreros** y **horas** son datos de entrada, mientras que **días** representa el valor de salida.

SALIDA	Días trabajados (días)

2.2.2 Fase 2: Diseño del algoritmo

El diseño del algoritmo puede ser representado mediante dos herramientas: los diagramas de flujo y el pseudocódigo. Aquí se hará uso de los pseudocódigos, ya que son los más parecidos al código fuente que necesitamos implementar en Java.

Se dice que un pseudocódigo representa el problema en pasos cortos de forma descendente, y que tiene un inicio y un fin, de tal manera que se pueden realizar las pruebas del problema con los datos obtenidos en la FASE 1.

A continuación, se seleccionarán los dos casos expuestos en la FASE 1 para representarlos en la fase del diseño del algoritmo.

Problema 1. Para poder enviar un paquete de 5 kg de peso a una población que está a 60 km de distancia, una empresa de transporte cobró $ 9.00. ¿Cuánto costará enviar un paquete de 15 kg a 200 km de distancia?

- Patrón de entrada, proceso y salida del problema:

ENTRADA	peso, distancia

PROCESO	Pseudocódigo:
	$\text{costo} = 9 \times \frac{\text{peso}}{5} \times \frac{\text{distancia}}{60}$
	Java:
	`double costo;`
	`costo-9*(peso*5) * (distancia/60);`

SALIDA	Costo

- Pseudocódigo de solución:

Líneas de pseudocódigo	Descripción
INICIO	Punto de inicio del pseudocódigo.
Leer peso, distancia	Solicita el valor de peso y la distancia.
Costo = 9 × (peso / 5) × (distancia / 60)	Calcula el costo según lo analizado.
Imprimir costo	Muestra el resultado del cálculo.
FIN	Punto final del pseudocódigo.

Problema 2. Doce obreros, trabajando 8 horas diarias, terminan un trabajo en 25 días. ¿Cuánto tardarán en hacer ese mismo trabajo 5 obreros laborando 10 horas diarias?

- Patrón de entrada, proceso y salida del problema:

ENTRADA	obreros, horas

PROCESO	Pseudocódigo:
	$\text{días} = 25 \times \frac{12}{\text{obreros}} \times \frac{8}{\text{horas}}$
	Java:
	`int dias;`
	`dias - 25 * (12/obreros) * (8/horas);`

SALIDA	Días

- Pseudocódigo de solución:

Líneas de pseudocódigo	Descripción
INICIO	Punto de inicio del pseudocódigo.
Leer obreros, horas	Solicita el valor de obreros y horas.
días = 25 × (12/obreros) × (8/horas)	Calcula la cantidad de días según la fórmula.
Imprimir días	Muestra el resultado del cálculo.
FIN	Punto final del pseudocódigo.

2.2.3 **Fase 3: Codificación del problema**

Para completar la fase de codificación se deben sustituir los comandos del pseudocódigo por las sentencias propias de Java, claro está, respetando las reglas y los patrones propios del lenguaje. Esto se irá aprendiendo conforme se avance en la lectura del libro.

A continuación, se toman como referencia los casos expuestos en la FASE 1.

> **Problema 1.** Para poder enviar un paquete de 5 kg de peso a una población que está a 60 km de distancia, una empresa de transporte cobra $ 9.00. ¿Cuánto costará enviar un paquete de 15 kg a 200 km de distancia?

Pseudocódigo de solución:

Líneas de pseudocódigo
INICIO
Leer peso, distancia
Costo = 9 × (peso / 5) × (distancia / 60)
Imprimir costo
FIN

Paso 1. Diseñe la siguiente interfaz:

Paso 2. Haga doble clic en el botón **Calcular** y coloque el siguiente código:

```
JButton btnCalcular = new JButton('CALCULAR');
btnCalcular.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        //Entrada
        double peso,distancia;
        peso = Double.parseDouble(txtPeso.getText());
        distancia = Double.parseDouble(txtDistancia.getText());

        //Proceso
        double costo;
        costo = 9 * (peso/5.0) * (distancia/60.0);

        //Salida
        lblCosto.setText('$'+String.format('%.2f', costo));
    }
});
```

Seguidamente se detalla qué función cumple cada línea de la función:

- **double peso,distancia;**

  Se declaran las variables **peso** y **distancia** del tipo **doublé**, puesto que recibirán valores decimales.

- **peso = Double.parseDouble(txtPeso.getText());**

  Se envía a la variable **peso** el valor ingresado en el control **txtPeso**; no hay que olvidar que dicho valor es considerado como cadena y que este debe ser convertido al mismo tipo de la variable, para ello se usa la sentencia **Double.parseDouble**.

- **distancia = Double.parseDouble(txtDistancia.getText());**

  Se envía a la variable **distancia** el valor ingresado en el control **txtDistancia**; no hay que olvidar que dicho valor es considerado como cadena y que este debe ser convertido al mismo tipo de la variable, para ello se usa la sentencia **Double.parseDouble**.

- **double costo;**

  Se declara la variable **costo** del tipo **doublé**; esta será la encargada de recibir el valor resultante del proceso.

- **costo = 9 * (peso/5.0) * (distancia/60.0);**

  La variable **costo** almacenará el resultado correspondiente a la fórmula que soluciona el problema.

- **lblCosto.setText("$"+String.format("%.2f", costo));**

  Se envía el valor almacenado en la variable **costo**, para ello se usa el método **setText** del control **lblCosto**; asimismo, para imprimir a dos decimales se usa el método **format** de la clase **String**, donde 2 representa a la cantidad de decimales que se desea imprimir.

> **Problema 2.** Doce obreros, trabajando 8 horas diarias, terminan un trabajo en 25 días. ¿Cuánto tardarán en hacer ese mismo trabajo 5 obreros laborando 10 horas diarias?

Pseudocódigo de solución:

Líneas de pseudocódigo
INICIO
Leer obreros, horas
días = 25 × (12 / obreros) × (8 / horas)
Imprimir días
FIN

Paso 1. Diseñe la siguiente interfaz:

Paso 2. Haga doble clic en el botón **Calcular** y coloque el siguiente código:

```
protected void actionPerformedBtnCalcular(ActionEvent arg0) {
    //Entrada
    double obreros,horas;
    obreros = Double.parseDouble(txtObreros.getText());
    horas = Double.parseDouble(txtHoras.getText());

    //Proceso
    double dias;
    dias = 25 * (12.0/obreros) * (8.0/horas);

    //Salida
    lblDias.setText(""+String.format("%.2f", dias));
}
```

Seguidamente se detalla qué función cumple cada línea de la función:

- **double obreros,horas;**

  Se declaran las variables **obreros** y **horas** del tipo **doublé**, ya que recibirán valores decimales.

- **obreros = Double.*parseDouble*(txtObreros.getText());**

  Se envía a la variable **obreros** el valor ingresado en el control **txtObreros**; no hay que olvidar que dicho valor es considerado como cadena y que este debe ser convertido al mismo tipo de la variable, para ello se usa la sentencia **Double.parseDouble**.

- **horas = Double.*parseDouble*(txtHoras.getText());**

  Se envía a la variable **horas** el valor ingresado en el control **txtHoras**; no hay que olvidar que dicho valor es considerado como cadena y que este debe ser convertido al mismo tipo de la variable, para ello se usa la sentencia **Double.parseDouble**.

- **double dias;**

  Se declara la variable **dias** del tipo **doublé**; esta será la encargada de recibir el valor resultante del proceso.

- **dias = 25 * (12.0/obreros) * (8.0/horas);**

  La variable **dias** almacenará el resultado correspondiente a la fórmula que soluciona el problema.

- **lblDias.setText(""+String.*format*("%.2f", dias));**

  Se envía el valor almacenado en la variable **dias**, para ello se usa el método **setText** del control **lblDias**; asimismo, para imprimir a dos decimales se usa el método **format** de la clase **String**, donde 2 representa a la cantidad de decimales que se desea imprimir.

2.2.4 Fase 4: Ejecución del programa

IDE	Forma de ejecución
NetBeans	**<Shift>** + **<F6>**
Eclipse Mars	Haga clic derecho sobre el formulario, luego seleccione **Run As** > **Java Application**.

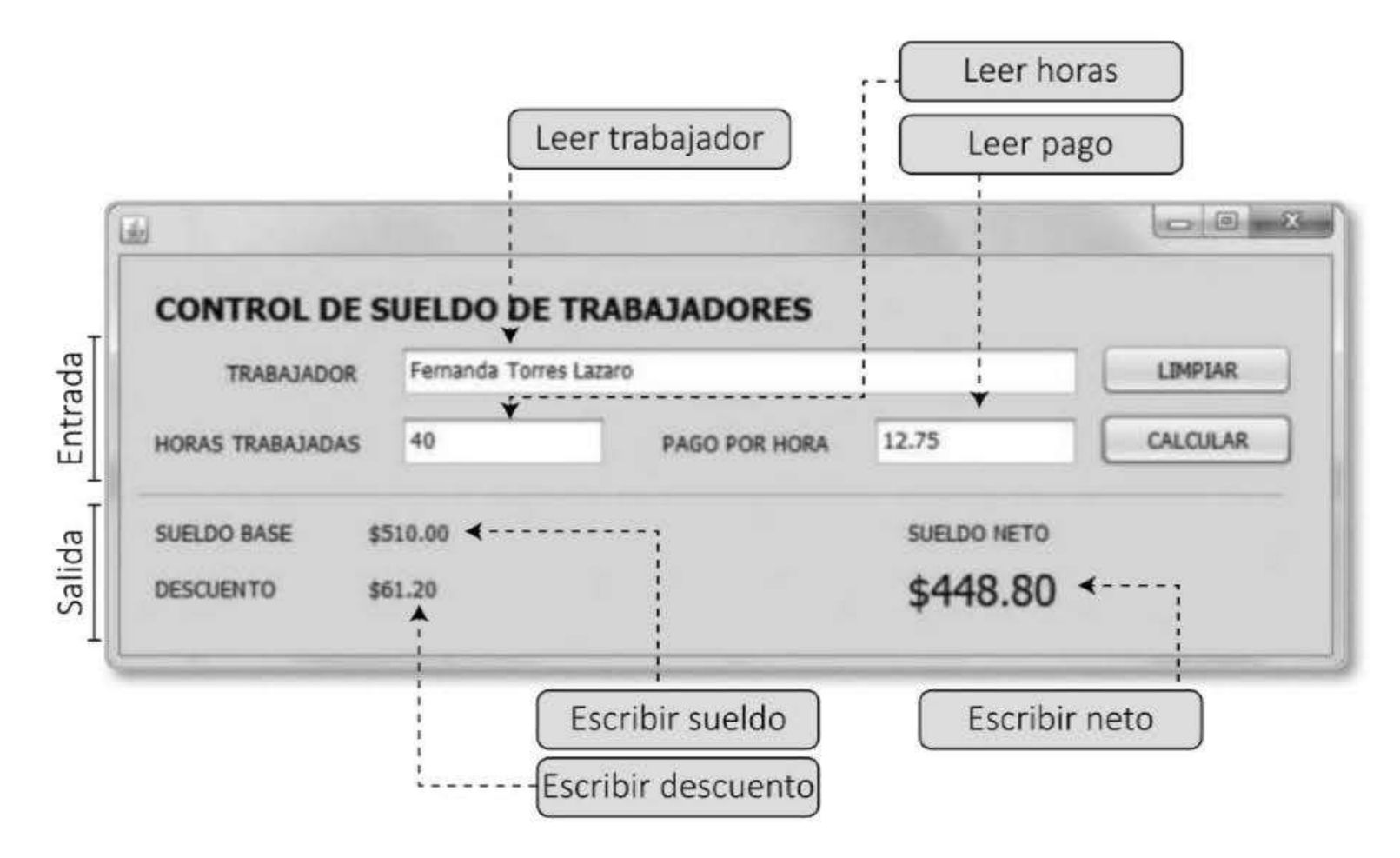
2.3 Asociar el pseudocódigo al formulario en Java

Para terminar de entender el proceso de implementación se realizará una analogía entre el pseudocódigo y la aplicación Java.

A continuación, se analiza el siguiente pseudocódigo del proceso que calcula el sueldo de los empleados.

Líneas de pseudocódigo
INICIO
Leer trabajador, horas, pago
Sueldo = horas x pago Descuento = sueldo * 12% Neto = sueldo – descuento
Imprimir sueldo, descuento, neto
FIN

El diseño de la GUI se muestra de la siguiente manera:

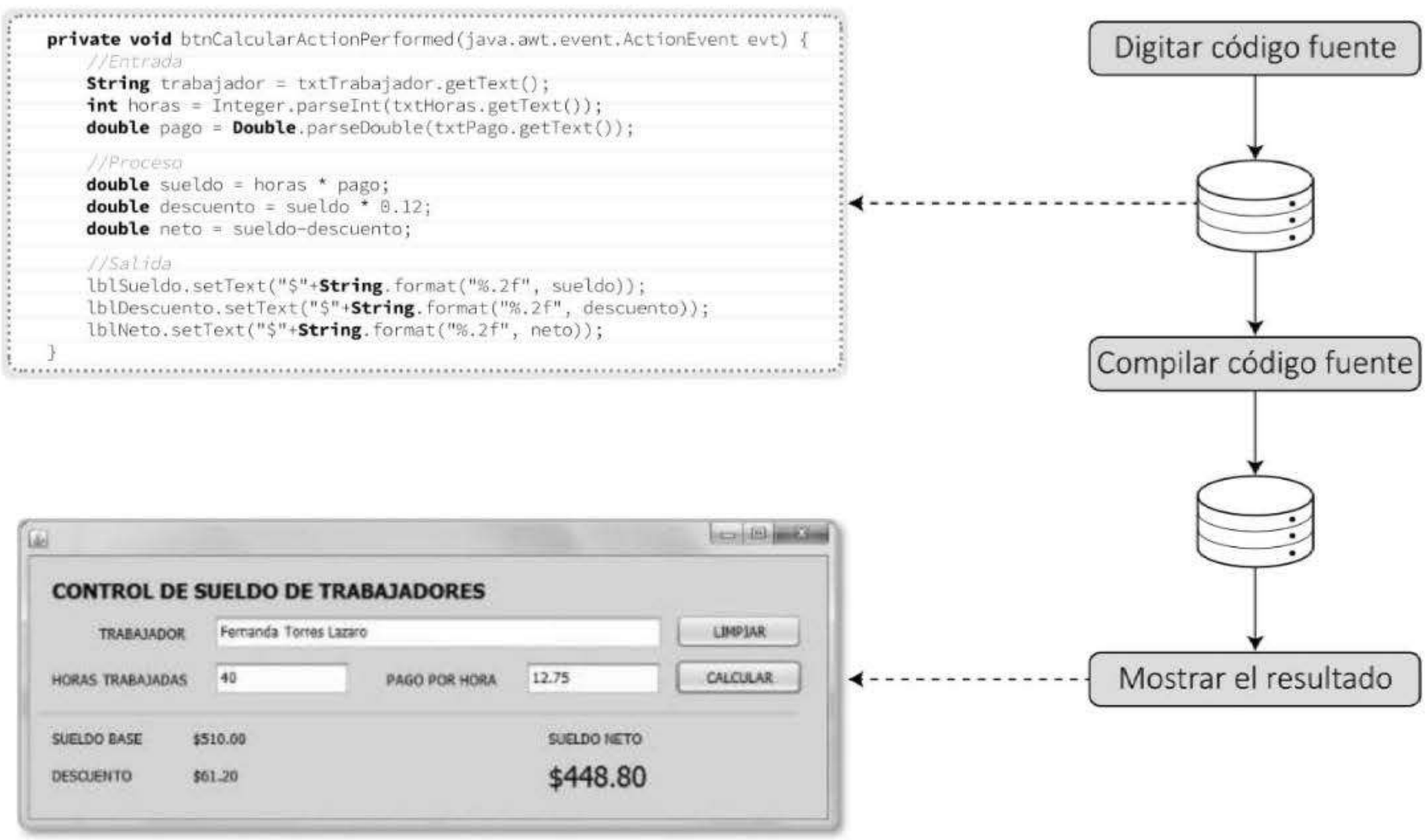


El código se debe colocar dentro del botón **Calcular**, y se muestra como sigue:

```
private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt) {
    //Entrada
    String trabajador = txtTrabajador.getText();
    int horas = Integer.parseInt(txtHoras.getText());
    double pago = Double.parseDouble(txtPago.getText());

    //Proceso
    double sueldo = horas * pago;
    double descuento = sueldo * 0.12;
    double neto = sueldo-descuento;

    //Salida
    lblSueldo.setText('$'+String.format('%.2f', sueldo));
    lblDescuento.setText('$'+String.format('%.2f', descuento));
    lblNeto.setText('$'+String.format('%.2f', neto));
}
```

Proceso de compilación

Luego de haber codificado el código Java se debe traducir a un lenguaje máquina; este proceso lo realiza el compilador de Java, llevando a cabo las siguientes actividades:



2.4 Metodología de programación con Java

El desarrollo de cualquier aplicación que dé una solución adecuada a un problema resulta ser un tanto complicado si se tiene en cuenta que existen diferentes formas de programar. Por tanto, será de suma importancia adecuarse a una metodología que nos permita programar adecuadamente.

Cuando se habla del uso de una metodología de programación, se hace referencia a un conjunto de reglas, principios o métodos que permiten tener una forma adecuada de desarrollo, tal conjunto forma parte de la cultura en programación e indica cómo esta última debe emplearse. Al final de un desarrollo se podrá ver que es posible integrar todas las metodologías en una misma aplicación, y estas pueden ser:

2.4.1 Programación estructurada

Es la programación orientada a mejorar el sentido del desarrollo utilizando solo funciones o procedimientos y tres estructuras como la secuencial, la selectiva y la repetitiva. Surge en la década de los sesenta. El paradigma de la programación estructurada se basa en el teorema desarrollado por Edsger Dijkstra, por eso es que solo se usan tres estructuras básicas de control.

```
String alumno = "ÁNGELA TORRES LÁZARO";
int nota1 = 17;
int nota2 = 18;

double promedio = (nota1 + nota2)/2.0;

String condicion="";
if (promedio >= 12.5)
    condicion = "APROBADO";
else
    condicion = "DESAPROBADO";

for(int i=1; i<=10; i++) {
    System.out.println(condicion);
}
```

Secuencial

Selectiva

Repetitiva

Figura 2.1 Código orientado a la programación estructurada.

- **Secuencial.** Se caracteriza porque el código sigue una línea secuencial al momento de ejecutarse; esto genera una especie de orden dentro del bloque de código, pero no amplía las capacidades que ofrece un lenguaje de programación como Java.
- **Selectiva.** Se caracteriza porque el código puede tomar distintos caminos según una determinada condición.
- **Repetitiva.** Se caracteriza porque el código puede incluirse en un ciclo de repeticiones que puede ser controlado por algún evento.

2.4.2 Programación modular

Es la programación orientada a dividir un programa en porciones de código más pequeñas, también llamadas módulos o subrutinas. Surge como una evolución de la programación estructurada. Se usa generalmente para aplicaciones mucho más complejas.

Con el uso de la programación modular no se deja atrás la estructurada; al contrario, esta se integra dentro de los módulos o subrutinas y hace que una programación sea más robusta y esté totalmente controlada por el programador; esto proviene de una técnica llamada *divide y vencerás*.

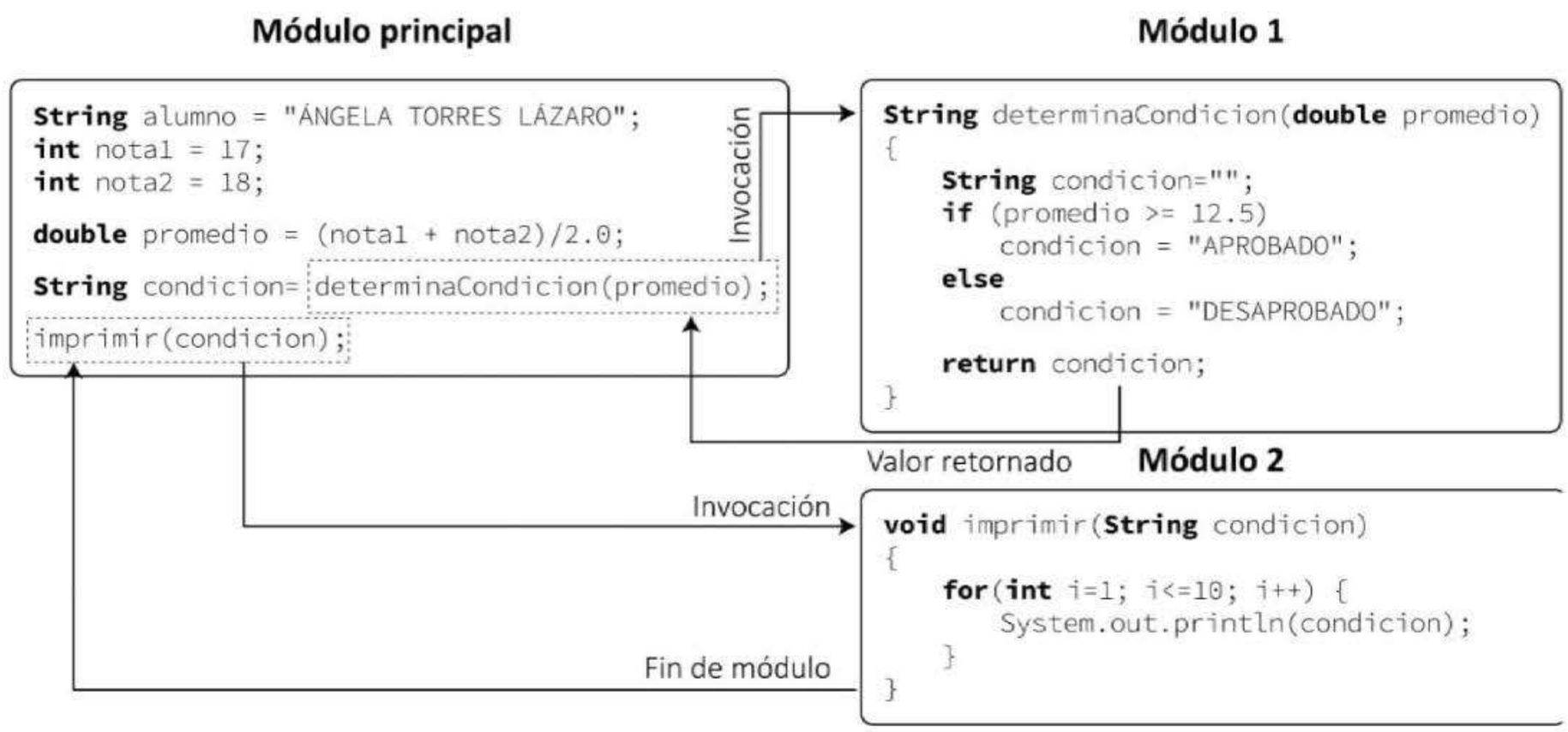


Figura 2.2 Código orientado a la programación modular.

2.4.3 Programación orientada a objetos

Es la programación orientada a la definición de clases, donde, además, por medio de objetos, se puede administrar la información de entrada y salida de una aplicación, permitiendo el uso de algunas especificaciones nuevas como la herencia, la abstracción, el polimorfismo, el acoplamiento y el encapsulamiento.

```
public class ProgramaPrincipal
{
    Promedio objP = Promedio();
    objP.setAlumno('ÁNGELA TORRES LÁZARO');
    objP.setN1(17);
    objP.setN2(18);

    double promedio = calculaPromedio();

    String condicion = determinaCondicion(promedio);
}
```

```
public class Promedio
{
    private String alumno;
    private int n1;
    private int n2;

    public String getAlumno() {
        return alumno;
    }

    public void setAlumno(String alumno) {
        this.alumno = alumno;
    }

    public int getN1() {
        return n1;
    }

    public void setN1(int n1) {
        this.n1 = n1;
    }

    public int getN2() {
        return n2;
    }

    public void setN2(int n2) {
        this.n2 = n2;
    }

    public int getN1() {
        return n1;
    }

    //Calculando el promedio
    public double calculaPromedio() {
        return (n1+n2)/2.0;
    }

    //Determinar la condición
    public String determinaCondicion() {
        if (calculaPromedio() >= 12.5)
            return 'APROBADO';
        else
            return 'DESAPROBADO';
    }
}
```

Asigna valor a los atributos privados

Envía el promedio como valor de retorno

Envía la condición de aprobado o desaprobado como valor de retorno

Figura 2.3 Código orientado a la programación con objetos.

2.5 Elementos que componen una aplicación Java

Una aplicación Java se compone de muchos elementos que el compilador tiene que reconocer y, principalmente, diferenciarlos. El desarrollador debe usar todos los elementos posibles para dar solución a un proceso de negocio, pero siempre respetando las reglas del lenguaje de programación como Java.

A continuación, se muestra el código de una aplicación básica en Java.

```
private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt) {
    //Entrada
    String trabajador = txtTrabajador.getText();
    int horas = Integer.parseInt(txtHoras.getText());
    double pago = Double.parseDouble(txtPago.getText());

    //Proceso
    double sueldo = horas * pago;
    double descuento = sueldo * 0.12;
    double neto = sueldo-descuento;

    //Salida
    lblSueldo.setText('$'+String.format('%.2f', sueldo));
    lblDescuento.setText('$'+String.format('%.2f', descuento));
    lblNeto.setText('$'+String.format('%.2f', neto));
}
```

Como se puede notar, el código mostrado tiene varios elementos que componen una aplicación Java. Estos son:

- **Definición de método.** Permite la definición de un método o procedimiento en Java. Dentro del método se puede especificar un conjunto de sentencias que tienen un objetivo específico; por ejemplo, imprimir un resultado o realizar cálculos.
- **Espacios en blanco.** Una tabulación o un cambio de línea en el código Java es considerado como un espacio en blanco que no altera la interpretación del código por parte del compilador; sin embargo, se sugiere no exagerar en el contenido con espacios en blanco, a menos que sea necesario, como cuando se definen las sentencias una debajo de la otra mediante cambios de línea (**\n**).
- **Llaves.** Define o enmarca el inicio de un bloque de código en Java, el cual es convertido en una unidad lógica de código que se puede utilizar en cualquier parte de la aplicación tomándolo como una sola sentencia. Esto se puede realizar para determinar bloques de código en sentencias como las repetitivas o selectivas.
- **Comentario de línea.** Define una línea de comentario, la cual no será interpretada por el compilador.
- **Sentencia.** Representa una actividad dentro del proceso que tiene un objetivo específico como asignar o imprimir un valor.

2.6 Los comentarios

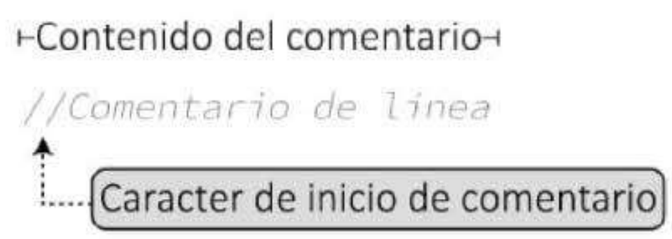
Un comentario es una forma de documentar el código dentro de una aplicación Java, lo cual conlleva al entendimiento del código frente a otros desarrolladores. Una buena práctica es documentar ciertas partes del código para no olvidar qué acción se está realizando.

Se recomienda colocar un comentario en las siguientes ocasiones:

- Al inicio de cada clase, método o antes de una variable
- Al inicio de cada fragmento de código
- Al inicio de una estructura selectiva o repetitiva

Los comentarios se pueden realizar de tres formas:

a) **Comentario de línea.** Se usan dos caracteres *slash* 'barra oblícua' (//) antes de iniciar el comentario, el cual podría incluir cualquier tipo de carácter. Algunos casos de referencia son:



Primera forma:

```
//Obtener el nombre del trabajador
String trabajador = txtTrabajador.getText();
```

Segunda forma:

```
String trabajador = txtTrabajador.getText(); //Obtener el nombre del trabajador
```

b) **Comentario de varias líneas.** Se usa la combinación de un carácter *slash* y un asterisco para iniciar el bloque de comentarios, y finaliza con un asterisco y un carácter *slash*, respetando el mismo orden expuesto. Algunos casos de referencia son:

Primera forma:

```
/* Obtener el nombre del trabajador */
String trabajador = txtTrabajador.getText();
```

Segunda forma:

```
String trabajador = txtTrabajador.getText(); /*Obtener el nombre del trabajador*/
```

c) **Comentario de documentación.** Se usa la combinación de un carácter *slash* y dos asteriscos al inicio, y finaliza con un asterisco y un slash; este tipo de comentario es importante para la documentación del proyecto en el JavaDoc. Se pueden usar las siguientes referencias de comentarios:

- **@author.** Define el nombre del creador de la aplicación.
- **@version.** Define la versión del código, normalmente se usa para clases o métodos.
- **@param.** Define un comentario sobre un parámetro de un método.
- **@return.** Define el valor de salida de un método.
- **@see.** Define una descripción especialmente cuando un bloque de código comentado se encuentra relacionado con otra clase o método.

Algunos casos de referencia son:

Primera forma: comentario que especifica el objetivo del proyecto y el autor del mismo.

```
/**
* Aplicación que determina el sueldo de un trabajador
* Según las horas de trabajo y un costo por hora
* @author Lic. Manuel Torres
* @version 1.0
*/
```

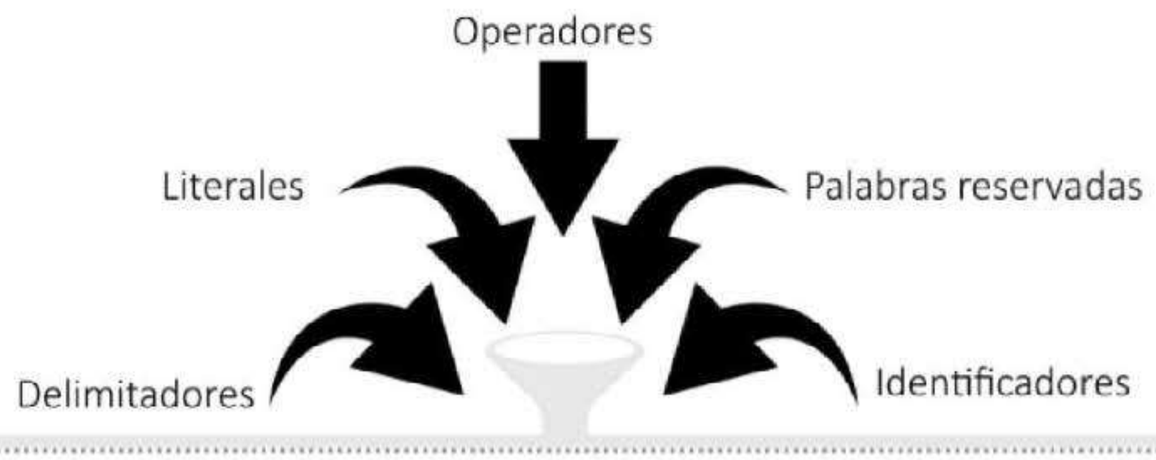
Segunda forma: comentario que especifica el contenido de las variables.

```
/**
* @param horas especifica las horas de trabajo
* @param pago especifica el costo por hora
*/

private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt) {
    String trabajador = txtTrabajador.getText();
    int horas = Integer.parseInt(txtHoras.getText());
    double pago = Double.parseDouble(txtPago.getText());
}
```

Si se necesita visualizar la documentación JavaDoc de su proyecto en NetBeans, desde el menú principal, se selecciona **Run > Generate Javadoc** (proyecto). El resultado se mostrará de la siguiente manera:

2.7 Elementos léxicos en Java

Una aplicación Java contiene un conjunto de elementos que se integran con un fin específico. Es necesario reconocer dichos elementos para poder tener la administración correcta de una aplicación; entre los principales elementos léxicos se puede mencionar a las palabras reservadas del lenguaje Java, los identificadores, los literales, los operadores y los delimitadores.




```
Public class ProgramaPrincipal
{
    Promedio objP - new Promedio();
    objP.setAlumno('ÁNGELA TORRES LÁZARO');
    objP.setN1(17);
    objP.setN2(18);

    double promedio - calculaPromedio();

    String condicion - determinaCondicion(promedio);

}
```

Figura 2.4 Elementos léxicos Java.

2.7.1 Palabras reservadas

Inicialmente, se debe entender que este conjunto de palabras pertenece a un diccionario interno de Java, el cual no puede ser usado de otra forma que no sea la especificada en su formato original. La lista de palabras reservadas con que cuenta Java es la siguiente:

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

2.7.2 Identificadores

Es el nombre que se le asigna a los elementos del código Java, como una clase, un método o una variable. Es una secuencia de caracteres que cumple con las siguientes reglas:

- Un identificador es único dentro de un bloque de código como método o a nivel de clase.
- El identificador debe darle un nombre representativo a su valor.
- Java distingue entre mayúsculas y minúsculas.
- Normalmente inicia por una letra, un guion bajo o el símbolo de dólar ($); los demás caracteres del identificador pueden ser la combinación de letras y números.
- No puede ser una palabra que se encuentra en la lista de palabras reservadas por Java.

A continuación, se analiza el siguiente problema:

> Los ingresos mensuales de una universidad particular se recaudan principalmente de las pensiones de los alumnos. El costo mensual por alumno se determina a partir de los salarios de los docentes, el costo de mantenimiento y otros gastos administrativos de la universidad. Se necesita una aplicación en Java que permita ingresar los montos aproximados para el año actual y así determinar el costo de matrícula y la cuota mensual de cada estudiante.

- Se determinan los posibles identificadores válidos y no válidos a partir del caso expuesto:

Descripción	Válidos	No válidos

Salario de docentes anual	`salario`
	`salario_docentes`
	`$_salario`
	`salario$`
		`salario docentes`
		`@salario`
		`150000`

Costo de mantenimiento anual	`costo`
	`costo_mantenimiento`
	`_costo`
	`COSTO`
		`costo mantenimiento`
		`%costo`
		`200000`

Gastos administrativos anuales	`gasto`
	`gasto_administrativo`
	`$_gasto`
	`GaStO`
		`gasto administrativo`
		`#gasto`
		`75000`

Costo de matrícula	`matricula`
	`costo_matricula`
		`costo matricula`
		`¿matricula`
		`750`

Costo mensual	`pensión`
	`costo_pension`
		`costo pensión`
		`&pensión`
		`750`

2.7.3 Literales

Un literal en Java representa un valor formado por una secuencia de caracteres válidos según el tipo de datos. Los principales literales en un código Java son:

a) **Literal de tipo entero.** Se expresa estrictamente como un valor numérico sin valores decimales. Cuando el valor es positivo, no será necesario especificarlo por el símbolo + (más); por el contrario, si el valor es negativo, obligatoriamente se colocará el símbolo – (menos).

Identificador	Literal
Nota	20
Edad	45
Cantidad	150
Stock	25 000
Pérdida	-500

b) **Literal real.** Puede expresarse como una parte entera y otra fraccionaria, tanto para valores positivos como negativos.

Identificador	Literal
sueldo_empleado	2500.00
interes_negativo	-0.0085
promedio_notas	15.75

c) **Literal carácter.** Se expresa como un valor de un solo carácter, el cual se referencia por medio de comillas simples.

Identificador	Literal
`\'`	Sirve para especificar una comilla simple.
`\"`	Sirve para especificar una comilla doble.
`\\`	Sirve para especificar una barra inclinada.
`\n`	Sirve para especificar un salto de línea.
`\t`	Sirve para especificar una tabulación.

d) **Literal cadena.** Puede expresarse como un conjunto de caracteres.

Identificador	LiteraL
Mensaje	`'Pago registrado correctamente'`
Condición	`'El estudiante \n no adeuda cuotas'`
Advertencia	`'\'Está seguro de salir de la aplicación \''`

e) **Literal booleano.** Se expresa mediante los valores **true** y **false**.

Identificador	Literal
Sexo	true
Condición	false

2.7.4 Operadores

Son símbolos que tienen una tarea específica al componer una expresión en Java. Este lenguaje cuenta con operadores de asignación, aritméticos binarios, aritméticos unarios de incremento y decremento, complejos, de comparación y lógicos, además del operador de concatenación y el ternario.

a) **Operadores de asignación.** Se caracterizan por asignar un valor a una variable; este valor se podría obtener a partir de una expresión.

Operador	Acción	Caso

=	Asignación	Asigna un valor entero a una variable.
		Primera forma:
		`int n;`
		`n = 10;`
		Segunda forma:
		`int n = 10;`

Otra forma de asignación es mediante el uso de expresiones. Se presentan los siguientes casos:

Asignación de una cadena de texto	`String mensaje;` `Mensaje = 'Error de ingreso de datos';`
Asignación de una expresión matemática	`double r;` `r = (12 + 27 + 15) / 3.0;`

Finalmente, se puede hacer una asignación múltiple de la siguiente manera:

Asignación múltiple	`int n, m;` `n = m = 10;`

b) **Operadores aritméticos binarios.** Se caracterizan por unir en una expresión dos elementos de programación.

Operador	Acción	Caso

+	Suma	`//Sumar dos valores enteros`
		`int n1 = 10;`
		`int n2 = 15;`
		`int suma = n1 + n2;`
		Resp: 25

-	Resta	`//Restar dos valores enteros`
		`int n1 = 20;`
		`int n2 = 10;`
		`int resta = n1-n2;`
		Resp: 10

*	Multiplicación	`//Mostrar el producto de dos valores numéricos enteros`
		`int n1 = 5;`
		`int n2 = 9;`
		`int multiplica = n1 * n2;`
		Resp: 45

Operador	Acción	Caso

/	División	//Mostrar el valor resultante de una división de dos valores enteros.
		int n1 = 1;
		int n2 = 2;
		double d = n1 / n2;
		Resp: 0
		double n1 = 1.0;
		int n2 = 2;
		double divide = n1 / n2;
		Resp: 0.5

%	Módulo	//Mostrar el resto de dividir dos valores enteros.
		int n1 = 4;
		int n2 = 3;
		int resto = n1 % n2;
		Resp: 1

c) **Operadores aritméticos unarios.** Se caracterizan por tener un solo operando, de modo que destacan la asignación de signo a un valor, ya sea positivo o negativo, y aumentan o disminuyen en uno su valor.

Operador	Acción	Caso

+	Positivo	int a = 1;
		int n = +a;
		//Resp: n = 1

--	Negativo	int a = 1;
		int n = -a;
		//Resp: n = -1

++	Prefija	int i = 0;
		i++;
		//Resp: i = 1
		//Incrementa i en 1, pero se evalúa al valor anterior al incremento.

++	Posfija	int i = 0;
		++i;
		//Resp: i = 1
		//Incrementa i en 1, pero se evalúa al valor posterior al incremento.

--	Prefija	int i = 10;
		i--;
		//Resp: i = 9
		//Decrementa i en 1, pero se evalúa al valor anterior al incremento.

--	Posfija	int i = 10;
		--i;
		//Resp: i = 9
		//Decrementa i en 1, pero se evalúa al valor posterior al incremento.

d) **Operadores complejos.** Se caracterizan por aumentar y asignar un nuevo valor a una variable.

Operador	Acción	Caso

+=	Acumulado positivo	`int n - 0;`
		`int a - 10;`
		`n +- a;`
		`n +- a;`
		`//Resp: n - 20`

-=	Acumulado negativo	`int n - 1000`
		`int a - 10;`
		`n -- a;`
		`//Resp: n - 990`

*=	Acumulado de productos	`int n - 1;`
		`int a - 2;`
		`n *- a;`
		`n *- a;`
		`//Resp: n - 4`

/=	Acumulado de divisiones	`double a - 1;`
		`double b - 2;`
		`double a /- b;`
		`//Resp: a - 0.5`

e) **Operadores de comparación.** Se caracterizan por efectuar una comparación lógica entre dos valores, de modo que el resultado puede ser **true** o **false**. Normalmente se usa dentro de una estructura selectiva como **if** o repetitiva como **for**.

Operador	Acción	Caso

==	Igualdad numérica o carácter	`//Evaluar un valor numérico`
		`if (n -- 10)`
		`//Evaluar un valor decimal`
		`if (sueldo -- 1000.00)`
		`//Evaluar un valor booleano`
		`if (estado -- true)`
		`//Evaluar si el valor n ya es 10 dentro del ciclo while`
		`while (n -- 10)`

>	Mayor que	`//Evaluar si el valor de N supera a 10`
		`if (n > 10)`
		`//Evaluar si el valor de N es superior a 10 dentro de un ciclo //de repeticiones`
		`while (n > 10)`

>=	Mayor o igual que	`//Evaluar si el valor sueldo es mayor o igual a 1000.00` `if (sueldo >- 1000.00)`

<	Menor que	`//Evaluar si el valor de N es inferior a 10`
		`if (n < 10)`
		`//Evaluar si el valor de N es inferior a 10 dentro de un ciclo //de repeticiones`
		`while (n < 10)`

Operador	Acción	Caso
<=	Menor o igual que	`//Evaluar si el valor sueldo es menor o igual a 1000.00` `if (sueldo <- 1000.00)`

!=	Diferente	`//Evaluar si el valor de N no es igual a 10`
		`if (n !- 10)`
		`//Evaluar si el valor N no llega al valor 10 en un ciclo de //repeticiones`
		`while (n !-10 )`

f) **Operadores lógicos.** Se caracterizan por unir y evaluar expresiones dentro de una estructura condicional o repetitiva. El resultado de la evaluación puede ser **true** o **false** y se basa en los siguientes modelos de tablas:

Tabla de verdad para Y lógica

p	q	p && q
V	V	V
V	F	F
F	V	F
F	F	F

Tabla de verdad para O lógica

p	q	p && q
V	V	V
V	F	V
F	V	V
F	F	F

Tabla de verdad para negación

p	!p
V	F
F	V

Operador	Acción	Caso
&&	Y lógica	`if (n>-1 && n<-10)` `/*Evalúa si el valor n se encuentra en el rango de 1 a 10*/`
||	O lógica	`if (n--1 || n--10)` `/*Evalúa si el valor n puede ser 1 o 10*/`
!	Negación	`if !(n--1)` `/*Si al evaluar n-1 resulta true, el símbolo de negación invierte el resultado a false*/`

g) **Operador de concatenación.** Se caracteriza por unir dos valores cuando estos son de tipo cadena de caracteres.

Operador	Acción	Caso
+	Concatenar	/*Mostrar la fecha concatenada a partir de valores como el día, mes y año desde variables de tipo String*/ **String** dia='02'; **String** mes='Febrero'; **String** año='2018'; **String** fecha = dia+mes+año; Resp: 02Febrero2018 Para agregarle espacios entre los valores se tiene que expresar de la siguiente forma **String** fecha = dia+' '+mes+' '+año; /*Mostrar la fecha concatenada a partir de valores como el día numérico, mes de tipo String y año de tipo int*/ **int** dia=2; **String** mes='Febrero'; **int** año=2018; **String** fecha = **String**.valueOf(día) + ' ' + mes++ ' ' +**String**.valueOf(año); Resp: 2 Febrero 2018

h) **Operador ternario.** Se caracteriza por realizar una actividad dependiendo de una condición; el trabajo es muy parecido al realizado por la sentencia selectiva **if**.

Operador	Acción	Caso
?	Condición	/*Mostrar el mensaje aprobado y desaprobado dependiendo del promedio de notas */ **double** promedio = 10; **String** condicion = ''; condicion>=12.5 ? 'Aprobado' : 'Desaprobado';

Precedencia de los operadores

Entre los operadores existe un nivel de prioridad que se debe tener en cuenta al momento de crear una expresión. Java los organiza primero por unarios, aritméticos, relacionales, booleanos y, por último, por los operadores de asignación. La tabla de precedencias es como sigue:

Tipo de operador	Operadores
Operadores posfijos	expr++ expr--
Operadores prefijos y unarios	++expr --expr +expr -expr !
Aritméticos binarios	* / %

Tipo de operador	Operadores
Aritméticos binarios	+ -
Comparación	< <= > >=
Igualdad y desigualdad	== !=
Y lógico	&&
O lógico	||
Operadores de asignación	= += -= *= /=

2.7.5 Delimitadores

Especifican la separación entre elementos del código Java, por lo que es importante reconocer cuándo usar uno u otro delimitador dentro de una aplicación. Se tiene la siguiente lista de separaciones:

Operador	Descripción

( )	Los paréntesis se pueden implementar para especificar una lista de parámetros y llamadas a un método; asimismo, pueden definir el orden de prioridad de una expresión y controlar las conversiones de tipo.
	■ Especificación de parámetros en un método. Ejemplo:
	`void imprimir(String trabajador, double sueldo)`
	■ Invocación de un método. Ejemplo:
	`imprimir('Fernanda Torres Lazaro',4500.00)`
	■ Definición de la precedencia en una expresión. Ejemplo:
	`double r = (1.0/2) * 2;`
	■ Controlando la conversión de tipo. Ejemplo:
	`int n = 10;`
	`float x = (float) n;`

Operador	Descripción

{ }	Las llaves definen un bloque de código dentro de una estructura o definición de clase; por otro lado, pueden inicializar valores a un arreglo.
	■ Asignar valores a una matriz. Ejemplo:
	`int[] edad = {45, 23, 11, 9};`
	■ Definición de bloque de código dentro de una clase, método o estructura. Ejemplo:
	`public class Persona { }`
	`public void calculaImporte(){ }`
	`String asignaRegalo(){ }`
	`If (edad>18){ }`
	`while(i<=10){ }`

[ ]	Los corchetes se usan para definir y establecer valores referenciados a una matriz. Por ejemplo:
	■ Declaración de arreglo unidimensional y bidimensional
	`String alumnos[];`
	`int A[][];`
	■ Asignar un valor a los arreglos
	`alumnos[1] = 'Fernanda Torres Lázaro';`
	`a[1][1] = 10;`

;	El punto y coma separa todo tipo de sentencia en una aplicación Java. Este determina el fin de una actividad para el compilador de Java.
	`//Sentencia de asignación`
	`double sueldo = 15200;`
	`//Sentencia repetitiva`
	`do{`
	`…`
	`}while(n<=10);`
	`//Separador de parámetros en la estructura for. Ejm: For para 10 número entero`
	`for (int i=1;i<=10;i++)`

,	La coma separa identificadores consecutivos que se pueden realizar en una declaración de variables o separa parámetros en un método dentro de una aplicación Java. Por ejemplo:
	`//Declarar 5 variables de tipo entero`
	`int n1,n2,n3,n4,n5;`
	`//Especificar 3 parámetros de tipo entero a un método`
	`void calculaPromedio(int n1, int n2, int n3)`
	`//Especificar varios parámetros en la estructura for.`
	Ejemplo:
	`for (int i=1 , int j=0 ; i<=10 , j<=9 ; i++ , j++)`

.	El punto separa el nombre de los paquetes de sus clases contenedoras; asimismo, permite invocar a atributos o métodos públicos de una clase a otra. Por ejemplo:
	`//Importando el paquete swing`
	`imports javax.swing.*;`
	`//Invocando al atributo público de una clase`
	`Persona objPersona = new Persona();`
	`objPersona.nombres = 'Ángela Torres L.';`

2.8 Los paquetes y las importaciones

Los paquetes son contenedores de clases, interfaces, formularios, imágenes y de conexión a base de datos, que permiten una administración más ordenada en una aplicación Java. Una de las características de la programación orientada a objetos es la encapsulación y los paquetes ofrecen un mecanismo de alto nivel. La definición del paquete a nivel de código se muestra de la siguiente manera:

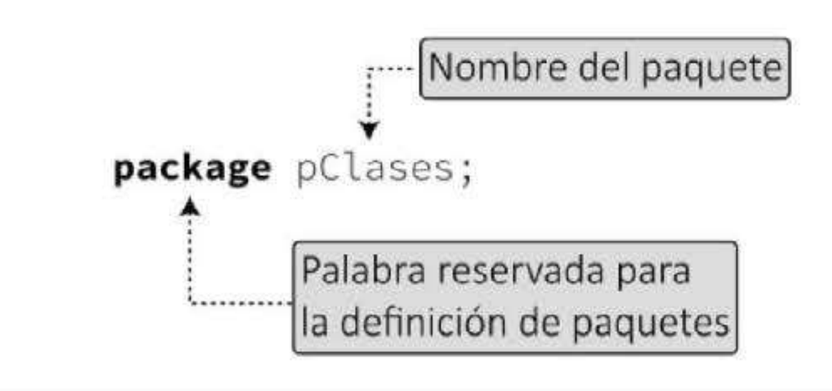


Figura 2.5 Definición del paquete **pClases**.

Este código se encuentra siempre antes de la definición de clases, pues indican que tal definición se encuentra en dicho paquete. Cuando se necesita invocar a una clase contenida en un determinado paquete, se usa la palabra reservada **import** de la siguiente manera:

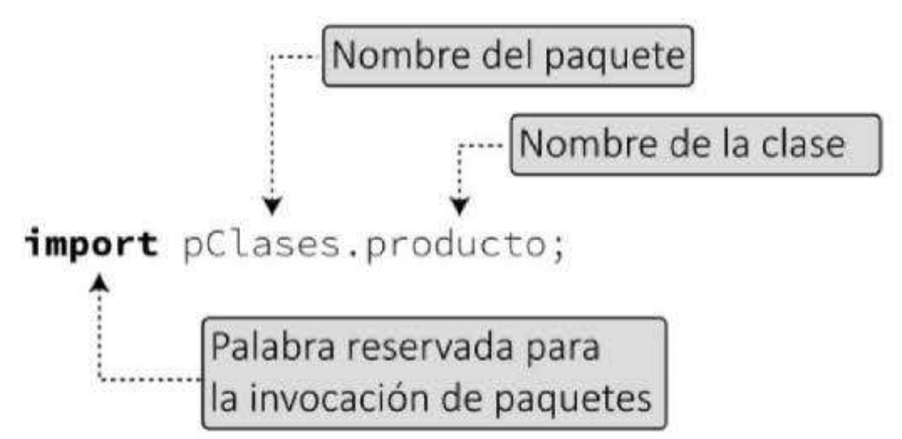


Figura 2.6 Invocando a la clase **producto** dentro del paquete **pClases**.

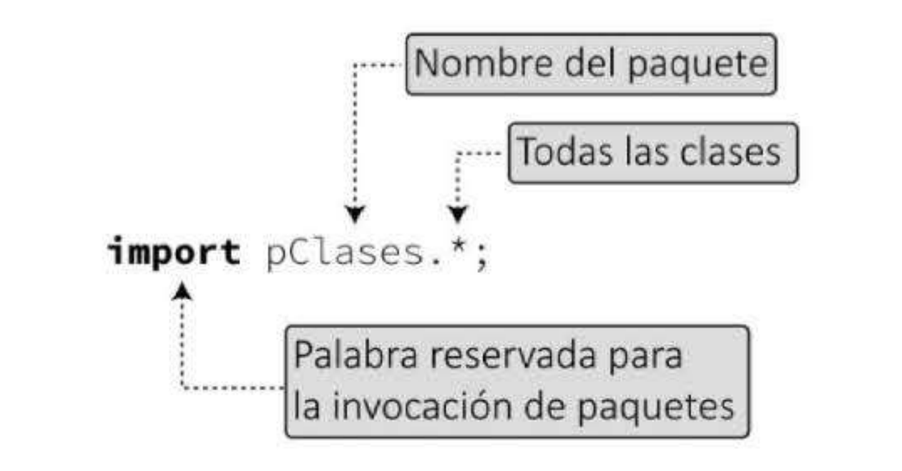


Figura 2.7 Invocando a todas las clases dentro del paquete **pClases**.

A continuación, se presenta una lista de clases con sus respectivos paquetes más usados en una aplicación Java.

2.8.1 **java.lang**

Contiene un conjunto de clases fundamentales para el lenguaje Java; se caracteriza por su importación de forma automática en una aplicación.

- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Math
- Number
- Object
- System
- Thread
- Void
- String

2.8.2 **Java.awt**

Contiene un conjunto de clases que permiten diseñar una interfaz gráfica de usuario a partir de entornos, gráficos, etc.

- Composite
- LayoutManager
- BorderLayout
- Paint
- Dialog
- Event
- Font
- Frame
- Graphics
- Image
- Label
- Menu
- Scrollbar

2.8.3 **Javax.swing**

Contiene un conjunto de clases modernas para el diseño de la interfaz gráfica del usuario, el cual presenta mejoras con respecto a la AWT.

- Action
- ActionMap
- Icon
- ListModel
- MenuElement
- JApplet
- JButton
- JCheckBox
- Jframe
- JMenu
- JLabel
- JPanel
- JTextField
- JTree
- Timer

2.8.4 **java.io**

Contiene clases que soportan flujos de entrada y salida en una aplicación Java.

- BufferedReader
- FileFilter
- BufferedWrite
- FilenameFilter
- ObjectInput
- Serializable
- DataOutputStream

2.8.5 Java.util

Contiene clases de utilidad para una aplicación Java como enumeraciones, vectores, fecha, hora, separación de token, generación de números aleatorios, propiedades del sistema, etc.

- Collection
- Comparator
- AbstracList
- Enumeration
- AbstractMap
- EventListener
- Iterator
- List
- ArrayList
- Collection
- SortedSet
- EventObject
- Random
- Stack
- Timer
- Vector
- Date

2.9 Tipos de datos

Representa un conjunto de valores que puede tomar una variable. Al definir un tipo de datos, se define la capacidad que puede tener un valor. Java no cuenta con una variedad de tipos de datos, pero deja la posibilidad de crear sus propios tipos de datos mediante una clase, como es el caso de la clase **String**.

Algunas características de los tipos de datos son las siguientes:

- Se tienen ocho tipos de datos primitivos agrupados en valores enteros, reales, caracteres y boleanos.
- Un tipo de datos determina el rango que una variable puede tener.
- La definición de una variable en Java no distingue la plataforma de trabajo.
- Java es un lenguaje de programación estrictamente *tipado*, es decir, toda variable debe ser declarada con su propio tipo de datos.

Según el tipo de información que se almacene dentro de una variable, se asignará un tipo de datos siguiendo dos tendencias:

a) **Tipos de datos primitivos.** Estos representan a un solo valor. También son llamados *escalares*. La forma de identificar un tipo de datos primitivo es que siempre se inicia con una letra minúscula.

b) **Tipos de datos por referencia.** Estos representan a un conjunto de valores, como, por ejemplo, los arreglos, las clases, las interfaces, etc. La forma de identificarlos es que siempre comienzan con una letra mayúscula.

A continuación, se muestran los tipos de datos primitivos agrupados por categorías:

Categoría	Tipos de datos
Enteros	byte (entero de 8 bits) short (entero de 16 bits) int (entero de 32 bits) long (entero largo de 64 bits)
Reales	float (coma flotante de precisión simple de 32 bits) double (coma flotante de precisión doble de 64 bits)

Categoría	Tipos de datos
Carácter	char (carácter)
Boleanos	boolean (lógico)

Ahora, se observan las capacidades de los tipos de datos primitivos:

Tipos de datos	Capacidad mínima	Capacidad máxima
byte	-128	+127
short	-32768	+32767
int	-2147483648	+2147483647
long	-9223372036854775808	+9223372036854775807
float	±3.40282347E+38	±1.40239846E-45
double	±1.79769313486231570E+308	±4.94065645841246544E-324
char	\u0000	\uFFFF

También se debe considerar el valor por defecto de cada tipo de datos primitivos. Esto podría ayudar en la programación cuando se declaran variables globales o de clase, ya que dicha declaración no necesita especificación de valores iniciales, pues su valor se asigna por defecto. He aquí una lista de dichos valores:

Tipo de datos	Valor por defecto
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u000'
boolean	False

2.10 Las variables

Una variable es una posición de memoria destinada a almacenar cierta información expresada en valores; dicho valor debe ser del mismo tipo que la variable. Asimismo, toda variable tiene la capacidad de ser modificable durante el ciclo de una aplicación Java.

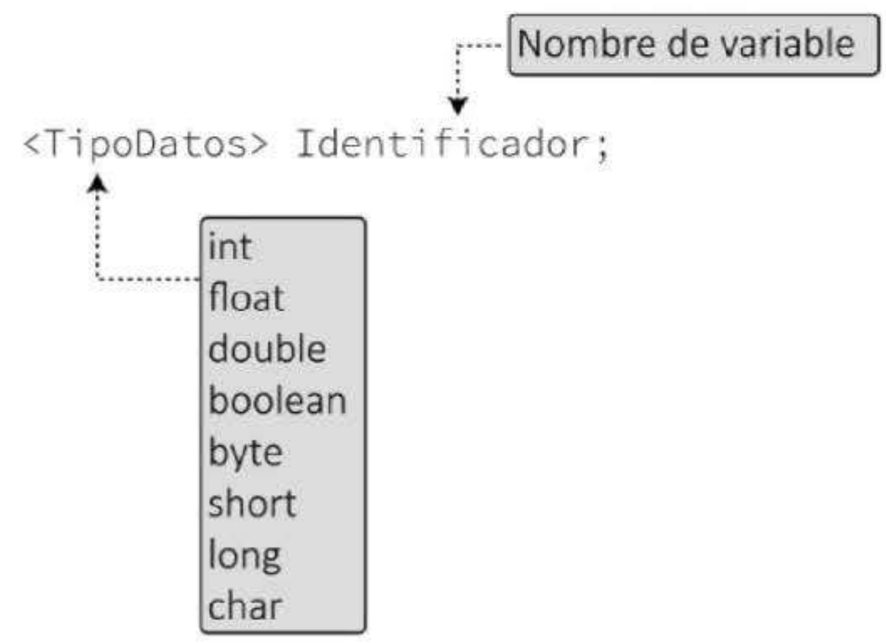
Java es considerado como un lenguaje de programación altamente *tipado*, es decir, todo lo que se use en la programación debe ser declarado usando identificadores.

Al declarar variables, se debe tener en cuenta lo siguiente:

- La asignación de un nombre depende de lo que se va a almacenar; por ejemplo, si se necesita almacenar un salario bruto, entonces se debe declarar la variable como salarioBruto.
- Si la variable declarada es totalmente en mayúsculas, así deberá usarse dentro de la aplicación; esto se debe a la sensibilidad de las mayúsculas y minúsculas en Java.

- Toda variable se compone de un nombre identificativo, un tipo de datos y un valor que contendrá.
- El tiempo de vida de una variable dependerá del uso que tenga dentro de una clase o método; es decir, al finalizar un método, también finaliza la variable.

Formato:



Normalmente la declaración de variables ocurre en las siguientes ocasiones:

a) **Antes de asignar un valor.** Cuando se necesita asignar un valor a una variable, esta debe ser declarada una línea antes de su uso o hacerlo directamente. Obsérvese el siguiente ejemplo:

Se inicia por declarar la variable **salario** de tipo **double** por tratarse de un monto de dinero, luego se le asigna el valor 2500. Al declarar la variable, la memoria separa un espacio para un determinado valor; y cuando se le asigna un valor, la memoria llena dicho espacio. Si en algún momento de la aplicación se necesita dicho valor, este será descargado desde la memoria.

b) **Dentro de una clase.** La variable declarada dentro de la definición de clase se considera como elemento o atributo de clase, por lo tanto, se puede usar dentro de cualquier método de la clase.

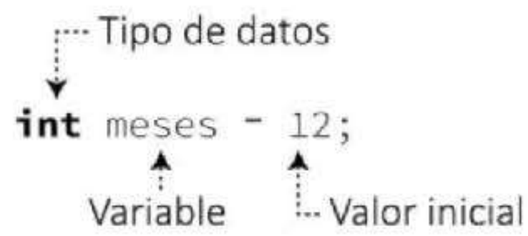
c) **Dentro de un método.** La variable declarada dentro de un método solo será accesible dentro de él.

```
public class empresa {
   ...
   double calculaDescuento() {
      double descuento;
      descuento - salario * 0.12;
      return descuento;
   }
}
```

Variable del método

2.10.1 Inicialización de variables

Cuando las variables son declaradas dentro de un método, o son usadas como contadores o acumuladores, deben ser inicializadas con un valor según su tipo de datos. Hay que considerar que el valor de inicialización debe ser el mismo que se declara. El formato de inicialización es como sigue:



Otra forma de inicializar las variables es declarar la variable y luego asignarle un valor, por ejemplo:

```
int meses;
meses-12;
```

Y, por otro lado, si la variable declarada es parte de un proceso condicional, es obligatorio inicializarlo, por ejemplo:

```
double descuento - 0;
if (monto > 1700) descuento - 0.12*monto;
```

2.10.2 Conversiones de tipo

Cuando se efectúa dentro del código una asignación de variable, se debe tener en cuenta que el valor cumpla con el rango especificado por el tipo de datos; esto se debe a una evaluación de compatibilidad que realiza el compilador. Existen dos tipos de conversiones:

1. **Conversión automática**

   Esta se realiza cuando una variable de un determinado tipo se asigna a otro tipo con la restricción de que ambos tipos sean compatibles, siempre y cuando no exista pérdida de información o que el tipo de datos destino sea más grande que el origen. Se analiza el siguiente caso:

```
//Aumentar dos puntos al promedio
int puntos - 2;
double promedio - 12.50;
promedio - promedio + puntos;
```

   La variable **puntos** se declara como tipo **entero**; mientras que **promedio**, como tipo **double**. Finalmente, **promedio** acumula el valor **puntos** en la variable **promedio**, haciendo una conversión automática de la variable **puntos** a tipo **double** para que sean compatibles en la acumulación.

2. **Conversión explícita**

Esta se realiza cuando un valor de tipo de datos de mayor capacidad es convertido a uno de inferior capacidad. A esto se le llama estrechamiento, debido a que se estrecha explícitamente el valor para que pueda caber en el tipo de datos destinado. El formato de conversión es como sigue:

```
variable = (tipo) valor;
```

Valor de asignación

Tipo de datos a convertir

La variable tiene un tipo de datos específico, mientras que el valor tiene otro tipo no compatible; entre paréntesis se especifica el tipo de datos que sea compatible al tipo de la variable para realizar una conversión correcta. Por ejemplo:

```
int n=100;
byte x=(byte)n;
```

La variable **n** de tipo **entero** tiene como valor asignado 100, y se necesita convertirlo al tipo **byte**, el cual es de menor capacidad; es aquí donde se realiza la conversión usando la técnica *casting* sobre el tipo de datos de la variable **n**.

Cuando se realizan las conversiones, se debe conocer exactamente la capacidad de cada uno de ellos, ya que todos los tipos de datos se pueden convertir, pero no se garantiza una conversión segura entre ellas.

Por ejemplo, si se tiene una variable de tipo **integer** (32 bits) y se desea convertirlo al tipo **byte** (8 bits), se perderán 24 bits en plena conversión. Esto podría ocasionar u na pérdida importante al valor original entero. La siguiente tabla muestra la norma que se necesita para las conversiones sin pérdida de información.

Tabla 2.3 Norma para las conversiones sin pérdida de información.

Tipo de datos origen	Tipo de datos destino
Byte	double, float, long, int, char, short
Short	double, float, long, int
Char	double, float, long, int
Int	double, float, long
long	double, float
float	double

```
//Convertir una variable de tipo short a otro de tipo double
short s =10;
double d = (double) s;
```

2. **Conversión explícita**

Esta se realiza cuando un valor de tipo de datos de mayor capacidad es convertido a uno de inferior capacidad. A esto se le llama estrechamiento, debido a que se estrecha explícitamente el valor para que pueda caber en el tipo de datos destinado. El formato de conversión es como sigue:

```
variable = (tipo) valor;
```

Valor de asignación

Tipo de datos a convertir

La variable tiene un tipo de datos específico, mientras que el valor tiene otro tipo no compatible; entre paréntesis se especifica el tipo de datos que sea compatible al tipo de la variable para realizar una conversión correcta. Por ejemplo:

```
int n=100;
byte x=(byte)n;
```

La variable **n** de tipo **entero** tiene como valor asignado 100, y se necesita convertirlo al tipo **byte**, el cual es de menor capacidad; es aquí donde se realiza la conversión usando la técnica *casting* sobre el tipo de datos de la variable **n**.

Cuando se realizan las conversiones, se debe conocer exactamente la capacidad de cada uno de ellos, ya que todos los tipos de datos se pueden convertir, pero no se garantiza una conversión segura entre ellas.

Por ejemplo, si se tiene una variable de tipo **integer** (32 bits) y se desea convertirlo al tipo **byte** (8 bits), se perderán 24 bits en plena conversión. Esto podría ocasionar u na pérdida importante al valor original entero. La siguiente tabla muestra la norma que se necesita para las conversiones sin pérdida de información.

Tabla 2.3 Norma para las conversiones sin pérdida de información.

Tipo de datos origen	Tipo de datos destino
Byte	double, float, long, int, char, short
Short	double, float, long, int
Char	double, float, long, int
Int	double, float, long
long	double, float
float	double

```
//Convertir una variable de tipo short a otro de tipo double
short s =10;
double d = (double) s;
```

Un ejemplo de pérdida de información en la conversión se puede ver en el siguiente caso:

```
double promedio = 10.50;
int prom = (int) promedio;
```

El resultado de imprimir la variable **prom** es el valor 10 y pierde todo valor fraccionario, ya que ha sido casteado a entero bajando la precisión de su valor.

2.11 La clase Integer

Es la clase más usada ya que posee métodos que permiten convertir un valor de tipo **int** en otro como **long**, **float** o **double**. La clase **Integer** envuelve un valor de tipo **int** en un objeto; este objeto contiene un campo simple cuyo tipo de datos es **int**.

Tabla 2.4 Principales métodos que se usarán en las aplicaciones.

MAX_VALUE	Devuelve el valor máximo que puede tener un tipo **int**. Ejemplo: `int n = Integer.MAX_VALUE;` El resultado de imprimir el valor **n** es 2147483647
MIN_VALUE	Devuelve el mínimo valor que puede tener un tipo **int**. Ejemplo: `int n = Integer.MIN_VALUE;` El resultado de imprimir el valor **n** es -2147483648

parseInt(String s)	Devuelve el valor de tipo **int** desde un valor de tipo **String**. Hay que tener en cuenta que el valor tiene que ser numérico; caso contrario, genera un error de excepción llamado NumberFormatException. Ejemplo:
	`String n = '100';`
	`int x = Integer.parseInt(n);`
	El resultado de imprimir la variable **x** es 100, pero como tipo **entero**; esto quiere decir que se puede usar dicho valor en una expresión como valor numérico entero.

toString()	Permite convertir un número entero en un objeto de tipo **String**. Ejemplo:
	`integer n = 100;`
	`String x=n.toString();`
	`JOptionPane.showMessageDialog(null, 'El Valor de N es: '+n.toString());`
	También se puede representar:
	`Integer n = 100;`
	`JOptionPane.showMessageDialog(null, 'El Valor de N es: '+n.toString());`
	El resultado de imprimir la variable **x** es 100, como valor de tipo **String**.

toString(int i)	Permite convertir un número entero en un objeto de tipo **String**, usando al número entero como parámetro. Ejemplo:
	`int n=Integer.MAX_VALUE;`
	`JOptionPane.showMessageDialog(null, 'El Valor de N es: '+Integer.toString(n));`

2.12 La clase String

Java posee gran capacidad para el manejo de cadenas dentro de sus clases **String** y **StringBuffer**. Un objeto **String** representa una cadena alfanumérica de un valor constante que no puede ser cambiada después de haber sido creada. Un objeto **StringBuffer** representa una cadena cuyo tamaño puede variar.

Los **String** son objetos constantes y, por lo tanto, muy baratos para el sistema. La mayoría de las funciones relacionadas con cadenas esperan valores **String** como argumentos y devuelven valores **String**.

Tabla 2.5 Principales métodos de la clase **String**.

<table>
<tr><th>Método</th><th>Explicación</th></tr>
<tr><td>charAt</td><td>Retorna el carácter de una posición específica de la cadena. Se debe tener en cuenta que cada carácter es almacenado en una posición determinada del arreglo unidimensional y que los espacios en blanco también ocupan un espacio. Por ejemplo, si se tiene la siguiente declaración:
String cadena = 'Programación Java';
<table><tr><td>P</td><td>r</td><td>o</td><td>g</td><td>r</td><td>a</td><td>m</td><td>a</td><td>c</td><td>i</td><td>ó</td><td>n</td><td></td><td>J</td><td>a</td><td>v</td><td>a</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td></tr></table>y se necesita imprimir los caracteres ubicados en la posición 1, 4 y 13; el código sería:
<pre>JOptionPane.showMessageDialog(null,'Posicion 1:'+cadena.
charAt(1));
JOptionPane.showMessageDialog(null,'Posicion 4:'+cadena.
charAt(4));
JOptionPane.showMessageDialog(null,'Pos. 13:'+cadena.
charAt(13));</pre>Ahora bien, si se quisiera imprimir el reverso de la cadena, sería de la siguiente forma:
<pre>for(int i=cadena.length()-1;i>=0;i--)
    JOptionPane.showMessageDialog(null,cadena.charAt(i));</pre>Primero se debe determinar la cantidad de caracteres que tiene la cadena; para ello se usa el método length() y se implementa la estructura for para que recorra en forma descendente. Luego se imprimen los caracteres por medio de un mensaje.</td></tr>
<tr><td>compareTo</td><td>Compara dos cadenas de forma lexicográfica. Emitirá un valor cero (0) si las cadenas comparadas son exactamente iguales; en caso contrario, enviará el valor uno (1). Por ejemplo, veamos cómo realizar la comparación entre dos cadenas de caracteres:
<pre>/* En el punto uno se declara y asigna con el valor Java a
la variable cadena, luego se crea la variable numérica i que
obtendrá el valor numérico de la comparación (0 iguales - 1 No
iguales).*/
String cadena='Java';
int i=cadena.compareTo('Java');

/* Luego se implementa la condicional if en la cual se compara
qué valor devolvió la comparación y de acuerdo a esto emitirá
mensajes*/
if (i==0)
    JOptionPane.showMessageDialog(null,'Son Iguales');
else
    JOptionPane.showMessageDialog(null,'No son Iguales');</pre></td></tr>
</table>

Método	Explicación

concat	Permite concatenar dos cadenas en una sola. Por ejemplo, veamos el código para concatenar dos cadenas de caracteres:
	`String cadena='Programación ';`
	`String nuevaCadena=cadena.concat('Java');`
	`JOptionPane.showMessageDialog(null,nuevaCadena);`
	El resultado de la concatenación es Programación Java.

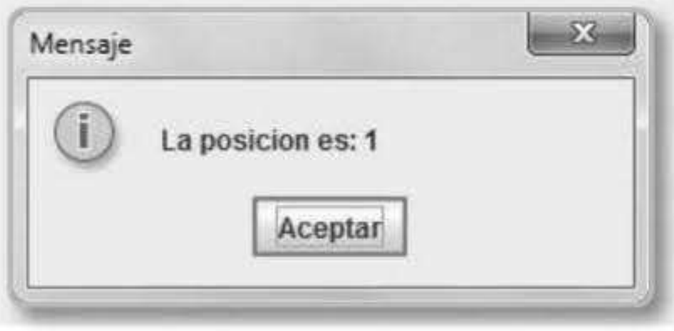
equals	Compara la cadena con un objeto especificado. Por ejemplo, veamos cómo comparar dos cadenas de caracteres.
	`/* Iniciamos declarando la variable cadena1 con el valor Java donde la primera letra es en mayúsculas y en la cadena2 con el valor java totalmente en minúsculas */`
	`String cadena1='Java';`
	`String cadena2='java';`
	`/* Luego se implementa la condición que compara la cadena1 con la cadena2 con el método equals cadena1.equals(cadena2) o también cadena2.equals(cadena1), al final se imprimen los mensajes dependiendo de la igualdad de las cadenas */`
	`if (cadena1.equals(cadena2))`
	`  JOptionPane.showMessageDialog(null,'Son Iguales');`
	`else`
	`  JOptionPane.showMessageDialog(null,'No son Iguales');`

equalsIgnoreCase	Compara una cadena con otra sin considerar las mayúsculas. Por ejemplo, veamos el código de logeo de usuario y su clave.
	`/* Iniciamos declarando la variable usuario y clave para poder asignar los valores que el usuario ingrese; se usa la instrucción showInputDialog que permite ingresar un valor textual*/`
	`String usuario=JOptionPane.showInputDialog(null,`
	`'Ingrese usuario: ');`
	`String clave=JOptionPane.showInputDialog(null,`
	`'Ingrese clave: ');`
	`/* Luego se compara el valor ingresado con el usuario y la clave predeterminada, al usar el método equalsIgnoreCase se podrá ingresar el nombre del usuario y la clave en mayúsculas o minúsculas.*/`
	`if (usuario.equalsIgnoreCase('mtorres') &&`
	`  clave.equalsIgnoreCase('abc'))`
	`  JOptionPane.showMessageDialog(null,'Bienvenido usuario');`
	`else`
	`  JOptionPane.showMessageDialog(null,'Error de usuario');`

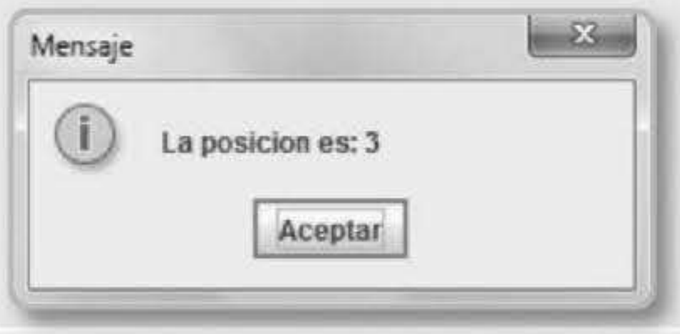
Método	Explicación

getChars	Copia caracteres de una cadena a un arreglo de caracteres. Por ejemplo, veamos el código para copiar una cadena a otra.
	`/* Iniciamos por declarar la variable 'vocales' inicializada con las vocales en minúsculas, luego se declara la variable cadena con el valor inicial AEIOU */`
	`char vocales[]={'a','e','i','o','u'};`
	`String cadena='AEIOU';`
	`/* Ahora usamos el método getChars en donde a la variable 'vocales' se le agrega los caracteres de la variable cadena desde la posición 1 hasta 3.*/`
	`cadena.getChars(1, 3, vocales, 1);`
	`/* Finalmente, se imprimen los caracteres de la variable vocales; para acceder a los caracteres se usa la estructura for */`
	`for(int i=0;i<5;i++)`
	`  JOptionPane.showMessageDialog(null, vocales[i]);`

hashCode	Retorna el código HASH para una cadena especificada.
	En el siguiente ejemplo se determina el número HASH asignado a la cadena **Programación Java**.
	`String cadena='Programación Java';`
	`int hash=cadena.hashCode();`
	`JOptionPane.showMessageDialog(null, 'El código HASH es: '+hash);`

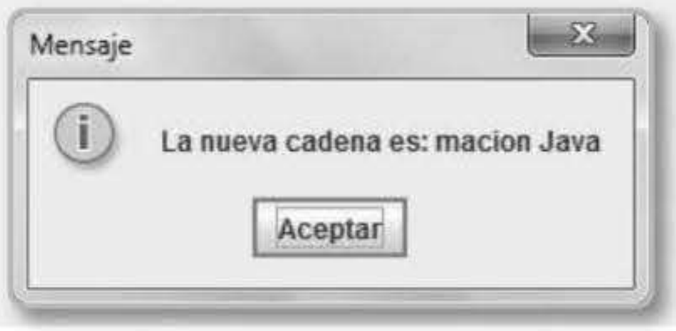
indexOf	Retorna la posición según el número de carácter ASCII incluido en una determinada cadena. Retornará -1 cuando el código buscado no se encuentra en la cadena.
	En el ejemplo que se muestra a continuación se determina la posición del carácter ASCII 97 (a minúsculas).
	`String cadena='Java';`
	`int valor=cadena.indexOf(97);`
	`JOptionPane.showMessageDialog(null, 'La posición es: '+valor);`
	 

Método	Explicación

lastIndexOf	Retorna la posición del carácter ASCII empezando por el final de la cadena. Retornará -1 cuando el código buscado no se encuentra en la cadena.
	
	En el siguiente ejemplo se determina la posición del carácter ASCII empezando por el final de la cadena.
	
	`String cadena='Java';`
	`int valor=cadena.lastIndexOf(97);`
	`JOptionPane.showMessageDialog(null, 'La posición es: '+valor);`
	
	 

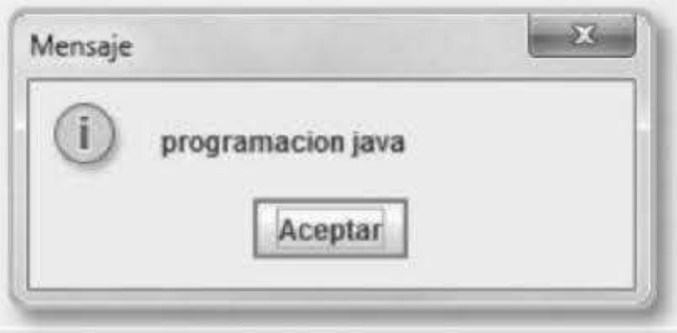
length	Retorna el número total de caracteres de una determinada cadena incluyendo espacios en blanco.
	
	En el ejemplo que se muestra a continuación se determina la longitud de la cadena **PROGRAMACIÓN JAVA**.
	
	`String cadena='PROGRAMACION JAVA';`
	`int longitud=cadena.length();`
	`JOptionPane.showMessageDialog(null, 'La longitud es: ' +`
	`longitud);`
	
	Resp.: 17 caracteres

replace	Reemplaza un carácter por otro dentro de una cadena.
	
	En el siguiente ejemplo se reemplazan todos los caracteres 'e' por la 'a'.
	
	`String cadena='Jeve';`
	`cadena=cadena.replace('e','a');`
	`JOptionPane.showMessageDialog(null, 'La nueva cadena es: ' +`
	`cadena);`
	
	Resp.: Java

subString	Permite retornar una parte de una cadena especificando la posición de la captura.
	
	En el ejemplo que se muestra a continuación se substrae la cadena **Programación Java** desde la posición 6.
	
	`String cadena='Programación Java';`
	`String nuevacadena=cadena.substring(6);`
	`JOptionPane.showMessageDialog(null, 'La nueva cadena es:' +`
	`nuevacadena);`
	
	 

Método	Explicación

toCharArray	Convierte el **String** en arreglo de caracteres. Por ejemplo, si se realiza la impresión de los caracteres de una cadena, entonces:
	`/* Iniciamos por declarar la variable cadena con el valor predeterminado 'Programación Java', luego este se envía a la variable arreglo con el método toCharArray */`
	`String cadena='Programación Java';`
	`char arreglo[]=cadena.toCharArray();`
	`/* Luego se imprimen los caracteres contenidos dentro del arreglo usando la estructura repetitiva for */`
	`for(int i=0;i<cadena.length();i++)`
	`  JOptionPane.showMessageDialog(null,arreglo[i]);`

toLowerCase	Permite convertir una cadena en minúsculas.
	En el siguiente ejemplo se convierte la cadena **PROGRAMACIÓN JAVA**.
	`String cadena='PROGRAMACIÓN JAVA';`
	`JOptionPane.showMessageDialog(null,cadena.toLowerCase());`
	 

toUpperCase	Permite convertir una cadena en mayúsculas.
	En el ejemplo que se muestra a continuación se convierte la cadena **programación java**.
	`String cadena='programación java';`
	`JOptionPane.showMessageDialog(null,cadena.toUpperCase());`

trim	Permite eliminar los espacios en blanco al final de una cadena de caracteres.
	En el siguiente ejemplo se permite eliminar los espacios en blanco contenidos dentro de la cadena **programación java**.
	`String cadena='programación java ';`
	`JOptionPane.showMessageDialog(null,cadena.trim());`

valueOf	Permite retornar la representación de la cadena.
	En el ejemplo que se muestra a continuación se tiene un cuadro combinado (Combobox) con 4 países predeterminados; al seleccionar uno de ellos, se devuelve el nombre del país seleccionado desde el Combobox.
	`String pais;`
	`pais=String.valueOf(cboPais.getSelectedItem());`
	`JOptionPane.showMessageDialog(null, 'El País seleccionado es: '`
	`  + pais);`

2.13 Manejo de la clase JOptionPane

Cuando se desarrolla aplicaciones Java, siempre se tendrá la necesidad de interactuar la aplicación con el usuario; esto se realiza mediante la interfaz implementada por el programador. En la tabla 2.6 se verán las diferentes formas que tiene la aplicación para responder al usuario, ya sea mediante un mensaje o solicitud de algún dato para la aplicación.

La clase **JOptionPane** permite:

- enviar mensajes informativos al usuario;
- mostrar mensajes de error;
- mostrar mensajes de advertencia;
- enviar solicitudes de selección;
- solicitar datos de entrada para la aplicación.

Para que funcionen los mensajes de una aplicación se necesita la librería javax.swing.JOptionPane.

Tabla 2.6 Métodos de la clase **JOptionPane**.

Método	Especificación y ejemplo

showMessageDialog	Este método permite mostrar un mensaje personalizado por el usuario y un botón de forma predeterminada. El método tiene como argumentos:
	`JOptionPane.showMessageDialog(Destino, 'Mensaje', 'Titulo',Icono)`
	■ **Destino del diálogo.** Este determina en dónde se va a colocar el cuadro de diálogo; se le asigna null para que pueda aparecer en el centro de la pantalla.
	■ **Mensaje.** Es el texto que se mostrará al usuario.
	■ **Título.** Es el texto mostrado en la barra de título del mensaje.
	■ **Ícono.** Define qué tipo de ícono se mostrará al usuario.
	Ejemplo 1. Mostrar el mensaje "Bienvenido a Java" mediante el cuadro de diálogo **JOptionPane**.
	`JOptionPane.showMessageDialog(null,'Bienvenido a Java');`

Ejemplo 2. En el siguiente código se muestra el uso del **JOptionPane** para mostrar un mensaje de salida y el valor de una variable.

```
int n=10;
JOptionPane.showMessageDialog(null, 'El valor de N es: '+n);
```

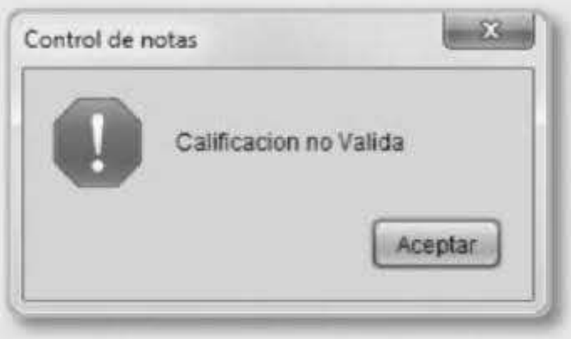
Para concatenar el texto a una variable se usa el operador +. En el siguiente código se usa el **JOptionPane** que muestra el ícono **Warning**.

```
double sueldo=1000;
JOptionPane.showMessageDialog(null, 'El sueldo es: '+sueldo,
'Control de Pagos', JOptionPane.WARNING_MESSAGE);
```

showMessageDialog

Ejemplo 3. En el siguiente código se muestra un mensaje de error al usuario cuando se valida el valor de una calificación.

```
byte calificación=-10;
if (calificación<0 || calificación>20)
JOptionPane.showMessageDialog(null, 'Calificación no Valida',
'Control de notas', JOptionPane.ERROR_MESSAGE);
```



Ejemplo 4. En el siguiente código se muestran los datos de un empleado en un solo mensaje, haciendo un cambio de línea por cada dato.

```
String cliente='Fernanda Torres L.';
String DNI='47521522';
double sueldo=2500.75;
JOptionPane.showMessageDialog(null,'Cliente: '+cliente+'\n'
+'DNI: '+DNI+'\n' +'Sueldo: '+sueldo, 'Control de Personal',
JOptionPane.INFORMATION_MESSAGE);
```

showOptionDialog	Este método permite mostrar un cuadro de diálogo, en el cual se podrá tener un mejor control de los botones presentados. Cuenta con los siguientes parámetros:
	`JOptionPane.showOptionDialog(Destino, 'Mensaje', 'Titulo',TipoOpcion, TipoMensaje,Icono, Opciones,valorInicial)`
	■ **Destino.** Aquí se asigna el contenedor del cuadro de diálogo; en forma predeterminada se le asigna null.
	■ **Mensaje.** Es el texto que visualizará el usuario.
	■ **Título.** Es el texto que se mostrará como título de la ventana de diálogo.
	■ **TipoOpcion.** Aquí se indica qué tipo de opción visualizará el usuario, como, por ejemplo, DEFAULTOPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION u OK_CANCEL_OPTION.
	■ **TipoMensaje.** Aquí se indica qué tipo de mensaje se mostrará al usuario; desde aquí se determina qué tipo de ícono está asignado al mensaje, como, por ejemplo, ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE o PLAIN_MESSAGE.
	■ **Ícono.** Aquí se determina el ícono a mostrar en el cuadro de diálogo; el valor predeterminado es null.
	■ **Opciones.** El valor predeterminado es null, el cual permite definir los textos de los botones según el TipoOpcion. En otros casos se puede implementar un array de objetos.
	■ **valorInicial.** Es la selección por defecto, la cual define que uno de los objetos del cuadro de diálogo tiene el foco activo.
	
	El método **showOptionDialog** devuelve un valor entero que determinará qué tipo de botón ha seleccionado el usuario. Así, por ejemplo, el botón **SI** tiene el valor 0, el botón **No** tiene el valor 1; y si no se selecciona ningún botón, el valor es -1.
	
	Ejemplo 1. Mostrar un mensaje de salida al usuario con valores predeterminados, dicho mensaje solicitará al usuario la salida segura del sistema. En caso se seleccione **SI**, se tendrá que salir de la aplicación; en caso contrario, se seguirá en la aplicación.
	`int r-JOptionPane.showOptionDialog(this, 'Esta seguro de salir...?','Sistema de Ventas',JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE,null,null,null);`
	`if (r--0) System.exit(0);`

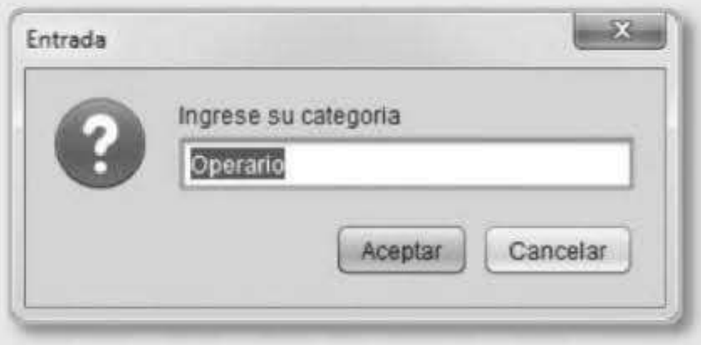
showOptionDialog	**Ejemplo 2.** Mostrar un mensaje de salida al usuario con valores personalizados en los botones y predeterminando al botón de salida.
	`int r-JOptionPane.showOptionDialog(this,'Esta seguro de salir...?', 'Sistema de Ventas', JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE,null,new Object[ ] { 'si salgo', 'no salgo'}, 'si salgo');`
	`if (r--0) System.exit(0);`

showInputDialog	Este método solicita un valor de tipo cadena al usuario, este dato debe ser capturado mediante una variable o atributo. Presenta el siguiente formato:
	`JOptionPane.showInputDialog(Object mensaje)`
	Ejemplo 1. Mostrar un cuadro de diálogo que solicite el nombre de un determinado empleado de forma básica.
	`String nombre - JOptionPane.showInputDialog('Ingrese su nombre');`
	`JOptionPane.showMessageDialog(null, 'El nombre ingresado es: '+nombre);`
	
	El método **showInputDialog** permite el ingreso de un valor; en este caso, el nombre es de tipo **String**, así que no necesita conversión. La siguiente ventana de diálogo muestra el valor capturado a partir del diálogo anterior.

showInputDialog

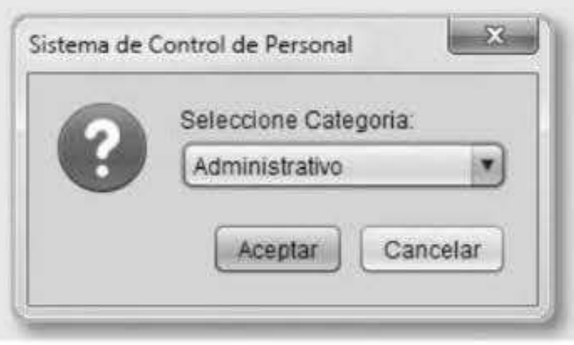
Ejemplo 2. Mostrar un cuadro de diálogo que solicite el ingreso de una categoría al empleado, mostrando de forma predeterminada el valor **Operario**.

```
String categoria - JOptionPane.showInputDialog('Ingrese su
                                    categoría','Operario');
JOptionPane.showMessageDialog(null,'La categoría ingresada
            es: '+categoría);
```



Ejemplo 3. Mostrar un cuadro de diálogo que permita seleccionar la categoría del empleado por medio de un cuadro combinado.

```
String categoriaSeleccionada - String.valueOf(JOptionPane.
                                    showInputDialog(null,
                                    'Seleccione Categoría:
                                    ', 'Sistema de
                                    Control de Personal',
                                    JOptionPane.QUESTION_
                                    MESSAGE, null, new
                                    Object[] {'Operario',
                                    'Administrativo',
                                    'Servicio'},
                                    'Administrativo'));
JOptionPane.showMessageDialog(null, 'La categoría
            seleccionada es: '+ categoriaSeleccionada);
```



2.14 La clase Math

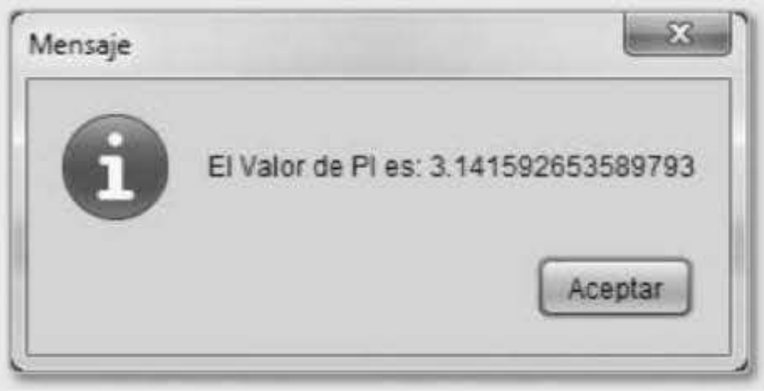
Pertenece a **java.lang.Math**, el cual contiene métodos estáticos que permiten realizar cálculos sobre funciones matemáticas en una aplicación Java. Esta clase es de visibilidad pública, es decir, se puede tener acceso a todos sus métodos, en cualquier parte de la aplicación. Se consideran los siguientes aspectos:

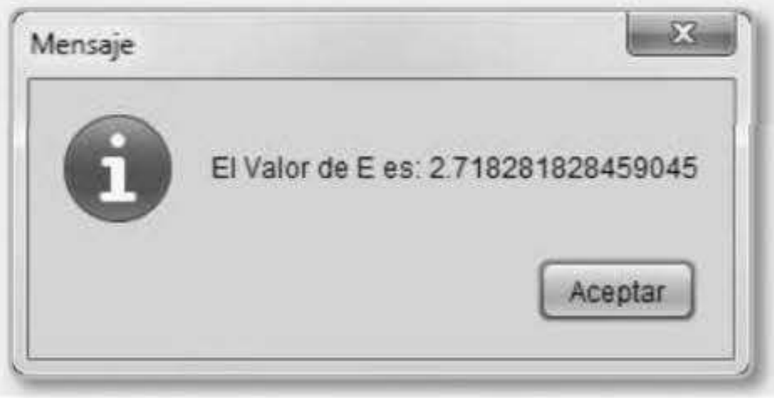
- Los métodos por ser estáticos no necesitan inicialización de valores.
- Los métodos seno (sin), coseno (cos) y tangente (tan) trabajan con ángulos radianes; por tanto, si son usados, tendrán que ser convertidos previamente.
- Un logaritmo neperiano (Ln) tiene la siguiente expresión en Java: **Math.log(x)**.
- Una exponenciación neperiana expresada matemáticamente como e^x tiene la siguiente expresión en Java: **Math.exp(x)**, donde *e* tiene un valor de 2.7182...

2.14.1 Métodos que representan a PI y a E

La clase **Math** hace referencia, especialmente, a estas dos constantes más usadas en funciones matemáticas, asumiendo una precisión de 15 decimales.

Método	Descripción

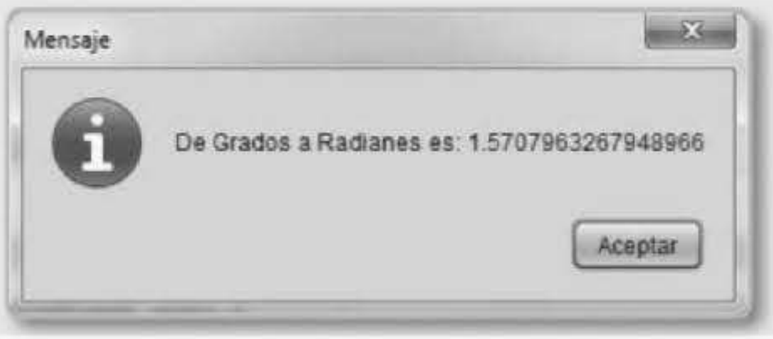
PI	Es un método que imprime el valor **PI** predeterminado en la clase **Math** de Java.
	`double PI=Math.PI;`
	`JOptionPane.showMessageDialog(null,'El Valor de PI es: '+PI);`
	 

e	Es un método que imprime el valor **e** predeterminado en la clase **Math** de Java.
	`double e=Math.E;`
	`JOptionPane.showMessageDialog(null,'El Valor de E es: '+e);`
	 

2.14.2 Métodos de conversión entre grados y radianes

Se hará referencia a la conversión entre grados y radianes.

Método	Descripción

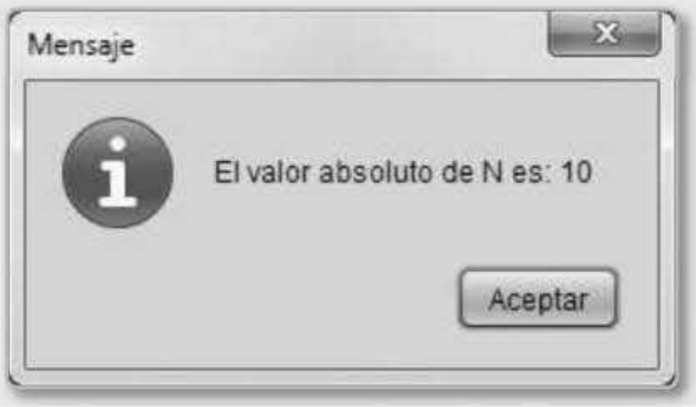
De grados a radianes	`double grados = 90;`
	`double radianes = Math.toRadians(grados);`
	`JOptionPane.showMessageDialog(null, 'De Grados a Radianes es: ' + radianes);`
	 

De radianes a grados	`double radianes = 1.5707963267948966;`
	`double grados = Math.toDegrees(radianes);`
	`JOptionPane.showMessageDialog(null,'De Radianes a Grados es: ' + grados);`

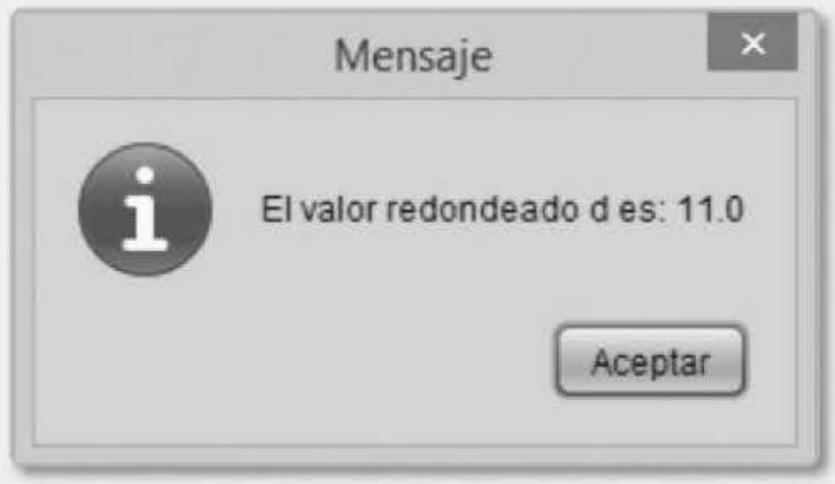
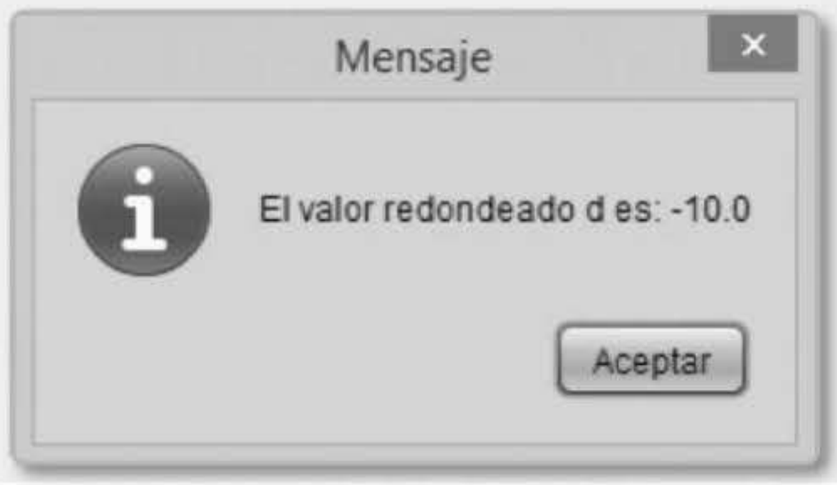
2.14.3 Métodos de la clase Math

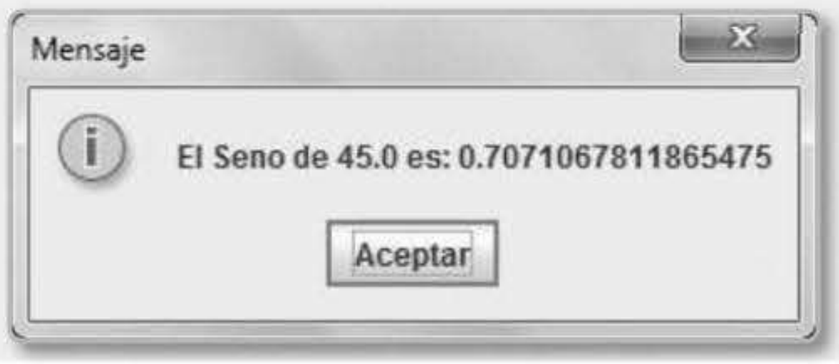
Contiene un conjunto de clases que permiten gestionar los valores numéricos de una aplicación. Se exponen, a continuación, una lista de clases implementadas de forma predeterminada:

Método	Descripción

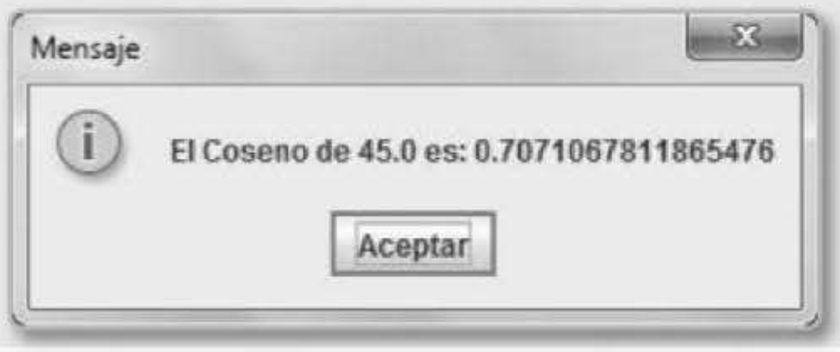
abs	Determina el valor absoluto de un número. Por ejemplo, el siguiente código determina el valor absoluto del valor -10.
	`int n = -10;`
	`JOptionPane.showMessageDialog(null, 'El valor absoluto de N es: ' + Math.abs(n));`
	 

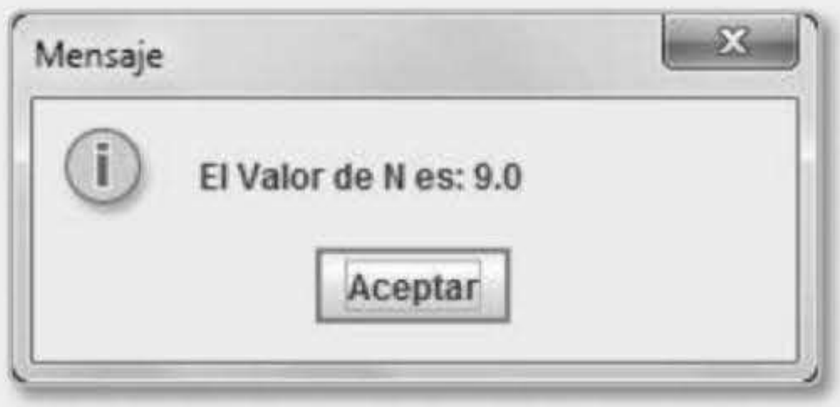
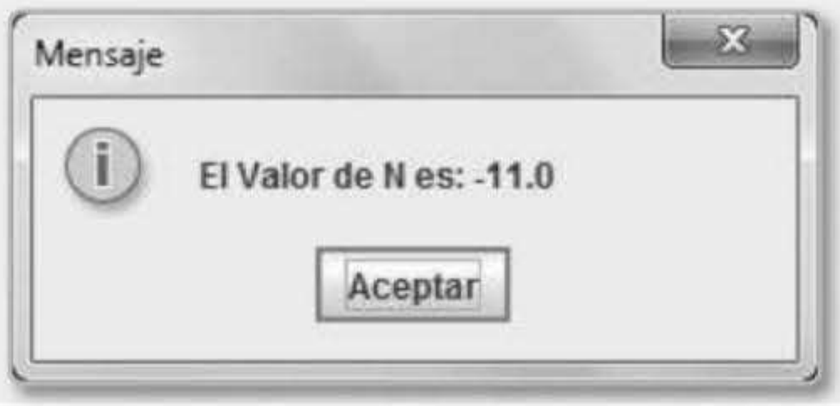
Método	Descripción

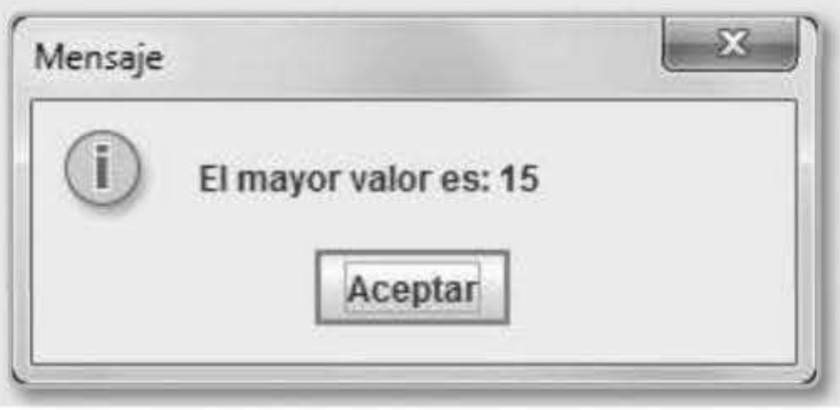
ceil	Permite redondear un valor decimal al entero más pequeño, no menor a sí mismo. Por ejemplo, el siguiente código determina el valor redondeado de 10.254.
	`double d = 10.254;`
	`JOptionPane.showMessageDialog(null, 'El valor redondeado d es: ' + Math.ceil(d));`
	 
	
	Asimismo, se probará con un valor negativo -10.95:
	`double d = -10.95;`
	`JOptionPane.showMessageDialog(null, 'El valor redondeado d es: ' + Math.ceil(d));`
	 

sin	Permite determinar el seno trigonométrico de un ángulo expresado en radianes. Por ejemplo, el siguiente código permite determinar el seno de 45°.
	`double grados = 45;`
	`double radianes = Math.toRadians(grados);`
	`JOptionPane.showMessageDialog(null, 'El Seno de '+grados +' es: ' + Math.sin(radianes));`
	 

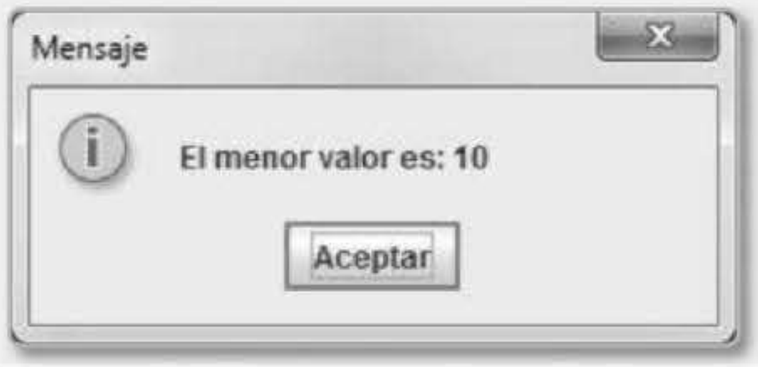
Método	Descripción

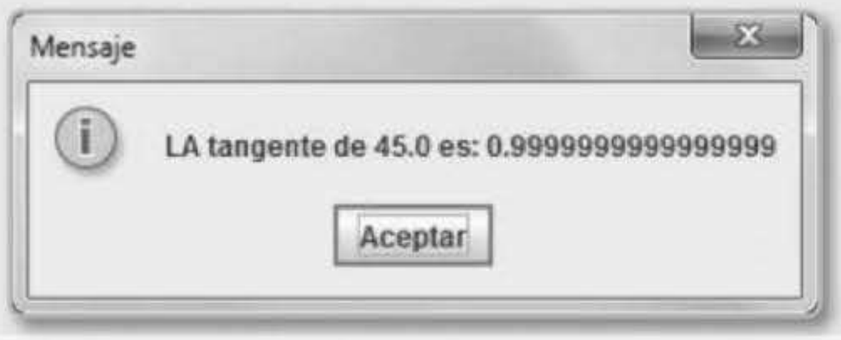
cos	Permite determinar el coseno trigonométrico de un ángulo expresado en radianes. Por ejemplo, el siguiente código permite calcular el coseno de 45°.
	`double grados = 45;`
	`double radianes = Math.toRadians(grados);`
	`JOptionPane.showMessageDialog(null, 'El Coseno de '+grados +' es: ' + Math.cos(radianes));`
	 

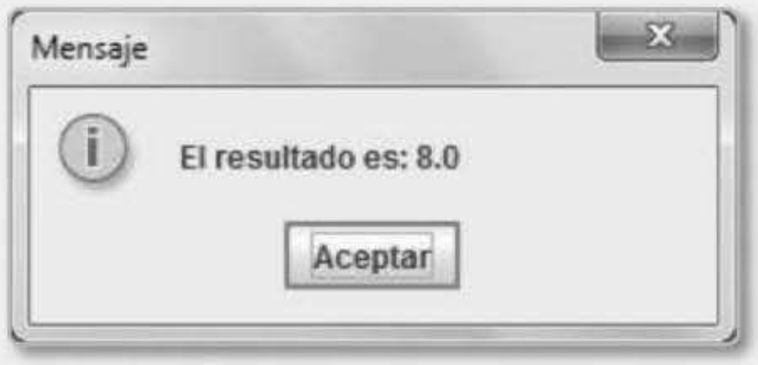
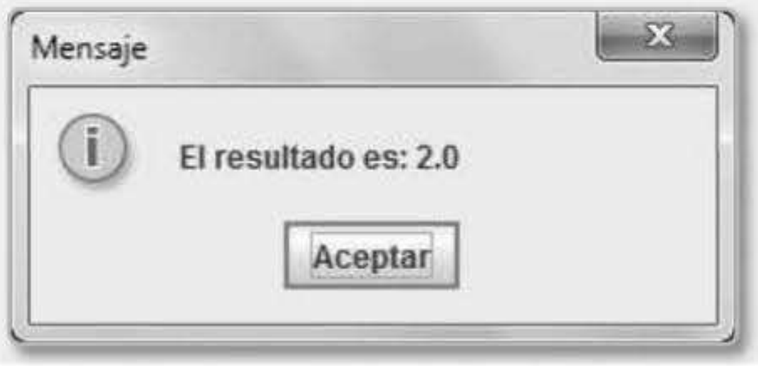
floor	Permite redondear un valor decimal al entero más grande no mayor a sí mismo. Por ejemplo, el siguiente código permite determinar el valor redondeado de 9.2.
	`double n=9.2;`
	`JOptionPane.showMessageDialog(null, 'El Valor de N es: ' + Math.floor(n));`
	 
	
	`double n=-10.5;`
	`JOptionPane.showMessageDialog(null, 'El Valor de N es: ' + Math.floor(n));`
	 

max	Permite determinar el máximo valor entre dos valores numéricos. Por ejemplo, el siguiente código determina el mayor valor entre 10 y 15.
	`int a=10;`
	`int b=15;`
	`JOptionPane.showMessageDialog(null, 'El mayor valor es: ' + Math.max(a,b));`
	 

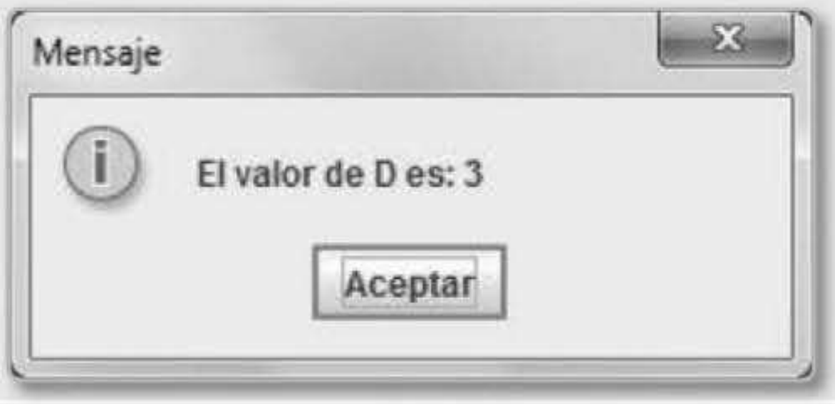
Método	Descripción

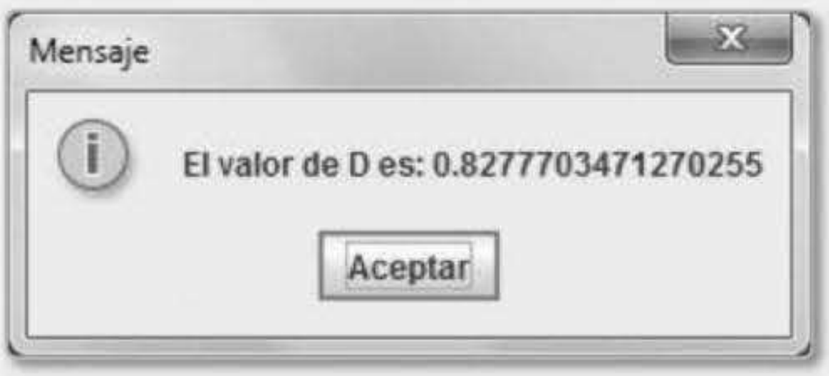
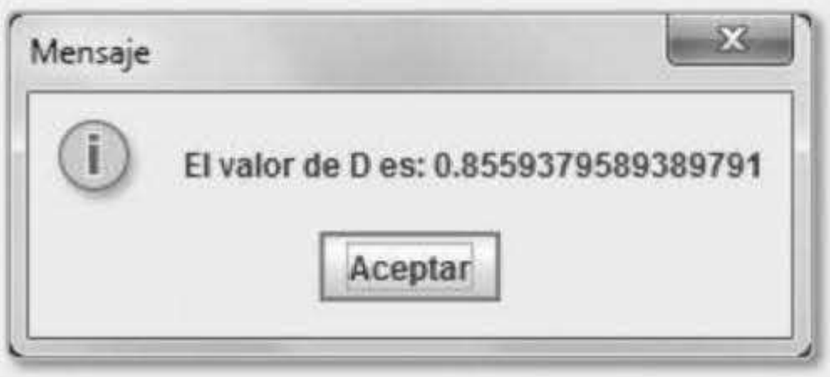
min	Permite determinar el mínimo valor entre dos valores numéricos. Por ejemplo, el siguiente código determina el menor valor entre 10 y 15.
	`int a=10;`
	`int b=15;`
	`JOptionPane.showMessageDialog(null, 'El menor valor es: ' +`
	`Math.min(a,b));`
	 

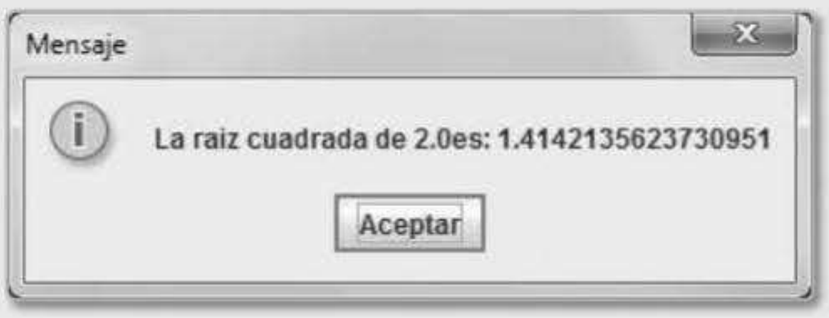
tan	Permite determinar la tangente trigonométrica de un ángulo expresado en radianes. Por ejemplo, el siguiente código determina la tangente de 45°.
	`double grados = 45;`
	`double radianes = Math.toRadians(grados);`
	`JOptionPane.showMessageDialog(null, 'La tangente de '+grados +' es: '`
	`+ Math.tan(radianes));`
	 

pow	Permite determinar la potencia *n* de un valor numérico entero. Por ejemplo, el siguiente código determina la potencia 3 del valor 2.
	`int base=2;`
	`int potencia=3;`
	`JOptionPane.showMessageDialog(null, 'El resultado es: ' +`
	`Math.pow(base,potencia));`
	 
	Asimismo, se probará con la raíz cubica de 8:
	`int base=8;`
	`JOptionPane.showMessageDialog(null, 'El resultado es: ' +`
	`Math.pow(base,(1/3.0)));`
	 

Método	Descripción

round	Permite expresar un número real con un número específico de decimales aplicando el exceso y defecto matemático. Por ejemplo, el siguiente código aplica el redondeo a cero decimales del valor 2.55.
	`double d=2.55;`
	`JOptionPane.showMessageDialog(null, 'El valor de D es: '+ Math.round(d));`
	 

random	Permite devolver un número aleatorio entre 0 y 1 excluyendo a estos valores. Por ejemplo, el siguiente código determina un número aleatorio cualquiera.
	`double d=Math.random();`
	`JOptionPane.showMessageDialog(null, 'El valor de D es: '+d);`
	 
	 

sqrt	Permite devolver la raíz cuadrada de un número entero. Por ejemplo, el siguiente código permite determinar la raíz cuadrada del valor 2.
	`double d=2;`
	`JOptionPane.showMessageDialog(null, 'La raíz cuadrada de '+d+'es: ' + Math.sqrt(d));`
	 

CASO DESARROLLADO 1: **Modelo de cotización bancaria (programación estructurada)**

El área de préstamos de un banco necesita una aplicación que permita simular una cotización de préstamo bancario y que, a través de los datos de la cotización, pueda mostrar a cuánto ascenderá la cuota inicial que debe depositar el cliente y la cuota mensual a pagar según el plazo en meses seleccionado por el cliente.

Consideraciones:

- Desarrolle la aplicación usando la metodología estructurada.
- Use el siguiente diseño de interfaz gráfica:

- Use los tipos de moneda, soles y dólares, en un cuadro combinado.
- En un cuadro combinado, use como tipo de bien: un automóvil, una vivienda y otros.
- Ingrese un número entero que represente el plazo en meses a solicitud del cliente.
- Ingrese el valor del bien para realizar los cálculos.
- Como regla de negocio se tiene que, para calcular la cuota inicial, aplicar el 20% sobre el valor del bien. Asimismo, se debe aplicar una tasa de interés del 12% sobre el monto del riesgo neto del préstamo.
- El monto del riesgo neto resulta de la diferencia entre el valor del bien y la cuota inicial.
- La cuota mensual resulta de añadir el 12% sobre el riesgo neto y dividirlo entre la cantidad especificada en el plazo en meses.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjMetodologia**.
3. Agregue el paquete **pFormularios**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmEstructurado**.

5. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

6. Agregue código al botón **Ver resultados de la simulación**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
package pFormularios;

public class frmEstructurado extends javax.swing.JFrame {

    /** Creates new form frmEstructurado */
    public frmEstructurado() {
        initComponents();

        //Llenando el tipo de moneda
        cboMoneda.addItem('SOLES');
        cboMoneda.addItem('DOLARES');

        //Llenando el tipo de bien
        cboBien.addItem('Automovil');
        cboBien.addItem('Vivienda');
        cboBien.addItem('Otro');
    }

    private void btnResultadosActionPerformed(java.awt.event.ActionEvent evt) {
        //Obtener los valores del formulario
        String moneda = String.valueOf(cboMoneda.getSelectedItem());
        String simbolo = moneda.equals('SOLES')?'S/':'$';

        String bien = String.valueOf(cboBien.getSelectedItem());
        int meses = Integer.parseInt(txtMeses.getText());
        double tasa = Double.parseDouble(lblTasa.getText());
        double inicial = Double.parseDouble(lblInicial.getText());
        double valor = Double.parseDouble(txtValor.getText());

        //Realizando los cálculos
        double cuotaInicial = valor * 0.2;
        double riesgo = valor-cuotaInicial;
        double cuotaMensual = (riesgo * 1.12 ) / meses
```

```
        //Imprimiendo los resultados
        lblCuotaInicial.setText(simbolo+String.
                        format('%.2f',cuotaInicial));
        lblRiesgo.setText(simbolo+String.format('%.2f',riesgo));
        lblCuotaMensual.setText(simbolo+String.
                        format('%.2f',cuotaMensual));
    }
}
```

7. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 2: **Modelo de cotización bancaria (programación modular)**

El área de préstamos de un banco necesita una aplicación que permita simular una cotización de préstamo bancario y que, a través de los datos de la cotización, pueda mostrar a cuánto ascenderá la cuota inicial que debe depositar el cliente y la cuota mensual a pagar según el plazo en meses seleccionado por el cliente.

Consideraciones:

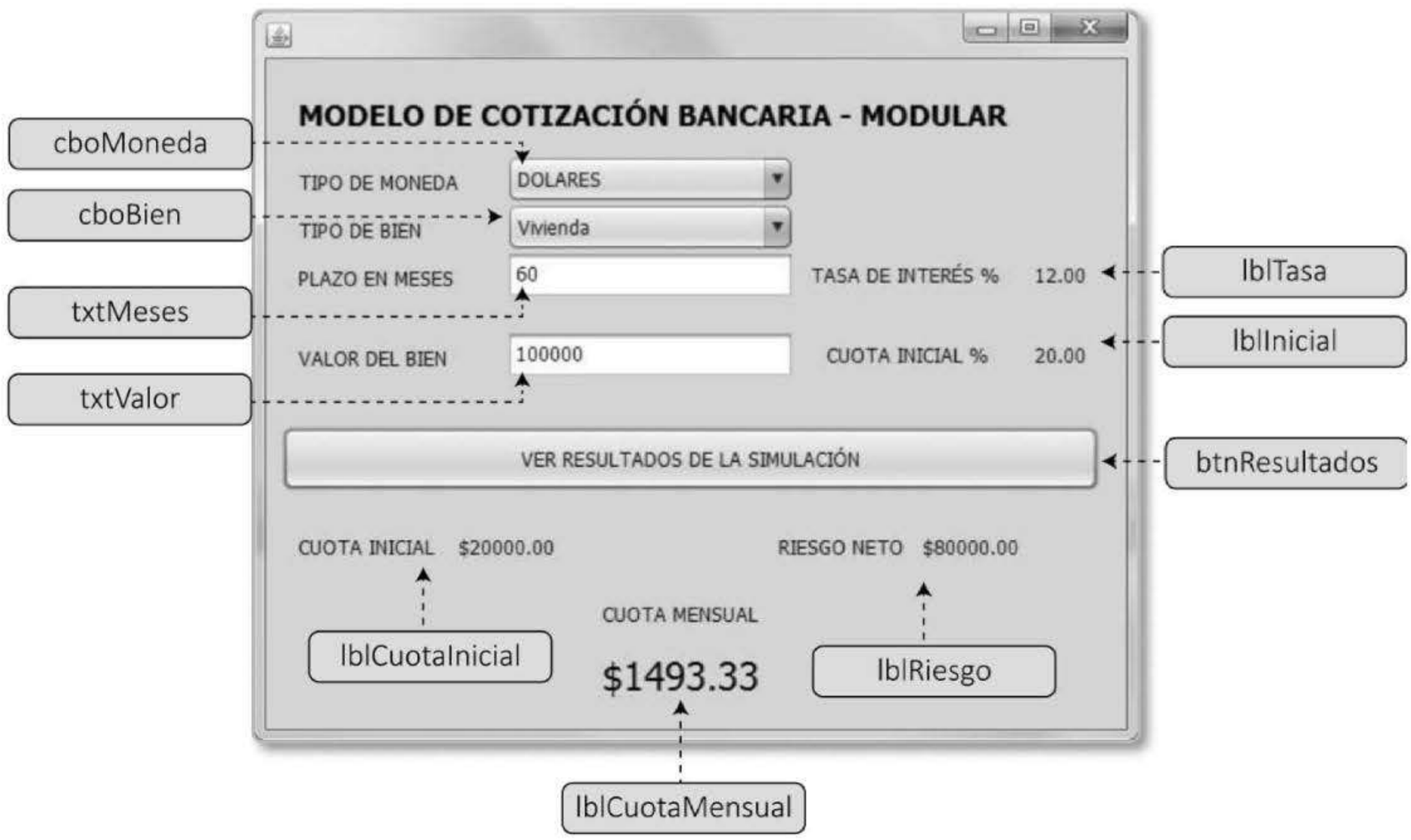
- La aplicación será desarrollada usando la metodología modular.
- Use el siguiente diseño de interfaz gráfica:

- Use los tipos de moneda, soles y dólares, en un cuadro combinado.
- En un cuadro combinado, use como tipo de bien: un automóvil, una vivienda y otros.

- Ingrese un número entero que represente el plazo en meses a solicitud del cliente.
- Ingrese el valor del bien para realizar los cálculos.
- Como regla de negocio se tiene que, para calcular la cuota inicial, aplicar el 20% sobre el valor del bien. Asimismo, se debe aplicar la tasa de interés del 12% sobre el monto del riesgo neto del préstamo.
- El monto del riesgo neto resulta de la diferencia entre el valor del bien y la cuota inicial.
- La cuota mensual resulta de añadir el 12% sobre el riesgo neto y dividirlo entre la cantidad especificada en el plazo en meses.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjMetodologia**.
3. Agregue el paquete **pFormularios**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmModular**.
5. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:



6. Agregue código al botón **Ver resultados de la simulación**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
package pFormularios;

public class frmModular extends javax.swing.JFrame {
    /** Creates new form frmEstructurado */
    public frmModular() {
        initComponents();
        //Llenando el tipo de moneda
        llenaTipoMoneda();
        //Llenando el tipo de bien
        llenaTipoBien();
    }

    //Modulo para llenar el tipo de moneda
    void llenaTipoMoneda(){
        cboMoneda.addItem('SOLES');
        cboMoneda.addItem('DOLARES');
    }

    //Modulo para llenar el tipo de bien
    void llenaTipoBien(){
        cboBien.addItem('Automovil');
        cboBien.addItem('Vivienda');
        cboBien.addItem('Otro');
    }

    private void btnResultadosActionPerformed(java.awt.event.ActionEvent
evt) {
        //Obtener los valores del formulario
        String moneda = getMoneda();
        String bien = getBien();
        int meses = getMeses();
        double tasa = getTasa();
        double inicial = getInicial();
        double valor = getValor();

        //Realizando los cálculos
        double cuotaInicial = calculaCuotaInicial(valor);
        double riesgo = calculaRiesgo(valor, cuotaInicial);
        double cuotaMensual = calculaCuotaMensual(riesgo, meses);

        //Imprimiendo los resultados
        lblCuotaInicial.setText(moneda+String.format('%.2f',cuotaInicial));
        lblRiesgo.setText(moneda+String.format('%.2f',riesgo));
        lblCuotaMensual.setText(moneda+String.format('%.2f',cuotaMensual));
    }

    //Modulo que captura los datos
    String getMoneda(){
        String moneda = String.valueOf(cboMoneda.getSelectedItem());
        return moneda.equals('SOLES')?'S/':'$';
    }

    String getBien(){
        return String.valueOf(cboBien.getSelectedItem());
    }

    int getMeses(){
        return Integer.parseInt(txtMeses.getText());
    }
```

```
double getTasa(){
        return Double.parseDouble(lblTasa.getText());
    }

    double getInicial(){
        return Double.parseDouble(lblInicial.getText());
    }

    double getValor(){
        return Double.parseDouble(txtValor.getText());
    }
    //Modulo para los calculos

    double calculaCuotaInicial(double valor){
        return valor * 0.2;
    }

double calculaRiesgo(double valor, double cuotaInicial){
        return valor-cuotaInicial;
    }

    double calculaCuotaMensual(double riesgo, int meses){
        return (riesgo * 1.12 ) / meses;
    }
}
```

7. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 3: **Modelo de cotización bancaria (programación orientada a objetos)**

El área de préstamos de un banco necesita una aplicación que permita simular una cotización de préstamo bancario y que, a través de los datos de la cotización, pueda mostrar a cuánto ascenderá la cuota inicial que debe depositar el cliente y la cuota mensual a pagar según el plazo en meses seleccionado por el cliente.

Consideraciones:

- La aplicación será desarrollada usando la metodología orientada a objetos.
- Use el siguiente diseño de interfaz gráfica:

- Use los tipos de moneda, soles y dólares, en un cuadro combinado.
- En un cuadro combinado, use como tipo de bien: un automóvil, una vivienda y otros.
- Ingrese un número entero que represente el plazo en meses a solicitud del cliente.
- Ingrese el valor del bien para realizar los cálculos.
- Como regla de negocio se tiene que, para calcular la cuota inicial, aplicar el 20% sobre el valor del bien. Asimismo, se debe aplicar la tasa de interés del 12% sobre el monto del riesgo neto del préstamo.
- El monto riesgo neto resulta de la diferencia entre el valor del bien y la cuota inicial.
- La cuota mensual resulta de añadir el 12% sobre el riesgo neto y dividirlo entre la cantidad especificada en el plazo en meses.

CASO DESARROLLADO 3: **Modelo de cotización bancaria (programación orientada a objetos)**

El área de préstamos de un banco necesita una aplicación que permita simular una cotización de préstamo bancario y que, a través de los datos de la cotización, pueda mostrar a cuánto ascenderá la cuota inicial que debe depositar el cliente y la cuota mensual a pagar según el plazo en meses seleccionado por el cliente.

Consideraciones:

- La aplicación será desarrollada usando la metodología orientada a objetos.
- Use el siguiente diseño de interfaz gráfica:

- Use los tipos de moneda, soles y dólares, en un cuadro combinado.
- En un cuadro combinado, use como tipo de bien: un automóvil, una vivienda y otros.
- Ingrese un número entero que represente el plazo en meses a solicitud del cliente.
- Ingrese el valor del bien para realizar los cálculos.
- Como regla de negocio se tiene que, para calcular la cuota inicial, aplicar el 20% sobre el valor del bien. Asimismo, se debe aplicar la tasa de interés del 12% sobre el monto del riesgo neto del préstamo.
- El monto riesgo neto resulta de la diferencia entre el valor del bien y la cuota inicial.
- La cuota mensual resulta de añadir el 12% sobre el riesgo neto y dividirlo entre la cantidad especificada en el plazo en meses.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al Proyecto: **pjMetodologia**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmOrientadoObjetos**.
5. Agregue una clase llamada **Préstamo** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

7. En la clase **Préstamo,** coloque el siguiente código:

```
package pClases;

public class Prestamo {
    //Atributos
    private String moneda;
    private String bien;
    private int meses;
    private double tasa;
    private double inicial;
    private double valor;

    //Métodos GET y SET
    public String getMoneda() {
        return moneda.equals('SOLES')?'S/':'$';
    }
    public void setMoneda(String moneda) {
        this.moneda = moneda;
    }
```

```
    public String getBien() {
        return bien;
    }
    public void setBien(String bien) {
        this.bien = bien;
    }


    public int getMeses() {
        return meses;
    }
    public void setMeses(int meses) {
        this.meses = meses;
    }
    public double getTasa() {
        return tasa;
    }
    public void setTasa(double tasa) {
        this.tasa = tasa;
    }
    public double getInicial() {
        return inicial;
    }
    public void setInicial(double inicial) {
        this.inicial = inicial;
    }
    public double getValor() {
        return valor;
    }
    public void setValor(double valor) {
        this.valor = valor;
    }
    //Método que calcula la cuota inicial
    public double calculaCuotaInicial(){
        return valor * 0.2;
    }
    //Método que calcula el monto riesgo
    public double calculaRiesgo(){
        return valor-calculaCuotaInicial();
    }
    //Método que calcula la cuota mensual
    public double calculaCuotaMensual(){
        return (calculaRiesgo() * 1.12 ) / meses;
    }
}
```

8. Agregue código al botón **Ver resultados de la simulación**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
package pFormularios;
import pClases.Prestamo;

public class frmOrientadaObjetos extends javax.swing.JFrame {

    /** Creates new form frmEstructurado */
    public frmOrientadaObjetos() {
        initComponents();
        //Llenando el tipo de moneda
        llenaTipoMoneda();
        //Llenando el tipo de bien
        llenaTipoBien();
    }

    //Modulo para llenar el tipo de moneda
    void llenaTipoMoneda(){
        cboMoneda.addItem('SOLES');
        cboMoneda.addItem('DOLARES');
    }

    //Modulo para llenar el tipo de bien
    void llenaTipoBien(){
        cboBien.addItem('Automovil');
        cboBien.addItem('Vivienda');
        cboBien.addItem('Otro');
    }

    private void btnResultadosActionPerformed(java.awt.event.ActionEvent evt) {
        //Objeto de la clase Préstamo
        Préstamo objP = new Préstamo();

        //Obtener los valores del formulario
        objP.setMoneda(String.valueOf(cboMoneda.getSelectedItem()));
        objP.setBien(String.valueOf(cboBien.getSelectedItem()));
        objP.setMeses(Integer.parseInt(txtMeses.getText()));
        objP.setTasa(Double.parseDouble(lblTasa.getText()));
        objP.setInicial(Double.parseDouble(lblInicial.getText()));
        objP.setValor(Double.parseDouble(txtValor.getText()));

        //Realizando los cálculos
        double cuotaInicial = objP.calculaCuotaInicial();
        double riesgo = objP.calculaRiesgo();
        double cuotaMensual = objP.calculaCuotaMensual();

        //Imprimiendo los resultados
        lblCuotaInicial.setText(objP.getMoneda()+String.
                                format('%.2f',cuotaInicial));
        lblRiesgo.setText(objP.getMoneda()+String.format('%.2f',riesgo));
        lblCuotaMensual.setText(objP.getMoneda()+String.
                                format('%.2f',cuotaMensual));
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CAPÍTULO

3

Clase Swing

Competencias

- Reconoce las clases que componen la colección **Swing**.
- Identifica el uso de las propiedades y métodos de clases contenidas en la colección **Swing**.

Contenido

3.1 Generalidades
3.2 Clases **Swing**
3.3 Clase **JFrame**
3.4 Clase **JLabel**
3.5 Clase **JTextField**
3.6 Clase **JTextArea**
3.7 Clase **JPasswordField**
3.8 Clase **Button**
3.9 Clase **JCheckBox**
3.10 Clase **JRadioButton**
3.11 Clase **JComboBox**
3.12 Clase **DefaultListModel**
3.13 Clase **JList**
3.14 Clase **JMenuBar**
3.15 Clase **JMenuItem**
3.16 Clase **Desktop Pane**

- Caso desarrollado 1: Manejo de la clase **JList**
- Caso desarrollado 2: Manejo de la clase **JComboBox**
- Caso desarrollado 3: Manejo de la clase **JMenuBar**

3.1 Generalidades

Cuando se desarrolla una aplicación Java, se piensa en diferentes tipos, como las aplicaciones de escritorio o plataforma, las aplicaciones móviles o las aplicaciones web; en todos los casos siempre se diseña una interfaz gráfica que permite modelar gráficamente un proceso de negocio por medio de controles visuales.

Inicialmente, en Java, el diseño se realizaba con las clases AWT (*Abstract Windows ToolKit*), que fueron usadas desde la primera versión de Java hasta el Java SE-2, el cual permitía implementar interfaces gráficas mediante controles, como botones, textos estáticos, cuadros de texto, etc., invocados desde el paquete **java.awt**. Luego, con la evolución del lenguaje, nace **Swing** con una propuesta evolucionada sobre el AWT; esta nueva versión forma parte de la JFC (*Java Foundation Class*), la cual contiene clases modernas para el desarrollo de aplicaciones visuales. No obstante, se recomienda que no se combinen clases de AWT con Swing en una misma aplicación; a pesar de que ambas clases son muy parecidas, no es una buena práctica integrarlas. Entre los elementos que diferencian a Swing de AWT se tienen los árboles, los tooltips, la barra de progreso, etc.

Por otro lado, la *interfaz gráfica de usuario*, más conocida como GUI, permite mostrar de forma ergonómica una aplicación. Aquí, para el desarrollo de nuestras aplicaciones, usaremos las clases contenidas en el paquete **javax.swing**.

3.2 Clases Swing

La programación basada en GUI determina el uso de un conjunto de clases que serán usadas en una aplicación Java y que desde allí se llamarán componentes u objetos. **Swing** ofrece un conjunto de clases, interfaces, recursos y otros elementos para la implementación de aplicaciones gráficas en Java. Su aparición se da con la versión Java 2 y, hoy por hoy, sigue en uso; además, los JDK actuales ya vienen incorporados con el paquete **Swing**.

- Para habilitar el uso de las clases contenidas en el paquete **Swing** se usa el siguiente código, pero se debe tener en cuenta que el símbolo asterisco representa a todas las clases dentro del paquete:

```
import javax.swing.*;
```

- También se puede acceder a los métodos o atributos de la clase directamente con el siguiente código:

```
import javax.swing.JFrame;
```

Hay que considerar que los framework para Java ya incorporan de forma visual las clases contenidas en el paquete **Swing**. A continuación, se muestra la estructura de clases que contiene el paquete **Swing** de Java:

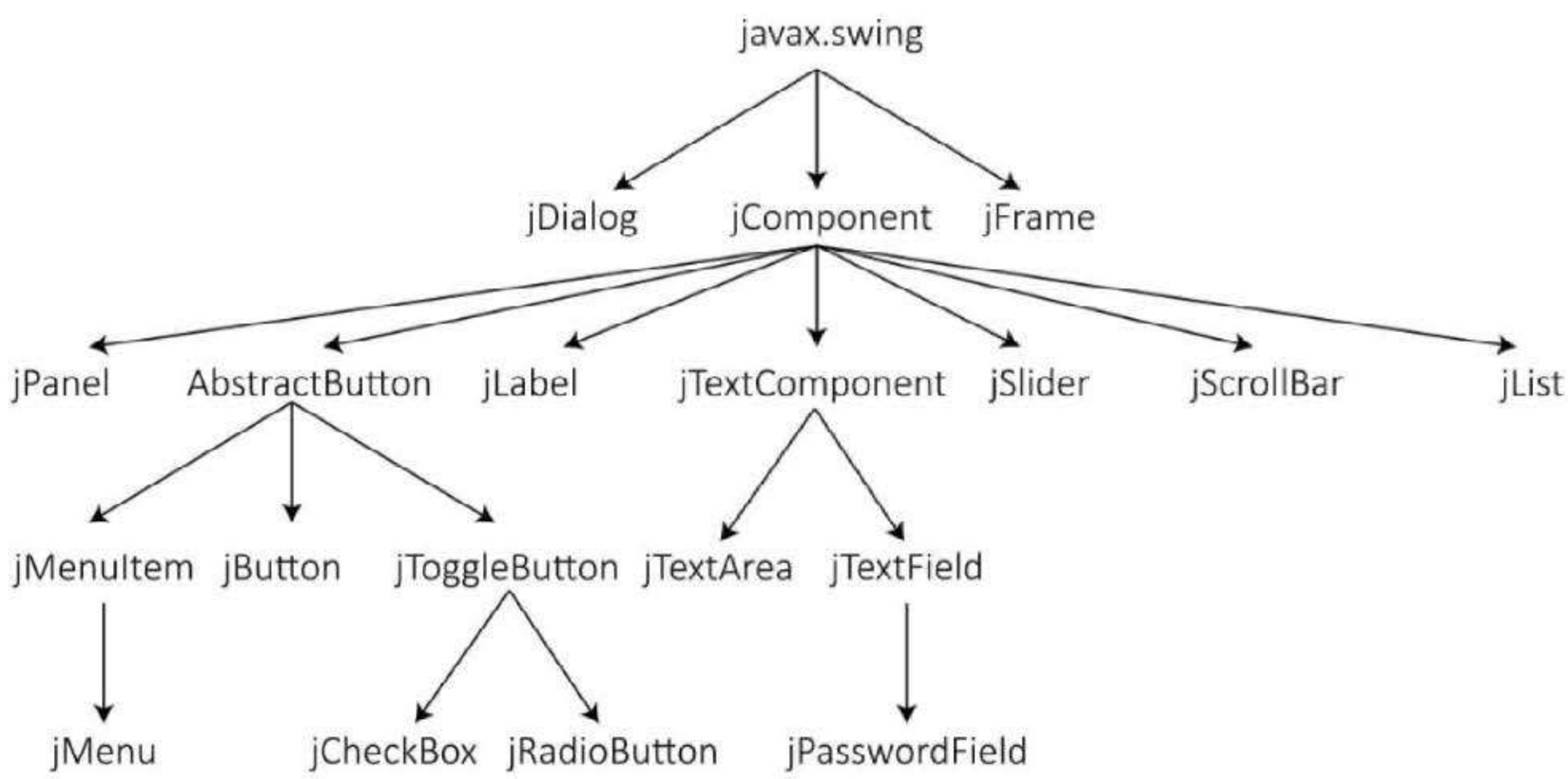


Figura 3.1 Estructura del paquete Swing de Java.
Fuente: <https://geekytheory.com/tutorial-14-java-swing-interfaces-graficas>

Para el caso de NetBeans las clases se muestran en el panel **Pallete**, tal como se observa en la siguiente imagen:

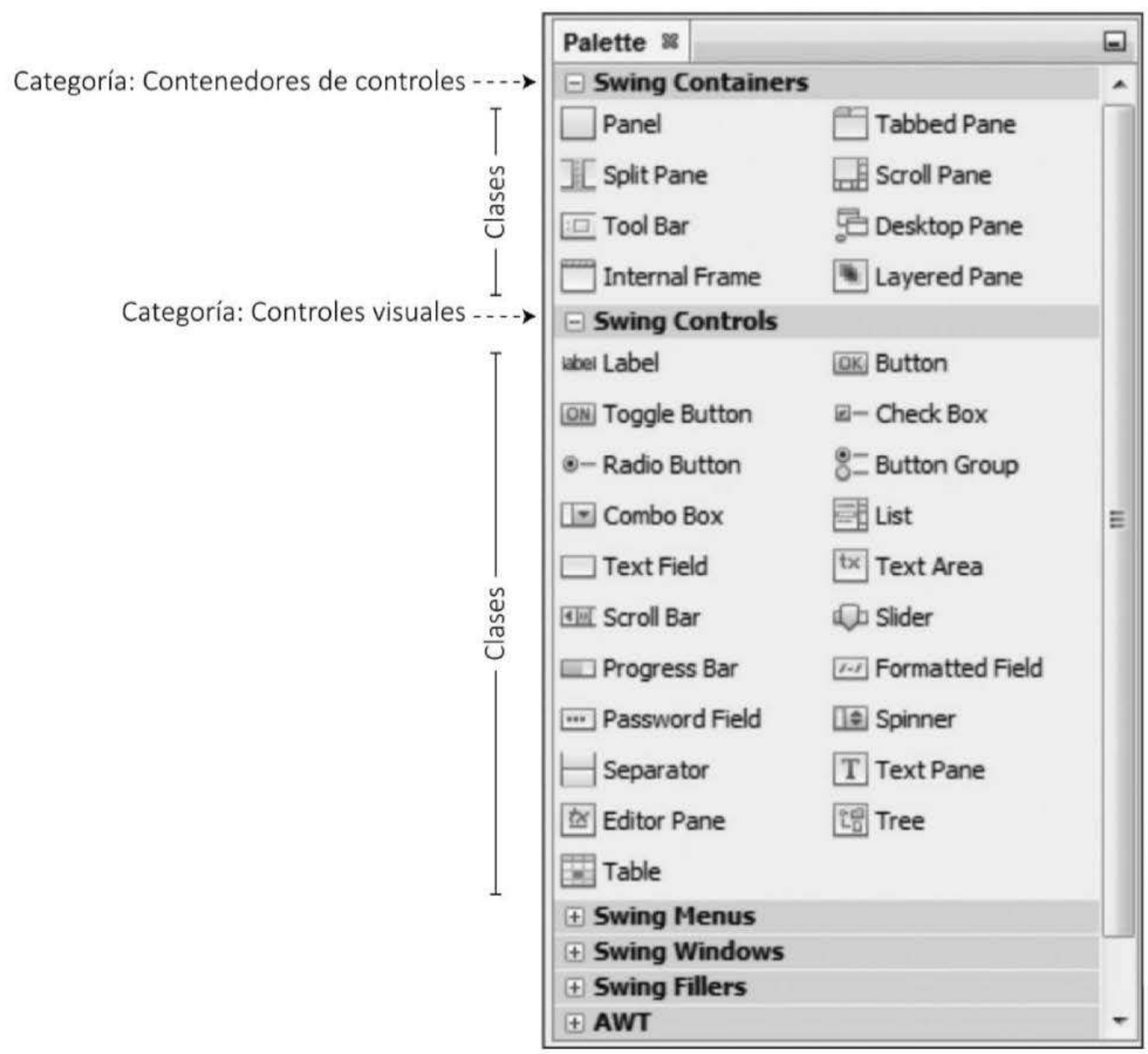


Figura 3.2 Paleta de controles de NetBeans.

Para el caso de Eclipse Mars las clases se muestran también dentro de **Pallete**, y la variación se encuentra en los nombres que presentan las categorías:

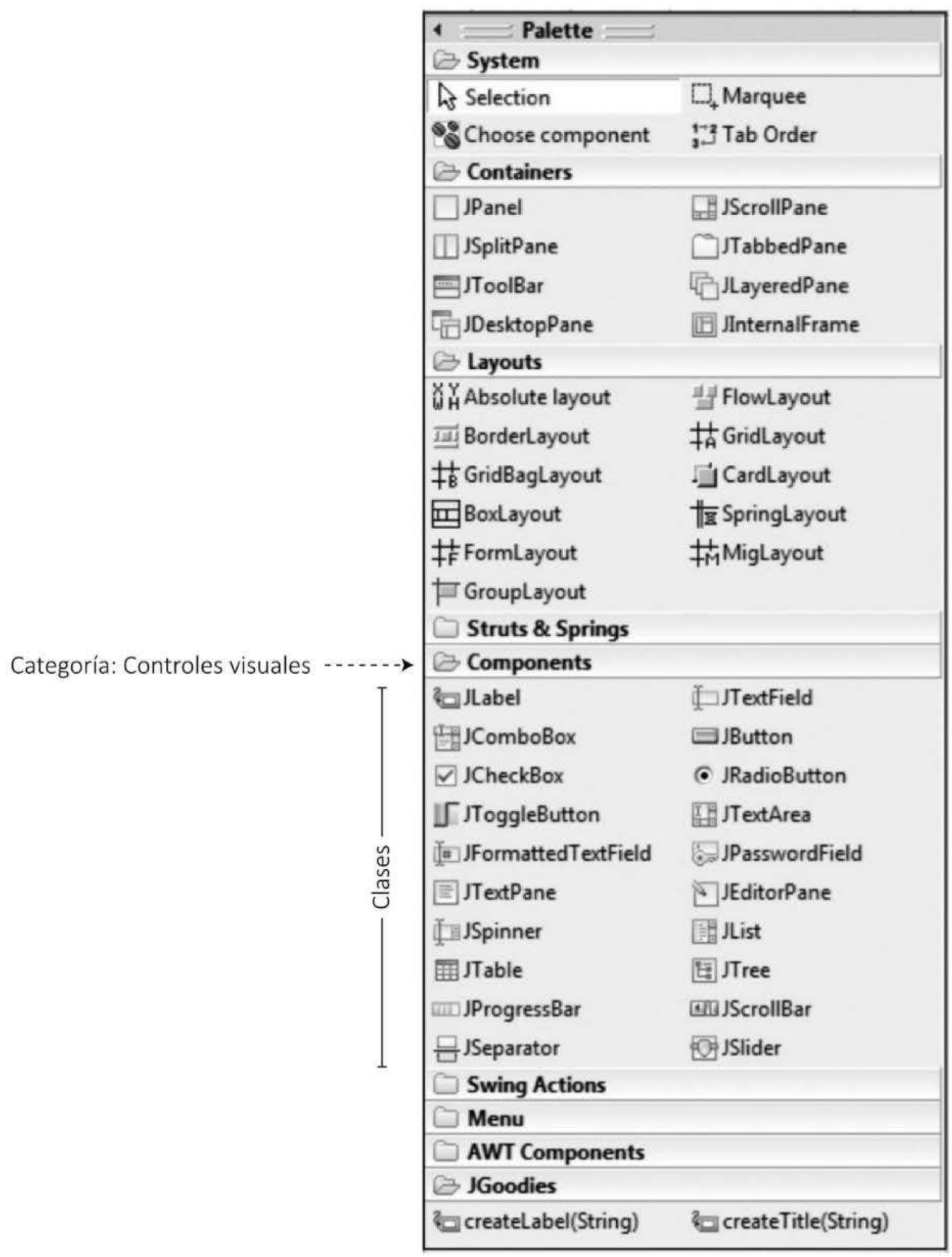


Figura 3.3 Paleta de controles de Eclipse Mars.

3.3 La clase JFrame

Esta clase encapsula una ventana contenedora. El aspecto inicial que mostrará será dependiendo del sistema operativo desde donde se esté ejecutando la aplicación.

Figura 3.4 JFrame con NetBeans.

Figura 3.5 JFrame con Eclipse Mars.

Como se puede visualizar, el **JFrame** en NetBeans no muestra la barra de título en tiempo de diseño, mientras que Eclipse Mars lo muestra vacío. Sin embargo, en ambos casos, al ejecutar la aplicación se mostrará dicha barra.

Principales propiedades

Propiedad	Descripción	Script

title	Permite definir un texto dentro de la barra de título de la ventana.	`/* Definir el título de la ventana como 'Control de pagos' a nivel de código */`
		`frmVentana.setTitle('Control de Pagos');`
		`//Usando el constructor de la clase JFrame`
		`JFrame frmVentana - new JFrame('Control de Pagos');`

background	Define el color de fondo de la ventana.	`/* Definir el color de fondo rojo */` `setBackground(new java.awt.Color(255,0,0));`
resizeable	Permite modificar el alto y ancho del formulario en tiempo de ejecución.	`/* Habilitar la modificación de alto y ancho del formulario*/` `setResizable(true);`

Principales métodos

setBounds(x,y,ancho,alto)	Permite posicionar el objeto **JFrame** dentro de la ventana. Por ejemplo, se puede definir la posición inicial 0,0 con un alto de 23 px y 90 px, con el siguiente código: `frmVentana.setBounds(0, 0, 90, 23);`
setLayout()	Permite determinar la forma de administrar los elementos contenidos dentro del control **JFrame**. Cuando se agrega un control en el **JFrame**, este deben ser independiente de la ventana; este efecto se nota cuando movemos de lugar un control dentro del formulario y los demás controles también se intentan mover. Para evitar dicho evento en NetBeans se debe tener en cuenta los siguientes pasos: ■ Haga clic sobre el fondo del formulario. ■ Desde la paleta **Properties** seleccione la ficha **Code**. ■ Seleccione **Generate Resize Code** desde la opción **Form Size Policy**.

	O también, se puede usar el siguiente código: `frmVentana.setLayout(null);`
setVisible()	Permite mostrar u ocultar el control **JFrame**. Esto puede ser usado para ocultar un **JFrame** y mostrar otro de acuerdo a algún evento dentro de la aplicación. Para ocultar el JFrame se puede usar el siguiente código: frmVentana.setVisible(false); y para hacerlo visible, este otro código: frmVentana.setVisible(true).

3.4 La clase JLabel

Esta clase permite incorporar un control de texto estático no modificable dentro de un **JFrame**. Tiene como objetivo mostrar información de una sola línea de texto o una imagen.

Principales propiedades

Propiedad	Descripción	Script

text	Define el contenido del control **JLabel**.	`/*Asignar el texto 'Sistema de Control de Productos'*/`
		`lblTitulo.setText('Sistema de Control de Productos');`
		`/*También se puede asignar un título al control JLabel usado el siguiente código: */`
		`JLabel lblTitulo = new JLabel('Sistema de Control de productos');`

foreground	Asigna un color al texto contenido en el control **JLabel**.	`/*Asignar el color azul al texto*/` `lblTitulo.setForeground(new java.awt.Color(0,0, 255));`
font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JLabel**.	`/*Configurar el tipo de letra Tahoma con estilo negrita a un tamaño de 20 puntos al control JLabel que muestra el título del JFrame, podemos usar el siguiente código: */` `lblTitulo.setFont(new Font('Trebuchet MS', Font.BOLD, 20));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JLabel**.	`/*Asignar el mensaje 'Productos' al JLabel título*/` `lblTitulo.setToolTipText('Hola');`
border	Define un tipo de borde al control **JLabel**.	`/*Asignar un borde de línea sólida de color negro al JLabel título */` `lblTitulo.setBorder(javax.swing.BorderFactory.createLineBorder(new java.awt.Color(0, 0, 0)));`

Propiedad	Descripción	Script

Icon	Define una imagen dentro del control **JLabel**.	`/*Asignar una imagen al JLabel lblFoto*/`
		`JLabel lblFoto = new JLabel(new ImageIcon('imagen1.jpg'));`
		Si la imagen tiene como extensión **gif**, al ejecutar la aplicación se podrá visualizar la animación. También se podría usar el siguiente código:
		`ImageIcon imagen = new ImageIcon('imagen1.jpg');`
		`lblFoto.setIcon(imagen);`

Principales métodos

getText()	Permite devolver el contenido del objeto **JLabel**. Por ejemplo, si se cuenta con un control llamado **lblN**, el cual muestra un valor de 1000, y se necesita capturar dicho valor, se puede usar el siguiente código: `int numero=Integer.parseInt(lblN.getText());`

setText("mensaje")	Permite asignar un texto al control **JLabel**. Por ejemplo, si se quiere asignar el título de impresión de un valor, se podría usar el siguiente código:
	`double precio = 2490;`
	`lblPrecio.setText('El precio es: '+precio);`

Casos de prueba

CONTROL DE VENTA DE PRODUCTOS SISTEMA DE VENTAS	El control **JLabel** permite definir los títulos dentro de una aplicación.
SUBTOTAL $ 1500.00 DESCUENTO $ 150.00 NETO A PAGAR $ 1350.00	El control **JLabel** permite mostrar los valores resultantes de una aplicación. Hay que tener en cuenta que el método **setText** define la información que se muestra en un control **JLabel**.

3.5 La clase JTextField

Esta clase permite incorporar un control de campo de texto en una sola línea, el cual permite ingresar un valor haciendo interactuar al usuario con la aplicación. Hay que tener en cuenta que se puede ingresar cualquier tipo de información por el control **JTextField** y que el programador deberá aplicar conversiones necesarias a los datos.

Principales propiedades

Propiedad	Descripción	Script
text	Define el contenido del control **JTextField**.	`/*Limpiar el texto contenido en el control txtDescripcion*/` `txtDescripcion.setText('');`

foreground	Asigna un color al texto contenido en el control **JTextField**.	`/*Asignar el color azul al texto contenido en el control txtDescripcion*/`
		`txtDescripcion.setForeground(`
		`new java.awt.Color(0,0, 255));`

font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JTextField**.	`/*Configurar un tamaño de 20 puntos al control txtDescripcion, podemos usar el siguiente código: */` `txtDescripcion.setFont(new Font('Tahoma',0, 20));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JTextField**.	`/*Asignar el mensaje 'Ingrese descripción del producto' al control txtDescripcion*/` `txtDescripcion.setToolTipText('Ingrese descripción del producto');`

border	Define un tipo de borde al control **JTextField**.	`/*Asignar un título tipo marco al control txtDescripcion que indique 'Descripción del producto' */`
		■ Seleccione el control **txtDescripcion**.
		■ Seleccione la propiedad **border**.
		■ Seleccione **Titled Border**.
		■ Coloque un texto descriptivo en **Title**.
		`txtDescripcion.setBorder(javax.swing.`
		`BorderFactory.`
		`createTitledBorder('Descripción de producto'));`
		Descripción de producto

Principales métodos

getText()	Permite devolver el contenido del objeto **JTextField**. Por ejemplo, si necesitamos obtener el valor ingresado en la caja de texto **txtCantidad**, se puede usar el siguiente código: `int cantidad - Integer.parseInt(txtCantidad.getText());`

setHorizontalAlignment()	Permite alinear el texto contenido dentro del control **JTextField**. Por ejemplo, el siguiente código nos permite alinear al centro, izquierda y derecha de la caja de texto **txtCantidad**.
	`txtCantidad.setHorizontalAlignment(JTextField.CENTER);`
	`txtCantidad.setHorizontalAlignment(JTextField.LEFT);`
	`txtCantidad.setHorizontalAlignment(JTextField.RIGHT);`

setEditable	Permite bloquear la edición de texto contenido en un control **JTextField**. Por ejemplo, si contamos con una caja de texto que contenga el monto **IVA** llamada **txtIVA** y debemos bloquear su contenido, se puede usar el siguiente código: `txtIVA.setEditable(false);`
setText(String texto)	Permite asignar un texto al control **JTextField**. Por ejemplo, para realizar la impresión de un valor resultante en el control **txtPromedio**, se puede usar el siguiente código: `double promedio = 10.50;` `txtPromedio.setText('El promedio es: '+promedio);`

Casos de prueba

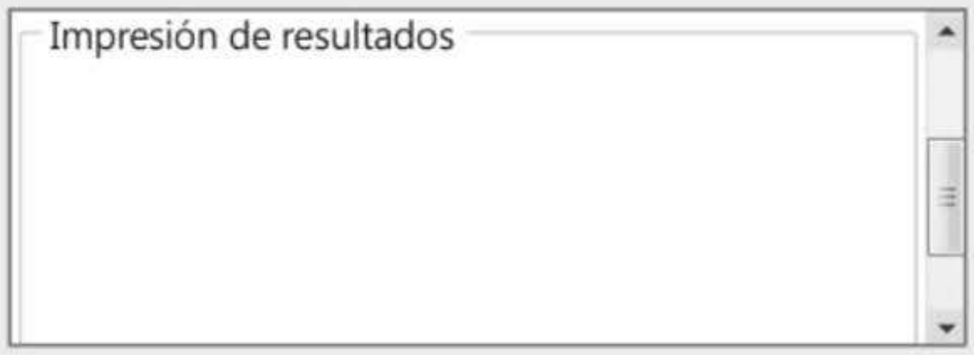
INGRESE MONTO	El control **JTextField** define el valor de ingreso que va a tener el usuario frente a la aplicación. No hay que olvidar que estos controles siempre deben tener un nombre y por nomenclatura deben iniciar en **txt**. En este caso lo llamaremos **txtMonto**.
INGRESE MONTO	Se puede definir un título al control **JTextField**, y así se evita colocar un **JLabel** por cada **JTextField**.

3.6 La clase JTextArea

Esta clase permite incorporar un control de campo de texto de líneas múltiples, el cual permite ingresar muchos valores textuales dentro del mismo control. Su uso se relaciona a impresiones de respuestas o informes.

Principales propiedades

Propiedad	Descripción	Script
text	Define el contenido del control **JTextArea**.	`/*Limpiar el texto contenido en el control txtImpresion*/` `txtImpresion.setText('');`
foreground	Asigna un color al texto contenido en el control **JTextArea**.	`/*Asignar el color rojo al texto contenido en el control txtImpresion */` `txtImpresion.setForeground(new java.awt.Color(255,0, 0));`
font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JTextArea**.	`/*Configurar un tamaño de 20 puntos al control txtImpresion, podemos usar el siguiente código: */` `txtImpresion.setFont(new Font('Tahoma',0, 20));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JTextArea**.	`/*Asignar el mensaje 'Mostrando valores resultantes' al control txtImpresion*/` `txtImpresion.setToolTipText('Mostrando valores resultantes');`

border	Define un tipo de borde al control **JTextArea**.	`/*Asignar un título tipo marco al control txtImpresion que indique 'Impresión de resultados' */`
		■ Seleccione el control **txtImpresion**.
		■ Seleccione la propiedad **border**.
		■ Seleccione **Titled Border**.
		■ Coloque un texto descriptivo en **Title**.
		`txtImpresion.setBorder(javax.swing.BorderFactory.createTitledBorder('Impresión de resultados'));`
		 

Principales métodos

append()	Permite añadir una línea de texto al final del control **JTextArea**. Por ejemplo, si se necesita hacer la impresión de una venta de productos a un determinado cliente como se muestra en la siguiente imagen:
	Impresión de resultados
	RESUMEN DE VENTA
	CLIENTE: Fernanda Torres Lázaro
	---------- 	
	SUBTOTAL: 2000.0
	DESCUENTO: 200.0
	NETO: 1800.0
	Entonces, el código sería de la siguiente manera:
	`String cliente-'Fernanda Torres Lázaro';`
	`double subtotal-2000;`
	`double descuento-0.1*subtotal;`
	`double neto - subtotal -descuento;`
	`txtImpresion.append('**RESUMEN DE VENTA**');`
	`txtImpresion.append('\nCLIENTE: '+cliente);`
	`txtImpresion.append('\n----------');`
	`txtImpresion.append('\nSUBTOTAL: '+subtotal);`
	`txtImpresion.append('\nDESCUENTO: '+descuento);`
	`txtImpresion.append('\nNETO: '+neto);`

getText()	Permite devolver el contenido del objeto **JTextArea**. Por ejemplo, si se necesita obtener todo el contenido de la caja de impresión, se podría usar el siguiente código: `String mensaje-txtImpresion.getText());`
setText(String texto)	Permite asignar un texto al control **JtextArea**. Si el control contiene algún texto, este se eliminará y mostrará la nueva asignación; asimismo, no permite la impresión línea debajo en el mismo control. Por ejemplo, si se quiere imprimir el título **RESUMEN DE VENTA** en el control **txtImpresion**, se puede usar el siguiente código: `txtImpresion.setText('** RESUMEN DE VENTA**');`
setBackground()	Permite asignar un color al fondo del control **JTextArea**. Por ejemplo, se puede asignar el color rojo de fondo al control **txtImpresion**. `txtImpresion.setBackground(new Color(255,00,00));`

Casos de prueba

`** RESUMEN DE VENTA **`
`----------`
`SUBTOTAL $1500.00`
`DESCUENTO $150.00`
`NETO A PAGAR $1350.00`
	El control **JTextArea** muestra el resumen de los valores resultantes de la aplicación; se recomienda combinar el método **setText** y **append**.
	`txtSalida.text('**RESUMEN DE VENTA**');`
	`txtSalida.append('\nSUBTOTAL: '+subtotal);`
	`txtSalida.append('\nDESCUENTO: ' +`
	`descuento);`
	`txtSalida.append('\nNETO A PAGAR: '+neto);`

3.7 La Clase JPasswordField

Esta clase permite incorporar un control de campo de texto tipo *password*, es decir, los caracteres ingresados se enmascaran mostrando símbolos.

Principales propiedades

Propiedad	Descripción	Script
text	Define el contenido del control **JPasswordField**.	`/*Limpiar el texto contenido en el control txtClave*/` `txtClave.setText('');`
font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JPasswordField**.	`/*Configurar un tamaño de 20 puntos al control txtClave, podemos usar el siguiente código: */` `txtClave.setFont(new Font('Tahoma',0, 20));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JPasswordField**.	`/*Asignar el mensaje 'Ingrese su contraseña' al control txtClave*/` `txtClave.setToolTipText('Ingrese contraseña');`

border	Define un tipo de borde al control **JPasswordField**.	`/*Asignar un titulo tipo marco al control txtClave que indique 'Ingrese contraseña' */`
		■ Seleccione el control **txtImpresion**.
		■ Seleccione la propiedad **border**.
		■ Seleccione **Titled Border**.
		■ Coloque un texto descriptivo en **Title**.
		`txtClave.setBorder(javax.swing.BorderFactory.createTitledBorder('Ingrese contraseña'));`
		Ingrese contraseña

Principales métodos

getText()	Permite devolver el contenido del objeto **JPasswordField**. Por ejemplo, el siguiente código obtiene la clave registrada en el control txtClave: `String clave - txtClave.getText();`
setEchoChar	Permite establecer un carácter al control **JPasswordField**, que por defecto es asterisco (*). Por ejemplo, el siguiente código asigna el carácter **#** como símbolo del control **txtClave**. `txtClave.setEchoChar('#');`

Casos de prueba

Ingrese contraseña	El control **JPasswordField** permitirá enmascarar la información que se ingrese. Hay que tener en cuenta que los valores ingresados podrán ser capturados o calculados como si se tratase de un control **JTextField**.

3.8 La clase JButton

Esta clase permite incorporar un control de tipo *button*, es decir, el usuario podrá ejecutar un proceso mediante el uso de botones. Hay que tener en cuenta que los botones deben estar incluidos dentro de la lista de eventos **ActionListener** de Java.

Principales propiedades

Propiedad	Descripción	Script
text	Define el contenido del control **JButton**.	`/*Asignar el texto 'Procesar' al control btnProcesar*/` `btnProcesar.setText('PROCESAR');`

Icon	Permite asignar una imagen al contenido del control **JButton**.	`/*Asignar una imagen al botón btnProcesar*/`
		PROCESAR
		■ Seleccione el botón desde el JFrame.
		■ Seleccione la propiedad **ICON**.
		■ Seleccione el botón **Import to Project...**
		■ Busque y seleccione su imagen.
		■ Siga la ruta **Next > Finish > Ok**.
		`btnProcesar.setIcon(new javax.swing.ImageIcon(getClass().getResource('/Brightness.gif')));`

font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JButton**.	`/*Configurar un tamaño de 20 puntos al texto mostrado en el control btnProcesar, podemos usar el siguiente código: */` `btnProcesar.setFont(new Font('Tahoma',0, 20));`
foreground	Asigna un color al texto contenido en el control **JButton**.	`/* Asignar el color rojo al texto contenido en el control btnProcesar */` `btnProcesar.setForeground(new java.awt.Color(255,0, 0));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JButton**.	`/*Asignar el mensaje 'Procesando' al control btnProcesar*/` `btnProcesar.setToolTipText('Procesando');`

Principales métodos

AddActionListener()	Permite agregar una acción de evento al **JButton**. Por ejemplo, para activar el clic sobre el botón **btnProcesar**, se usa el siguiente código:
	`btnProcesar.addActionListener(this);`
	`public void actionPerformed( ActionEvent e ){`
	`  if(e.getSource()--btnProcesar){`
	`  //Colocar el código que ejecutara el botón`
	`  }`
	`}`

setVisible()	Permite mostrar u ocultar el control **JButton**. Por ejemplo, se puede ocultar el botón, por algún evento dentro de la aplicación, usando el siguiente código:
	`btnProcesar.setVisible(false)`
	Y para mostrar nuevamente el botón, se usará el siguiente código:
	`btnProcesar.setVisible(true).`

Casos de prueba

CALCULAR	El control **JButton** permite gestionar los cálculos dentro de una aplicación y, asimismo, mostrar los resultados. Para esto, los controles del formulario deben estar bien definidos.
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z TODOS	También se puede usar como una botonera dentro de la aplicación. Por ejemplo, al seleccionar un botón, se deben mostrar los productos por la inicial de la letra seleccionada.

3.9 La clase JCheckBox

Esta clase permite incorporar un control de tipo cuadro de chequeo, en la cual un usuario podrá seleccionar una o más opciones *Check*.

Principales propiedades

Propiedad	Descripción	Script
text	Define el contenido del control **JCheckBox**.	`/*Asignar el texto 'Acepto los términos' al control chkOpcion*/` `chkOpcion.setText('Acepto los términos');`

Propiedad	Descripción	Script
font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JCheckBox**.	`/*Configurar un tamaño de 20 puntos al texto mostrado en el control chkOpcion, podemos usar el siguiente código: */` `chkOpcion.setFont(new Font('Tahoma',0, 20));`
foreground	Asigna un color al texto contenido en el control **JCheckBox**.	`/*Asignar el color rojo al texto contenido en el control chkOpcion */` `chkOpcion.setForeground(new java.awt.Color(255,0, 0));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JCheckBox**.	`/*Asignar el mensaje 'Seleccione opción' al control chkOpcion*/` `chkOpcion.setToolTipText('Seleccione opción');`
buttonGroup	Define que un conjunto de controles **JCheckBox** se encuentren agrupados.	`/*Asignar el control chkOpcion al grupo btgReglas */` `btgReglas.add(chkOpcion);`
selected	Define que el control **JCheckBox** aparezca previamente seleccionado.	`/*Asignar el control chkOpcion como un check seleccionado */` `chkOpcion.setSelected(true);`

Principales métodos

AddItemListener()	Permite agregar una acción de evento al **JCheckBox**. Por ejemplo, si se quiere seleccionar el **chkOpcion** para que realice una actividad en la aplicación, se puede utilizar el siguiente código:
	`chkOpcion.addItemListener(this);`
	`public void itemStateChanged( ItemEvent e ){`
	`  if(chkOpcion.isSelected()){`
	`  //Código a ejecutar`
	`  }`
	`}`

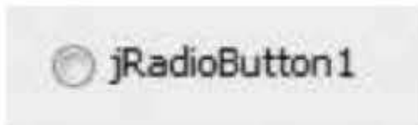
isSelected()	Permite determinar si el objeto **JCheckBox** ha sido seleccionado. Por ejemplo, si se quiere verificar cómo el control **chkOpcion** se encuentra seleccionado, se puede utilizar el siguiente código: `if (chkOpcion.isSelected())`

Casos de prueba

☐ Primaria ☐ Secundaria ☑ Técnico ☐ Superior	El control **JCheckBox** permitirá seleccionar una o más opciones según sea el caso. Por ejemplo, podríamos marcar los cuadros primaria, secundaria y técnico, ya que es el ciclo de evolución educativa que tenemos.

3.10 La clase JRadioButton

Esta clase permite incorporar un control de tipo opción. Comúnmente son usadas para que los usuarios puedan seleccionar una opción de un conjunto de opciones.



Principales propiedades

Propiedad	Descripción	Script
text	Define el contenido del control **JRadioButton**.	`/*Asignar el texto 'Acepto los terminos' al control rbTerminos*/` `rbTerminos.setText('Acepto los terminos');`
font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JRadioButton**.	`/*Configurar un tamaño de 20 puntos al texto mostrado en el control rbTerminos, podemos usar el siguiente código: */` `rbTerminos.setFont(new Font('Tahoma',0, 20));`
foreground	Asigna un color al texto contenido en el control **JRadioButton**.	`/*Asignar el color azul al texto contenido en el control rbTerminos */` `rbTerminos.setForeground(new java.awt.Color(0,0,255));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JRadioButton**.	`/*Asignar el mensaje 'Acepte los términos' al control rbTerminos*/` `rbTerminos.setToolTipText('Acepte los términos');`
buttonGroup	Define que un conjunto de controles **JRadioButton** se encuentren agrupados.	`/*Asignar el control rbTerminos al grupo btgReglas */` `rbTerminos.add(rbTerminos);`
selected	Define que el control **JRadioButton** aparezca previamente seleccionado.	`/*Asignar el control rbTerminos como un radio button seleccionado */` `rbTerminos.setSelected(true);`

Principales métodos

AddItemListener()	Permite agregar una acción de evento al **JRadioButton**. Por ejemplo, al seleccionar Imprimir en Pantalla desde el control **rbImprimir**, se podría usar el siguiente código:
	`rbImprimir.addItemListener(this);`
	`public void itemStateChanged( ItemEvent e ){`
	`  if(rbImprimir.isSelected()){`
	`  //Código para la impresión en pantalla`
	`  }`
	`}`

isSelected()	Permite determinar si el objeto **JRadioButton** ha sido seleccionado. Por ejemplo, para determinar que el control **rbImprimir** se encuentra activo o seleccionado, se puede usar el siguiente código: `if (rbImprimir.isSelected())`

Casos de prueba

◉ Soltero ○ Casado ○ Viudo ○ Divorciado	El control **JRadioButton** permitirá seleccionar una sola opción de un grupo de opciones. Hay que tener en cuenta que siempre debe estar asociada a un control **buttonGroup**. Por ejemplo, el estado civil de una persona siempre será una sola opción.

3.11 La clase JComboBox

Esta clase permite crear un objeto de tipo cuadro combinado. Este control permitirá controlar muchos valores dentro de un mismo control.

Principales propiedades

Propiedad	Descripción	Script

model	Define el contenido inicial del control **JComboBox**.	`/*Asignar las categorías de productos al cuadro combinado cboCategorias*/`
		`cboCategorias.addItem('Bebidas');`
		`cboCategorias.addItem('Golosinas');`
		`cboCategorias.addItem('Lacteos');`

font	Asigna un tipo de letra, tamaño y estilo al texto contenido en un **JComboBox**.	`/*Configurar un tamaño de 20 puntos al texto mostrado en el control cboCategorias, podemos usar el siguiente código: */` `cboCategorias.setFont(new Font('Tahoma',0, 20));`
foreground	Asigna un color al texto contenido en el control **JComboBox**.	`/*Asignar el color azul al texto contenido en el control cboCategorias */` `cboCategorias.setForeground(new java.awt.Color(0,0,255));`
ToolTipText	Define un mensaje de ayuda que se muestra en un marco de color amarillo al posicionar el puntero del mouse encima del control **JComboBox**.	`/*Asignar el mensaje 'Seleccione categoría' al control cboCategorias */` `cboCategorias.setToolTipText('Seleccione categoría');`
SelectedIndex	Define un elemento predeterminado de un conjunto de elementos por medio de su índice.	`/*Asignar como opción predeterminada a lácteos desde el control cboCategorias*/` `cboCategorias.selectedIndex(2);`

Propiedad	Descripción	Script
SelectedItem	Define un elemento predeterminado de un conjunto de elementos por medio de su contenido.	`/*Asignar como opción predeterminada a lácteos desde el control cboCategorias*/` `cboCategorias.selectedItem('Lacteos');`

Principales métodos

addItemListener()	Permite agregar a la lista contenedora de eventos un elemento de la lista contenida en el **JComboBox**.

addItem(elemento)	Permite añadir un nuevo elemento al contenedor **JComboBox**.
	Ej.: Agregar las siguientes categorías de empleados:
	
	`cboCategoria.addItem('Administrativo');`
	`cboCategoria.addItem('Operario');`
	`cboCategoria.addItem('Jefe');`

getItemCount()	Determina el total de ítems contenidos dentro del control **JComboBox**.
	Ej.: Mostrar el número de elementos del control **cboCategoria**.
	`int n=cboCategoria.getItemCount();`
	`JOptionPane.showMessageDialog(null, 'Número de elementos es: ' + n);`

getSelectedIndex()	Permite retornar el número de índice del elemento seleccionado desde el control **JComboBox**.
	Ej.: Determinar la posición del elemento seleccionado en el control **cboCategoria**.
	`int pos=cboCategoria.getSelectedIndex();`
	`JOptionPane.showMessageDialog(null, 'La posición del elemento es: ' + pos);`

getSelectedItem()	Permite retornar el elemento seleccionado desde el control **JComboBox**.
	Ej.: Mostrar el elemento seleccionado desde el cuadro combinado **cboCategoria**.
	`String categoria=(String)cboCategoria.getSelectedItem();`
	`JOptionPane.showMessageDialog(null, 'La categoría seleccionada es:' + categoria);`
	O también:
	`String categoria = String.valueOf(cboCategoria.getSelectedItem());`
	`JOptionPane.showMessageDialog(null, 'La categoría seleccionada es: '+categoria);`

setBounds(x,y, ancho,alto)	Permite posicionar el objeto **JComboBox** dentro del control **JFrame**.
	Ej.: Posicionar el cuadro combinado en (15, 15) y (90, 23).
	`cboCategoria.setBounds(15, 15, 90, 23);`
	O en su defecto se puede usar:
	`cboCategoria.setLocation(15,15);`
	`cboCategoria.setSize(90,23);`

setEditable()	Establece la edición sobre el control **JComboBox**. Ej.: Habilitar la edición sobre el control **cboCategoria**. `cboCategoria.setEditable(true);`
setEnabled()	Permite establecer la habilitación o no del control **JComboBox**. Ej.: Inhabilitar el control **cboCategoria**. `cboCategoria.setEnabled(false);`

setVisible()	Permite mostrar u ocultar el control **JComboBox**.
	Ej.: Mostar y ocultar el cuadro combinado de **Categoría**.
	`Mostrar > cboCategoria.setVisible(true);`
	`Ocultar > cboCategoria.setVisible(false);`

Casos de prueba

CRÉDITO CRÉDITO CONTADO SIN DEFINICIÓN	El control **JComboBox** permitirá seleccionar uno de los elementos presentados en la lista. Por ejemplo, se podrá seleccionar **crédito**, **contado** o **sin definición** en una aplicación que controle el tipo de pago que el cliente puede realizar.

TIPO DE PAGO
CRÉDITO
CRÉDITO
CONTADO
SIN DEFINICIÓN
	Se puede definir un título al control **JComboBox** para indicar qué valores podemos seleccionar. Se puede asignar un título al cuadro combinado usando el siguiente código:
	`cboTipo.setBorder(javax.swing.BorderFactory.`
	`  createTitledBorder('TIPO DE PAGO'));`
	O también realizando los siguientes pasos:
	■ Seleccione la propiedad **border**.
	■ Seleccione **Titled Border**.
	■ Coloque un texto descriptivo en **Title**.

3.12 La clase DefaultListModel

Un **DefaultListModel** es un tipo de contenedor para listas, que se encarga de administrar los valores de una lista, así como su incorporación, eliminación y actualización.

Métodos Constructores

DeafultListModel ()	Construye un modelo vacío. Por ejemplo, si se necesita almacenar información de categoría en un modelo, se puede usar el siguiente código:
	`DefaultListModel moCategoria - new DefaultListModel();`
	`moCategoria - new DefaultListModel();`

Métodos

addElement()	Permite añadir un valor al modelo. Por ejemplo, si se añade las categorías de un empleado al modelo **moCategoria**, se puede usar el siguiente código:
	`moCategoria.addElement('Jefe');`
	`moCategoria.addElement('Operario');`
	`moCategoria.addElement('Administrador');`
	`moCategoria.addElement('Tesorero');`

getSize()	Permite devolver el número de elementos contenidos en un modelo. Por ejemplo, para capturar la cantidad de elementos del modelo moCategoria, se puede usar el siguiente código: `int elementos - moCategoria.getSize();`

getElementAt()	Permite obtener un elemento del modelo desde un índice específico. Por ejemplo, para obtener el primer elemento del modelo **moCategoria** se usa el siguiente código:
	`int indice-0;`
	`String categoria - moCategoria.getElementAt(indice);`
	Si se necesita sumar todos los elementos contenidos en el modelo **moSubtotal**, se usa el siguiente código:
	`int n-moNumero.size();`
	`double subtotal-0;`
	`for(int i-0;i<n;i++){`
	`  subtotal +- Double.parseDouble(moSubtotal.`
	`  getElementAt(i).toString());`
	`}`

remove()	Permite eliminar un elemento ubicado en una posición del índice del modelo. Por ejemplo, para eliminar el primer elemento contenido en el modelo **moCategoria**, se puede usar el siguiente código:
	`int posicion-0;`
	`moCategoria.removeITem(posicion);`

size()	Permite determinar el tamaño total de los elementos contenidos en el modelo. Por ejemplo, para capturar la cantidad de elementos del modelo **moCategoria**, se puede usar el siguiente código: `int total - moCategoria.size();`

3.13 La clase JList

Esta clase permite crear un objeto tipo lista, en la cual se puede almacenar valores que serán visibles y accesibles por el usuario.

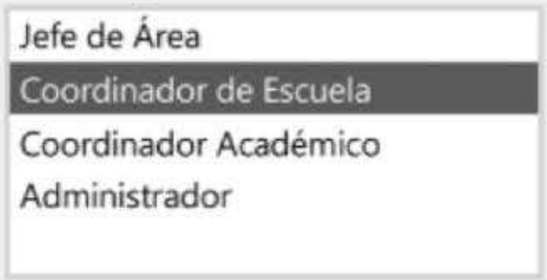
Principales propiedades

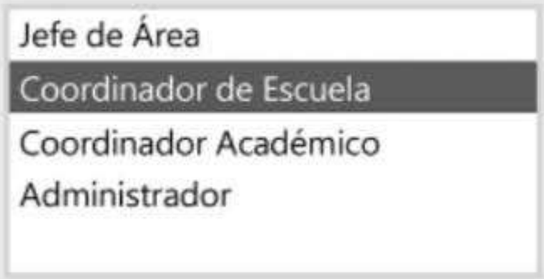
Propiedad	Descripción	Script

model	Define el contenido inicial del control **JList**.	`/*Asignar las categorías de productos al control JList*/`
		`DefaultListModel moC = new DefaultListModel();`
		`moC.addElement('Bebidas');`
		`moC.addElement('Golosinas');`
		`moC.addElement('Lacteos');`
		`lstCategoria.setModel(moC);`

Métodos constructores

JList()	Construye un objeto **JList** vacío. Por ejemplo: `lstProductos = new JList();`

JList (ListModel Modelo)	Construye un objeto **JList** con un conjunto de valores por medio de un modelo. Por ejemplo, si se asigna las categorías de empleado, el código podría ser:
	`DefaultListModel moCategorias=new DefaultListModel();`
	`moCategorias.addElement('Jefe de Area');`
	`moCategorias.addElement('Coordinador de Escuela');`
	`moCategorias.addElement('Coordinador Academico');`
	`moCategorias.addElement('Administrador');`
	`JList lstCategoria = new JList(moCategorias);`
	`lstCategoria.setBounds(10,10,200,100);`
	`add(lstCategoria);`
	 

JList (Object[] Lista)	Se podría usar una matriz para llenar el control. Por ejemplo, se declara el arreglo **objCategorias** y luego se agrega directamente al control **JList**.
	`Object objCategorias[] = new Object[4];`
	`objCategorias[0]='Jefe de Área';`
	`objCategorias[1]='Coordinador de Escuela';`
	`objCategorias[2]='Coordinador Académico';`
	`objCategorias[3]='Administrador';`
	`JList lstCategoria=new JList(objCategorias);`
	`lstCategoria.setBounds(10,10,200,100);`
	`add(lstCategoria);`
	 

Métodos

addListSelectionListener()	Permite añadir un receptor de selección al control **JList**.
clearSelection()	Permite borrar la selección asignada al control **JList**.
getSelectedIndex()	Permite devolver el índice del primer elemento seleccionado en el control **JList**.
getSelectedIndex()	Permite devolver una matriz de tipo entero con todos los elementos formados por los índices seleccionado en el control **JList**.
getSelectedValue()	Permite devolver el primer valor seleccionado desde el control **JList**; devolverá nulo si no ha seleccionado nada en el control.
getSelectionMode()	Permite definir las listas simples y múltiples en un control **JList**.
setListData(Object[] Lista)	Permite construir un modelo de lista a partir de una matriz de objetos, y después aplica el **setModel** al modelo.
setModel(ListModel modelo)	Permite asignar al modelo que representa el contenido de la lista, y limpia la selección.

3.14 La clase JMenuBar

Esta clase permite crear un objeto tipo menú, el cual se posiciona en la parte superior de un **JFrame**. Hay que tener en cuenta que solo añade la barra de menú mostrando los títulos de las opciones del menú, pero no presenta subopciones, estas deben ser añadidas con el objeto **JMenuItem**.

Figura 3.6 **JMenuBar** con Eclipse Mars.

Figura 3.7 **JMenuBar** con NetBeans.

Principales propiedades

Propiedad	Descripción	Script
Text	Modifica el título del menú seleccionado.	`/*Asignar el título Listado al control JMenu1*/` `jMenu1.setText('Listado');`
Icon	Asigna una imagen tipo ícono en el lado derecho del título del menú.	`/*Asignar un icono al control JMenu1*/` `jMenu1.setIcon(new ImageIcon(getClass().getResource('/pFormularios/Calendar.png')));`
Enabled	Asigna la habilitación o inhabilitación del botón de menú.	`/*Asignar la habilitación True al control JMenu1*/` `jMenu1.setEnabled(false);`

Opciones que se presentan en la implementación de un menú con JMenuBar

- **Agregar un nuevo elemento de menú.** Haga clic derecho sobre la barra de menú y seleccione **Add menú**.

- **Modificar el título del elemento de menú.** Seleccione un elemento y modifique la propiedad **Text** desde el panel de propiedades.

- **Agregar un elemento al menú.** Arrastre un control **Menú ítem** desde el panel **Swing Menús** y suéltelo encima de un determinado menú; inicialmente se muestra como **JMenuItem1**.

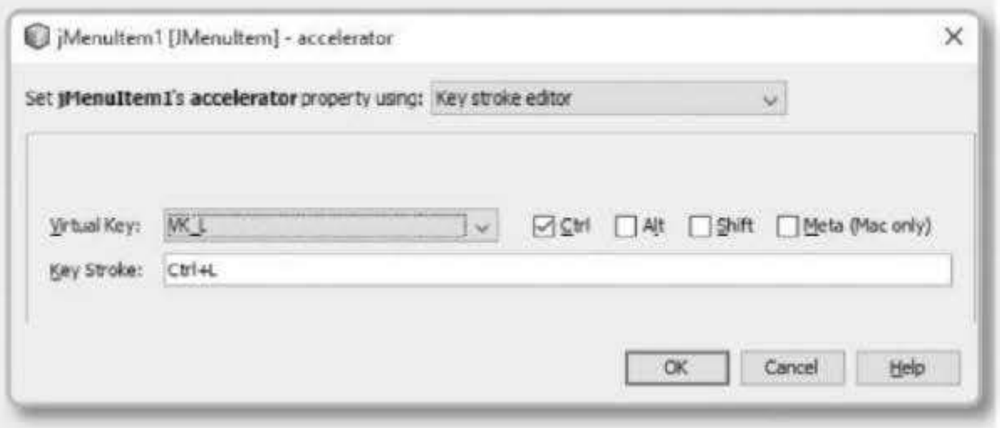
- **Asignar un nombre al elemento.** Seleccione un elemento de menú, haga clic derecho sobre **Change Variable Name** y asigne un nombre, como, por ejemplo, **miListado**.
- **Asignar un ícono al elemento de menú.** Seleccione un elemento de menú y desde las propiedades elija un ícono en la propiedad **icon**.

3.15 La clase JMenuItem

Esta clase permite crear una opción de menú sobre un control **JMenuBar**. Un ejemplo se podría dar en el menú **Mantenimiento**, como se muestra en la siguiente imagen:

Principales propiedades

PROPIEDAD	DESCRIPCIÓN	SCRIPT
Name	Define el nombre del control.	`/*Asignar el nombre miListado a una opción del menú */` `JMenuItem miListado - new JMenuItem();`
Text	Modifica el título del elemento de menú seleccionado.	`/*Asignar el título 'Listado de Clientes' al control JMenuItem1*/` `jMenuItem1.setText('Listado de Clientes');`
Icon	Asigna una imagen tipo ícono en el lado derecho del título del menú.	`/*Asignar un ícono al control JMenu1*/` `jMenu1.setIcon(new ImageIcon(getClass().getResource('/pFormularios/Calendar.png')));`

Accelerator	Asigna una ruta corta por medio del teclado a una opción del menú.	`/*Asignar CTRL+L al control JMenuItem1*/`
		 
		
		`jMenuItem1.setAccelerator(KeyStroke.getKeyStroke(java.awt.event.KeyEvent.VK_L, java.awt.event.InputEvent.CTRL_MASK));`

3.16 Clase Desktop Pane

Esta clase permite crear un contenedor de formularios de tipo internal. Su principal objetivo es implementar una base repositoria de controles de tipo formulario. Es de suma importancia su incorporación dentro del proceso de creación de menús en una aplicación Java.

Principales propiedades

Propiedad	Descripción	Script
Name	Es el nombre del control **Desktop Pane**.	`/*Asignar el nombre dpMenu al control Desktop Pane*/` `JDesktopPane dpMenu - new JDesktopPane();`

Principales métodos

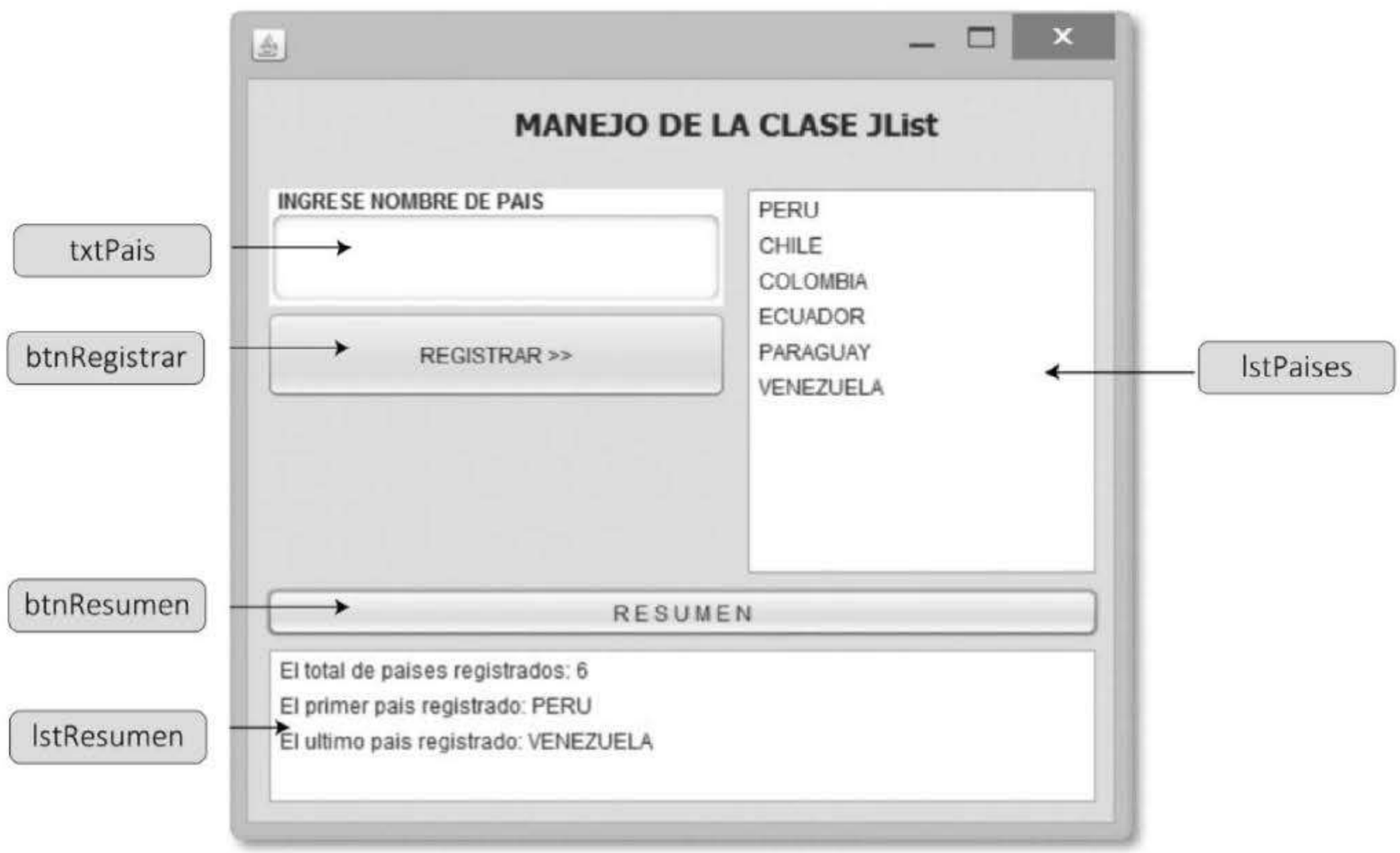
Propiedad	Descripción	Script
add	Permite agregar un formulario al control **Desktop Pane**.	`/*Agregar el formulario frmListado al control DeskTop Pane dpMenu*/` `dpMenu.add(frmListado);`

CASO DESARROLLADO 1: **Manejo de la clase JList**

Implemente una aplicación que permita registrar un determinado país desde un cuadro de texto y que luego pueda mostrarse dentro de una lista. Finalmente, se debe presentar en un cuadro resumen el total de países registrados y, además, indicar el primer y último país registrado.

Consideraciones:

Utilice la siguiente interfaz para la aplicación:



Solución:

```
package pFormularios;
import javax.swing.DefaultListModel;

public class frmListado extends javax.swing.JFrame {

    //Definición del modelo
    DefaultListModel moPaises;
    DefaultListModel moResumen;

    public frmListado() {
        initComponents();

        //Creando los objetos de tipo DefaultListModel
        moPaises = new DefaultListModel();
        moResumen = new DefaultListModel();

        //Asociar el modelo a los controles de tipo JList
        lstPaises.setModel(moPaises);
        lstResumen.setModel(moResumen);
    }

    private void btnRegistrarActionPerformed(java.awt.event.ActionEvent evt) {
        //Capturar el país ingresado
        String pais = txtPais.getText();
```

```
        //Enviando el pais al modelo
        moPaises.addElement(pais);

        //Limpiar la caja para un nuevo registro
        txtPais.setText('');

        //Devolver el enfoque del puntero del mouse
        txtPais.requestFocus();
    }

    private void btnResumenActionPerformed(java.awt.event.ActionEvent evt)
{
        //Capturar el total de elementos del modelo
        int totalElementos - moPaises.getSize();

        //Capturar el primer pais registrado
        String primero-moPaises.getElementAt(0).toString();
        String ultimo-moPaises.getElementAt(totalElementos-1).toString();

        //Enviar al modelo
        moResumen.addElement('El total de paises registrados: ' +
                    totalElementos);
        moResumen.addElement('El primer pais registrado: '+primero);
        moResumen.addElement('El ultimo pais registrado: '+ultimo);
    }
}
```

CASO DESARROLLADO 2: **Manejo de la clase JComboBox**

Implemente una aplicación que permita seleccionar un determinado producto desde un cuadro combinado (**JComboBox**), y que al seleccionarlo muestre el precio asociado a dicho producto en un control **JLabel**.

Consideraciones:

Utilice la siguiente interfaz para la aplicación:

Solución:

```
package pFormularios;
public class frmVenta extends javax.swing.JFrame {

    public frmVenta() {
        initComponents();
        llenaProductos();
    }

    //Capturar el producto seleccionado
    void llenaProductos(){
        cboProducto.addItem('Producto 1');
        cboProducto.addItem('Producto 2');
        cboProducto.addItem('Producto 3');
        cboProducto.addItem('Producto 4');
    }
```

```
        //Enviando el pais al modelo
        moPaises.addElement(pais);

        //Limpiar la caja para un nuevo registro
        txtPais.setText('');

        //Devolver el enfoque del puntero del mouse
        txtPais.requestFocus();
    }

    private void btnResumenActionPerformed(java.awt.event.ActionEvent evt)
{
        //Capturar el total de elementos del modelo
        int totalElementos - moPaises.getSize();

        //Capturar el primer pais registrado
        String primero-moPaises.getElementAt(0).toString();
        String ultimo-moPaises.getElementAt(totalElementos-1).toString();

        //Enviar al modelo
        moResumen.addElement('El total de paises registrados: ' +
                    totalElementos);
        moResumen.addElement('El primer pais registrado: '+primero);
        moResumen.addElement('El ultimo pais registrado: '+ultimo);
    }
}
```

CASO DESARROLLADO 2: **Manejo de la clase JComboBox**

Implemente una aplicación que permita seleccionar un determinado producto desde un cuadro combinado (**JComboBox**), y que al seleccionarlo muestre el precio asociado a dicho producto en un control **JLabel**.

Consideraciones:

Utilice la siguiente interfaz para la aplicación:

Solución:

```
package pFormularios;
public class frmVenta extends javax.swing.JFrame {

    public frmVenta() {
        initComponents();
        llenaProductos();
    }

    //Capturar el producto seleccionado
    void llenaProductos(){
        cboProducto.addItem('Producto 1');
        cboProducto.addItem('Producto 2');
        cboProducto.addItem('Producto 3');
        cboProducto.addItem('Producto 4');
    }
```

```
    private void cboProductoItemStateChanged(java.awt.event.ItemEvent evt) {
        //Capturar el producto seleccionado
        String producto = String.valueOf(cboProducto.getSelectedItem());
        //Capturar el producto seleccionado
        if (producto.equals('Producto 1')) lblPrecio.setText('15.00');
        if (producto.equals('Producto 2')) lblPrecio.setText('25.00');
        if (producto.equals('Producto 3')) lblPrecio.setText('10.00');
        if (producto.equals('Producto 4')) lblPrecio.setText('5.00');
    }
```

CASO DESARROLLADO 3: Manejo de la clase JMenuBar

Implemente una aplicación que permita invocar a formularios de tipo internal dentro de un control **Desktop Menú**.

Consideraciones:

Use la siguiente interfaz para la aplicación:

Figura 3.8 Menú principal.

Solución:

En la ventana menú principal, los elementos que la componen no necesariamente deben tener un nombre asignado, por lo tanto, se dejará como se presentan.

Figura 3.9 Elementos del menú **Listado**.

En las opciones que presenta el **Listado**, sí se debe asignar un nombre de objeto a cada opción del menú: **miGeneralProductos** para la opción **General de productos**, **miProductosCategoria** para la opción **Productos por categoría** y **miProductosPrecio** para la opción **Productos por precio**.

Figura 3.10 Elementos del menú **Mantenimiento**.

Se asigna los nombres **miNuevoProducto** para la opción **Nuevo producto**, **miModificaProducto** para la opción **Modifica producto** y **miEliminaProducto** para la opción **Elimina producto**.

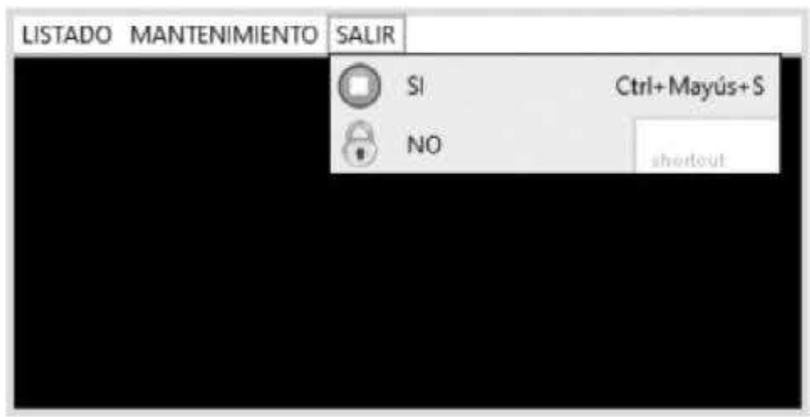
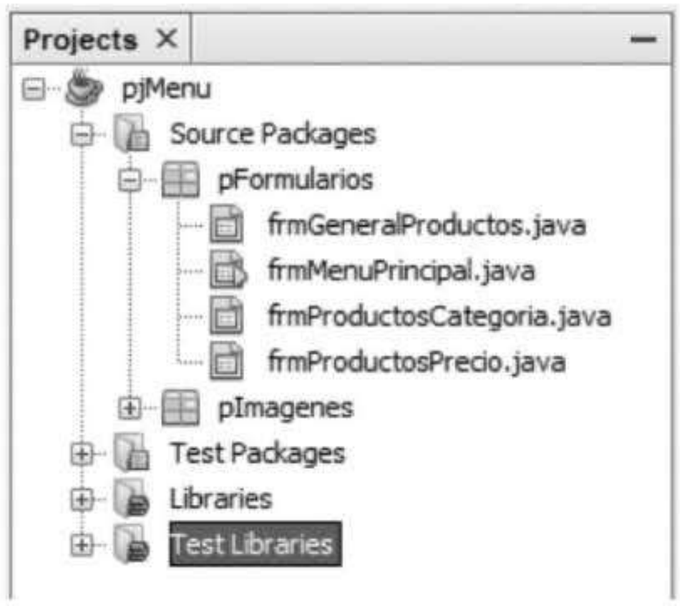


Figura 3.11 Elementos del menú **Salir**.

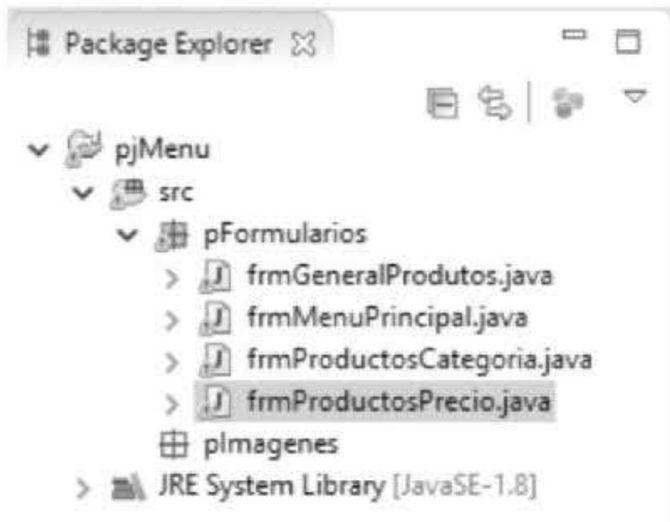
Se asigna los nombres **miSi** para la opción **Sí** y **mino** para la opción **No** del menú de opciones de salida de la aplicación. El proyecto debe tener el siguiente aspecto:

- En Netbeans



Se puede apreciar que solo existe un formulario **JFrame** llamado **frmMenuPrincipal**; los demás formularios son de tipo **JInternalFrame**.

- En Eclipse Mars



A continuación, se presenta el código que permite anexar los formularios internal dentro del control **Desktop Pane** y mostrarlos dentro de él mediante el método show.

```
package pFormularios;

public class frmMenuPrincipal extends javax.swing.JFrame {

    public frmMenuPrincipal() {
        initComponents();
    }

    private void miGeneralProductosActionPerformed(java.awt.event.ActionEvent evt) {
    frmGeneralProductos objG=new frmGeneralProductos();
        dpMenu.add(objG);
        objG.show();
    }

    private void miProductosxCategoriaActionPerformed(java.awt.event.ActionEvent evt) {
    frmProductosCategoria objC=new frmProductosCategoria();
        dpMenu.add(objC);
        objC.show();
    }

    private void miProductosxPrecioActionPerformed(java.awt.event.ActionEvent evt) {
    frmProductosPrecio objP=new frmProductosPrecio();
        dpMenu.add(objP);
        objP.show();
    }
}
```

Programación orientada a objetos

CAPÍTULO

4

Competencias

- Reconoce los conceptos básicos de programación orientada a objetos.
- Implementa aplicaciones con atributos privados y públicos
- Implementa aplicaciones con método constructor.

Contenido

4.1 Generalidades

4.2 Conceptos iniciales de programación orientada a objetos

4.3 Paquetes Java

4.4 Clases en Java

4.5 Método constructor

4.6 Referencia **this**

4.7 Objetos en Java

4.8 Métodos **get** y **set**

- Caso desarrollado 1: Casa de cambio (con atributos públicos de clase)
- Caso desarrollado 2: Casa de cambio (con atributos privados de clase)
- Caso desarrollado 3: Casa de cambio (con métodos **get** y **set**)
- Caso desarrollado 4: Casa de cambio (con método constructor)

4.1 Generalidades

La *programación orientada a objetos* surge como un intento de dominar la complejidad que tiene la construcción de un *software*. Tiene un enfoque específico con respecto a las aplicaciones, pues se orienta al manejo de objetos, superando el estilo de programación secuencial o modular.

Tradicionalmente, la forma de enfrentar una complejidad del *software* era usando la programación modulada, en la cual un bloque de código se podía reutilizar solo dentro de la misma aplicación. La programación orientada a objetos tiene el mismo principio, pero lo que la diferencia de los demás estilos de programación es que va más allá de la modulación de su código. Sus características fundamentales son:

- Abstracción
- Encapsulamiento
- Herencia
- Polimorfismo
- Reutilizar código

Por tanto, se puede decir que la programación orientada a objetos en Java es una forma especial de implementar aplicaciones, más cercana a cómo expresaríamos las cosas en la vida real. Con la programación orientada a objetos se tiene que aprender a pensar las cosas de una manera distinta a lo tradicional, ya que nuestras aplicaciones las veremos en términos de clases, objetos y métodos.

Si nos remontamos a la historia de cómo nace la programación orientación a objetos, podemos encontrar que surge en Noruega, por los años 60; se inicia con un lenguaje de programación llamado Simula 67, el cual introduce los conceptos de clases y subclases en su desarrollo, algo parecido a la programación orientada a objetos del presente. Uno de los inconvenientes para el desarrollo de un *software* era que no lograban culminarse con éxito, ya que solo se podía avanzar una parte del análisis, dejando un vacío dentro del desarrollo y mostrando un tanto compleja su terminación. La programación orientada a objetos propone una nueva metodología de desarrollo en la que sobresale el ocultamiento de la información; la idea era encapsular cada una de las variables globales de una aplicación en un solo módulo que hoy llamamos clases, junto a sus operaciones o métodos, siendo así la única forma de acceder a su contenido.

A inicios de los años 80 se amplió el lenguaje C y se pasó a llamar C++ que tenía como soporte a la programación orientada a objetos, abriendo un camino hacia los nuevos lenguajes orientados a objetos que contamos en la actualidad. En esa misma década se desarrollaron otros lenguajes orientados a objetos como Objective C, Common Lisp Object System (CIOS), object Pascal, Ada y otros.

Los años 90 son considerados como la consolidación de la metodología de programación orientada a objetos, la cual es la forma más cercana de programar y acercarnos a la realidad. Así, en el año 1996, surge Java como una evolución del lenguaje C++, ya que la primera posee todas sus características y funcionalidades y las explota comercialmente como el lenguaje del futuro.

4.2 Conceptos iniciales de programación orientada a objetos

4.2.1 Abstracción

Tiene como objetivo principal mostrar un comportamiento particular de un objeto, pero no detalla ni muestra cuáles son las actividades que se realizan para llegar al comportamiento deseado. Es decir, solo se enfoca en la visión externa de un objeto, separando el comportamiento específico del mismo. Finalmente, se puede decir que la abstracción muestra las características esenciales de un determinado objeto, las cuales distinguen a este de los demás objetos. la figura 4.1 muestra un ejemplo de abstracción del objeto automóvil.

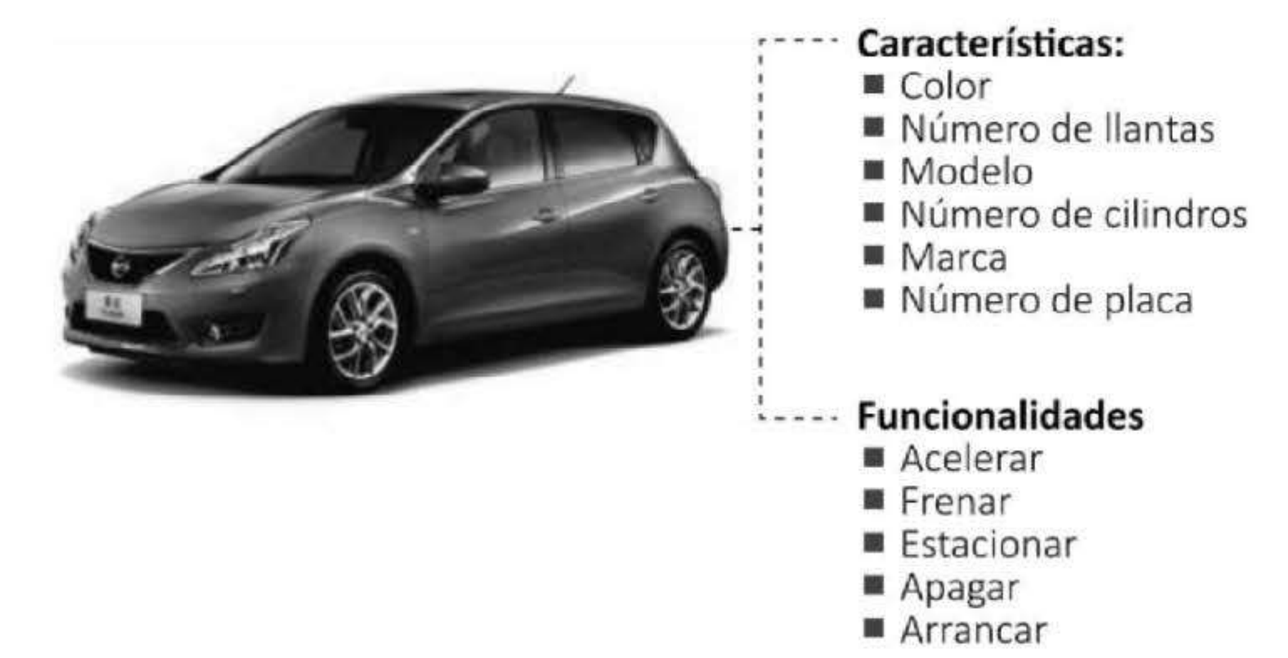


Figura 4.1 Abstracción de un automóvil.

4.2.2 Encapsulamiento

Tiene como objetivo almacenar, en una misma sección, los elementos obtenidos a partir de una abstracción, sabiendo que dichos elementos solo pueden ser características o funcionalidades. El encapsulamiento secciona las características o funcionalidades de su posible diseño, ello garantiza la integridad de los datos que contiene un determinado objeto.

Figura 4.2 Encapsulamiento de funcionalidades entre un usuario del banco y su cajero automático.

4.2.3 Herencia

Tiene como objetivo transmitir código de una clase a otra con cierto grado de seguridad. Para soportar este mecanismo se debe identificar qué clase será padre, es decir, la clase de la cual heredarán otras; y por otro lado, se deben identificar las clases hijas, las cuales obtienen la información de la clase padre, evidenciando la herencia entre ambos.

La herencia también es llamada subclase, ya que existen clases que heredarán funcionalidad y características de una clase padre. Una de las características es que las subclases o clases hijas pueden, asimismo, definir nuevas características y funcionalidades.

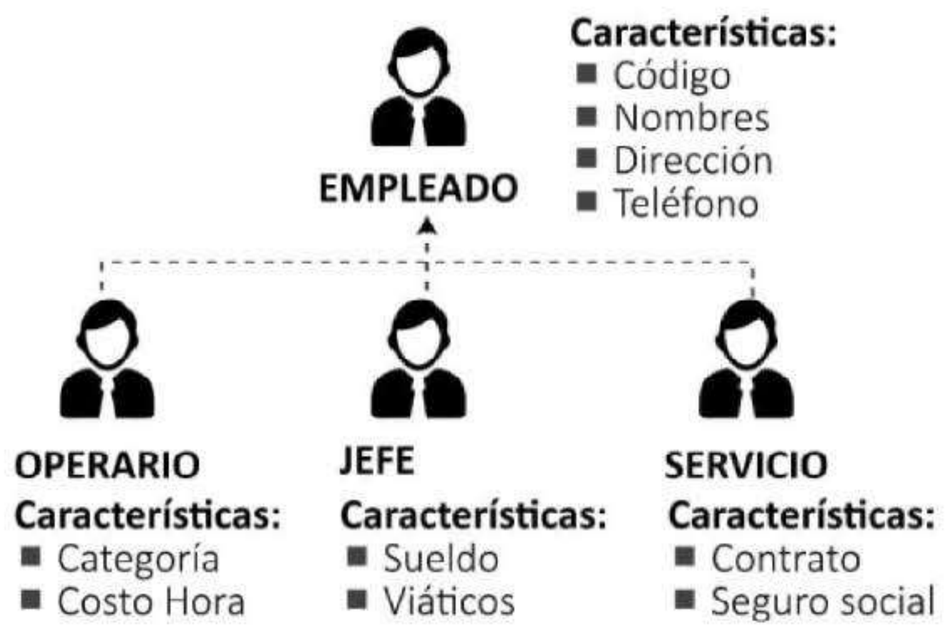


Figura 4.3 Herencia entre empleados de una misma empresa.

4.2.4 Polimorfismo

Es uno de los pilares de la programación orientada a objetos, el cual permite implementar métodos con el mismo nombre, pero de diferente accionar. Se podría, por ejemplo, nombrar la forma para calcular el sueldo de los empleados de una empresa, los cuales se distinguen por el tipo de sueldo que perciben: unos tienen un salario fijo y otros ganan por comisión. En ambos casos se debe calcular el sueldo, pero se tiene que realizar de diferente forma.

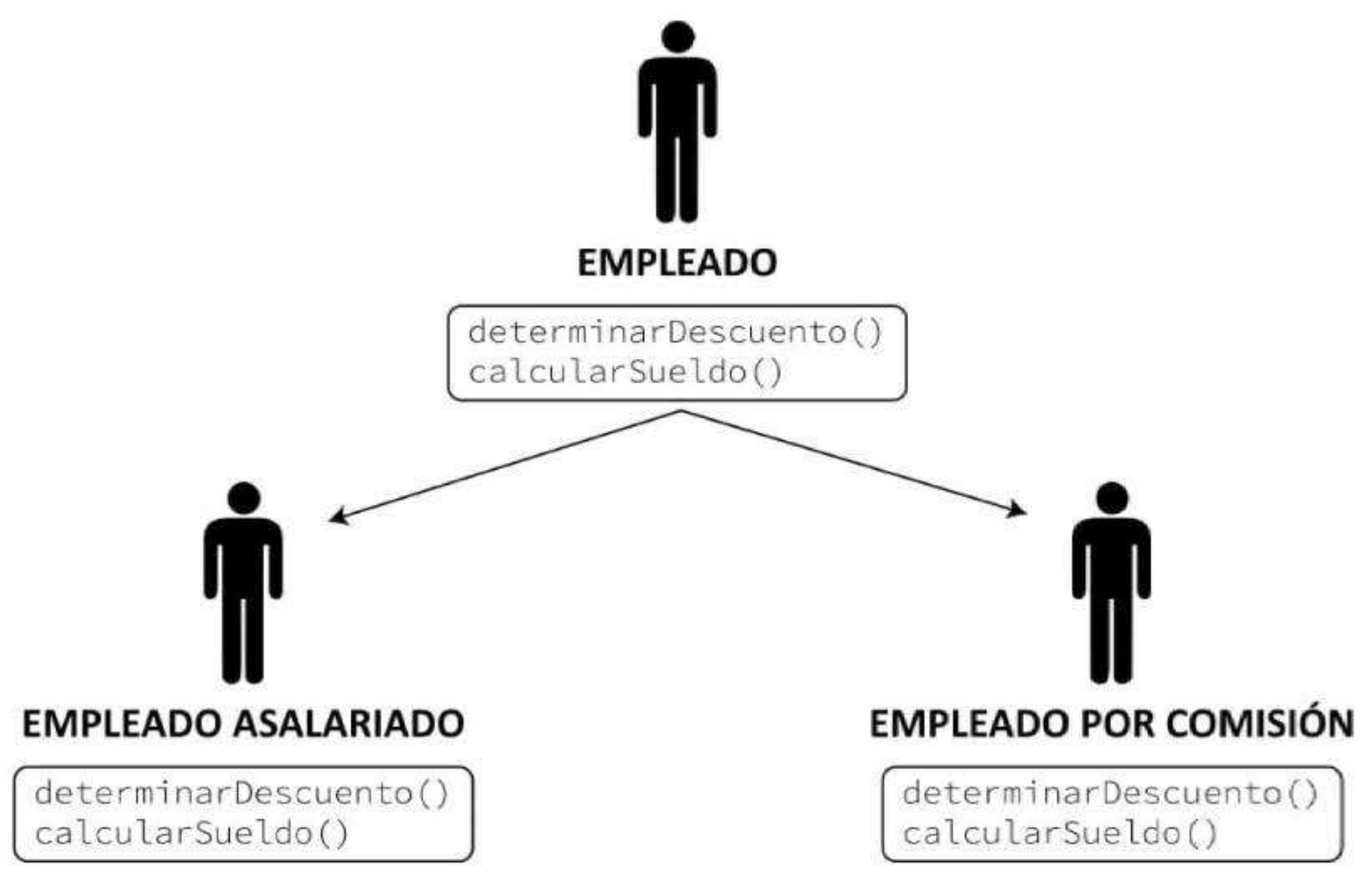


Figura 4.4 Métodos polimórficos de los empleados de una empresa.

Se puede observar que para implementar polimorfismos se deben heredar los códigos de una clase base y desde allí sobrescribir sus métodos; haciendo que, en las clases derivadas, dichos métodos tengan diferente implementación, es decir, diferente accionar desde un mismo método común.

Cuando se ejecuta un programa orientado a objetos, ocurren tres acciones:

1. Se crean los objetos cuando se necesitan.
2. Los mensajes se envían desde un objeto y se reciben en otro.
3. Se borran los objetos cuando ya no son necesarios y se recupera la memoria ocupada por ellos.

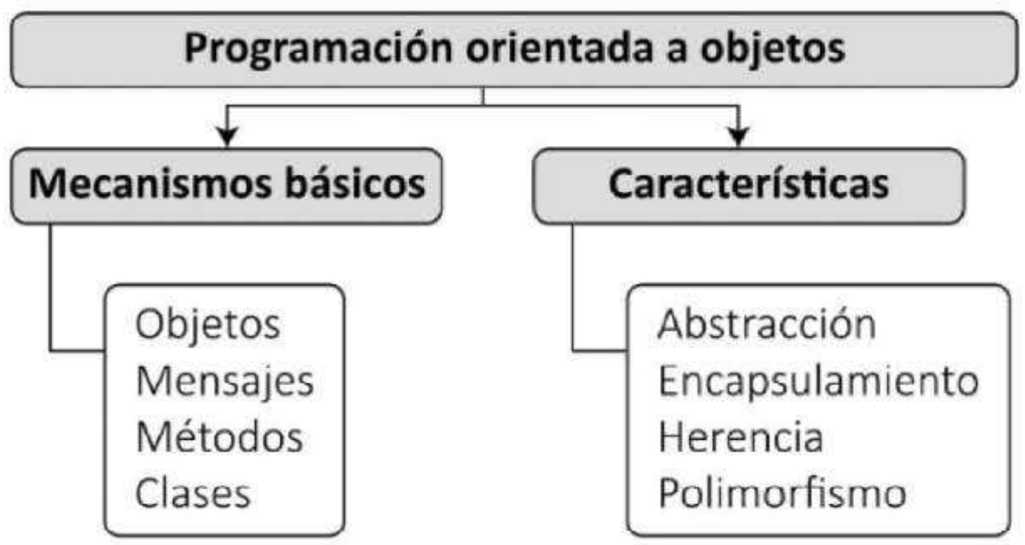


Figura 4.5 Mecanismos y características de la programación orientada a objetos (POO).

Finalmente, se puede decir que la programación orientada a objetos usa mecanismos básicos para su implementación, los cuales interactúan entre sí, como los objetos, los mensajes, los los métodos y las clases. De igual manera, presenta características como la abstracción, el encapsulamiento, la herencia y el polimorfismo.

4.3 Paquetes Java

Las aplicaciones que se desarrollan en Java inicialmente suelen ser de pocos elementos, pero conforme vamos usando Java notaremos que se necesita cierto orden dentro de un proyecto; es decir, las clases y formularios que se pueden crear, o las muchas imágenes que se pueden usar dentro de un mismo proyecto, únicamente pueden ser organizados por medio de los paquetes, que no son más que simples carpetas.

Los paquetes en Java o *packages* son considerados como la forma adecuada de poder organizar clases, interfaces, formularios, conexión a base de datos o imágenes de manera lógica. Se podría decir que los paquetes:

- organizan las clases que podrían estar asociadas por algún objetivo específico;
- evitan la duplicidad de clases;
- muestran de manera clara la estructura de un proyecto Java.

Cuando se crea una aplicación en NetBeans, los paquetes se encuentran dentro de la carpeta llamada **Source Packages**. Si no se define un nombre de paquete, automáticamente se le asigna un paquete por defecto; esto es muy similar al hecho de crear una carpeta en Windows y no asignarle nombre. El nombre mostrado es **default package**, tal como se muestra en la imagen 4.6.

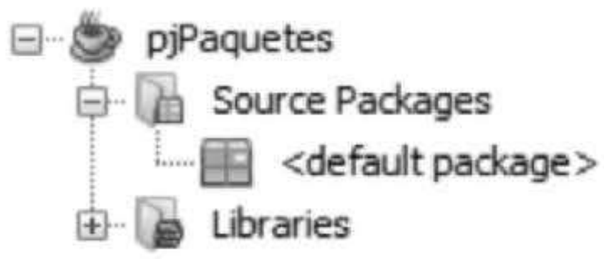


Figura 4.6 Paquete por defecto de NetBeans.

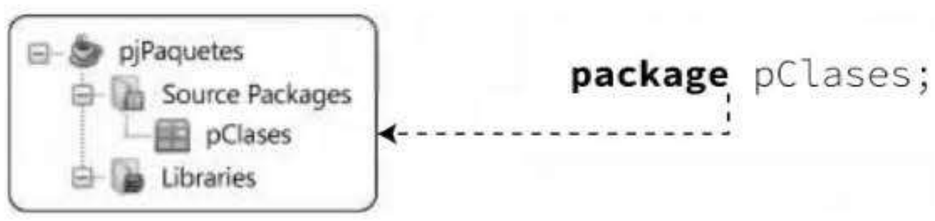
Una buena práctica es tener una aplicación organizada; por lo tanto, se podría haber creado una estructura de paquetes tal como se muestra en la siguiente imagen:

Figura 4.7 Paquetes organizados en NetBeans.

La distribución de paquetes se debe a una conveniencia propia de la aplicación. Por ejemplo, se crea el paquete **pClases** para almacenar todas las clases usadas en la aplicación; de la misma forma el paquete **pFormularios** almacenará todos los formularios **JFrame** que se incorporen al proyecto. Asimismo, se podrían crear paquetes dentro de paquetes, que se llamarían subpaquetes, pero esto será decisión de cada quien.

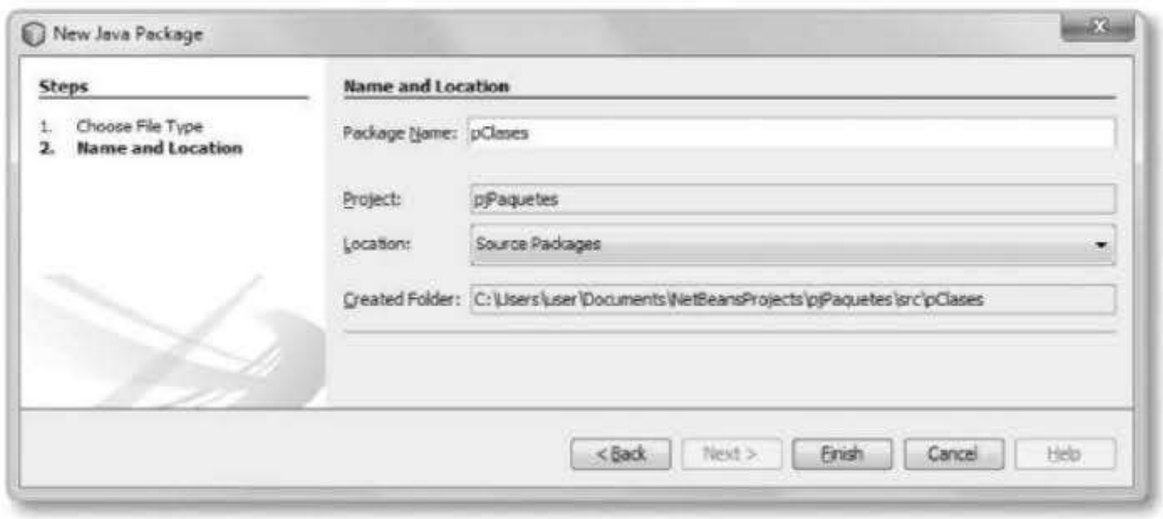
4.3.1 Creación de un paquete

La forma para crear paquetes dentro de una aplicación dependerá muchas veces del framework donde se encuentre desarrollando Java, pero a nivel de código se crea de la siguiente manera:



Los pasos para crear un paquete en NetBeans

- Haga clic derecho sobre la carpeta **Source Package**.
- Seleccione la ruta **New > Java Package....**



- Asigne un nombre en la sección **Package Name**. Luego haga clic en el botón **Finish**.

  Cuando se crea una clase dentro del paquete, se notará que la sentencia **package pClases** se encuentra en la primera línea, como se muestra en el siguiente código:

```
package pClases;
public class Cliente {
}
```

Esto indica que toda la implementación de la clase se encuentra dentro del paquete **pClases**; si otra clase tiene la misma línea inicial de código, entonces también pertenecerá al mismo paquete, como, por ejemplo, la definición de la clase **Producto** en el mismo paquete **pClases**.

```
package pClases;
public class Producto {
}
```

Entonces, se tiene que entender que todas las clases especificadas sobre el mismo paquete se encontrarán organizadas dentro del mismo. Cada paquete contendrá sus propias clases, pero eso no significa que no se encuentren asociadas, ya que un paquete no aísla sus elementos, sino que simplemente los organiza.

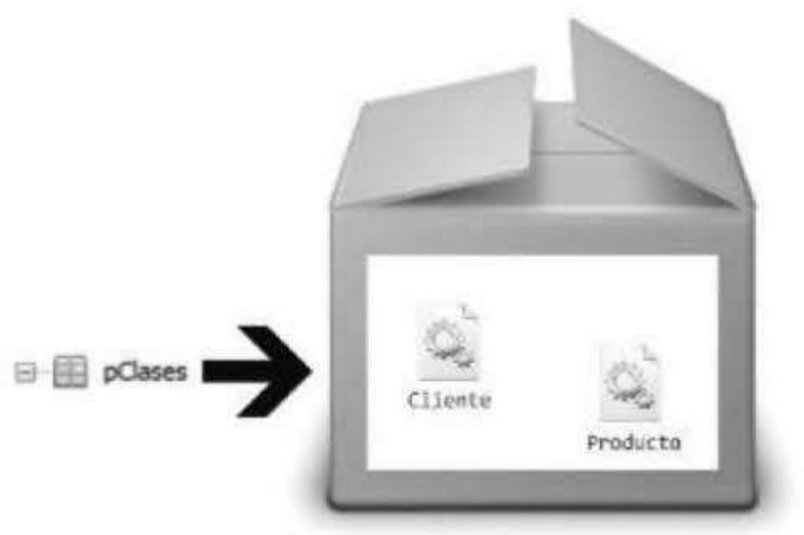


Figura 4.8 Contenido del paquete **pClases**.

4.3.2 Importación de paquetes

La importación conlleva a la apertura de una o más clases contenidas dentro del paquete; es decir, se podrá usar las clases que fueron implementadas con visibilidad pública. El formato de uso del paquete es como sigue:

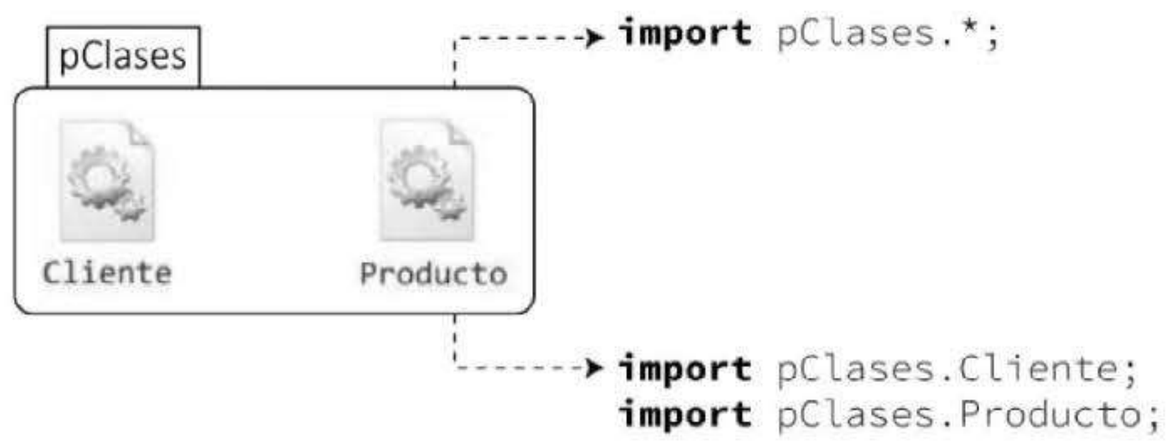


Figura 4.9 Importación de paquetes.

Todos los IDE tienen el mismo formato de importación; en algunos casos se realiza en forma automática, como sucede con NetBeans, que al momento de crear una clase se especifica a qué paquete se destinará la clase, mientras que la sentencia **import** aparece de forma automática dentro del código fuente.

En la imagen anterior las clases **Cliente** y **Producto** se encuentran dentro del mismo paquete. Así pues, para invocar a los métodos contenidos en dichas clases, se puede especificar el nombre de la clase al final de la sentencia o colocar el operador asterisco, el cual hace referencia a todas las clases contenidas en el paquete.

4.3.3 Alcance de los elementos contenidos en un paquete

Si se tiene en cuenta que la construcción de un paquete conlleva a una organización de clases, entonces también se debe considerar el alcance o la visibilidad que tiene frente a los demás paquetes. Por ejemplo, si se necesita hacer referencia de un paquete a otro, se debe considerar que dicho elemento sea *privado*, *público*, *protegido* o *sin alcance definido*. Se expone, a continuación, un cuadro que muestra la situación que puede tener un elemento dentro de un paquete:

Situación del elemento contenido en el paquete	Visibilidad privada *(private)*	Visibilidad pública *(public)*	Visibilidad protegida *(protected)*	Sin visibilidad
Dentro de la misma clase.	SÍ	SÍ	SÍ	SÍ
Desde una subclase.	NO	SÍ	SÍ	NO
Desde una clase implementada dentro del mismo paquete.	NO	SÍ	SÍ	SÍ
Desde el exterior de la misma clase.	NO	SÍ	NO	NO

En la gráfica anterior, se muestra el alcance que pueden tener los miembros de una clase; todo programador debe tener en cuenta que tanto los atributos como los métodos trabajan con un fin en común, y que la definición del alcance es muy importante para la aplicación.

4.4 Clases en Java

Una clase es el elemento básico de la programación orientada a objetos, representa a una plantilla desde la cual se pueden crear objetos.

Técnicamente una clase es la descripción de un tipo de objeto aplicando la abstracción como pilar de la programación orientada a objetos; esto quiere decir que la definición de un objeto es la clase. Cuando programamos sobre un objeto y se le define sus características y funcionalidades, en realidad lo que estamos haciendo es programar en una clase.

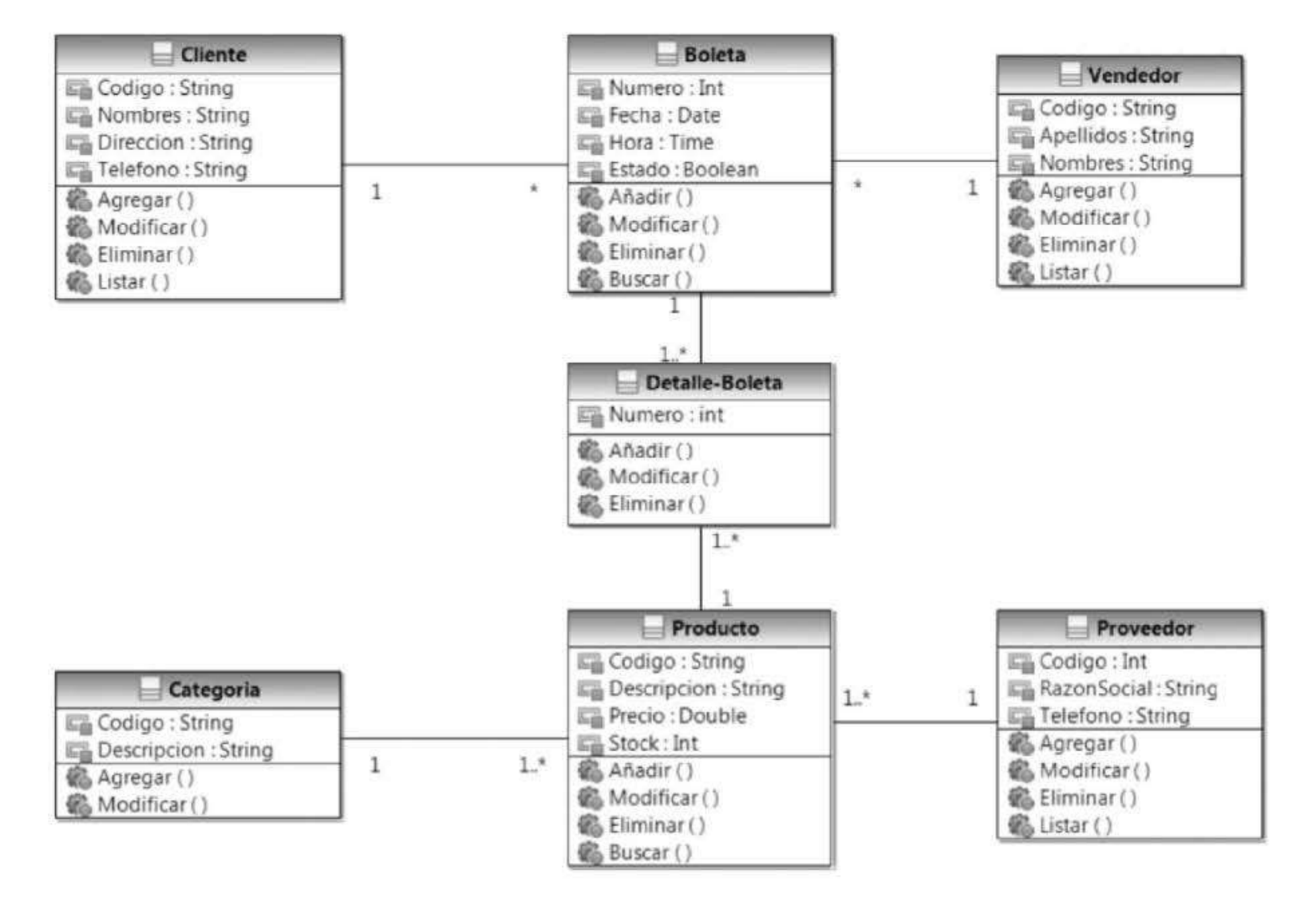


Figura 4.10 Composición de clases en un sistema de ventas.

4.4.1 Estructura de una clase

En la programación, la clase es representada por tres compartimentos, como se visualizan en la siguiente figura:

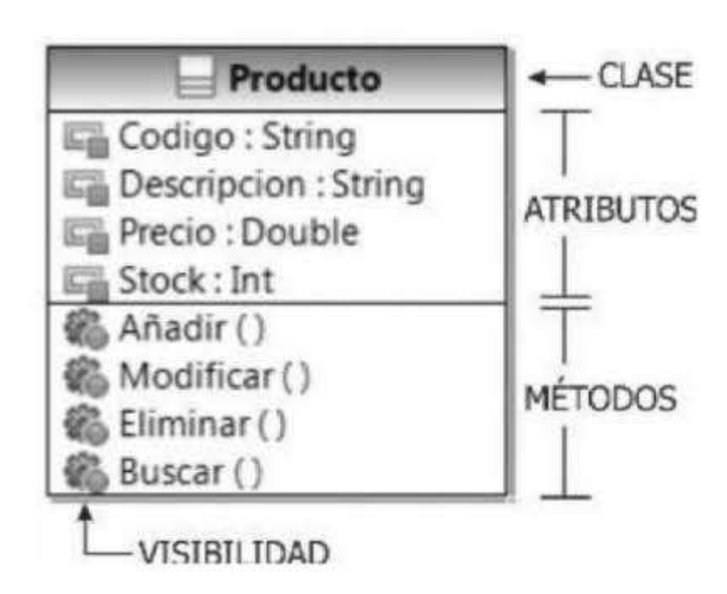


Figura 4.11 Estructura de la clase **Producto**.

A nivel de código, la implementación de la clase se visualiza de la siguiente manera:

```
package pClases;

public class Producto {
    private String codigo;
    private String descripcion;
    private double precio;
    private int stock;

    public void añadirProducto() {

    }
    public void eliminarProducto() {

    }
    public void buscarProducto() {

    }
}
```

Clase

Atributos

Métodos

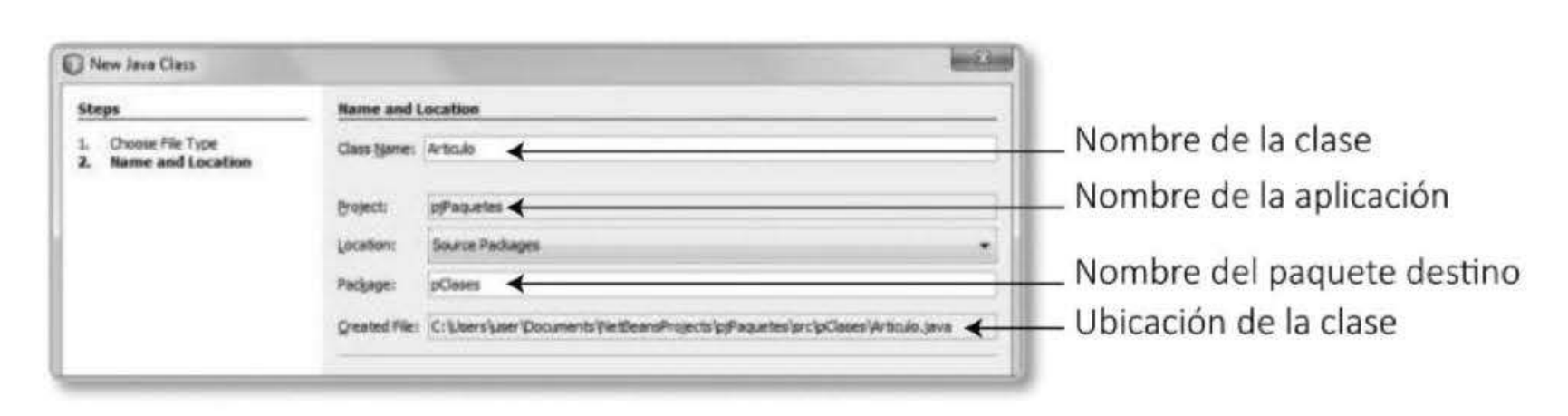
Visibilidad

Figura 4.12 Código Java implementado la clase **Producto**.

4.4.2 Agregar una clase a un paquete en NetBeans

Cuando se desarrolla una aplicación Java, se debe tener una estructura de paquetes para dividir, en agrupaciones lógicas, todos los componentes de la aplicación. Ahora vamos a incorporar clases a nuestros paquetes con NetBeans. Para ello se realizan los siguientes pasos:

- Haga clic derecho sobre un determinado paquete.
- Luego siga la ruta **New > Java Class**.



- **Class Name.** Es el nombre asignado a la clase.
- **Project.** Muestra el nombre del proyecto (no es modificable).
- **Location.** Es el lugar destino de las clases; en este caso debe estar seleccionado **Source Packages**.
- **Package.** Muestra el nombre del paquete de destino, es decir, en dónde se almacenará la clase.
- **Created File.** Muestra la ruta del archivo creado (no es modificable).

4.4.3 Atributos de una clase

Un atributo de una clase es la característica que puede tener un objeto. Cada atributo soporta un valor único dentro de la aplicación. El formato que se presenta para declarar un atributo se muestra como sigue:

Donde:

- **Visibilidad.** Define la visibilidad que puede tener un atributo frente a otras clases; esta puede ser *private*, *public*, *protected* o *sin definición*.
- **TipoDatos.** Es el tipo de datos que se le asigna al atributo.
- **Atributo.** Es el nombre que se le asignará al atributo de clase.

Por ejemplo, si se tiene la clase **Producto** y se necesita declarar el atributo código, podríamos tener las siguientes opciones:

- `int codigo;`

  Al declarar un código sin visibilidad, solo podrá accederse desde la misma clase y no desde el exterior de la clase.

- `public int codigo;`

  Al declarar un código como public, todas las clases del mismo proyecto podrán hacer referencia al valor contenido en dicho atributo.

- `private int codigo;`

  Al declarar un código como private, solo podrá accederse desde la clase que se definió. Hay que mencionar que la implementación del atributo como privado es lo más recomendado, puesto que se protege la información contenida en el atributo, es decir, no será manipulada.

- `protected int codigo;`

  Al declarar un código como protected, solo podrá accederse desde la clase que se definió y las clases descendientes de esta; esto se da cuando hay una relación de herencia entre las clases.

Si declaramos la clase **Producto**, con atributos privados, se vería de la siguiente manera:

En Java se tendría el siguiente script:

```
import pClases;

public class Producto{
    private String codigo;
    private String descripcion;
    private double precio;
    private int stock;
}
```

Hay que tener en cuenta que al crear un objeto de la clase **Producto** no se podrá tener acceso a sus atributos; para esto debe implementarse métodos de acceso público que se asocien a cada atributo como los métodos **GET** y **SET**. El script que declara los atributos de la clase **Producto** como públicas es:

En Java se tendría el siguiente script:

```
import pClases;

public class Producto{
    public String codigo;
    public String descripcion;
    public double precio;
    public int stock;
}
```

4.4.4 Métodos en las clases

Un método representa a la funcionalidad que puede tener una clase; se podría decir que la lógica que necesita una aplicación debería estar encapsulada dentro de la clase mediante métodos. Se presentan dos formatos para la implementación de métodos en Java:

Figura 4.13 Estructura de un método vacío.

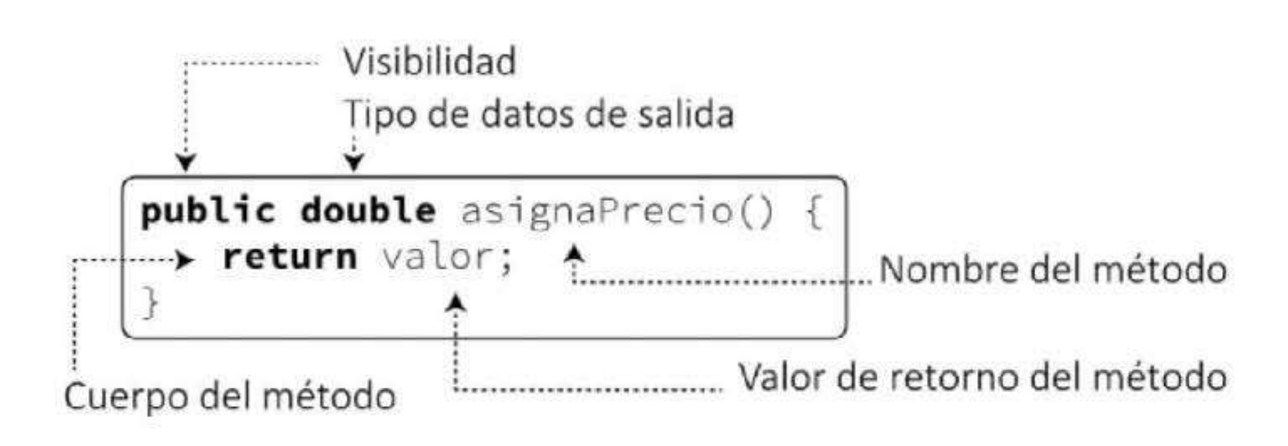


Figura 4.14 Estructura de un método con valor de retorno.

Donde:

- **Visibilidad.** Es el nivel de acceso que se define al método; este puede ser *private*, *public*, *protected* o *sin definición*.
- **TipoDatos.** Todo método tiene un objetivo dentro de la aplicación; al definir el tipo de datos, se le indica al método qué valor debe retornar, y cuando no retorna valor alguno, se coloca la cláusula **void**.
- **Nombre del método.** Es el nombre que se le asigna al método de clase. Hay que tener en cuenta que existen métodos con parámetros o sin ellos; esto dependerá de la necesidad del método.
- **Valor de retorno.** Aquí se define qué valor retornará el método cuando se encuentre asociado a una variable. Hay que tener en cuenta que al definir un método tipo **void** no se considerará el valor de retorno por tratarse de un método vacío.

Por ejemplo, si se necesita declarar el método **asignaPrecio** de la clase **Producto**, tendríamos las siguientes posibilidades de implementación:

- `double asignaPrecio(){}`

  Al declarar el método **asignaPrecio** *sin visibilidad*, solo será accesible por los métodos que se implementen dentro de la clase.

- `public double asignaPrecio(){}`

  Al declarar como *público* el método **asignaPrecio**, será accesible desde todos los métodos que se encuentren en cualquier clase de la misma aplicación.

- `private double asignaPrecio(){}`

  Al declarar como *privado* el método **getCodigo**, solo será accesible dentro de la misma clase.

- `protected double asignaPrecio(){}`

  Al declarar como *protegido* el método **getCodigo**, será accesible desde la misma clase y las clases descendientes.

4.4.5 Métodos sin valor de retorno

Son considerados como métodos vacíos y que no devuelven respuesta alguna a quien los invoca, pero sí realizan una actividad dentro de la aplicación.

Figura 4.15 Definición de un método sin valor de retorno.

Estos métodos se pueden implementar cuando se necesite realizar una asignación a un determinado parámetro o para actividades de impresión. Obsérvese, por ejemplo, el método que permita asignar un valor a un parámetro dentro de la clase **producto**:

```
import pclases;

public class producto{
    //Atributo
    private String descripcion;

    //Metodo
    public void asignaDescripcion(String des) {
        descripcion = des;
    }
}
```

El método **asignaDescripcion** recibe como parámetro la descripción del producto, y este será enviado al atributo privado **descripción**. Desde la aplicación, el código se muestra de la siguiente manera:

```
Producto objPro = new Producto();
objPro.asignaDescripcion('Lavadora 16Kg');
```

4.4.6 Métodos con valor de retorno

Son considerados como métodos que devuelven un solo valor como resultado, es decir, estos métodos siempre tienen un solo objetivo.

Figura 4.16 Definición de un método con valor de retorno.

Estos métodos se pueden implementar cuando se necesite realizar cálculos sobre determinados atributos de la clase. Obsérvese, por ejemplo, cómo asignar un precio a un determinado producto:

```
import pclases;

public class producto{
    //Metodo
    public double asignaPrecio(String descripcion){
        double precio=0;

        if (descripción.equals('Lavadora'))
            precio = 1500;

        return precio;
    }
}
```

El método **asignaPrecio** recibe como parámetro la descripción del producto para poder determinar de qué producto se trata; y usando la estructura selectiva **if** se le asigna un monto tipo precio. Desde la aplicación, el código se muestra de la siguiente manera:

```
Producto objPro = new Producto();
double precio = objPro.asignaPrecio('Lavadora');
```

4.5 Método constructor

Un constructor es un método especial dentro de la clase, el cual se encargará de asignar valores iniciales a los parámetros privados de la clase. Hay que tener en cuenta que esta actividad solo lo realiza una vez por ejecución, vale decir que, si necesita actualizar o modificar el valor asignado por el constructor, tendrá que usar los métodos **set**.

Figura 4.17 Definición de un método constructor.

Donde:

- **Visibilidad.** Representa el alcance del método **constructor**; casi siempre se declara como *public*, ya que este método será invocado desde fuera de la clase que se implementó.
- **Nombre del método.** Es el nombre asignado al método **constructor**, para poder ser reconocido como constructor debe tener el mismo nombre de la clase.
- **Cuerpo del método.** Aquí se pueden definir inicializaciones de atributos privados de clase.

Si se cuenta con el siguiente código:

```
import pClases;

public class Producto{
    private String codigo;
    private String descripcion;
    private double precio;
    private int stock;
}
```

Y se necesita implementar el método **constructor** de la clase **Producto**, podríamos usar el siguiente código:

```
public Producto(String codigo, String descripcion, double precio, int stock){
    this.codigo - codigo;
    this.descripcion - descripcion;
    this.precio - precio;
    this.stock - stock;
}
```

4.6 Referencia this

Permite resolver ambigüedades entre los atributos de una clase y los parámetros de los métodos. Muchos métodos de un objeto necesitan acceder a los datos miembros de este y a otras propiedades del objeto. Para hacer referencia explícitamente a una referencia desde el método se utiliza la palabra reservada **this**. La especificación de la referencia **this** es de la siguiente forma:

```
this.atributo;
this.metodo();
```

Donde:

- **this.atributo.** Hace referencia a un atributo de clase.
- **this.m**étodo**.** Hace referencia a un método de clase.

Por ejemplo, si se tiene el método constructor **Empleado**, que tiene como parámetros el código, nombres y edad, que casualmente tienen el mismo nombre que los atributos privados de la clase, debemos referirnos a ellos por medio de la referencia **this**:

```
//Atributos privados
private String codigo;
private String nombres;
private int edad;

//Método constructor
public Empleado(String codigo, String nombres, int edad) {

    //Manejo de la referencia this
    this.codigo-codigo;
    this.nombres-nombres;
    this.edad-edad;
}
```

4.7 Objetos en Java

Los objetos son copias o ejemplares de una clase. El hecho de crear un objeto permite tener acceso a los atributos y métodos definidos en la clase. La acción de crear un objeto de una clase es llamada instancia de clase. Obsérvese la siguiente imagen:

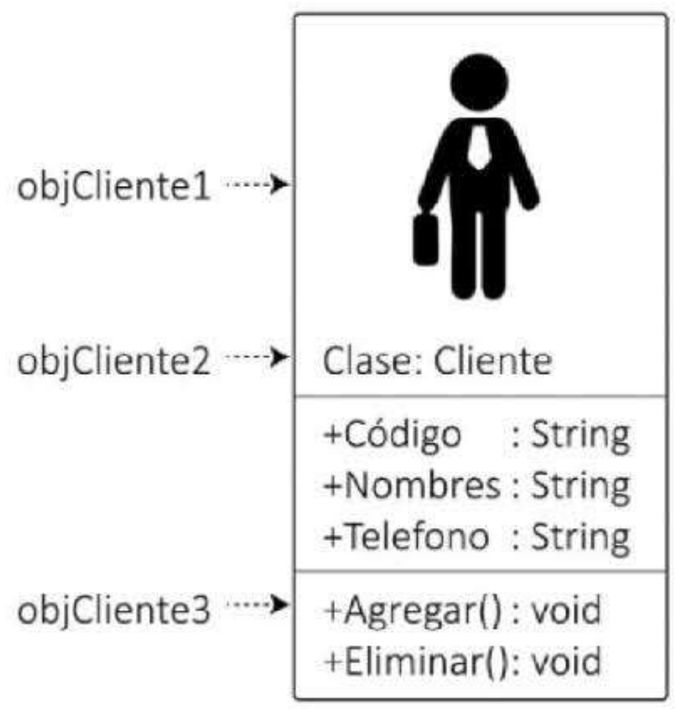


Figura 4.18 Objetos de la clase **Cliente**.

Cuando se implementa una clase se definen atributos y métodos que podrán ser accesibles mediante objetos; estos tendrán la misión de asociarse a la clase. Se presentan dos formas de crear un objeto de clase:

Clase
Nombre del objeto

```
Cliente objCliente - new Cliente();
```

Método constructor
Operador

Figura 4.19 Creando un objeto en una sola línea.

Clase
Nombre del objeto

```
Cliente objCliente;
objCliente - new Cliente();
```

Método constructor
Operador

Figura 4.20 Creando un objeto en dos líneas.

Donde:

- **Clase.** Es el nombre de la clase del cual se necesita crear un objeto. Hay que tener en cuenta que dicha clase se debe encontrar en el mismo paquete; en caso contrario, se debe usar la sentencia **import**.
- **Nombre del objeto.** Es el nombre que se le asigna al objeto, hay que asignar un nombre que haga referencia a la clase de donde proviene.

- **Operador.** Es el operador **new** el que permite crear el objeto de clase.
- **Método constructor.** Es la especificación que se realiza para la inicialización de los valores de clase.

Por ejemplo, si se van a crear tres objetos de la clase **Cliente**, podríamos usar los siguientes códigos como alternativa:

```
Cliente objCliente1 - new Cliente();
Cliente objCliente2 - new Cliente();
Cliente objCliente3 - new Cliente();
```

Los objetos se declaran uno por cada línea, haciendo llamadas múltiples a la clase y a su método **constructor**.

```
Cliente objCliente1, objCliente2, objCliente3;

objCliente1 - new Cliente();
objCliente2 - new Cliente();
objCliente3 - new Cliente();
```

Los objetos se crean basándose en una sola declaración de objetos.

4.7.1 Referenciar a los atributos de clase

Cuando se crea un objeto de una clase, en realidad lo que se quiere es tener acceso a sus atributos o métodos. Aquí se debe considerar la visibilidad que estos tengan, pues solo se podrá acceder a los elementos públicos. El formato para acceder a los atributos se muestra en la siguiente imagen:

```
Cliente objCliente;
objCliente - new Cliente();

objCliente.nombre - "Angela Victoria";
```

Atributo público

Valor asignado

Objeto de clase

Figura 4.21 Accediendo a los atributos de una clase.

Donde:

- **Objeto de clase.** Es el nombre del objeto declarado como instancia de una clase.
- **Atributo público.** Es la referencia al atributo público de la clase asociada. Hay que tener en cuenta que se le puede asignar un valor o el resultado de una expresión, solo se debe considerar que el tipo de datos devuelto por el valor o por la expresión sea el mismo que se especificó en el atributo.
- **Valor asignado.** Es el valor que se le asigna al atributo; dicho valor será usado dentro de los métodos de la clase para devolver algún valor resultante.

4.7.2 Referenciar a los métodos de clase

Los métodos de clase siempre tienen un objetivo específico en la aplicación. El formato para acceder a los métodos se muestra en la siguiente imagen:

```
Cliente objCliente;
objCliente - new Cliente();

objCliente.agregaCliente("Angela Victoria");
```

Método público

Parámetro

Objeto de clase

Figura 4.22 Referencia a los métodos de una clase.

Donde:

- **Objeto de clase.** El objeto es la referencia a la clase. El punto es obligatorio al momento de invocar algún método público de la clase, tiene que considerar el tipo de valor devuelto por método.
- **Método público.** Se hace referencia al nombre del método que se implementa en la clase con visibilidad pública.
- **Parámetro.** Son los valores que necesita el método para realizar un determinado proceso; la cantidad de valores enviados debe ser igual al número de parámetros especificados en la clase y, además, deben ser del mismo tipo de datos.

4.8 Métodos get y set

Cuando se declaran atributos, hay una diferencia entre una declaración pública y otra privada, ya que el nivel de acceso se refleja al crear un objeto de la clase y tener acceso a los atributos. Entonces, se puede decir que el uso de los métodos **get** y **set** permitirán tener un mejor acceso a los atributos privados dentro de una clase.

4.8.1 Método set

Permite asignar un valor a un determinado atributo de clase, la idea principal es dar un valor a un atributo que no es accesible por contar con visibilidad privada. Dentro de la clase hace que un atributo privado tenga un valor desde el exterior. Su formato se presenta de la siguiente manera:

Visibilidad

Método

Nombre del método

Valor exterior

```
public void setAtributo(tipo parámetro){
    atributoPrivado - Parámetro;
}
```

Atributo de clase

Valor exterior

Figura 4.23 Implementación del método **set**.

Donde:

- **Visibilidad.** Representa el alcance del método **set**; casi siempre se declara como *public*, ya que los valores provienen del exterior de la clase que lo implementa.
- **Método void.** La misión de este método es asignar un valor desde el exterior y no devolver nada desde el método.
- **setAtributo.** Es el nombre del método, siempre se inicia con la palabra **set** seguido del nombre del parámetro privado.
- **Valor exterior.** Es el valor recibido desde el exterior, que será asignado al parámetro privado de la clase.
- **Atributo de clase.** Es la referencia al atributo privado definido en la clase.

Por ejemplo, si se tiene la declaración del atributo privado **Descripción** de la clase **Producto** y se necesita implementar su método **set**, podríamos tener las siguientes alternativas:

```
public class Producto {
    private String descripcion;
}
```

La declaración del atributo **descripción** se realiza después de la creación de la clase.

```
public void setDescripcion(String desc){
    descripcion = desc;
}
```

El parámetro **desc** obtiene el valor que proviene del exterior y le asigna al atributo privado **descripción**.

```
public void setDescripcion(String sDescripcion){
    this.descripcion = sDescripcion;
}
```

En ocasiones se puede diferenciar el parámetro del método con el atributo privado colocando la primera letra del tipo de datos en el parámetro.

```
public void setDescripcion(String descripcion){
    this.descripcion = descripcion;
}
```

Finalmente, si el parámetro y el atributo privado de la clase tienen el mismo nombre, nos podríamos confundir en la referencia, para diferenciarlos se usa la palabra reservada **this**, la cual hace referencia a todos los atributos privados de la clase.

4.8.2 Método get

Si bien los métodos **set** tienen la misión de asignar un valor a un atributo privado, es el método **get** el que hace evidente la asignación devolviendo el valor asignado a dicho atributo.

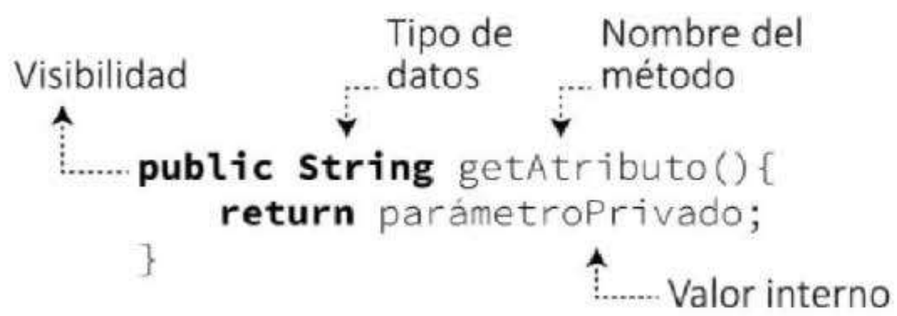


Figura 4.24 Estructura del método **get**.

Donde:

- **Visibilidad.** Representa el alcance del método **get**; siempre se declara como *public*, ya que este método será invocado desde fuera de la clase.
- **TipoDatos.** Es el tipo de datos del valor que devolverá el método. Hay que tener en cuenta que el parámetro privado y su método **get** deben tener el mismo tipo de datos.
- **getAtributo.** Es el nombre que se le asigna al método **get**, casi siempre se inicia con la palabra **get** seguido del nombre del parámetro privado.
- **Valor interno.** Aquí se especifica qué valor devolverá el método; casi siempre se coloca el nombre del parámetro directamente, pero algunos programadores usan el operador **this** para hacer referencia a un miembro de la clase.

Por ejemplo, si se tiene la declaración del atributo privado **Descripción** de la clase **Producto** y se necesita implementar su método **get**, podríamos tener las siguientes alternativas:

```
public class Empleado {
    private String descripcion;
}
```

La declaración del atributo **descripción** se realiza después de la creación de la clase.

```
public String getDescripcion(){
    return descripcion;
}
```

El método tiene como valor de devolución el atributo **descripción**.

```
public String getDescripcion(){
    return this.descripcion;
}
```

La palabra reservada **this** hace referenciar a todos los atributos privados de la clase, y por este medio se puede especificar qué atributos se devolverán por los métodos **get**.

4.8.3 Implementación de métodos get y set con NetBeans

Se presenta la clase **Producto** con los atributos **Código**, **Descripción**, **Precio** y **Stock**, como se muestra en la siguiente imagen:

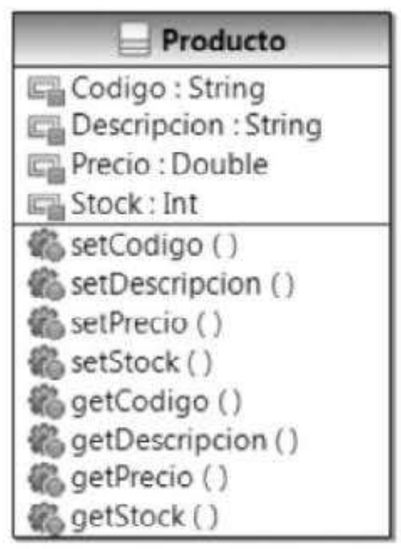


Figura 4.25 Estructura de la clase **Producto**.

Para poder crear los métodos **get** y **set** en NetBeans, se deben realizar los siguientes pasos:

1. Declare los atributos privados dentro de la clase **Producto**.

```
package pClases;

public class Producto {
    private String codigo;
    private String descripcion;
    private double precio;
    private int stock;
}
```

2. Haga clic derecho sobre la clase. Luego siga la ruta **Refactor > Encapsulate Fields**.

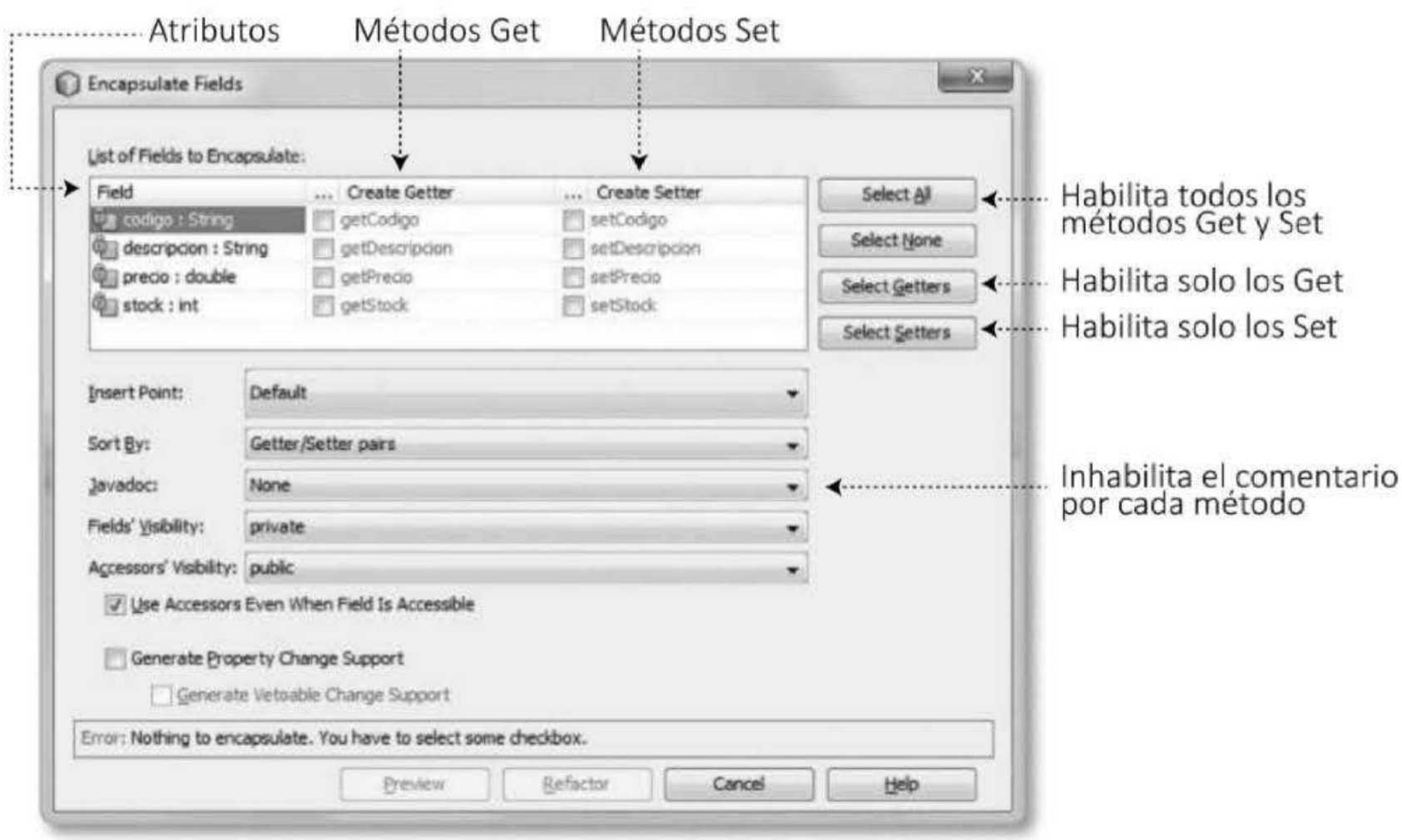


Figura 4.26 Ventana de encapsulado de campos.

3. Seleccione los métodos que necesite implementar.
4. Seleccione **None** en la opción **Javadoc** para no mostrar comentarios sobre cada método implementado.
5. Finalmente, haga clic el botón **Refactor** para implementar los métodos. El código se muestra de la siguiente manera:

```
package pClases;
public class Producto {
    private String codigo;
    private String descripcion;
    private double precio;
    private int stock;

    public String getCodigo() {
        return codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

CASO DESARROLLADO 1: **Casa de cambio (con atributos públicos de clase)**

Una casa de cambio de dinero necesita una aplicación que le permita agilizar sus procesos de consultas sobre el tipo de cambio. Para ello, la casa de cambio usa solo tres tipos de monedas: soles, dólares y marcos, respectivamente. Implemente una aplicación que determine el monto total del dinero en euros de un cliente que necesita realizar el cambio de soles, euros y marcos. Utilice la siguiente tabla de conversión:

1 dólar	3.51 soles
1 dólar	1.09 euros
1 dólar	2.12 marcos

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con atributos públicos.
- Use el siguiente diseño de interfaz gráfica:

- Debe ingresar un monto en soles, dólares y marcos, respectivamente.
- Cree una clase llamada **Casa** que defina los atributos referentes a los montos ingresados.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion41**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmCasaCambio**.
5. Agregue una clase llamada **Casa** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE CAMBIOS – CASA DE CAMBIO
2	JTextField	txtSoles
3	JTextField	txtDolares
4	JTextField	txtMarcos
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Casa** coloque el siguiente código:

```
package pClases;

public class Casa {

    //Atributos publicos
    public double soles;
    public double dolares;
    public double marcos;

    //Métodos
    public double calculaMontoEuros(){
        return (soles/3.51+marcos/2.12+dolares)*1.09;
    }
}
```

8. Agregue código al botón **Procesar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
    //Creando el objeto de la clase Casa
    Casa objC = new Casa();

    //Enviando los valores a los atributos públicos
    objC.soles = Double.parseDouble(txtSoles.getText());
    objC.dolares = Double.parseDouble(txtDolares.getText());
    objC.marcos = Double.parseDouble(txtMarcos.getText());

    //Realizando los cálculos
    double euros = objC.calculaMontoEuros();

    //Mostrando los resultados en la lista
    DefaultListModel moR = new DefaultListModel();
    moR.addElement('** RESUMEN DE CAMBIO **');
    moR.addElement('--------------------------');
    moR.addElement('MONTO SOLES: ' + String.format('%.2f', objC.soles));
    moR.addElement('MONTO DOLARES: ' + String.format('%.2f', objC.dolares));
    moR.addElement('MONTO MARCOS: ' + String.format('%.2f', objC.marcos));
    moR.addElement('--------------------------');
    moR.addElement('EL MONTO TOTAL EN EUROS ES:' + String.format('%.2f',
                   euros));

    lstResumen.setModel(moR);
}
```

9. Agregue código al botón **Limpiar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
    txtSoles.setText('');
    txtDolares.setText('');
    txtMarcos.setText('');
    txtSoles.requestFocus();
}
```

10. Agregue código al botón **Salir**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
    int r = JOptionPane.showOptionDialog(this, 'Estas seguro de salir...?',
                        'Casa de Cambio', JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE, null, null, null);

    if (r == 0) {
        System.exit(0);
    }
}
```

11. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 2: Casa de cambio (con atributos privados de clase)

Una casa de cambio de dinero necesita una aplicación que le permita agilizar sus procesos de consultas sobre el tipo de cambio. Para ello, la casa de cambio usa solo tres tipos de monedas: soles, dólares y marcos, respectivamente. Implemente una aplicación que determine el monto total del dinero en euros de un cliente que necesita realizar el cambio de soles, euros y marcos. Utilice la siguiente tabla de conversión:

1 dólar	3.51 soles
1 dólar	1.09 euros
1 dólar	2.12 marcos

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con atributos privados.
- Use el siguiente diseño de interfaz gráfica:

- Debe ingresar un monto en soles, dólares y marcos, respectivamente.
- Cree una clase llamada **Casa** que defina los atributos referentes a los montos ingresados.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion42**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmCasaCambio**.
5. Agregue una clase llamada **Casa** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne los nombres a los controles como se muestra en la imagen:

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE CAMBIOS – CASA DE CAMBIO
2	JTextField	txtSoles
3	JTextField	txtDolares
4	JTextField	txtMarcos
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Casa** coloque el siguiente código:

```
package pClases;

public class Casa {
    //Atributos privados
    private double soles;
    private double dolares;
    private double marcos;

    //Métodos que asignar un valor a los atributos privados
    public void asignaSoles(double s){
        soles = s;
    }

    public void asignaDolares(double d){
        dolares = d;
    }

    public void asignaMarcos(double m){
        marcos = m;
    }

    //Metodos
    public double calculaMontoEuros(){
        return (soles/3.51+marcos/2.12+dolares)*1.09;
    }
}
```

6. Diseñe la siguiente interfaz y asigne los nombres a los controles como se muestra en la imagen:

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE CAMBIOS – CASA DE CAMBIO
2	JTextField	txtSoles
3	JTextField	txtDolares
4	JTextField	txtMarcos
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Casa** coloque el siguiente código:

```
package pClases;

public class Casa {
    //Atributos privados
    private double soles;
    private double dolares;
    private double marcos;

    //Métodos que asignar un valor a los atributos privados
    public void asignaSoles(double s){
        soles = s;
    }

    public void asignaDolares(double d){
        dolares = d;
    }

    public void asignaMarcos(double m){
        marcos = m;
    }

    //Metodos
    public double calculaMontoEuros(){
        return (soles/3.51+marcos/2.12+dolares)*1.09;
    }
}
```

8. Agregue código al botón **Procesar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
    //Creando el objeto de la clase Casa
    Casa objC = new Casa();

    //Enviando los valores a los métodos públicos
    objC.asignaSoles(Double.parseDouble(txtSoles.getText()));
    objC.asignaDolares(Double.parseDouble(txtDolares.getText()));
    objC.asignaMarcos(Double.parseDouble(txtMarcos.getText()));

    //Realizando los cálculos
    double euros = objC.calculaMontoEuros();

    //Mostrando los resultados en la lista
    DefaultListModel moR = new DefaultListModel();
    moR.addElement('** RESUMEN DE CAMBIO **');
    moR.addElement('-------------------------');
    moR.addElement('MONTO SOLES: ' + String.format('%.2f',
    Double.parseDouble(txtSoles.getText())));
    moR.addElement('MONTO DOLARES: ' + String.format('%.2f',
    Double.parseDouble(txtDolares.getText())));
    moR.addElement('MONTO MARCOS: ' + String.format('%.2f',
    Double.parseDouble(txtMarcos.getText())));
    moR.addElement('-------------------------');
    moR.addElement('EL MONTO TOTAL EN EUROS ES:' + String.format('%.2f', euros));

    lstResumen.setModel(moR);
}
```

9. Agregue código al botón **Limpiar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
    txtSoles.setText('');
    txtDolares.setText('');
    txtMarcos.setText('');
    txtSoles.requestFocus();
}
```

10. Agregue código al botón **Salir**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
    int r = JOptionPane.showOptionDialog(this, 'Estas seguro de salir...?',
                        'Casa de Cambio', JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE, null, null, null);

    if (r == 0) {
        System.exit(0);
    }
}
```

11. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 3: **Casa de cambio (con métodos get y set)**

Una casa de cambio de dinero necesita una aplicación que le permita agilizar sus procesos de consultas sobre el tipo de cambio. Para ello, la casa de cambio usa solo tres tipos de monedas: soles, dólares y marcos, respectivamente. Implemente una aplicación que determine el monto total del dinero en euros de un cliente que necesita realizar el cambio de soles, euros y marcos. Utilice la siguiente tabla de conversión:

1 dólar	3.51 soles
1 dólar	1.09 euros
1 dólar	2.12 marcos

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- Debe ingresar un monto en soles, dólares y marcos, respectivamente.
- Cree una clase llamada **Casa** que defina los atributos referentes a los montos ingresados.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion43**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmCasaCambio**.
5. Agregue una clase llamada **Casa** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE CAMBIOS – CASA DE CAMBIO
2	JTextField	txtSoles
3	JTextField	txtDolares
4	JTextField	txtMarcos
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Casa** coloque el siguiente código:

```
package pClases;

public class Casa {

    //Atributos privados
    private double soles;
    private double dolares;
    private double marcos;

    //Métodos Get y Set
    public double getSoles() {
        return soles;
    }

    public void setSoles(double soles) {
        this.soles = soles;
    }

    public double getDolares() {
        return dolares;
    }

    public void setDolares(double dolares) {
        this.dolares = dolares;
    }

    public double getMarcos() {
        return marcos;
    }
```

```
        public void setMarcos(double marcos) {
            this.marcos = marcos;
        }

        //Método para calcular el monto total en Euros
        public double calculaMontoEuros(){
            return (getSoles()/3.51+getMarcos()/2.12+getDolares())*1.09;
        }
    }
```

8. Agregue código al botón **Procesar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        //Creando el objeto de la clase Casa
        Casa objC = new Casa();

        //Enviando los valores a los atributos públicos
        objC.setSoles(Double.parseDouble(txtSoles.getText()));
        objC.setDolares(Double.parseDouble(txtDolares.getText()));
        objC.setMarcos(Double.parseDouble(txtMarcos.getText()));

        //Realizando los cálculos
        double euros = objC.calculaMontoEuros();

        //Mostrando los resultados en la lista
        DefaultListModel moR = new DefaultListModel();
        moR.addElement('** RESUMEN DE CAMBIO **');
        moR.addElement('————————————————————');
        moR.addElement('MONTO SOLES: ' + String.format('%.2f', objC.getSoles()));
        moR.addElement('MONTO DOLARES: ' + String.format('%.2f', objC.getDolares()));
        moR.addElement('MONTO MARCOS: ' + String.format('%.2f', objC.getMarcos()));
        moR.addElement('————————————————————');
        moR.addElement('EL MONTO TOTAL EN EUROS ES:' + String.format('%.2f', euros));

        lstResumen.setModel(moR);
    }
```

9. Agregue código al botón **Limpiar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtSoles.setText('');
        txtDolares.setText('');
        txtMarcos.setText('');
        txtSoles.requestFocus();
    }
```

10. Agregue código al botón **Salir**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
    int r = JOptionPane.showOptionDialog(this, 'Estas seguro de salir...?',
                        'Casa de Cambio', JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE, null, null, null);
    if (r == 0) {
        System.exit(0);
    }
}
```

11. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 4: Casa de cambio (con método constructor)

Una casa de cambio de dinero necesita una aplicación que le permita agilizar sus procesos de consultas sobre el tipo de cambio. Para ello, la casa de cambio usa solo tres tipos de monedas: soles, dólares y marcos, respectivamente. Implemente una aplicación que determine el monto total del dinero en euros de un cliente que necesita realizar el cambio de soles, euros y marcos. Use la siguiente tabla de conversión:

1 dólar	3.51 soles
1 dólar	1.09 euros
1 dólar	2.12 marcos

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con el método **constructor**.
- Use el siguiente diseño de interfaz gráfica:

- Debe ingresar un monto en soles, dólares y marcos, respectivamente.
- Cree una clase llamada **Casa** que defina los atributos referentes a los montos ingresados.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al Proyecto: **pjAplicacion44**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmCasaCambio**.
5. Agregue una clase llamada **Casa** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE CAMBIOS – CASA DE CAMBIO
2	JTextField	txtSoles
3	JTextField	txtDolares
4	JTextField	txtMarcos
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Casa** coloque el siguiente código:

```
package pClases;
public class Casa {
    //Atributos privados
    private double soles;
    private double dolares;
    private double marcos;

    //Método constructor
    public Casa(double soles, double dolares, double marcos){
        this.soles = soles;
        this.dolares = dolares;
        this.marcos = marcos;
    }
```

```
    //Métodos Get y Set
    public double getSoles() {
        return soles;
    }

    public void setSoles(double soles) {
        this.soles = soles;
    }

    public double getDolares() {
        return dolares;
    }

    public void setDolares(double dolares) {
        this.dolares = dolares;
    }

    public double getMarcos() {
        return marcos;
    }

    public void setMarcos(double marcos) {
        this.marcos = marcos;
    }

    //Método para calcular el monto total en Euros
    public double calculaMontoEuros(){
        return (getSoles()/3.51+getMarcos()/2.12+getDolares())*1.09;
    }
}
```

8. Agregue código al botón **Procesar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {

    //Capturando los valores del formulario
    double soles = Double.parseDouble(txtSoles.getText());
    double dolares = Double.parseDouble(txtDolares.getText());
    double marcos = Double.parseDouble(txtMarcos.getText());

    //Creando el objeto de la clase Casa
    Casa objC = new Casa(soles,dolares,marcos);

    //Realizando los cálculos
    double euros = objC.calculaMontoEuros();

    //Mostrando los resultados en la lista
    DefaultListModel moR = new DefaultListModel();
    moR.addElement('** RESUMEN DE CAMBIO **');
    moR.addElement('-----------------------------');
    moR.addElement('MONTO SOLES: ' + String.format('%.2f', objC.getSoles()));
    moR.addElement('MONTO DÓLARES: ' + String.format('%.2f', objC.getDolares()));
    moR.addElement('MONTO MARCOS: ' + String.format('%.2f', objC.getMarcos()));
    moR.addElement('-----------------------------');
    moR.addElement('EL MONTO TOTAL EN EUROS ES:' + String.format('%.2f', euros));
    lstResumen.setModel(moR);
}
```

9. Agregue código al botón **Limpiar**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
    txtSoles.setText('');
    txtDolares.setText('');
    txtMarcos.setText('');
    txtSoles.requestFocus();
}
```

10. Agregue código al botón **Salir**; para ello, desde la vista de diseño, haga doble clic sobre dicho botón.

```
private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {

    int r - JOptionPane.showOptionDialog(this, 'Estas seguro de salir...?',
                    'Casa de Cambio', JOptionPane.YES_NO_OPTION,
                    JOptionPane.QUESTION_MESSAGE, null, null, null);
    if (r -- 0) {
        System.exit(0);
    }
}
```

11. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

Estructura de secuencia

CAPÍTULO

5

Competencias

- Reconoce los principales pasos para la implementación de estructuras secuenciales.
- Implementa aplicaciones Java usando una estructura secuencial integrada a la programación orientada a objetos.

Contenido

5.1 Generalidades

5.2 Estructura secuencial

5.3 Componentes de una estructura secuencial

- Caso desarrollado 1: Cambio de temperaturas
- Caso desarrollado 2: Conversión de medidas
- Caso desarrollado 3: Pago de empleados
- Caso desarrollado 4: Venta de productos

5.1 Generalidades

El cuerpo de un programa en Java está compuesto por un conjunto de sentencias que tienen por misión realizar una o más acciones según lo especificado por el programador. Estas sentencias se implementan en forma de secuencia dependiendo del objetivo de la aplicación; aquí hay que recalcar que el orden de estas será responsabilidad directa del programador.

También se debe considerar que una sentencia Java hace referencia a un cálculo, captura de datos, impresión de valores, almacenado de datos, etc.; y estos se ejecutan uno a uno siguiendo el orden implementado. En la figura 5.1 obsérvese un ejemplo de una estructura secuencial.

Figura 5.1 Diagrama de flujo de la estructura secuencial.

Como se verá, la secuencia es proporcional al objetivo de la aplicación, ya que para todo programa se deben declarar variables que se usarán en la aplicación. Entre las sentencias o expresiones, se pueden capturar los datos desde un formulario; una vez capturados, se debe empezar con los cálculos necesarios en la aplicación y, finalmente, con la impresión de los resultados. Todo proceso tiene un inicio y un fin.

5.2 Estructura secuencial

La estructura secuencial es aquella en la que una sentencia sigue a otra en secuencia; esto permite enviar las sentencias una a continuación de otra al compilador de Java para que sea interpretado y ejecutado en el mismo orden.

```
                public class ProgramaPrincipal   <------ Inicio del proceso
                {
Secuencia       Promedio objP = new Promedio();
                objP.setAlumno('Ángela Torres Lázaro');
Secuencia       objP.setN1(17);                          Secuencia
                objP.setN1(18);
Secuencia       double promedio = calculaPromedio();     Secuencia
                String condicion = determinaCondicion(promedio);
                }   <------ Fin del proceso
```

Figura 5.2 Secuencia de un proceso.

5.3 Componentes de una estructura secuencial

Cuando se implemente un desarrollo de aplicación, comenzaremos a notar que existen elementos dentro de la programación que son muy parecidos en todas las aplicaciones. Algunos de los componentes que encontraremos son los siguientes:

5.3.1 Declaración

Java es un lenguaje tipado, por lo tanto, todo lo que se pretenda usar dentro de un código de programación debe ser declarado antes de usarse. El formato que permite declarar una variable es:

<TIPO DE DATOS> NOMBRE DE LA VARIABLE;

5.3.2 Asignación

La asignación consiste en asignar un valor a una variable en un determinado momento de la programación, de modo que puede ser usado como un nuevo elemento por la siguiente sentencia. Se puede realizar una asignación en estas situaciones:

- Cuando se asigna un valor a una variable, como, por ejemplo, $n = 15$.
- Cuando se especifica un contador, como, por ejemplo, $c = c + 1$.
- Cuando se especifica un acumulador, como, por ejemplo, $a = a$ + valor.
- Cuando se especifica una expresión, como, por ejemplo, $e = c + b*(2/4)$.

5.3.3 Lectura u obtención de datos

La lectura hace referencia a la forma de capturar un valor ingresado por el usuario, para nuestro caso usamos el control **Jtextfield**. A continuación, se presentan algunos casos de cómo obtener información a partir de controles:

- **Obtener un valor textual desde un cuadro de texto**

```
String descripcion;
descripcion = txtDescripcion.getText();
```

La variable **descripción** fue declarada como **String**, por lo tanto, no necesita ningún tipo de conversión al recibir un valor del mismo, como lo ofrece el control **txtDescripcion**. Por otro lado, el control a un método llamado **getText** permite obtener el valor ingresado en la caja de texto.

- **Obtener un valor numérico entero desde un cuadro de texto**

```
int numero;
numero = Integer.parseInt(txtNumero.getText());
```

La variable **número** fue declarada como entero, por lo que su valor asignado también debe ser entero; sin embargo, se entiende que el control **txtNumero** siempre devolverá un valor textual; por lo tanto, se necesita una conversión de valor usando **Integer.parseInt**, el cual convierte un valor numérico en forma de texto en un valor numérico de tipo **int**.

- **Obtener un valor numérico decimal desde un cuadro de texto**

```
double decimal;
decimal = Double.parseDouble(txtDecimal.getText());
```

La variable **decimal** fue declarada como **double**, por lo que su valor asignado también debe ser **double**; sin embargo, se entiende que el control **txtDecimal** siempre devolverá un valor textual; por lo tanto, se necesita una conversión de valor usando **Double.parseDouble**, el cual convierte un valor numérico decimal en forma de texto en un valor numérico de tipo **double**.

- **Obtener un valor textual desde un cuadro combinado**

```
String opcion;
opcion = String.valueOf(cboOpcion.getSelectedItem());
```

La variable **opción** fue declarada como **String**, por lo que su valor asignado también debe ser **String**; sin embargo, el control **cboOpcion** devuelve un valor textual no compatible con **String**; por lo tanto, se necesita una conversión de valor usando **StringValueOf**, el cual hace compatibles a las variables **String**.

5.3.4 Escritura de datos

La escritura hace referencia a la forma de mostrar un resultado por pantalla; esta se puede realizar de varias maneras. A continuación, se muestran algunos casos de cómo mostrar información:

- **Mostrar información en un control JLabel**

```
double subtotal = 1400;
lblSubtotal.setText('El subtotal es: '+subtotal);
```

El método **setText** del control **lblSubtotal** permite asignar un valor de salida, es decir, mostrará el valor del subtotal de forma estática, sin modificaciones.

- **Mostrar información en un control JTextField**

```
double subtotal = 1400;
txtSubtotal.setText('El subtotal es: '+subtotal);
```

El método **setText** del control **txtSubtotal** permite asignar un valor de salida, es decir, mostrará el valor del subtotal de forma editable, ya que es la naturaleza del control.

- **Mostrar información en un control JList**

```
double subtotal = 1400;
DefaultListModel moR = new DefaultListModel();
moR.addElement('El subtotal es: '+subtotal);
lstResumen.setModel(moR);
```

Para añadir un elemento de impresión o salida en un control **Jlist** primero se debe crear un objeto de la clase **DefaultListModel**. Es a partir de este objeto que se podrá enviar información a la lista. Se comienza por enviar toda la información al modelo usando el método **addElement** y, al final, se envía la información al control **JList** usando el método **setModel**.

CASO DESARROLLADO 1: **Cambio de temperaturas**

Implemente una aplicación Java que permita leer una temperatura en grados Celsius o centígrados (C) y la convierta a su equivalente en grados Fahrenheit (F), grados Kelvin (K) y grados Rankine (R). Utilice las siguientes fórmulas:

Farenheit	9 Celsius / 5 + 32
Kelvin	Rankine – 187
Rankine	Celsius + 460

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- En un cuadro de texto, debe ingresar la temperatura en Celsius.
- Cree una clase llamada **Temperatura** que defina el atributo referente a la temperatura ingresada.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion51**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmComversion**.
5. Agregue una clase llamada **Temperatura** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Figura 5.3 Controles del formulario de conversión.

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONVERSIÓN DE TEMPERATURAS
2	JTextField	txtCelsius
3	JButton	btnProcesar
4	JButton	btnLimpiar
5	JButton	btnSalir
6	JList	lstResumen

7. En la clase **Temperatura** coloque el siguiente código:

```
package pClases;
public class Temperatura {
    //Atributo
    private double celsius;
    //Métodos Get Set
    public double getCelsius() {
        return celsius;
    }
    public void setCelsius(double celsius) {
        this.celsius = celsius;
    }
    //Método de conversión de grados
    public double determinaFarenheit(){
        return (9 * celsius)/5 + 32;
        }
    public double determinaRankine(){
        return celsius + 460;
    }
    public double determinaKelvin(){
        return determinaRankine()-187;
    }
}
```

8. El código del formulario **frmConversion** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Temperatura;

public class frmConversion extends javax.swing.JFrame {
    //Definición del modelo para la lista
    DefaultListModel moR = new DefaultListModel();

    public frmConversion() {
        initComponents();
    }

    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        //Objeto de la clase Temperatura
        Temperatura objT = new Temperatura();
        //Enviando información a la clase
        objT.setCelsius(Double.parseDouble(txtCelsius.getText()));

        //Mostrar el resultado
        moR.addElement('** RESUMEN DE TEMPERATURA **');
        moR.addElement('-------------------------');
        moR.addElement('TEMPERATURA CELSIUS:  ' + String.format('%.2f',
                    objT.getCelsius()));
        moR.addElement('-------------------------');
        moR.addElement('TEMPERATURA FARENHEIT:' + String.format('%.2f',
                    objT.determinaFarenheit()));
        moR.addElement('TEMPERATURA RANKINE:' + String.format('%.2f',
                    objT.determinaRankine()));
        moR.addElement('TEMPERATURA KELVIN:' + String.format('%.2f',
                    objT.determinaKelvin()));
        lstResumen.setModel(moR);
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estas seguro de
                            salir...?', 'Temperatura', JOptionPane.
                            YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE,
                            null, null, null);
        if (r == 0) {
            System.exit(0);
        }
    }

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtCelsius.setText('');
        moR.clear();
        txtCelsius.requestFocus();
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 2: **Conversión de medidas**

Implemente una aplicación Java que permita convertir una cantidad especificada en pulgadas a pies, a yardas, a centímetros y a metros. Utilice las equivalencias de la siguiente tabla:

1 yarda	3 pies
1 pie	12 pulgadas
1 pulgada	2.54 centímetros
1 metro	100 centímetros

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- En un cuadro de texto, debe ingresar el valor en pulgadas.
- Cree una clase llamada **Medidas** que defina el atributo referente a la temperatura ingresada.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion52**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmMedidas**.
5. Agregue una clase llamada **Medidas** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

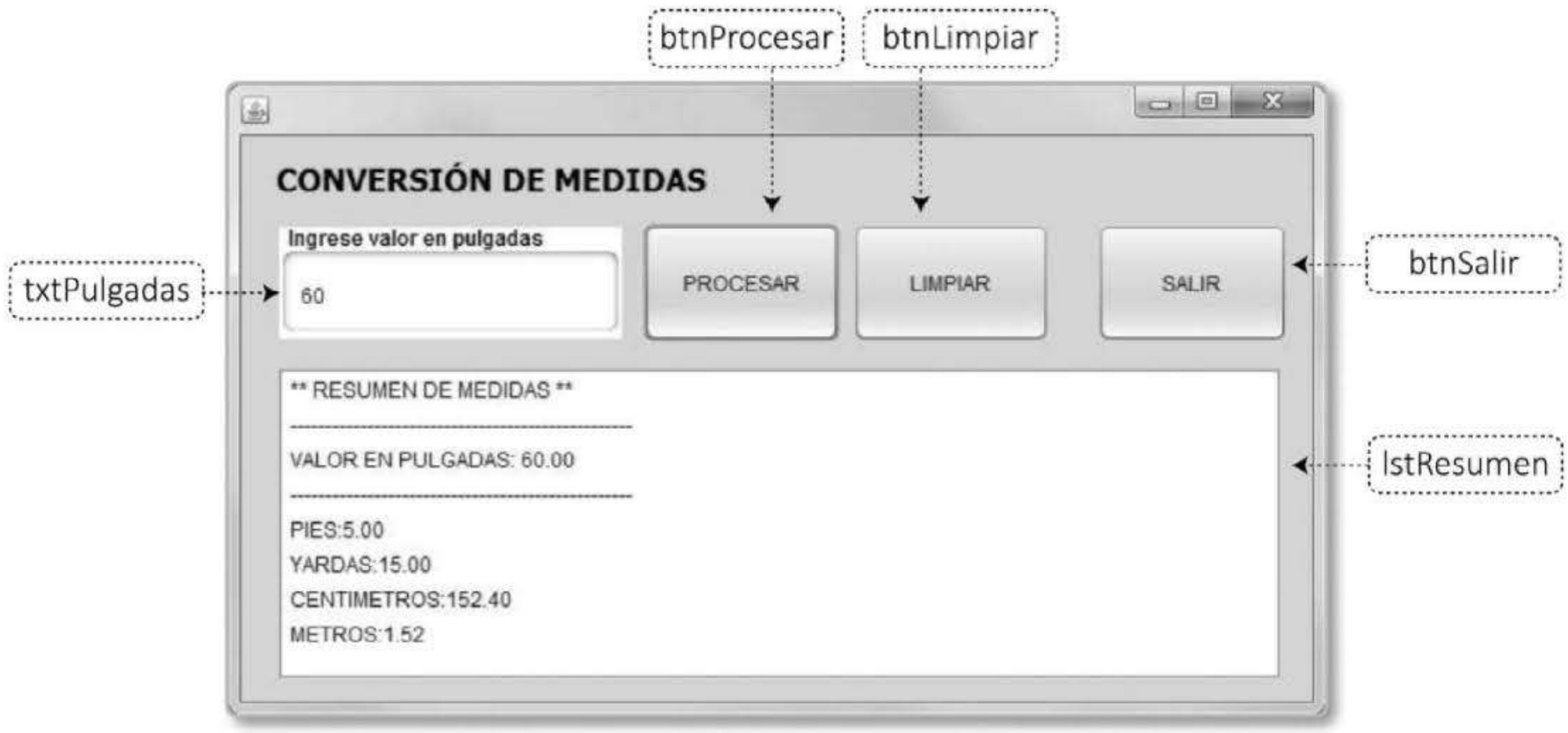


Figura 5.4 Controles del formulario de medidas

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONVERSIÓN DE MEDIDAS
2	JTextField	txtPulgadas
3	JButton	btnProcesar
4	JButton	btnLimpiar
5	JButton	btnSalir
6	JList	lstResumen

7. En la clase **Medidas** coloque el siguiente código:

```
package pClases;

public class Medidas {
    //Atributo
    private double pulgadas;

    //Métodos Get y Set
    public double getPulgadas() {
        return pulgadas;
    }
    public void setPulgadas(double pulgadas) {
        this.pulgadas - pulgadas;
    }

    //Métodos de conversión
    public double determinaPies(){
        return pulgadas/12;
    }
    public double determinaYardas(){
        return 3 * determinaPies();
    }
```

```
        public double determinaCentimetros(){
            return pulgadas*2.54;
        }
        public double determinaMetros(){
            return determinaCentimetros()/100;
        }
    }
```

8. El código del formulario **frmMedidas** se muestra como sigue:

```
package pFormularios;
import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Medidas;
public class frmMedidas extends javax.swing.JFrame {
    //Definición del modelo para la lista
    DefaultListModel moR = new DefaultListModel();
    public frmMedidas() {
        initComponents();
    }
    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        //Objeto de la clase Medidas
        Medidas objM = new Medidas();
        //Enviar el valor en pulgadas a la clase medidas
        objM.setPulgadas(Double.parseDouble(txtPulgadas.getText()));
        //Imprimir los resultados
        moR.addElement('** RESUMEN DE MEDIDAS **');
        moR.addElement('-------------------------');
        moR.addElement('VALOR EN PULGADAS: ' + String.format('%.2f', objM.
            getPulgadas()));
        moR.addElement('-------------------------');
        moR.addElement('PIES:' + String.format('%.2f', objM.
            determinaPies()));
        moR.addElement('YARDAS:'+String.format('%.2f',objM.
            determinaYardas()));
        moR.addElement('CENTIMETROS:'+String.format('%.2f', objM.
            determinaCentimetros()));
        moR.addElement('METROS:'+String.format('%.2f',objM.
            determinaMetros()));
        lstResumen.setModel(moR);
    }
    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtPulgadas.setText('');
        txtPulgadas.requestFocus();
        moR.clear();
    }
    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estas seguro de salir...?',
                                'Medidas', JOptionPane.YES_NO_OPTION,
                                JOptionPane.QUESTION_MESSAGE, null, null, null);
        if (r == 0) {
            System.exit(0);
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 3: **Pago de empleados**

Implemente una aplicación en Java que permita calcular el salario de un empleado. Al respecto se sabe que el empleado cobra un precio fijo por hora y se le descuenta el 12.5% por concepto de impuesto sobre la renta. La aplicación debe pedir el nombre del empleado, las horas trabajadas y el sueldo que cobra por hora. Se pide información sobre el sueldo bruto, descuento de renta y el salario a pagar.

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con el método **constructor**.
- Use el siguiente diseño de interfaz gráfica:

- En cuadros de textos, debe ingresar el nombre del empleado, la cantidad de horas trabajadas y el sueldo por hora.
- Cree una clase llamada **Pago** que defina los atributos referentes a los datos del pago de empleados.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion53**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmPago**.
5. Agregue una clase llamada **Pago** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Figura 5.5 Controles del formulario de pago.

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE PAGO DE EMPLEADOS
2	JTextField	txtEmpleado
3	JTextField	txtHoras
4	JTextField	txtPagoHora
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Pago** coloque el siguiente código:

```
package pClases;

public class Pago {
    //Atributos
    private String empleado;
    private int horas;
    private double pagoHora;

    //Método constructor
    public Pago(String empleado, int horas, double pagoHora) {
        this.empleado = empleado;
        this.horas = horas;
        this.pagoHora = pagoHora;
    }
```

```
    //Métodos Get y Set
    public String getEmpleado() {
        return empleado;
    }

    public void setEmpleado(String empleado) {
        this.empleado = empleado;
    }

    public int getHoras() {
        return horas;
    }

    public void setHoras(int horas) {
        this.horas = horas;
    }

    public double getPagoHora() {
        return pagoHora;
    }

    public void setPagoHora(double pagoHora) {
        this.pagoHora = pagoHora;
    }

    //Métodos propios de la aplicación
    public double calculaSueldoBruto(){
        return pagoHora * horas;
    }

    public double calculaDescuento(){
        return calculaSueldoBruto()*0.125;
    }

    public double calculaSalario(){
        return calculaSueldoBruto()-calculaDescuento();
    }
}
```

8. El código del formulario **frmPago** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Pago;

public class frmPago extends javax.swing.JFrame {

    //Definición del modelo para la lista
    DefaultListModel moR = new DefaultListModel();

    public frmPago() {
        initComponents();
    }

    private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt) {

        //Objeto de la clase Pago
        String empleado = txtEmpleado.getText();
        int horas = Integer.parseInt(txtHoras.getText());
        double pagoHora = Double.parseDouble(txtPagoHora.getText());
```

```
        //Enviando los valores a la clase por medio del método constructor
        Pago objP = new Pago(empleado, horas, pagoHora);

        //Imprimir los resultados
        moR.addElement('** RESUMEN DE PAGO **');
        moR.addElement('-------------------------');
        moR.addElement('EMPLEADO: ' + objP.getEmpleado());
        moR.addElement('HORAS TRABAJADAS: ' + objP.getHoras());
        moR.addElement('PAGO POR HORA: ' + objP.getPagoHora());
        moR.addElement('-------------------------');
        moR.addElement('SALARIO BRUTO:' + String.format('%.2f',
                        objP.calculaSueldoBruto()));
        moR.addElement('DESCUENTO:' + String.format('%.2f',
                        objP.calculaDescuento()));
        moR.addElement('SALARIO:' + String.format('%.2f', objP.
calculaSalario()));
        lstResumen.setModel(moR);
    }

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt)
{
        txtEmpleado.setText('');
        txtHoras.setText('');
        txtPagoHora.setText('');
        txtEmpleado.requestFocus();
        moR.clear();
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt)
{
        int r = JOptionPane.showOptionDialog(this, 'Estas seguro de
        salir...?', 'Pago de empleados', JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE, null, null, null);
        if (r == 0) {
            System.exit(0);
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 4: **Venta de productos**

Implemente una aplicación en Java que permita determinar los montos que se generan de la venta de libros. La aplicación debe permitir seleccionar el producto y una cantidad para determinar los cálculos; asimismo, cuando se selecciona un producto el precio deberá mostrarse de forma automática. Los productos que son parte del proceso de venta son los siguientes:

Descripción del producto	Precio unitario $
Visual Basic 2015	$50.00
Java Fundamentos	$45.50
Aplicaciones web con PHP	$55.00
HTML5 y CSS3	$35.50

Asimismo, véanse los siguientes descuentos según la cantidad comprada:

Cantidad	Porcentaje de descuento
≤ 5	0%
Entre 6 y 10	5%
Entre 11 y 20	10%
> 20	30%

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- Debe seleccionar el producto a comprar e ingresar una determinada cantidad.
- Cree una clase llamada **Venta** que defina los atributos referentes a los datos de la venta de productos.
- Valide todos los valores ingresados mostrando un mensaje de error si fuera necesario.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre del proyecto: **pjAplicacion54**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmVenta**.
5. Agregue una clase llamada **Venta** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

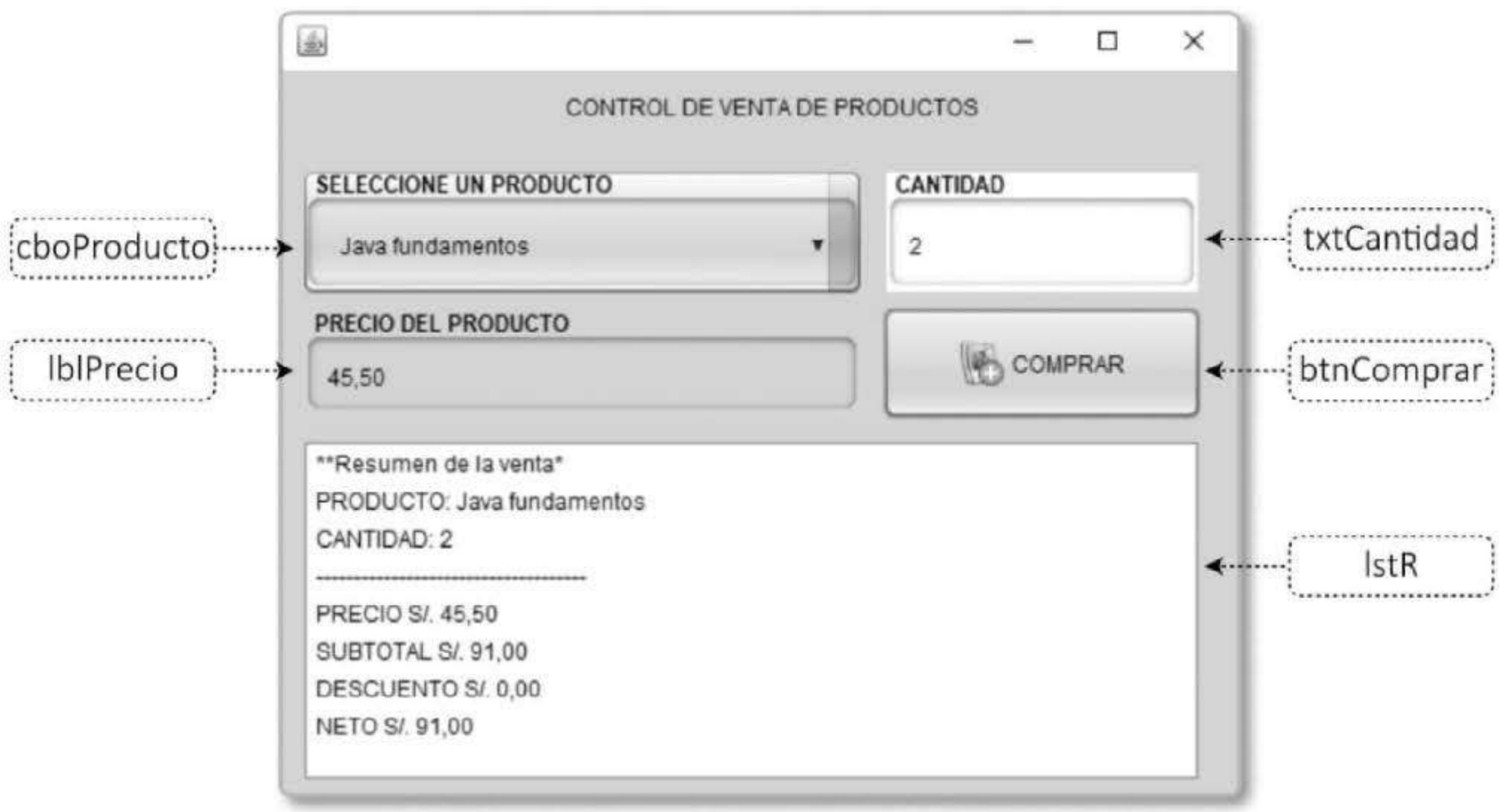


Figura 5.6 Controles del formulario de venta.

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE VENTA DE PRODUCTOS
2	JComboBox	cboProducto
3	JTextField	txtCantidad
4	JLabel	lblPrecio
5	JButton	btnComprar
8	JList	lstR

7. En la clase **Venta** coloque el siguiente código:

```
package pClases;
public class Venta {

    //Atributos de la clase
    private String producto;
    private int cantidad;

    //Encapsulamiento de atributos
    public String getProducto() {
        return producto;
    }

    public void setProducto(String producto) {
        this.producto = producto;
    }
    public int getCantidad() {
        return cantidad;
    }
```

```
        public void setCantidad(int cantidad) {
            this.cantidad = cantidad;
        }

        //Métodos de la clase
        //Determinar el precio de los productos
        public double asignaPrecio() {
            if (producto.equals('Visual Basic 2015')) {
                return 50;
            } else if (getProducto().equals('Java fundamentos')) {
                return 45.50;
            } else if (getProducto().equals('Aplicaciones web con PHP')) {
                return 55;
            } else if (getProducto().equals('HTML5 y CSS3')) {
                return 35.50;
            } else {
                return 0;
            }
        }

        //Calcular el subtotal
        public double calculaSubtotal() {
                                    return getCantidad() * asignaPrecio();
        }

        //Calcular el descuento según la cantidad comprada
        public double calculaDescuento() {
            if (getCantidad() <= 5) {
                return 0;
            } else if (getCantidad() > 5 && getCantidad() <= 10) {
                return calculaSubtotal() * 0.05;
            } else if (getCantidad() > 10 && getCantidad() <= 20) {
                return calculaSubtotal() * 0.10;
            } else {
                return calculaSubtotal() * 0.3;
            }
        }

        //Calcula el neto a pagar por la venta
        public double calculaNeto() {
            return calculaSubtotal()-calculaDescuento();
        }
    }
```

8. El código del formulario **frmVenta** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Venta;

public class frmVenta extends javax.swing.JFrame {

    //Creando un objeto de la clase Venta
    //Clase Objeto Método constructor
    Venta objVenta = new Venta();
    public frmVenta() {
        initComponents();
        llenaProductos();
    }
```

```
        //Metodo que llena los productos en el combo
        void llenaProductos() {
            cboProducto.addItem('Visual Basic 2015');
            cboProducto.addItem('Java fundamentos');
            cboProducto.addItem('Aplicaciones web con PHP');
            cboProducto.addItem('HTML5 y CSS3');
        }

        private void btnComprarActionPerformed(java.awt.event.ActionEvent evt) {
            //Enviando los valores a la clase Venta
            if (valida().equals('')) {
                objVenta.setProducto(String.valueOf(cboProducto.
                                    getSelectedItem()));
                objVenta.setCantidad(Integer.parseInt(txtCantidad.getText()));

                //Obtener los valores resultantes desde la clase
                double precio = objVenta.asignaPrecio();
                double subtotal = objVenta.calculaSubtotal();
                double descuento = objVenta.calculaDescuento();
                double neto = objVenta.calculaNeto();

                //Imprimir
                DefaultListModel moR = new DefaultListModel();
                moR.clear();
                moR.addElement('**Resumen de la venta*');
                moR.addElement('PRODUCTO: ' + objVenta.getProducto());
                moR.addElement('CANTIDAD: ' + objVenta.getCantidad());
                moR.addElement('-------------------');
                moR.addElement('PRECIO S/. ' + String.format('%.2f', precio));
                moR.addElement('SUBTOTAL S/. ' + String.format('%.2f', subtotal));
                moR.addElement('DESCUENTO S/. ' + String.format('%.2f', descuento));
                moR.addElement('NETO S/. ' + String.format('%.2f', neto));
                lstR.setModel(moR);
            } else {
                JOptionPane.showMessageDialog(null, 'El error está en ' +
                                              valida());
            }
        }

        //Método que muestra el precio al seleccionar un producto
        private void cboProductoItemStateChanged(java.awt.event.ItemEvent evt) {
            String producto = String.valueOf(cboProducto.getSelectedItem());
            objVenta.setProducto(producto);
            double precio = objVenta.asignaPrecio();
            lblPrecio.setText('' + String.format('%.2f', precio));
        }

        //Método que valida el ingreso de valores
        String valida() {
            if (txtCantidad.getText().equals('')) {
                return 'cantidad de producto';
            } else if (Integer.parseInt(txtCantidad.getText()) < 0) {
                return 'el valor de la cantidad';
            } else {
                return '';
            }
        }
    }
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CAPÍTULO

6

Estructura de selección

Competencias

- Implementa aplicaciones usando la sentencia **if** de forma simple, doble y doblemente enlazada.
- Implementa aplicaciones usando la sentencia **switch**.

Contenido

6.1 Generalidades
6.2 Operadores de relación
6.3 Operadores lógicos
6.4 Instrucción **if** simple
6.5 Instrucción **if** doble
6.6 Instrucción **if** doblemente enlazada
6.7 Sentencia **switch**
6.8 Implementar una condición lógica
- Caso desarrollado 1: Renta de autos (usando **if** doblemente enlazado)
- Caso desarrollado 2: Pago de trabajadores (usando **if** simple)
- Caso desarrollado 3: Control de hectáreas de algodón y maíz (usando **if** doble)
- Caso desarrollado 4: Control de consumo de agua (usando **if** doblemente enlazado)
- Caso desarrollado 5: Control de pensión de estudiantes (usando **switch**)

6.1 Generalidades

Las aplicaciones que hemos desarrollado hasta el momento se enfocan en la programación secuencial, es decir, una sentencia se ejecuta después de otra. La ejecución comienza con la primera sentencia de la función y prosigue hasta la última. Este estilo de programación es adecuado para resolver problemas sencillos; pero, conforme se va avanzando en el tema de la programación y el desarrollo de las aplicaciones, necesitaremos de las sentencias selectivas como elemento estratégico para la obtención de resultados lógicos en una aplicación.

Por otro lado, cabe mencionar que la aplicación con desarrollo secuencial es de gran utilidad cuando la aplicación así lo requiera; pero, en determinado momento, la aplicación requerirá un número de posibles alternativas de solución en lugar de un caso particular.

De allí se desprende las estructuras de decisión o también llamadas de selección, alternativas o condicionales, que se utilizan para tomar decisiones lógicas mediante la evaluación de una condición, donde, luego del resultado, se realiza una u otra instrucción en la aplicación.

La implementación de una aplicación conlleva a un flujo de ejecución de sentencias, las cuales se dividen en tres grandes categorías en función del flujo de ejecución: las secuenciales, vistas en el capítulo anterior, las selectivas y las repetitivas. En este capítulo se hará referencia al uso de las estructuras selectivas o condicionales usando **if** y **switch** en todas sus formas y, asimismo, integrándolas a la programación orientada a objetos.

6.2 Operadores de relación

Es importante reconocer los operadores de relación porque estos componen una condición dentro de la estructura selectiva, ya que es por aquí donde la estructura fluye durante el proceso de la aplicación.

A continuación, se muestra la lista operadores válidos para Java:

Operador	Definición	Ejemplo

>	Mayor que	`double sueldo - 4500;`
		`if (sueldo > 4500)`
		Se declara la variable **sueldo** con un valor inicial de 4500, que luego será condicionado con la estructura selectiva **if** simple, comparando si el valor asignado a sueldo es superior a 4500.
		`int edad - 23;`
		`if (edad>17)`
		Se declara la variable **edad** con un valor inicial de 23, que luego será condicionado con la estructura selectiva **if** simple, comparando si la edad es superior a 17.

Operador	Definición	Ejemplo

>=	Mayor o igual que	`double sueldo -5000;`
		`if (sueldo >- 4500)`
		Se declara la variable **sueldo** con un valor inicial de 5000, que luego será condicionado con la estructura selectiva **if** simple, comparando si el valor asignado a sueldo es superior o igual a 4500.
		`int edad - 23;`
		`if (edad >- 18)`
		Se declara la variable **edad** con un valor inicial de 23, que luego será condicionado con la estructura selectiva **if** simple, comparando si la edad es superior o igual a 18.

<	Menor que	`double precio - 50;`
		`if (precio < 20)`
		Se declara la variable **precio** con un valor inicial de 50, que luego será condicionado con la estructura selectiva **if** simple, comparando si dicho precio es inferior a 20.
		`double promedio - 17.10;`
		`if (promedio < 12.50)`
		Se declara la variable **promedio** con un valor inicial de 17.10, que luego será condicionado con la estructura selectiva **if** simple, comparando si el promedio es menor a 12.50.

<=	Menor o igual que	`double precio - 50;`
		`if (precio <- 19.99)`
		Se declara la variable **precio** con un valor inicial de 50, que luego será condicionado con la estructura selectiva **if** simple, comparando si dicho precio es menor o igual a 19.99.
		`double promedio - 17.10;`
		`if (promedio <- 12.49)`
		Se declara la variable **promedio** con un valor inicial de 17.10, que luego será condicionado con la estructura selectiva **if** simple, comparando si el promedio es menor o igual a 12.49.

==	Igual	`String categoria - 'Bebidas';`
		`if (categoría --'Golosinas')`
		Se declara la variable **categoría** con un valor inicial de **Bebidas**, que luego será condicionado para comparar si el valor ingresado es **Golosinas**; en este caso, el resultado es **false**. Otra solución podría ser de la siguiente manera:
		`String categoria - 'Bebidas';`
		`if (categoria.equals('Golosinas'))`
		`int cantidad - 20;`
		`if (cantidad -- 20)`
		El valor asignado a la variable **cantidad** se compara en la estructura condicional; en este caso, el resultado es **true**.

Operador	Definición	Ejemplo

!=	Diferente o no es igual	`String categoria = 'Operario';`
		`if (categoria != 'Operario')`
		La variable **categoría** se verifica si es diferente a su mismo valor; en este caso, el resultado es **false** porque el valor sí es operario.

?	Condicional	`double nota = 20;`
		`String condición = nota < 12.5 ? 'Desaprobado': 'Aprobado';`
		El código determina si la nota obtenida por un alumno en una evaluación es aprobada o no. Se debe tener en cuenta que la nota mínima aprobatoria es 12.5, mostrando el mensaje **Aprobado** porque el valor 20 es superior a 12.5.

6.3 Operadores lógicos

Otro factor importante dentro de la condición de una estructura condicional es el uso de los operadores lógicos, los cuales permiten unir dos o más condiciones en una sola unidad lógica, es decir, el resultado siempre será **true** o **false**.

Operador	Definición	Ejemplo

&&	Representa la Y lógica, el cual evalúa que ambas condiciones sean de tipo **true** para realizar una determinada acción.	`double sueldo = 1000;`
		`if (sueldo>=1000 && sueldo<=1500)`
		El código determina si los empleados de una empresa ganan entre $1000.00 y $1500.00

||	Representa la O lógica, el cual ejecuta una determinada acción solo si una de las condiciones es de tipo **true**.	`String categoria = 'Jefe';`
		`if (categoria == 'Jefe' || categoria == 'Gerente')`
		El código determina si los empleados de una empresa tienen la categoría **Jefe** o **Gerente**.

!	Representa la negación del resultado de una condición.	`String categoria = 'Jefe';`
		`if (!(categoria == 'Jefe' || categoria == 'Gerente'))`
		El código determina si los empleados de una empresa no son de tipo **Jefe** ni **Gerente**.

6.4 Instrucción if simple

Si se tiene en cuenta que toda evaluación emite como resultado un valor lógico, ya sea **true** o **false**, cuando la instrucción es simple solo evalúa uno de los dos resultados, pero no los dos; eso quiere decir que solo se tomará interés en los valores de un extremo de la condición.

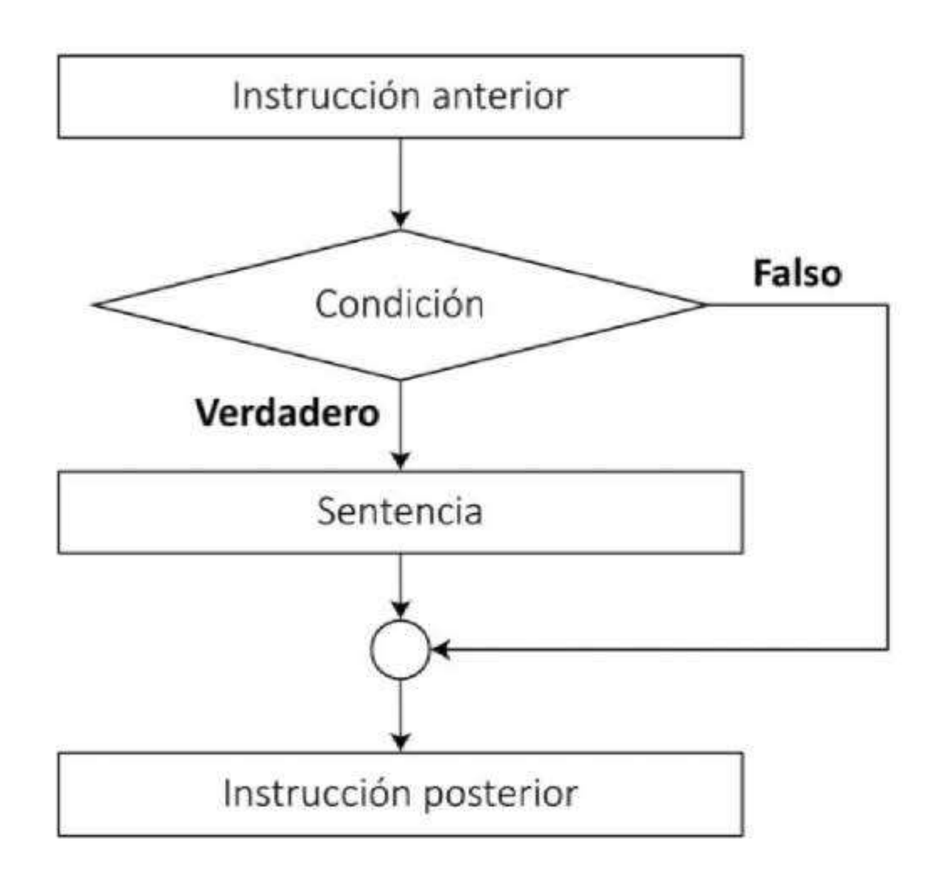
Obsérvese los siguientes formatos:

Formato 1:

```
if (condición) sentencia;
```

Formato 2:

```
if (condición) {
    sentencia1;
    sentencia2;
}
```



Donde:

- **Condición.** Es la expresión que determina las acciones a realizar si el resultado de esta devuelve **true** o verdadero. Asimismo, hay que considerar que es aquí donde se usan los operadores lógicos y relacionales.
- **Sentencia.** Es la expresión que se ejecutará si y solo si la condición evaluada es verdadera. Se debe tener en cuenta que, si se trata de una sola sentencia, no será necesario el uso de la pareja de llaves como se observa en el formato uno. Si se implementa más de dos sentencias, obligatoriamente deben estar encerrados entre llaves, tal como se visualiza en el formato dos.

6.5 Instrucción if doble

Se le denomina doble cuando se especifica sentencias en ambos lados de la condición, es decir, habrá código cuando el resultado de la evaluación sea **true** y también cuando sea **false**. Se debe considerar que, si son más de dos sentencias o instrucciones, deben estar encerradas entre llaves.

Obsérvese los siguientes formatos:

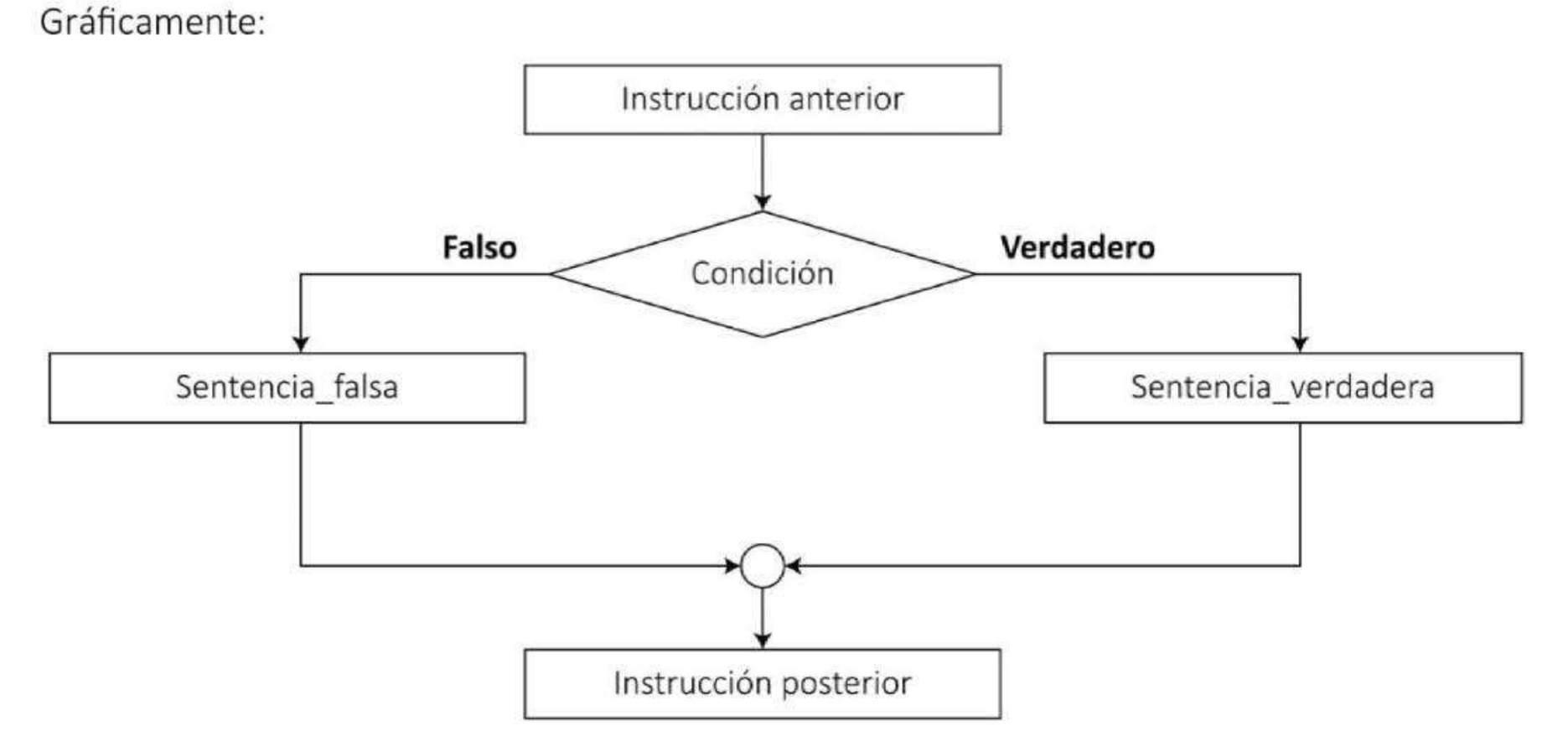
Primer formato:

```
if (condición) sentencia_verdadera; else sentencia_falsa;
```

Segundo formato:

```
if (condición) {
sentencia_verdadera1;
    sentencia_verdadera2;
}
else {
    sentencia_falsa1;
    sentencia_falsa2;
}
```

Gráficamente:



Donde:

- **Condición.** Es la expresión que determina las acciones a realizar si el resultado de esta devuelve **true** o **false**. Asimismo, hay que considerar que es aquí donde se usan los operadores lógicos y relacionales.
- **Sentencia_verdadera.** Son sentencias que se ejecutarán solo si la condición evaluada resulta **true**.
- **Sentencia_falsa.** Son sentencias que se ejecutarán solo si la condición evaluada resulta **false**.

6.6 Instrucción if doblemente enlazada

Se le llama doblemente enlazada cuando la condición es evaluada y el resultado **true** ejecuta un conjunto de sentencias, mientras que el resultado **false** implementa nuevamente una nueva evaluación dentro de la condición. Es como especificar condiciones sucesivas hasta que se encuentre un resultado adecuado.

Obsérvese los siguientes formatos:

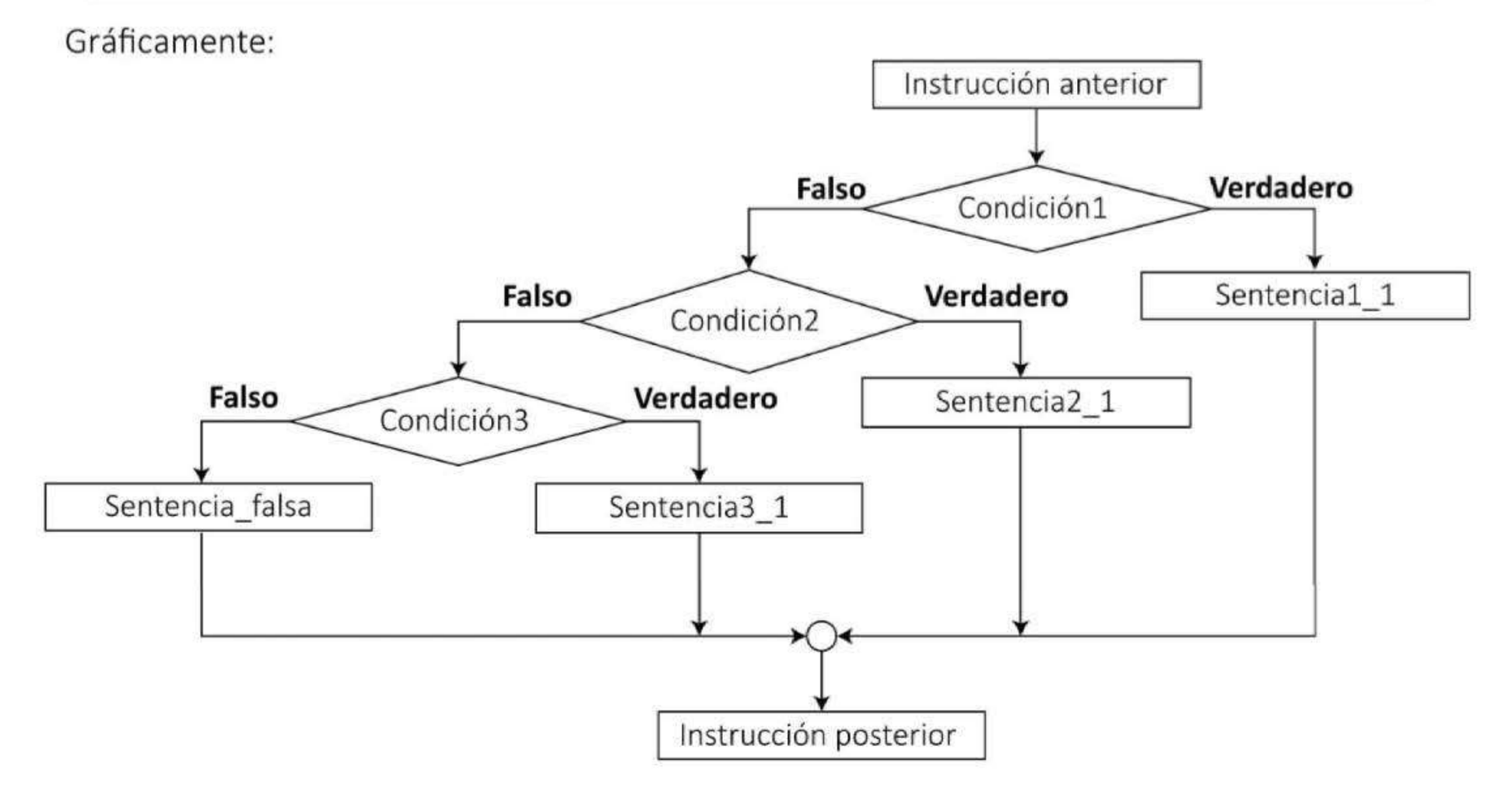
Primera forma:

```
if (condición1)
    sentencia1_1;
else if (condición2)
    sentencia2_1;
else if (condición3)
    sentencia3_1;
else
    sentencia_falsa;
```

Segunda forma:

```
if (condición1){
    sentencia1_1;
    sentencia1_2;
}
else if (condición2) {
    sentencia2_1;
    sentencia2_2;
}
else if (condición3) {
    sentencia3_1;
    sentencia3_2;
}
else{
    sentencia_Falsa1;
    sentencia_Falsa2;
}
```

Gráficamente:



Donde:

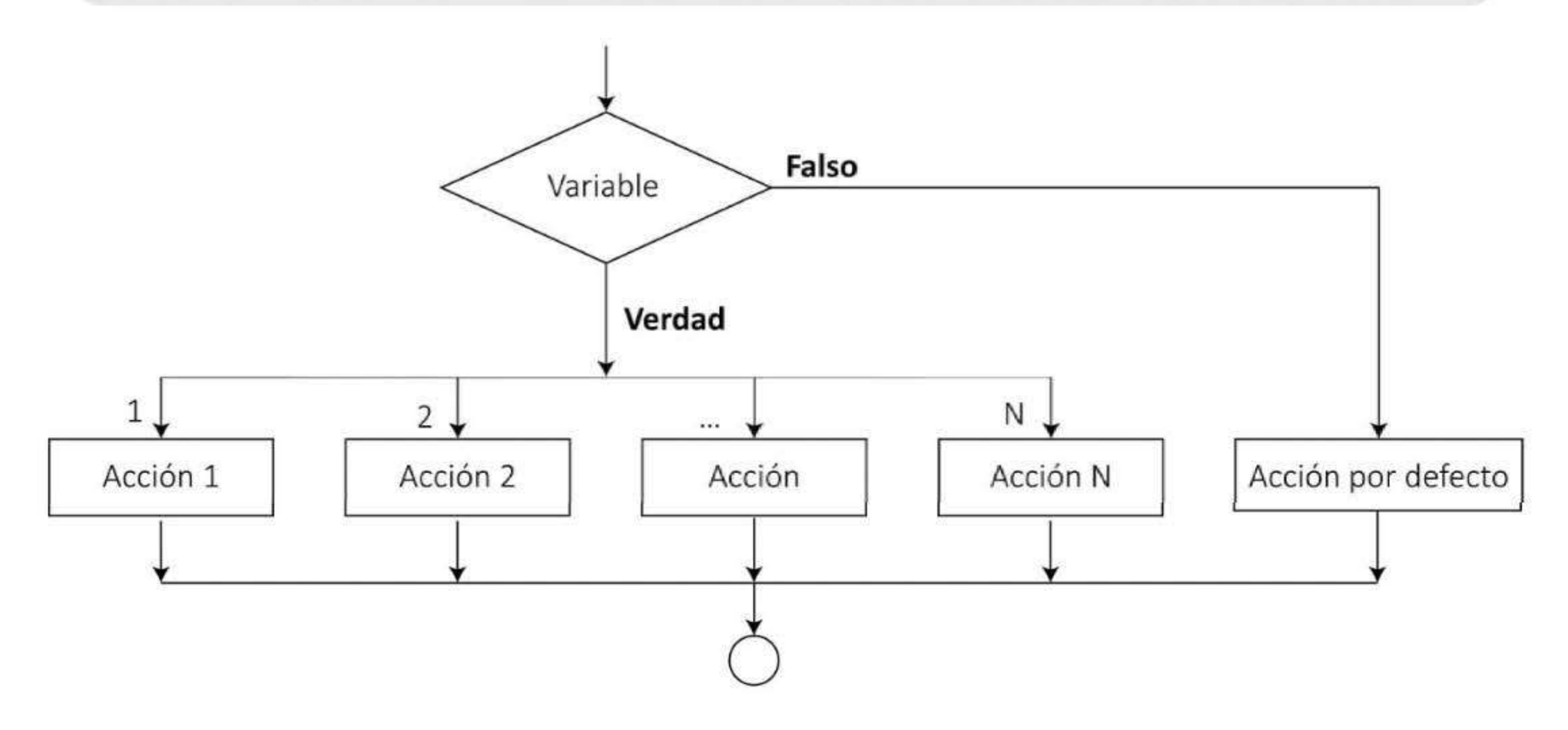
- **Condición1, Condición2, Condición3.** Determina la cantidad de evaluaciones por la que puede pasar una variable, es decir, las posibles opciones que puede contener.
- **Sentencia.** Representa la o las sentencias que se ejecutarán única y exclusivamente si la condición evaluada es verdadera; en caso contrario, seguirá evaluando hasta que la condición sea verdadera.
- **Sentencia_falsa.** Representa la o las sentencias que se ejecutarán cuando ninguna de las condiciones resulta verdadera; también se les denomina sentencias ejecutadas por defecto.

6.7 Sentencia switch

La sentencia **switch** en una aplicación Java es una instrucción de decisión múltiple, es decir, el compilador busca el valor contenido en una variable contra una lista de constantes definidos por el desarrollador, que pueden ser de tipo **int** o **char**. Cuando el compilador encuentra el valor de igualdad entre la variable y la constante, se ejecutará un grupo de instrucciones asociadas a dicha constante; si no encuentra el valor de igualdad, entonces se ejecutarán instrucciones asociadas a un **default**.

Formato:

```
switch(variable){
    case valor1: acción1; break;
    case valor2: acción2; break;
    …
    case valorN: acciónN; break;
    default: acción x defecto
}
```



Una sentencia **switch** se ejecuta de la siguiente forma:

- Se evalúa la variable especificada y se compara con cada una de las constantes definidas por el programador.
- Si la variable es igual a la constante definida, entonces se ejecutan las acciones definidas para dicha constante y se termina la instrucción usando la función break.
- No hay necesidad de especificar llaves por cada bloque de instrucciones.

¿Cómo implementar un switch?

Al implementar una instrucción **switch**, se debe considerar:

- ¿Qué tipo de dato se va a evaluar?
- ¿Cuántas constantes se implementarán?
- ¿Qué acciones se ejecutarán por cada caso?

6.8 Implementar una condición lógica

Es importante determinar una buena condición dentro de un proceso lógico, pues esto llevará a los mejores resultados dentro de la aplicación.

A continuación, se presentan algunos casos de implementación de métodos haciendo uso de la estructura **if**:

Para las condiciones if simples

- Determinar el nombre de la categoría de un producto según la siguiente tabla:

Categoría	Descripción
A	Golosinas
B	Lácteos
C	Bebidas
D	Abarrotes

```
String categoria, descripcion;
categoria = 'B';

if (categoria.equals('A')) descripcion = 'Golosinas';
if (categoria.equals('B')) descripcion = 'Lácteos';
if (categoria.equals('C')) descripcion = 'Bebidas';
if (categoria.equals('D')) descripcion = 'Abarrotes';
```

- Aplicar la siguiente tabla de descuentos con respecto al monto a pagar por la compra de productos en una tienda comercial:

Monto a pagar	Porcentaje de descuento
Menor o igual a $100.00	5.0
Superior a $100.00 e inferior a $200.00	10.75
Superior a $200.00 e inferior a $300.00	12.50
Superior a $300.00	14.65

```
double monto, descuento=0;
sueldo = 1750.00;

if (sueldo <= 100)                    descuento = monto * (5.0/100);
if (sueldo > 100 && sueldo <= 200)    descuento = monto * (10.75/100);
if (sueldo > 200 && sueldo <= 300)    descuento = monto * (12.5/100);
if (sueldo > 300)                     descuento = monto * (14.65/100);
```

- Aplicar la siguiente tabla de asignación de costo hora y descuentos con respecto al alquiler de cochera según el tipo de auto:

Tipo de auto	Costo hora	Descuento
Automóvil	$5.00	12%
Camioneta	$3.00	10%
Ómnibus	$2.00	11%
Motocicleta	$1.00	9%

```
String tipo;
double costoHora, descuento;

if (tipo.equals('Automovil')) {
    costoHora = 5;
    descuento = 0.12;
}
if (tipo.equals('Camioneta')) {
    costoHora = 3;
    descuento = 0.1;
}
if (tipo.equals('Omnibus')) {
    costoHora = 2;
    descuento = 0.11;
}
if (tipo.equals('Motocicleta')) {
    costoHora = 1;
    descuento = 0.09;
}
```

Para las condiciones if dobles

- Determinar los siguientes mensajes según la realización de los pagos de cuotas en una universidad, de acuerdo a la siguiente tabla:

Condición	Mensaje
PAGÓ	"Estimado estudiante, usted se encuentra al día en sus pagos".
NO PAGÓ	"Estimado estudiante, recuerde que el retraso en sus cuotas genera moras".

```
String condicion, mensaje;

if (condicion.equals('PAGÓ'))
    mensaje='Estimado estudiante usted se encuentre al dia, en sus pagos';
else
    mensaje='Estimado estudiante recuerde que el retraso en sus cuotas
    generan moras';
```

- Aplicar descuentos de tipo AFP al importe de sueldo usando la siguiente tabla:

Tipo de AFP	Descuento 1	Descuento 2
Integra	10.0%	2.0%
Prima	11.0%	1.75%

```
double sueldo, descuento1, descuento2;
String tipoAFP;

if (tipoAFP.equals('Integra')){
    descuento1 = sueldo * (10.0/100);
    descuento2 = sueldo * (2.0/100);
} else {
    descuento1 = sueldo * (11.0/100);
    descuento2 = sueldo * (1.75/100);
}
```

Para las condiciones if doblemente enlazadas

- Determinar el nombre de la categoría de un empleado según la siguiente tabla:

Categoría	Descripción
A	Jefe
B	Administrativo
C	Apoyo
D	Operario

```
String categoria, descripcion='';

if (categoria.equals('A'))
    descripcion = 'Jefe';
else if (categoria.equals('B'))
    descripcion = 'Administrativo';
else ir (categoria.equals('C'))
    descripcion = 'Apoyo';
else if (categoria.equals('D'))
    descripcion = 'Operario';
```

Para las condiciones if dobles

- Determinar los siguientes mensajes según la realización de los pagos de cuotas en una universidad, de acuerdo a la siguiente tabla:

Condición	Mensaje
PAGÓ	"Estimado estudiante, usted se encuentra al día en sus pagos".
NO PAGÓ	"Estimado estudiante, recuerde que el retraso en sus cuotas genera moras".

```
String condicion, mensaje;

if (condicion.equals('PAGÓ'))
    mensaje='Estimado estudiante usted se encuentre al dia, en sus pagos';
else
    mensaje='Estimado estudiante recuerde que el retraso en sus cuotas
    generan moras';
```

- Aplicar descuentos de tipo AFP al importe de sueldo usando la siguiente tabla:

Tipo de AFP	Descuento 1	Descuento 2
Integra	10.0%	2.0%
Prima	11.0%	1.75%

```
double sueldo, descuento1, descuento2;
String tipoAFP;

if (tipoAFP.equals('Integra')){
    descuento1 = sueldo * (10.0/100);
    descuento2 = sueldo * (2.0/100);
} else {
    descuento1 = sueldo * (11.0/100);
    descuento2 = sueldo * (1.75/100);
}
```

Para las condiciones if doblemente enlazadas

- Determinar el nombre de la categoría de un empleado según la siguiente tabla:

Categoría	Descripción
A	Jefe
B	Administrativo
C	Apoyo
D	Operario

```
String categoria, descripcion='';

if (categoria.equals('A'))
    descripcion = 'Jefe';
else if (categoria.equals('B'))
    descripcion = 'Administrativo';
else ir (categoria.equals('C'))
    descripcion = 'Apoyo';
else if (categoria.equals('D'))
    descripcion = 'Operario';
```

- Aplicar la siguiente tabla de descuentos con respecto al sueldo de los empleados en una empresa:

Sueldo	Porcentaje de descuento
Menor o igual a $750.00	7.50
Superior a $750.00 e inferior a $1400.00	12.50
Superior a $1400.00 e inferior a $2100.00	14.50
Superior a $2100.00	16.50

```
double sueldo, descuento-0;

if (sueldo <- 750)
    descuento - sueldo * (7.5/100);
else if (sueldo > 750 && sueldo <- 1400)
    descuento - sueldo * (12.5/100);
else if (sueldo > 1400 && sueldo <- 2100)
    descuento - sueldo * (14.5/100);
else if (sueldo > 2100)
    descuento - sueldo * (16.5/100);
```

- Aplicar la siguiente tabla de asignación de costo hora y descuentos con respecto a la categoría de los empleados en una empresa:

Categoría	Costo hora	Descuento
Jefe	$50.00	18%
Administrativo	$30.00	15%
Apoyo	$20.00	12%
Operario	$7.50	10%

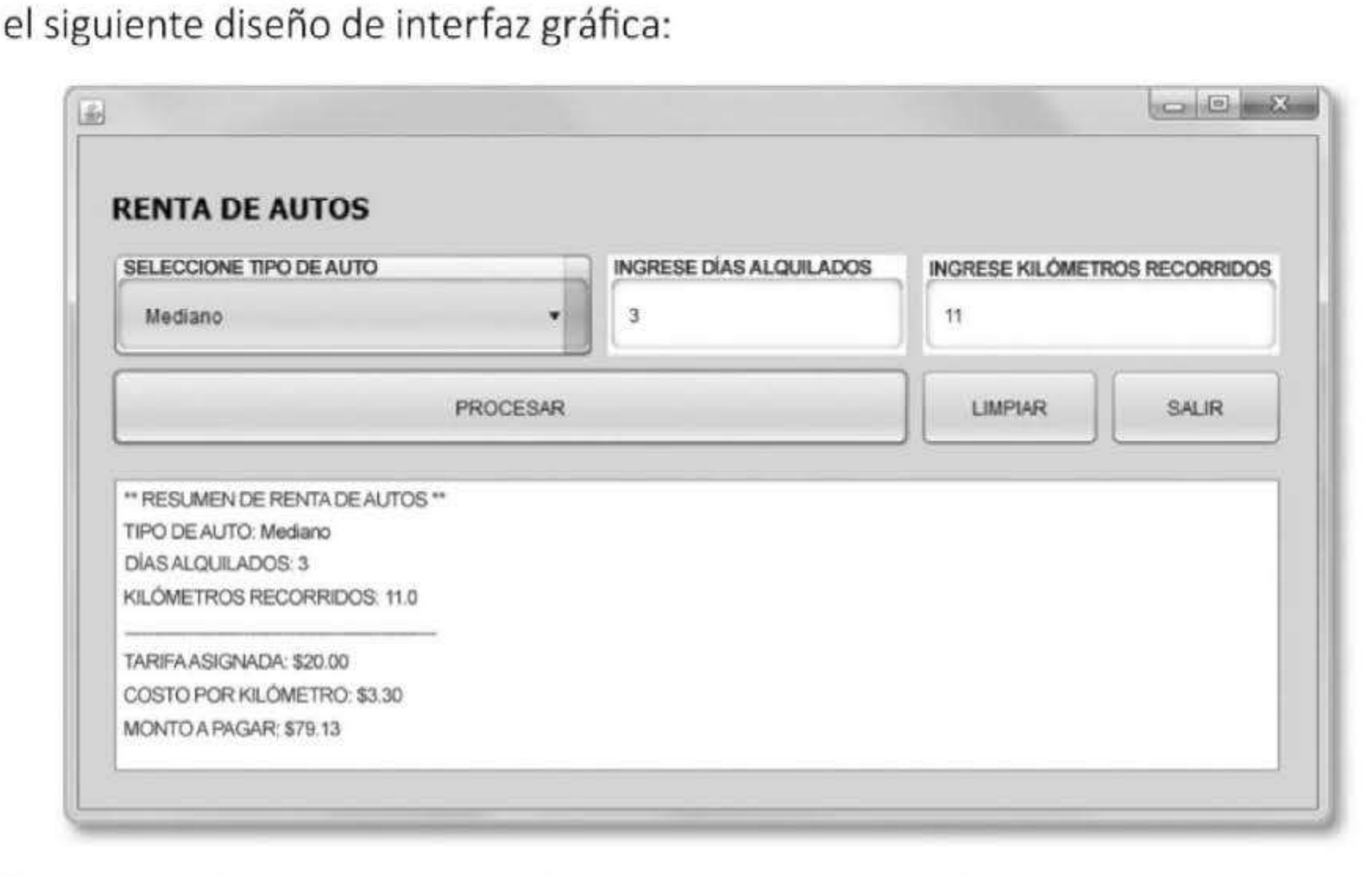
```
String categoria;
double costoHora, descuento;

if (categoria.equals('Jefe'))       {
    costoHora - 50;
    descuento - 0.18;
} else if (categoria.equals('Administrativo')) {
    costoHora - 30;
    descuento - 0.15;
} else if (categoria.equals('Apoyo')) {
    costoHora - 20;
    descuento - 0.12;
} else if (categoria.equals('Operario'))    {
    costoHora - 7.5;
    descuento - 0.1;
}
```

CASO DESARROLLADO 1: Renta de autos (usando if doblemente enlazado)

Una empresa renta autos de tres tipos: pequeños, medianos y grandes. La tarifa del alquiler por día es $ 15 en el auto pequeño, $ 20 en el auto mediano y $ 30 en el auto grande. Además, la empresa cobra $ 0.20/km recorrido en el auto pequeño, $ 0.30/km recorrido en el auto mediano y $ 0.40/km recorrido en el auto grande. Si la cantidad de kilómetros recorridos por el auto supera los 10 km, se le aumentará un 2.5 % sobre el monto a pagar por el cliente.

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:



- Desde un control cuadro combinado, debe seleccionarse el tipo de auto.
- Debe ingresar el número de días alquilados y el valor de kilómetros recorridos en cuadros de textos.
- Cree una clase llamada **Renta** que defina los atributos referentes a los datos de la renta de autos.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion61**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmRenta**.
5. Agregue una clase llamada **Renta** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

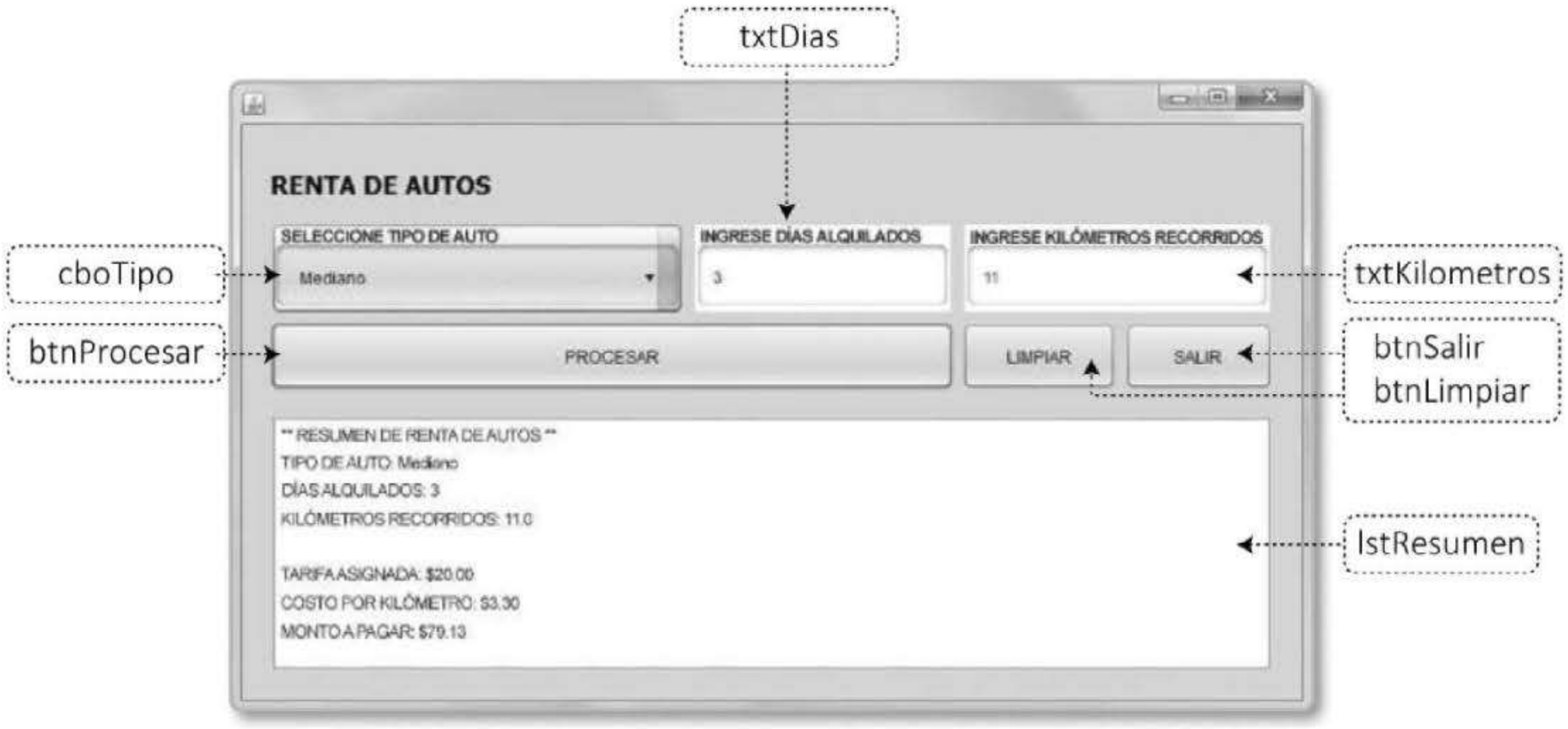


Figura 6.1 Controles del formulario de renta de autos.

Resumen de controles:

N.º	Clase	Control
1	JLabel	RENTA DE AUTOS
2	JComboBox	cboTipo
3	JTextField	txtDias
4	JTextField	txtKilometros
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Renta** coloque el siguiente código:

```
package pClases;

public class Renta {
    //Atributos
    private String tipo;
    private int dias;
    private double kilometros;

    //Métodos Get y Set
    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
```

```
        public int getDias() {
            return dias;
        }

        public void setDias(int dias) {
            this.dias = dias;
        }

        public double getKilometros() {
            return kilometros;
        }

        public void setKilometros(double kilometros) {
            this.kilometros = kilometros;
        }

        //Métodos propios de la aplicación
        public double determinaTarifa() {
            double tarifa = 0.0;
            if (tipo.equals('Pequeño')) {
                tarifa = 15;
            } else if (tipo.equals('Mediano')) {
                tarifa = 20;
            } else if (tipo.equals('Grande')) {
                tarifa = 30;
            } else {
                tarifa = 0;
            }
            return tarifa;
        }

        public double calculaCosto() {
            if (tipo.equals('Pequeño')) {
                return 0.2 * kilometros;
            } else if (tipo.equals('Mediano')) {
                return 0.3 * kilometros;
            } else if (tipo.equals('Grande')) {
                return 0.4 * kilometros;
            } else {
                return 0;
            }
        }

        public double calculaMonto() {
            double monto = (determinaTarifa() * dias) + calculaCosto();
            double aumento = 0;
            if (kilometros> 10) {
                aumento = monto * (25 / 100.0);
            }
            monto = monto + aumento;
            return monto;
        }
    }
```

8. El código del formulario **frmRenta** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import pClases.Renta;

public class frmRenta extends javax.swing.JFrame {

    //Definición del modelo
    DefaultListModel moR = new DefaultListModel();

    public frmRenta() {
        initComponents();
        llenaTipoAuto();
    }

    //Método para llenar el cuadro combinado
    void llenaTipoAuto(){
        cboTipo.addItem('Pequeño');
        cboTipo.addItem('Mediano');
        cboTipo.addItem('Grande');
    }

    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        //Objeto de la clase Renta
        Renta objR = new Renta();

        //Enviando los valores a la clase
        objR.setTipo(String.valueOf(cboTipo.getSelectedItem()));
        objR.setDias(Integer.parseInt(txtDias.getText()));
        objR.setKilometros(Double.parseDouble(txtKilometros.getText()));

        //Recuperando los valores a imprimir
        double tarifa = objR.determinaTarifa();
        double costo = objR.calculaCosto();
        double monto = objR.calculaMonto();

        //Imprimir
        moR.addElement('** RESUMEN DE RENTA DE AUTOS **');
        moR.addElement('TIPO DE AUTO: '+objR.getTipo());
        moR.addElement('DÍAS ALQUILADOS: '+objR.getDias());
        moR.addElement('KILÓMETROS RECORRIDOS: '+objR.getKilometros());
        moR.addElement('————————————————————');
        moR.addElement('TARIFA ASIGNADA: $' + String.format('%.2f',tarifa));
        moR.addElement('COSTO POR KILÓMETROS: $' +
                       String.format('%.2f',costo));
        moR.addElement('MONTO A PAGAR: $' + String.format('%.2f',monto));
        lstResumen.setModel(moR);
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 2: **Pago de trabajadores (usando if simple)**

Determine la cantidad de dinero que recibirá un trabajador por concepto de las horas extras trabajadas en una empresa. Al respecto se sabe que cuando las horas de trabajo exceden de 40, el resto se considera horas extras y estas se pagan el doble de una hora normal cuando no exceden de 8; si las horas extras exceden de 8, se pagan las primeras 8 el doble de lo que se paga por una hora normal, y el resto se paga el triple.

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- Debe ingresar el nombre del trabajador, las horas de trabajo y la tarifa por hora en cuadros de textos.
- Cree una clase llamada **Pago**, que defina los atributos referentes a los datos del pago del trabajador.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion62**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmPago**.
5. Agregue una clase llamada **Pago** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Figura 6.2 Controles del formulario de pago de trabajadores.

Resumen de controles:

N.º	Clase	Control
1	JLabel	PAGO DE TRABAJADORES
2	JTextField	txtTrabajador
3	JTextField	txtHoras
4	JTextField	txtTarifa
5	JButton	btnProcesar
6	JButton	btnLimpiar
7	JButton	btnSalir
8	JList	lstResumen

7. En la clase **Pago** coloque el siguiente código:

```
package pClases;
public class Pago {
    //Atributos
    private String trabajador;
    private double tarifa;
    private int horas;
    //Métodos Get y Set
    public String getTrabajador() {
        return trabajador;
    }
    public void setTrabajador(String trabajador) {
        this.trabajador = trabajador;
    }
```

```
    public double getTarifa() {
        return tarifa;
    }

    public void setTarifa(double tarifa) {
        this.tarifa = tarifa;
    }

    public int getHoras() {
        return horas;
    }

    public void setHoras(int horas) {
        this.horas = horas;
    }

    //Métodos propios de la aplicación
    public int determinaExtras() {
        int extras = 0;
        if (horas > 40) {
            extras = horas-40;
        }
        return extras;
    }

    public int determinaExcesoExtras() {
        int excede = 0;
        int extras = determinaExtras();
        if ( extras> 8) {
            excede = extras-8;
        }
        return excede;
    }

    public double calculaMonto(){
        return ((horas-determinaExtras()) * tarifa) + (determinaExtras()-
                determinaExcesoExtras())  *  (tarifa  *  2)  +
                (determinaExcesoExtras()) * (tarifa * 3);
    }
}
```

8. El código del formulario **frmPago** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Pago;

public class frmPago extends javax.swing.JFrame {

    //Definir el modelo
    DefaultListModel moR = new DefaultListModel();

    public frmPago() {
        initComponents();
    }
```

```
    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
    //Objeto de la clase Pago
        Pago objP = new Pago();

        //Enviando los valores a la clase
        objP.setTrabajador(txtTrabajador.getText());
        objP.setHoras(Integer.parseInt(txtHoras.getText()));
        objP.setTarifa(Double.parseDouble(txtTarifa.getText()));

        //Recuperando información para imprimir
        int extras = objP.determinaExtras()-objP.determinaExcesoExtras();
        int exceso = objP.determinaExcesoExtras();
        double monto = objP.calculaMonto();

        //Imprimir
        moR.clear();
        moR.addElement('**RESUMEN DE PAGO A TRABAJADORES**');
        moR.addElement('TRABAJADOR: '+objP.getTrabajador());
        moR.addElement('HORAS: '+objP.getHoras());
        moR.addElement('TARIFA: '+objP.getTarifa());
        moR.addElement('-----------------------');
        moR.addElement('HORAS EXTRAS: '+extras);
        moR.addElement('HORAS EXTRAS DE EXCESO: '+exceso);
        moR.addElement('MONTO A RECIBIR: '+monto);
        lstResumen.setModel(moR);
    }

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtTrabajador.setText('');
        txtHoras.setText('');
        txtTarifa.setText('');
        txtTrabajador.requestFocus();
        moR.clear();
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estás seguro de
                                salir...?', 'Pago de trabajadores',
                                JOptionPane.YES_NO_OPTION, JOptionPane.
                                QUESTION_MESSAGE, null, null, null);
        if (r == 0) {
            System.exit(0);
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 3: Control de hectáreas de algodón y maíz (usando if doble)

Un granjero tiene 150 hectáreas disponibles para plantar maíz y algodón. Las semillas de algodón le cuestan $ 3.00 por hectárea y las de maíz $ 5.00 por hectárea. Implemente una aplicación que permita determinar cuál es el costo que invertirá el granjero por la compra de cierta cantidad de hectáreas de algodón y maíz, sabiendo que la suma de las hectáreas no debe exceder a 150 hectáreas. Además, deberá determinar en qué semilla se invirtió más dinero.

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- Debe ingresar el nombre del trabajador, las horas de trabajo y la tarifa por hora en cuadros de textos.
- Cree una clase llamada **Pago** que defina los atributos referentes a los datos del pago del trabajador.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion63**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmHectareas**.
5. Agregar una clase llamada **Calculos** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

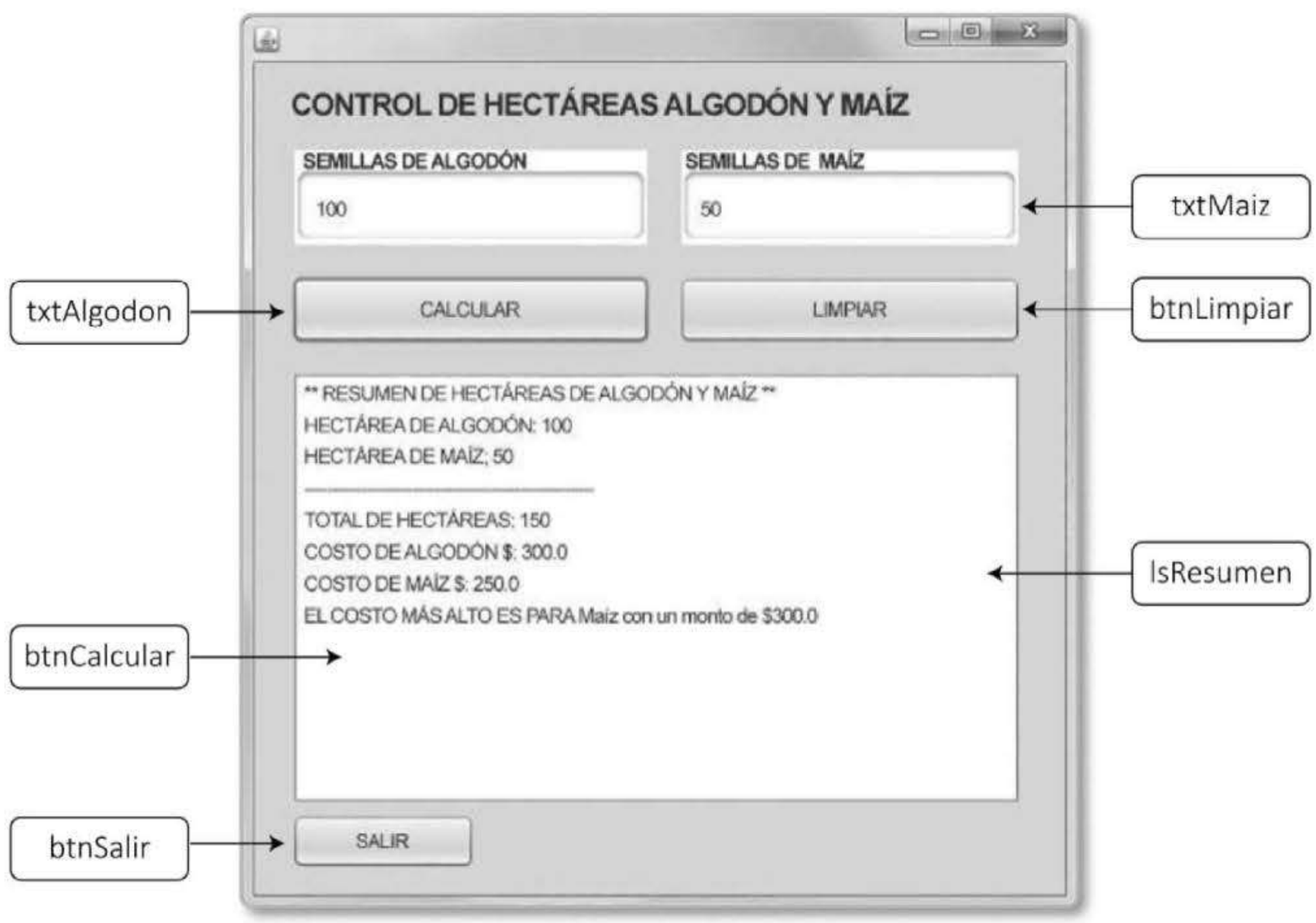


Figura 6.3 Controles del formulario de control de hectáreas.

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE HECTÁREAS ALGODÓN Y MAÍZ
2	JTextField	txtAlgodon
3	JTextField	txtMaiz
4	JButton	btnCalcular
5	JButton	btnLimpiar
6	JButton	btnSalir
7	JList	lstResumen

7. En la clase **Cálculos** coloque el siguiente código:

```
package pClases;

public class Calculos {
    //Atributos
    private int algodon;
    private int maiz;

    //Métodos Get y Set
    public int getAlgodon() {
        return algodon;
    }

    public void setAlgodon(int algodon) {
        this.algodon = algodon;
    }

    public int getMaiz() {
        return maiz;
    }

    public void setMaiz(int maiz) {
        this.maiz = maiz;
    }

    //Métodos propios
    public double calculaCostoAlgodon(){
        return algodon*3;
    }

    public double calculaCostoMaiz(){
        return maiz*5;
    }

    public String determinaProductoMasAlto(){
        if (calculaCostoAlgodon()<calculaCostoMaiz())
            return 'Algodón';
        else
            return 'Maíz';
    }

    public double determinaCostoMasAlto(){
        if (calculaCostoAlgodon()>calculaCostoMaiz())
            return calculaCostoAlgodon();
        else
            return calculaCostoMaiz();
    }
}
```

8. El código del formulario **frmHectareas** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Calculos;

public class frmHectareas extends javax.swing.JFrame {
    //Creando el modelo
    DefaultListModel moR = new DefaultListModel();
    public frmHectareas() {
        initComponents();
    }
```

```
    private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt) {
    //Objeto de la clase Cálculos
        Calculos objC = new Calculos();

        int algodon = Integer.parseInt(txtAlgodon.getText());
        int maiz = Integer.parseInt(txtMaiz.getText());

        //Comparar si no exceden a 150
        if (algodon + maiz > 150) {
            JOptionPane.showMessageDialog(null, 'Modifique los valores
                                                 para que no exceda a 150
                                                 hectáreas');
        } else {
            objC.setAlgodon(algodon);
            objC.setMaiz(maiz);
            int total = algodon + maiz;
            double costoAlgodon = objC.calculaCostoAlgodon();
            double costoMaiz = objC.calculaCostoMaiz();
            String productoCostoAlto = objC.determinaProductoMasAlto();
            double costoAlto = objC.determinaCostoMasAlto();

            //Imprimir
            moR.clear();
            moR.addElement('** RESUMEN DE HECTÁREAS DE ALGODÓN Y MAIZ **');
            moR.addElement('HECTÁREA DE ALGODÓN: ' + algodon);
            moR.addElement('HECTÁREA DE MAÍZ: ' + maiz);
            moR.addElement('-----------------------');
            moR.addElement('TOTAL DE HECTÁREAS: ' + total);
            moR.addElement('COSTO DE ALGODÓN $: ' + costoAlgodon);
            moR.addElement('COSTO DE MAÍZ $: ' + costoMaiz);
                            moR.addElement('EL COSTO MÁS ALTO ES PARA '
                           + productoCostoAlto + ' con un monto de $' +
                           costoAlto);
            lstResumen.setModel(moR);
        }
    }

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
    txtAlgodon.setText('');
        txtMaiz.setText('');
        txtAlgodon.requestFocus();
        moR.clear();
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estás seguro de
                                 salir...?', 'Hectáreas', JOptionPane.YES_NO_
                                 OPTION, JOptionPane.QUESTION_MESSAGE, null,
                                 null, null);
        if (r == 0) {
            System.exit(0);
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 4: Control de consumo de agua (usando if doblemente enlazado)

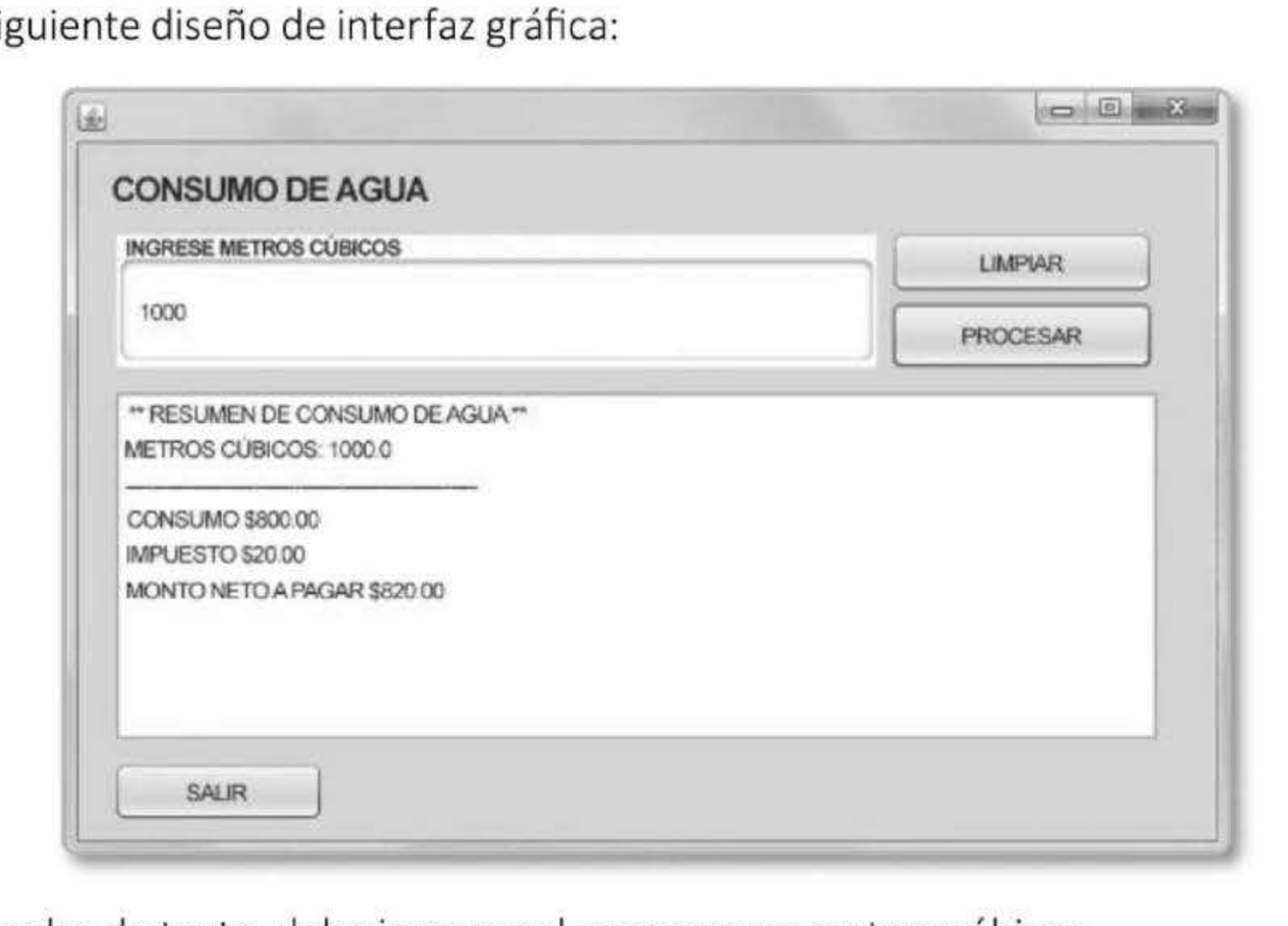
Debido a la escasez de agua en una determinada ciudad, se pone en práctica un sistema de cobro de agua que penalizará su consumo excesivo de la forma que se indica en la siguiente tabla:

Consumo (m^3)	Dólares / m^3
Menor a 100	0.15
Entre 100 y 499	0.20
Entre 500 y 1000	0.35
Más de 1000	0.80

El usuario deberá ingresar un consumo en metros cúbicos para poder determinar el monto a pagar. Se sabe que además se le incrementará un impuesto de 2.5% del monto, solo si sobrepasa los $ 600.00.

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:



- En un cuadro de texto, debe ingresar el consumo en metros cúbicos.
- Cree una clase llamada **Consumo** que defina los atributos referentes a los datos del consumo de agua.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion64**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmConsumo**.
5. Agregue una clase llamada **Consumo** en el paquete pClases.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Figura 6.4 Controles del formulario de consumo de agua.

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONSUMO DE AGUA
2	JTextField	txtConsumo
3	JButton	btnProcesar
4	JButton	btnLimpiar
5	JButton	btnSalir
6	JList	lstResumen

7. En la clase **Consumo** coloque el siguiente código:

```
package pClases;

public class Consumo {

    //Atributo
    private double consumo;

    //Métodos Get y Set
    public double getConsumo() {
        return consumo;
    }

    public void setConsumo(double consumo) {
        this.consumo = consumo;
    }

    //Métodos propios de la aplicación
    public double calculaConsumo() {
        if (consumo < 100) {
            return 0.15 * consumo;
        } else if (consumo >= 100 && consumo < 500) {
            return 0.2 * consumo;
        } else if (consumo >= 500 && consumo < 1000) {
            return 0.35 * consumo;
        } else {
            return 0.8 * consumo;
        }
    }

    public double calculaImpuesto() {
        if (calculaConsumo() > 600)
            return 2.5 / 100 * calculaConsumo();
        else
        return 0;
    }

    public double calculaNeto(){
        return calculaConsumo() + calculaImpuesto();
    }
}
```

8. El código del formulario **frmConsumo** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Consumo;

public class frmConsumo extends javax.swing.JFrame {

    //Modelo para listado
    DefaultListModel moR = new DefaultListModel();

    public frmConsumo() {
        initComponents();
    }
```

```
    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        //Objeto de la clase Consumo
        Consumo objC - new Consumo();
        //Enviando los valores a la clase
        objC.setConsumo(Double.parseDouble(txtConsumo.getText()));
        //Obteniendo los calculos
        double subtotal - objC.calculaConsumo();
        double impuesto - objC.calculaImpuesto();
        double neto - objC.calculaNeto();
        //Imprimir
        moR.clear();
        moR.addElement('** RESUMEN DE CONSUMO DE AGUA **');
        moR.addElement('METROS CUBICOS: '+objC.getConsumo());
        moR.addElement('————————————————————————-');
        moR.addElement('CONSUMO $'+String.format('%.2f',subtotal));
        moR.addElement('IMPUESTO $'+String.format('%.2f',impuesto));
        moR.addElement('MONTO NETO A PAGAR $'+String.format('%.2f',neto));
        lstResumen.setModel(moR);
    }
    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
    txtConsumo.setText('');
        txtConsumo.requestFocus();
        moR.clear();
    }
    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r - JOptionPane.showOptionDialog(this, 'Estás seguro de
                                salir...?', 'Consumo de Agua', JOptionPane.
                                YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE,
                                null, null, null);
        if (r -- 0) {
            System.exit(0);
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 5: Control de pensión de estudiantes (usando switch)

Implemente una aplicación que permita controlar los pagos que realiza un estudiante de una universidad particular; en donde el estudiante debe realizar un pago mensual por concepto de pensión, de acuerdo a un monto especificado según su categoría:

Categoría del estudiante	Monto pensión $
A	550.00
B	500.00
C	460.00
D	400.00

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- En cuadros de textos, debe ingresar el nombre del estudiante y su promedio.
- Debe seleccionar la categoría del alumno desde un cuadro combinado.
- Aplique un descuento según el promedio obtenido:

  18 a 20 > 15 %

  16 a 17.99 > 12 %

  14 a 15.99 > 10 %

  Menor a 13.99 no hay descuento
- Cree una clase llamada **Pensión** que defina los atributos referentes a los datos del control de pensión de estudiantes.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion65**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmPension**.
5. Agregue una clase llamada **Pensión** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Figura 6.5 Controles del formulario de control de pensión de estudiantes.

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE PENSIÓN DE ESTUDIANTES
2	JTextField	txtEstudiante
3	JTextField	txtPromedio
4	JComboBox	cboCategoria
4	JButton	btnProcesar
5	JButton	btnLimpiar
6	JButton	btnSalir
7	JList	lstResumen

7. En la clase **Pensión** coloque el siguiente código:

```
package pClases;

public class Pension {
    //Atributos
    private String estudiante;
    private String categoria;
    private double promedio;

    //Métodos Get y Set
    public String getEstudiante() {
        return estudiante;
    }
```

```
    public void setEstudiante(String estudiante) {
        this.estudiante = estudiante;
    }

    public String getCategoria() {
        return categoria;
    }

    public void setCategoria(String categoria) {
        this.categoria = categoria;
    }

    public double getPromedio() {
        return promedio;
    }

    public void setPromedio(double promedio) {
        this.promedio = promedio;
    }

    //Métodos propios
    public double calculaPension(){
        switch(getCategoria()){
            case 'A': return 550;
            case 'B': return 500;
            case 'C': return 460;
            default: return 400;
        }
    }

    public double calculaDescuento(){
        if(getPromedio()<-13.99)
            return 0;
        else if(getPromedio()<-15.99)
            return 0.1*calculaPension();
        else if(getPromedio()<-17.99)
            return 0.12*calculaPension();
        else
            return 0.15*calculaPension();
    }

    public double calculaNuevaPension(){
        return calculaPension()-calculaDescuento();
    }
}
```

8. El código del formulario **frmPension** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Pension;

public class frmPension extends javax.swing.JFrame {

    //Modelo para listado
    DefaultListModel moR = new DefaultListModel();

    public frmPension() {
        initComponents();
        llenaCategoria();
    }
```

```
    //Método que llena las categorías del estudiante
    void llenaCategoria(){
        cboCategoria.addItem('A');
        cboCategoria.addItem('B');
        cboCategoria.addItem('C');
        cboCategoria.addItem('D');
    }

    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        //Objeto de la clase Pensión
        Pension objP = new Pension();

        //Enviando valores a la clase
        objP.setEstudiante(txtEstudiante.getText());

        objP.setCategoria(String.valueOf(cboCategoria.getSelectedItem()));
        objP.setPromedio(Double.parseDouble(txtPromedio.getText()));

        //Obteniendo los cálculos
        double pension = objP.calculaPension();
        double descuento = objP.calculaDescuento();
        double nuevaPension = objP.calculaNuevaPension();

        //Imprimir
        moR.clear();
        moR.addElement('** RESUMEN DE PAGO DE PENSIONES **');
        moR.addElement('ESTUDIANTE: '+objP.getEstudiante());
        moR.addElement('CATEGORÍA: '+objP.getCategoria());
        moR.addElement('PROMEDIO: '+objP.getPromedio());
        moR.addElement('----------------------');
        moR.addElement('PENSIÓN $'+String.format('%.2f',pension));
        moR.addElement('DESCUENTO $'+String.format('%.2f',descuento));
        moR.addElement('NUEVA PENSIÓN $'+
                       String.format('%.2f',nuevaPension));
        lstResumen.setModel(moR);

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtEstudiante.setText('');
        txtPromedio.setText('');
        cboCategoria.setSelectedIndex(0);
        moR.clear();
        txtEstudiante.requestFocus();
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estás seguro de
                            salir...?', 'Pensión',JOptionPane.YES_NO_
                            OPTION, JOptionPane.QUESTION_MESSAGE, null,
                            null, null);
        if (r == 0) {
            System.exit(0);
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

Estructura de repetición

CAPÍTULO

7

Competencias

- Implementa aplicaciones usando los contadores y acumuladores de programación.
- Implementa aplicaciones usando estructuras de repetición.

Contenido

7.1 Generalidades

7.2 Contadores

7.3 Acumuladores

7.4 Estructuras repetitivas

7.5 Variables y métodos de clase: modificador **static**

- Caso desarrollado 1: Control de registro de participantes (usando contadores)
- Caso desarrollado 2: Registro de alquiler (usando acumuladores)
- Caso desarrollado 3: Registro de venta (usando **for**)
- Caso desarrollado 4: Control de pago (usando modificador **static**)

7.1 Generalidades

Los contadores y acumuladores tienen la misión de realizar cálculos acumulativos que con una expresión simple no se pueden implementar, y es que obligatoriamente toda variable contadora o acumuladora debe ser inicializada en un valor. Por otro lado, si estas variables son atributos de clase, entonces deben ser declarados como estáticos.

Asimismo, las estructuras repetitivas se utilizan cuando se quiere que un conjunto de sentencias se ejecute un cierto número de veces. También se denomina *bucle* o *ciclo de repeticiones* a todo proceso que se repite un cierto número de veces dentro de un método en Java. Principalmente existen dos tipos de estructuras repetitivas: la sentencia **for**, que define el ciclo de repeticiones claramente en su formato, y la sentencia **while**, que condiciona el ciclo de repeticiones.

7.2 Contadores

Es una variable de tipo entero, cuyo valor acumula las veces que se pasa por ella. Se suele denominar contador cuando la variable incrementa o decrementa de 1 en 1.

Formato:

Donde:

- **Valor constante.** Para que se considere un contador, este valor debe ser constante; normalmente para conteos positivos debe ser +1 y para conteos negativos –1.
- **Variable contadora.** Es el último valor del contador.
- **Variable contadora actual.** Es el nuevo valor del contador después del conteo.

Hay que tener en cuenta que cuando vimos el tema de operadores en Java se listó los operadores aritméticos unarios de incremento y decremento. Se presentan, a continuación, algunas analogías sobre los contadores:

Contar a los empleados de una empresa.		
`cEmpleados = cEmpleados + 1;`	`cEmpleados++;`	`cEmpleados+=1;`

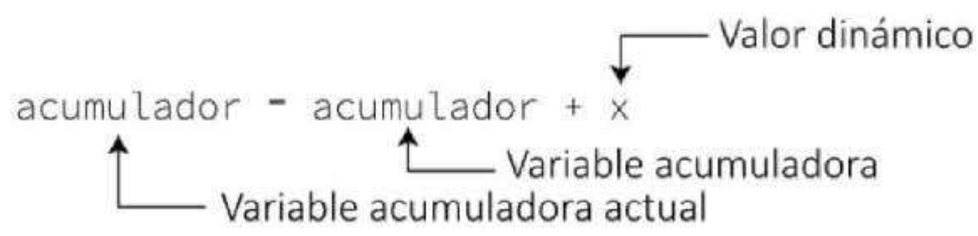
Contar el número de notas registradas, para este caso se aplica un conteo en forma de decremento.		
`cNotas = cNotas-1;`	`cNotas–;`	`cNotas-=1;`

Contar el número de veces que el usuario realiza un clic sobre un botón, para este caso se usa una constante 2.		
`cBoton = cBoton + 2;`	`cBoton++;` `cBoton++;`	`cBoton+=2;`

7.3 Acumuladores

Un acumulador o totalizador suele utilizarse para acumular resultados producidos en las iteraciones de un bucle o de llamadas sucesivas. Se parece en estructura a los contadores, con la diferencia que los acumuladores no tienen un valor constante sino más bien variable.

Formato:



Donde:

- **Valor dinámico.** Este valor tiene que ser una variable que tenga diferentes valores cada vez que se invoca a un acumulador.
- **Variable acumuladora.** Es el último valor del acumulador.
- **Variable acumuladora actual.** Es el nuevo valor del acumulador después de una acumulación de valor.

Algunos ejemplos de acumuladores son:

Acumular las edades de los empleados.	
aEdad - aEdad + edad;	aEdad +- Edad;

Acumular los sueldos de los empleados.	
aSueldo - aSueldo + sueldo;	aSueldo +- sueldo;

7.4 Estructuras repetitivas

Son aquellas que permiten repetir una secuencia de instrucciones o expresiones un número determinado de veces, dicha cantidad será especificada por el programador. También son llamados bucles o ciclos que permiten repetir las instrucciones un determinado número de veces.

Algunos casos donde las estructuras de tipo repetición pueden encajar son los siguientes:

- Imprimir un mismo documento muchas veces.
- Establecer un límite de veces de ingreso de un valor a un usuario; por ejemplo, en un cajero automático solo está permitido errar 3 veces el ingreso de la clave.

Un ejemplo práctico podría ser el caso de unos atletas corriendo sobre una pista de atletismo, en la cual tienen que dar cierto número de vueltas para completar una carrera.

Figura 7.1 ciclo de repeticiones en un circuito de carrera.

Una estructura repetitiva tiene un objetivo específico siempre y cuando se integre a los contadores o acumuladores, por ejemplo:

- Determinar el número de vueltas realizadas por un atleta.
- Determinar el total acumulado de kilómetros por cada vuelta.
- Determinar la máxima aceleración realizada por toda la carrera.

Hay que tener en cuenta que las estructuras de repetición están presentes en diferentes actividades que realizamos, como, por ejemplo, pasear en bicicleta y dar vueltas en un parque o dirigirnos a nuestro trabajo todos los días.

7.4.1 Estructura de repetición for

La estructura **for** ejecuta un número determinado de veces un conjunto de sentencias o instrucciones y controla de forma automática el número de iteraciones que esta realiza. Su formato se basa en la siguiente estructura:

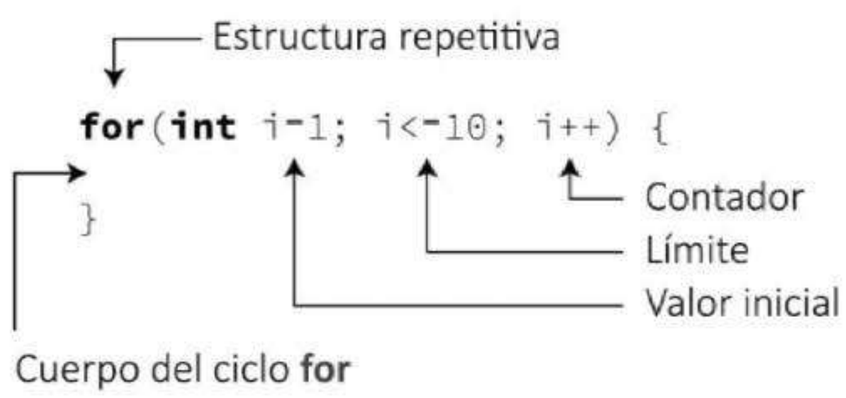


Figura 7.2 Estructura de la sentencia **for**.

Donde:

- **Valor inicial.** Define el punto de inicio del ciclo de repeticiones. Hay que tener en cuenta que solo se puede usar el operador igual para dicha asignación y que la declaración de la variable se puede realizar a nivel externo del ciclo **for**.
- **Límite.** Define el valor final del ciclo de repeticiones. Los operadores que se pueden usar son >, >=, <, <=.

- **Contador.** Indica la forma del ciclo de repeticiones. Así pues, cuando se dan ciclos de forma ascendente, se usa el operador ++; mientras que, en el caso de ciclos descendentes, el operador – –.
- **Cuerpo del ciclo for.** Aquí se definen las sentencias o instrucciones que se repetirán según el límite impuesto al ciclo de repeticiones.

De forma gráfica, la sentencia **for** se muestra de la siguiente manera:

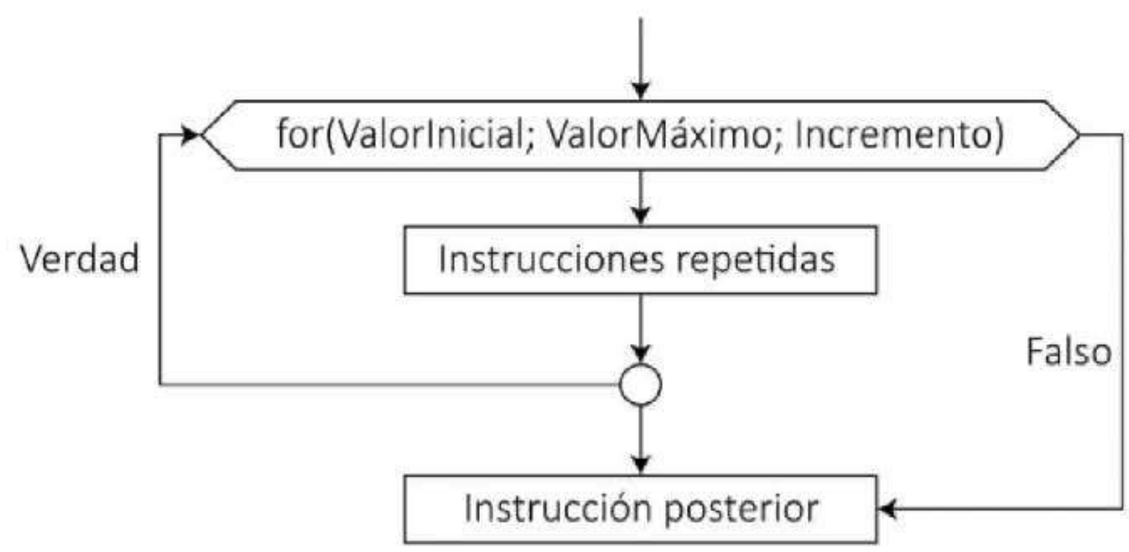


Figura 7.3 Diagrama de la sentencia **for**.

A continuación, se presenta una lista de situaciones para el ciclo **for**:

- Listar los 10 primeros números usando la estructura **for**.

```
for(int i=1;i<=10;i++)
{
    JOptionPane.showMessageBox('null',i);
}
```

- Listar los 10 primeros números en forma descendente usando la estructura **for**.

```
for(int i=10;i>=1;i--)
{
    JOptionPane.showMessageBox('null',i);
}
```

- Listar los divisores de un determinado número entero.

```
int n = 10;
for(int i=1;i<=n;i++)
{
    if (n % i == 0) JOptionPane.showMessageBox('null',i);
}
```

7.4.2 Estructura de repetición while

La estructura **while** ejecuta un número determinado de veces un conjunto de sentencias o instrucciones y controla el número de iteraciones que esta realiza mediante una condición lógica, es decir, el ciclo de repeticiones inicia bajo una condición. Su formato se basa en la siguiente estructura:

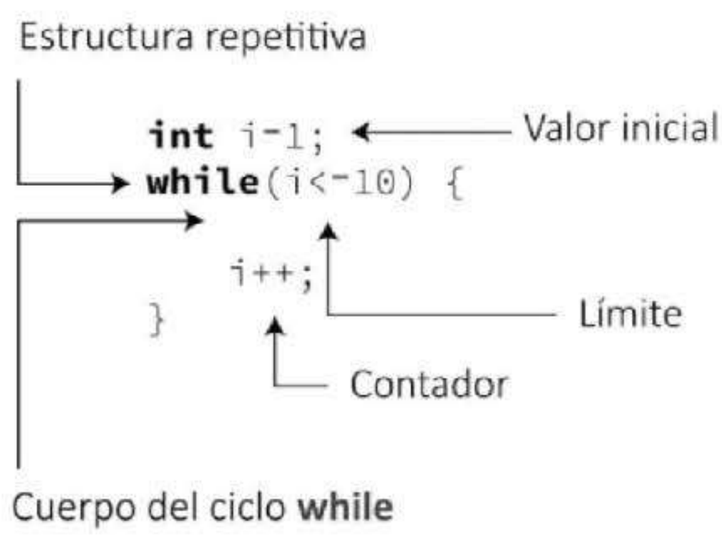


Figura 7.4 Estructura de la sentencia **while**.

Donde:

- **Valor inicial.** Aquí se define el valor de inicio que tendrá el ciclo de repeticiones con **while**.
- **Límite.** Define el valor final del ciclo de repeticiones. Los operadores que se pueden usar son >, >=, <, <=.
- **Contador.** Indica la forma del ciclo de repeticiones. Así pues, cuando se dan ciclos de forma ascendente, se usa el operador ++; mientras que, en el caso de ciclos descendentes, el operador – –.
- **Cuerpo del ciclo while.** Aquí se definen las sentencias o instrucciones que se repetirán según el límite impuesto al ciclo de repeticiones.

De forma gráfica, la estructura **while** se muestra de la siguiente manera:

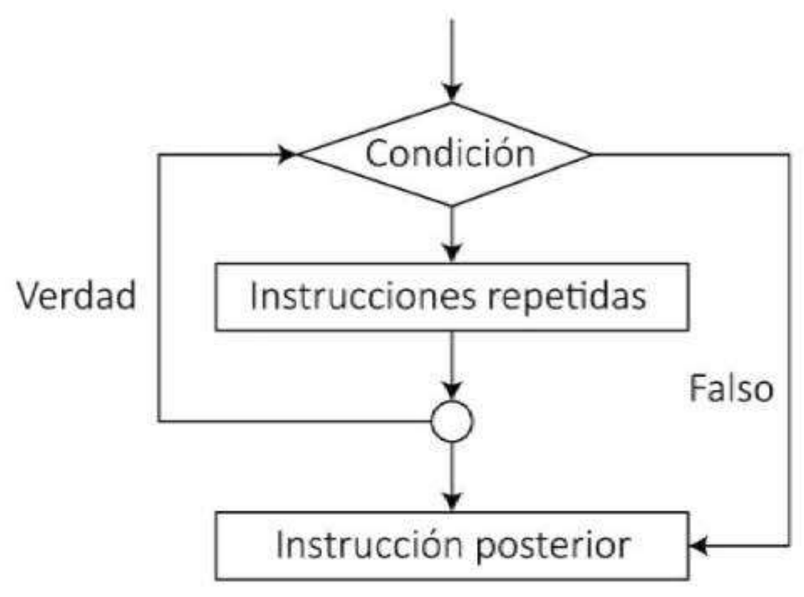


Figura 7.5 Diagrama de la estructura **while**.

A continuación, se presenta una lista de situaciones para el ciclo **while**:

- Listar los 10 primeros números usando la estructura **while**.

```
int i=1;
while(i<=10)
{
    JOptionPane.showMessageBox('null', i);
    i++;
}
```

- Listar los 10 primeros números en forma descendente usando la estructura **while**.

```
int i=10;
while(i>=1)
{
    JOptionPane.showMessageBox('null',i);
    i--;
}
```

- Listar los divisores de un determinado número entero.

```
int n = 10;
int i = 1;
while(i<=n)
{
    if (n % i == 0) JOptionPane.showMessageBox('null',i);
    i++;
}
```

7.4.3 Estructura de repetición do... while

La estructura **do... while** ejecuta un número determinado de veces un conjunto de sentencias o instrucciones y controla el número de iteraciones que esta realiza mediante una condición lógica. La condición siempre se evalúa al final de la estructura. Su formato se basa en la siguiente estructura:

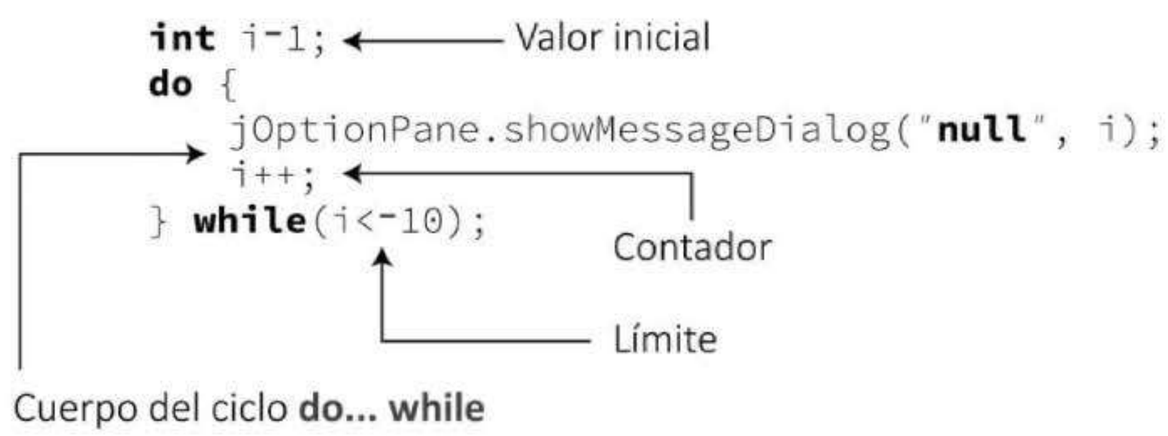


Figura 7.6 Estructura de la sentencia **do... while**.

Donde:

- **Valor inicial.** Aquí se define el valor de inicio que tendrá el ciclo de repeticiones con **do... while**. Hay que tener en cuenta que dicho valor se evalúa al final de la estructura **while**; y mientras sea verdadera la condición, el ciclo seguirá.

- **Límite.** Define el valor final del ciclo de repeticiones. Los operadores que se pueden usar son >, >=, <, <=.
- **Contador.** Indica la forma del ciclo de repeticiones. Así pues, cuando se dan ciclos de forma ascendente, se usa el operador ++; mientras que, en el caso de ciclos descendentes, el operador – –.
- **Cuerpo del ciclo do while.** Aquí se definen las sentencias o instrucciones que se repetirán según el límite impuesto al ciclo de repeticiones.

De forma gráfica, la estructura **do... while** se muestra de la siguiente manera:

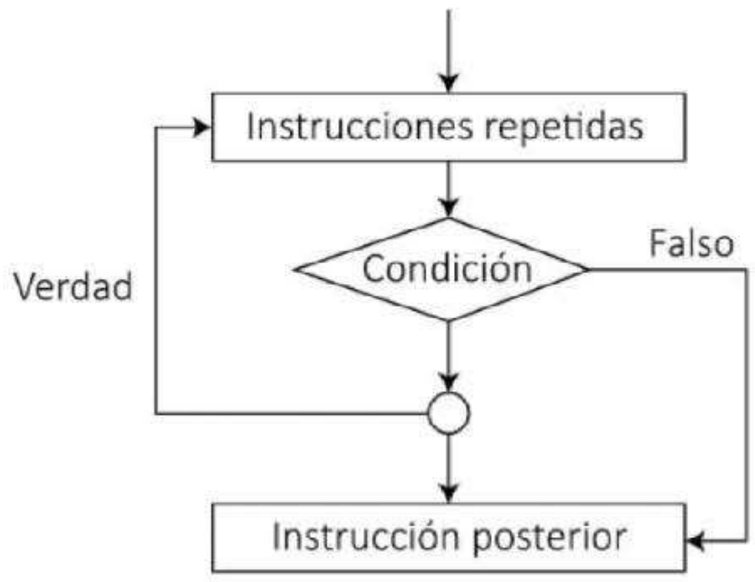


Figura 7.7 Diagrama de la estructura **do while**.

A continuación, se presenta una lista de situaciones para el ciclo **do while**:

- Listar los 10 primeros números usando la estructura **do while**.

```
int i-1;
do {
    JOptionPane.showMessageBox('null',i);
    i++;
} while(i<-10);
```

- Listar los 10 primeros números en forma descendente usando la estructura **do while**.

```
int i-10;
do {
    JOptionPane.showMessageBox('null',i);
    i–;
} while(i>-1);
```

- Listar los divisores de un determinado número entero.

```
int n - 10;
int i - 1;
do {
    if (n % i -- 0) JOptionPane.showMessageBox('null',i);
    i++;
} while(i<-n);
```

7.5 Variables y métodos de clase: modificador static

En Java, los atributos que contiene el modificador **static** en su implementación son conocidos como atributos de clase. Uno de los posibles usos del modificador **static** es para compartir el valor de una variable miembro entre objetos de una misma clase.

Algunas características del modificador **static** son las siguientes:

- Sus atributos son compartidos por todas las instancias de la clase.
- Permite definir clases que pueden ser llamadas desde otras clases sin usar objetos.
- Permite compartir un valor de una variable miembro entre objetos de una misma clase. La referencia por medio de **this** a las variables estáticas genera un error, puesto que la variable estática no puede ser accedida por ningún objeto.

7.5.1 Variable de clase

Es una variable que se crea una sola vez para todos los objetos de la misma clase.

Formato:

```
visibilidad static tipoDatos nombreAtributo = valorInicial;
```

Donde:

- **Visibilidad.** Puede ser asignada como pública o privada; si es declarada como privada, necesitará un método **get** que permitirá devolver el valor.
- **Static.** Modificador que asigna una variable de clase.
- **tipoDatos.** Es el tipo de datos de la variable de clase.
- **nombreAtributo.** Es el nombre asignado a la variable de clase.
- **valorInicial.** Si el tipo de datos es numérico, se le asignará el valor cero; en caso de ser una cadena, se le asignará comillas dobles.

Ejemplo. Se necesita contabilizar las veces que se crea el objeto **Empleado** en una aplicación.

La solución se da en tres pasos:

- Primero, se debe implementar la clase **Empleado**.

```
public class Empleado{
    private String codigo;
    private String nombres;
    private int edad;
    private static int contador=0;
}
```

- Segundo, se debe implementar el método **constructor**.

```
public Empleado(String codigo,String nombres,int edad){
    this.codigo = codigo;
    this.nombres = nombres;
    this.edad = edad;
    contador++;
}
```

- Tercero, se debe implementar un método que devolverá el valor que obtiene la variable de clase contador.

```
public int getContador(){
    return contador;
}
```

- Para obtener el valor de la variable de clase, se coloca el siguiente script:

```
Empleado objEmp = new Empleado();
JOptionPane.showMessageDialog(null, 'El total de Empleados es: ' +
                                objEmp.getContador());
```

7.5.2 Métodos estáticos

Los métodos **static** permiten optimizar la memoria en tiempo de ejecución por no tener que crear instancias para acceder a métodos comunes de los objetos.

Formato para la implementación de un método estático:

```
Visibilidad static tipoDatos nombreMetodo(){
    return valor;
}
```

Donde:

- **Visibilidad.** Es el alcance (público, privado o protegido) que tiene el método estático implementado.
- **static.** Define el método de clase.
- **tipoDatos.** Es el valor devuelto por el método de clase; hay que tener en cuenta que el valor devuelto a través del return sea del mismo tipo implementado.
- **nombreMetodo.** Es el nombre asignado al método de clase.
- **return valor.** Es el valor devuelto por el método.

Formato para la invocación de un método estático:

```
Variable = NombreClase.NombreMetodoEstatico();
```

Para la invocación del método de clase no será necesario importar el paquete, ya que el método será llamado directamente desde la clase.

Ejemplo. Se necesita contabilizar las veces que se crea el objeto **Empleado** en una aplicación.

Veamos el script:

- Primero, se debe implementar la clase **Empleado**.

```
public class Empleado{
    private String codigo;
    private String nombres;
    private int edad;
    private static int contador=0;
}
```

- Segundo, se debe implementar el método **constructor**.

```
public Empleado(String codigo,String nombres,int edad){
    this.codigo = codigo;
    this.nombres = nombres;
    this.edad = edad;
    contador++;
}
```

- Tercero, se debe implementar un método de clase que devolverá el valor que obtiene la variable de clase contador.

```
public static int getContador(){
    return contador;
}
```

- Para invocar al método, se coloca el siguiente script:

```
JOptionPane.showMessageDialog(null, 'El total de Empleados es: '
                                + Empleado.getContador());
```

7.5.3 Inicializadores de variables de clase

Los atributos **static** son inicializados una sola vez dentro de la clase en forma automática.

Formato para la inicialización de variables de clase:

```
static {
    variableClase=valorInicial;
}
```

Donde:

- **variableClase.** Es el atributo declarado como estático.
- **valorInicial.** Es el valor inicial del atributo estático.

CASO DESARROLLADO 1: **Control de registro de participantes (usando contadores)**

Implemente una aplicación que permita controlar los asistentes a una fiesta; para ello deberá ingresar la edad, sexo (masculino y femenino) y estado civil (soltero, casado, viudo y divorciado) del asistente. Luego muestre las siguientes estadísticas:

- Total de asistentes
- Total de personas mayores de edad
- Total de personas menores de edad
- Total de hombres
- Total de mujeres
- Total de solteros
- Total de casados
- Total de viudos
- Total de divorciados
- Porcentaje de hombres
- Porcentaje de mujeres

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**, el método **constructor** y los atributos estáticos para los conteos.
- Use el siguiente diseño de interfaz gráfica:

- En un cuadro de texto, debe ingresar la edad del participante.
- El tipo de sexo debe ser seleccionado a partir del control **JCheckbox**.
- El estado civil debe ser seleccionado a partir del control **JRadioButton**.
- Cree una clase llamada **Participantes** que defina los atributos referentes a los datos del participante, además de definir los atributos estáticos.
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion71**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmParticipantes**.
5. Agregue una clase llamada **Participantes** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Figura 7.8 Controles del formulario de control de personas.

Resumen de controles:

N.º	Clase	Control
1	JLabel	REGISTRO DE PARTICIPANTES
2	JTextField	txtEdad
3	JCheckbox	chkMasculino
4	JRadiobutton	rbSoltero
5	JRadiobutton	rbCasado
6	JRadiobutton	rbViudo
7	JRadiobutton	rbDivorciado
8	JButtongroup	bgEstado
9	JButton	btnProcesar
10	JButton	btnLimpiar
11	JButton	btnSalir
12	JList	lstResumen

7. En la clase **Participantes** coloque el siguiente código:

```
package pClases;

public class Participantes {
    //Atributos
    private int edad;
    private String masculino;
    private String estadoCivil;

    private static int total=0;
    private static int tHombre=0;
    private static int tMujeres=0;
    private static int tMayores=0;
    private static int tMenores=0;
    private static int tSolteros=0;
    private static int tCasados=0;
    private static int tViudos=0;
    private static int tDivorciados=0;

    public Participantes(){
        total++;
    }

    //Métodos Get y Set
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }

    public String getMasculino() {
        return masculino;
    }

    public void setMasculino(String masculino) {
        this.masculino = masculino;
    }

    public String getEstadoCivil() {
        return estadoCivil;
    }

    public void setEstadoCivil(String estadoCivil) {
        this.estadoCivil = estadoCivil;
    }

    //Métodos get de devolución de conteos
    public static int getTotal() {
        return total;
    }

    public static int gettHombre() {
        return tHombre;
    }

    public static int gettMujeres() {
        return tMujeres;
    }

    public static int gettMayores() {
        return tMayores;
    }
```

```
    public static int gettMenores() {
        return tMenores;
    }

    public static int gettSolteros() {
        return tSolteros;
    }

    public static int gettCasados() {
        return tCasados;
    }

    public static int gettViudos() {
        return tViudos;
    }

    public static int gettDivorciados() {
        return tDivorciados;
    }

    //Métodos propios
    public void determinaSexo(){
        if (masculino.equals('Masculino'))
            tHombre++;
        else
            tMujeres++;
    }

    public void determinaEdad(){
        if (edad>-18)
            tMayores++;
        else
            tMenores++;
    }

    public void determinaEstado(){
        if (estadoCivil.equals('Soltero'))
            tSolteros++;
        else if (estadoCivil.equals('Casado'))
            tCasados++;
        else if (estadoCivil.equals('Viudo'))
            tViudos++;
        else
            tDivorciados++;
    }
}
```

8. El código del formulario **frmParticipantes** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Participantes;

public class frmParticipantes extends javax.swing.JFrame {
    //Modelo para listado
    DefaultListModel moR - new DefaultListModel();
    public frmParticipantes() {
        initComponents();
    }
```

```
    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        if (valida().equals('')) {
            //Objeto de la clase Participantes
            Participantes objP - new Participantes();

            //Enviando información a la clase
            objP.setEdad(Integer.parseInt(txtEdad.getText()));
            if (chkMasculino.isSelected()) {
                objP.setMasculino('Masculino');
            } else {
                objP.setMasculino('Femenino');
            }

            if (rbSoltero.isSelected()) {
                objP.setEstadoCivil('Soltero');
            } else if (rbCasado.isSelected()) {
                objP.setEstadoCivil('Casado');
            } else if (rbViudo.isSelected()) {
                objP.setEstadoCivil('Viudo');
            } else {
                objP.setEstadoCivil('Divorciado');
            }

            //Invocando los conteos
            objP.determinaEdad();
            objP.determinaSexo();
            objP.determinaEstado();

            //Obtener los resultados
            int total - objP.getTotal();
            int totalMayor - objP.gettMayores();
            int totalMenor - objP.gettMenores();
            int totalHombres - objP.gettHombre();
            int totalMujeres - objP.gettMujeres();
            int totalSolteros - objP.gettSolteros();
            int totalCasados - objP.gettCasados();
            int totalViudos - objP.gettViudos();
            int totalDivorciados - objP.gettDivorciados();
            int porcentajeHombres - (totalHombres * 100) / total;
            int porcentajeMujeres - (totalMujeres * 100) / total;

            moR.clear();
            moR.addElement('** ESTADÍSTICAS **');
            moR.addElement('El Total de Asistentes es: ' + total);
            moR.addElement('El Total de Personas mayores es: ' + totalMayor);
            moR.addElement('El Total de Personas menores es: ' + totalMenor);
            moR.addElement('————————————————————');
            moR.addElement('El Total de Hombres es: ' + totalHombres);
            moR.addElement('El Total de Mujeres es: ' + totalMujeres);
            moR.addElement('————————————————————');
            moR.addElement('El Total de Solteros es: ' + totalSolteros);
            moR.addElement('El Total de Casados es: ' + totalCasados);
            moR.addElement('El Total de Viudos es: ' + totalViudos);
            moR.addElement('El Total de Divorciados es: ' + totalDivorciados);
            moR.addElement('————————————————————');
            moR.addElement('% de Hombres es: ' + (totalHombres * 100) / total);
            moR.addElement('% de Mujeres es: ' + (totalMujeres * 100) / total);
            lstResumen.setModel(moR);
        } else {
            JOptionPane.showMessageDialog(null, valida());
        }
    }
```

```
    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtEdad.setText('');
        txtEdad.requestFocus();
        moR.clear();
        rbSoltero.setSelected(true);
        chkMasculino.setSelected(false);
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estás seguro de
                                salir...?', 'Acumuladores y Contadores',
                                JOptionPane.YES_NO_OPTION, JOptionPane.
                                QUESTION_MESSAGE, null, null, null);
        if (r == 0) {
            System.exit(0);
        }
    }

    //Método que valida los controles
    String valida() {
            if (txtEdad.getText().equals('')
                || Integer.parseInt(txtEdad.getText()) < 0) {
            return 'Edad del Asistente..!!!';
        } else if (!(rbSoltero.isSelected() || rbCasado.isSelected()
                || rbViudo.isSelected() || rbDivorciado.isSelected())) {
            return 'Estado Civil del Asistente..!!!';
        } else {
            return '';
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 2: **Registro de alquiler (usando acumuladores)**

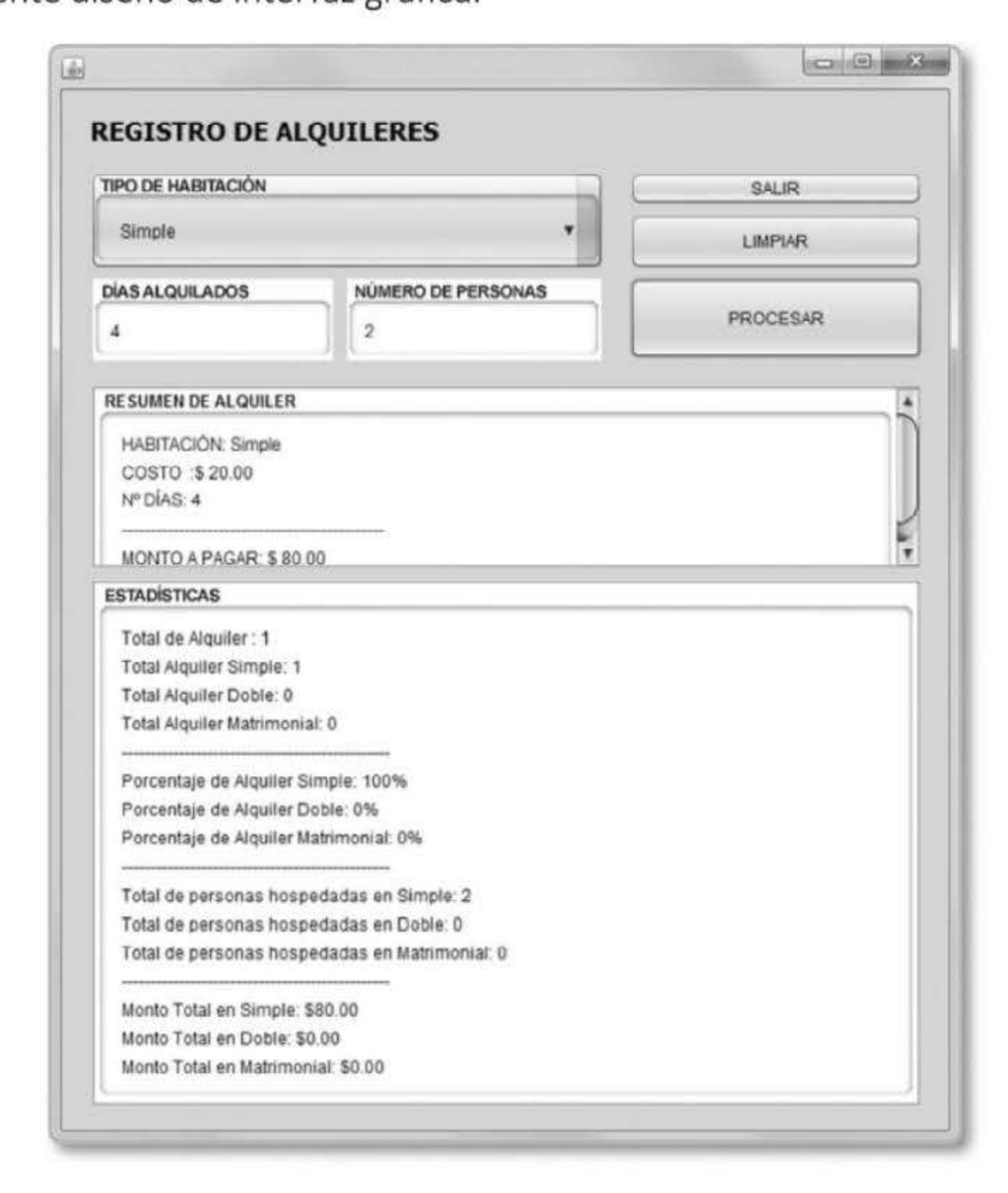
Implemente una aplicación que permita apoyar al área de recepción de un hotel. La aplicación debe registrar los alquileres que se realizan y el número de personas que se hospedan. El hotel cuenta con los siguientes tipos de habitación:

Tipo de habitación	Costo $
Simple	$ 20.00
Doble	$ 40.00
Matrimonial	$ 100.00

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**, el método **constructor** y los atributos estáticos para los conteos y acumulaciones.

- Use el siguiente diseño de interfaz gráfica:



- Desde el control cuadro combinado, debe seleccionarse el tipo de habitación.
- Debe ingresar el total de días alquilados y el número de personas en control de texto.
- Cree una clase llamada **Alquiler** que defina los atributos referentes a los datos del alquiler, además de definir los atributos estáticos para los conteos y acumulaciones.
- Debe mostrar los siguientes valores:
  - Costo por alquiler
  - Total de alquileres por tipo de habitación
  - Reporte porcentual de alquileres por tipo de habitación
  - Reporte de personas hospedadas por tipo de habitación
  - Reporte de montos acumulados por tipo de habitación
- Muestre el resultado de la aplicación en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion72**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmAlquiler**.

5. Agregue una clase llamada **Alquiler** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

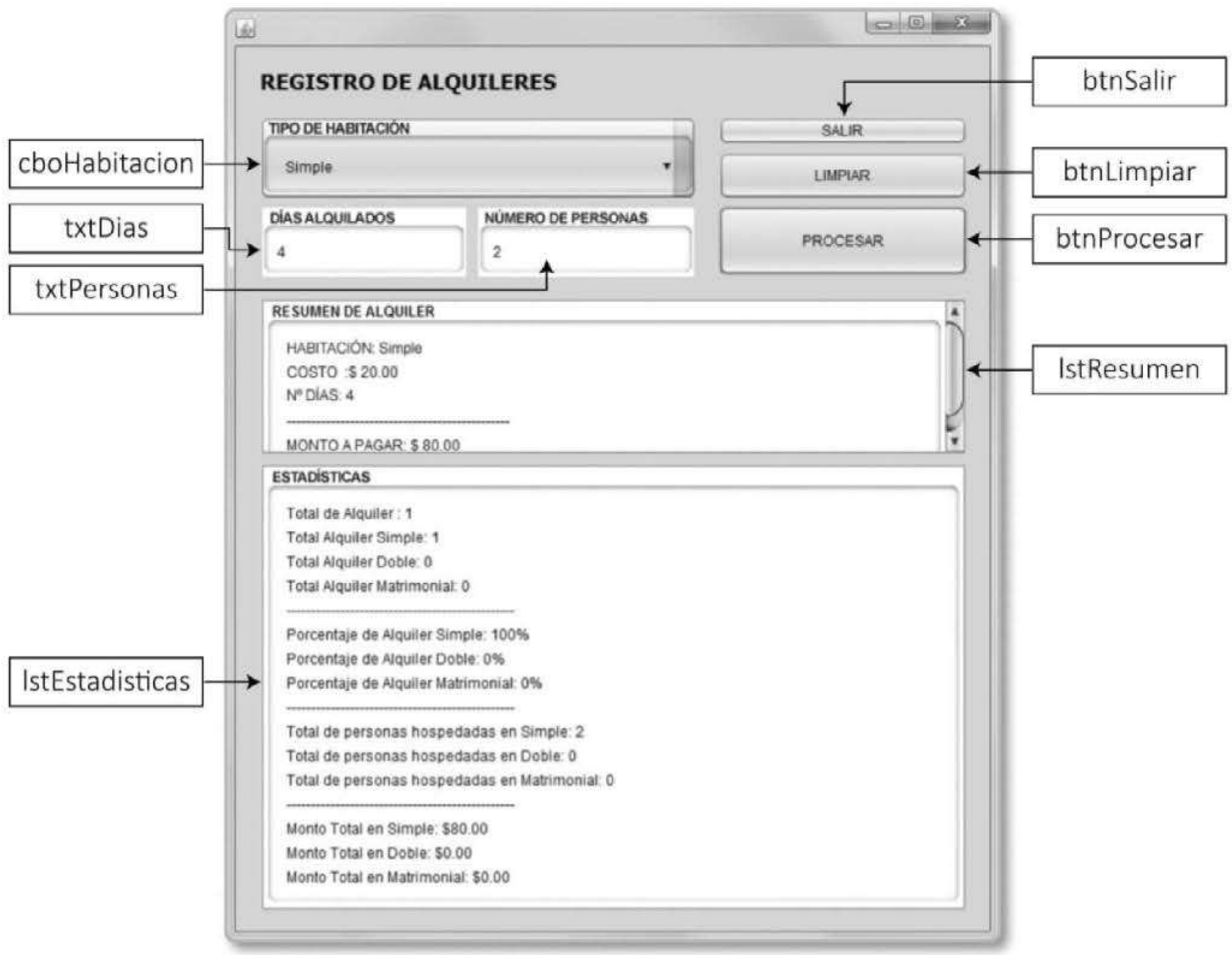


Figura 7.9 Controles del formulario de control de habitaciones.

Resumen de controles:

N.º	Clase	Control
1	JLabel	REGISTRO DE ALQUILERES
2	JCombobox	cboHabitacion
3	JTextField	txtDias
4	JTextField	txtPersonas
5	JList	lstResumen
6	JList	lstEstadisticas
7	JButton	btnProcesar
8	JButton	btnLimpiar
9	JButton	btnSalir

7. En la clase **Alquiler** coloque el siguiente código:

```
package pClases;

public class Alquiler {
    //Atributos
    private String habitacion;
    private int dias;
    private int personas;

    //Atributos estáticos
    private static int total;
    private static int tSimple;
    private static int tDoble;
    private static int tMatrimonial;
    private static int pSimple;
    private static int pDoble;
    private static int pMatrimonial;
    private static int mSimple;
    private static int mDoble;
    private static int mMatrimonial;

    //Método constructor
    public Alquiler(){
        total++;
    }

    //Métodos Get y Set
    public String getHabitacion() {
        return habitacion;
    }

    public void setHabitacion(String habitacion) {
        this.habitacion = habitacion;
    }

    public int getDias() {
        return dias;
    }

    public void setDias(int dias) {
        this.dias = dias;
    }

    public int getPersonas() {
        return personas;
    }

    public void setPersonas(int personas) {
        this.personas = personas;
    }

    //Métodos Get para atributos estáticos
    public static int getTotal() {
        return total;
    }

    public static int gettSimple() {
        return tSimple;
    }

    public static int gettDoble() {
        return tDoble;
    }
```

```
    public static int gettMatrimonial() {
        return tMatrimonial;
    }

    public static int getpSimple() {
        return pSimple;
    }

    public static int getpDoble() {
        return pDoble;
    }

    public static int getpMatrimonial() {
        return pMatrimonial;
    }

    public static int getmSimple() {
        return mSimple;
    }

    public static int getmDoble() {
        return mDoble;
    }

    public static int getmMatrimonial() {
        return mMatrimonial;
    }

    //Métodos propios de la aplicación
    public double asignaCosto() {
        switch (habitacion) {
            case 'Simple':
                return 20;
            case 'Doble':
                return 40;
            default:
                return 100;
        }
    }

    public double calculaMonto(){
        return dias*asignaCosto();
    }

    public void conteos(){
        switch(habitacion){
            case 'Simple':
                tSimple++;
                pSimple+=personas;
                mSimple+=calculaMonto();
                break;
            case 'Doble':
                tDoble++;
                pDoble+=personas;
                mDoble+=calculaMonto();
                break;
            default:
                tMatrimonial++;
                pMatrimonial+=personas;
                mMatrimonial+=calculaMonto();
        }
    }
}
```

8. El código del formulario **frmAlquiler** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Alquiler;

public class frmAlquiler extends javax.swing.JFrame {
    //Definición de modelos
    DefaultListModel moR = new DefaultListModel();
    DefaultListModel moE = new DefaultListModel();

    public frmAlquiler() {
        initComponents();
        llenaHabitacion();
    }

    //Método que llena los tipos de habitación
    void llenaHabitacion(){
        cboHabitacion.addItem('Simple');
        cboHabitacion.addItem('Doble');
        cboHabitacion.addItem('Matrimonial');
    }

    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        //Objeto de la clase Alquiler
        Alquiler objA = new Alquiler();

        //Enviando los valores a la clase
        objA.setHabitacion(String.valueOf(cboHabitacion.getSelectedItem()));
        objA.setDias(Integer.parseInt(txtDias.getText()));
        objA.setPersonas(Integer.parseInt(txtPersonas.getText()));

        //Invocando a los procesos
        double monto = objA.calculaMonto();
        objA.conteos();

        int total = objA.getTotal();
        int cSimple = objA.gettSimple();
        int cDoble = objA.gettDoble();
        int cMatrimonial = objA.gettMatrimonial();

        int porSimple = (cSimple * 100) / total;
        int porDoble = (cDoble * 100) / total;
        int porMatrimonial = (cMatrimonial * 100) / total;

        int pSimple = objA.getpSimple();
        int pDoble = objA.getpDoble();
        int pMatrimonial = objA.getpMatrimonial();

        double mSimple = objA.getmSimple();
        double mDoble = objA.getmDoble();
        double mMatrimonial = objA.getmMatrimonial();

        //Imprimir
        moR.clear();
        moR.addElement('HABITACIÓN :' + objA.getHabitacion());
        moR.addElement('COSTO :$ ' + String.format('%.2f', objA.asignaCosto()));
        moR.addElement('Nº DÍAS :' + objA.getDias());
        moR.addElement('————————————————————');
        moR.addElement('MONTO A PAGAR: $ ' + String.format('%.2f', monto));
        lstResumen.setModel(moR);
        moE.clear();
```

```
        moE.addElement('Total de Alquiler : ' + total);
        moE.addElement('Total Alquiler Simple: ' + cSimple);
        moE.addElement('Total Alquiler Doble: ' + cDoble);
        moE.addElement('Total Alquiler Matrimonial: ' + cMatrimonial);
        moE.addElement('-----------------------');
        moE.addElement('Porcentaje de Alquiler Simple: ' + porSimple + '%');
        moE.addElement('Porcentaje de Alquiler Doble: ' + porDoble + '%');
        moE.addElement('Porcentaje de Alquiler Matrimonial:'+porMatrimonial+'%');
        moE.addElement('-----------------------');
        moE.addElement('Total de personas hospedadas en Simple: ' + pSimple);
        moE.addElement('Total de personas hospedadas en Doble: ' + pDoble);
        moE.addElement('Total de personas hospedadas en Matrimonial:'+pMatrimonial);
        moE.addElement('-----------------------');
        moE.addElement('Monto Total en Simple: $'+String.format('%.2f',mSimple));
        moE.addElement('Monto Total en Doble: $'+ String.format('%.2f', mDoble));
        moE.addElement('Monto Total en Matrimonial: $'
                    + String.format('%.2f', mMatrimonial));
        lstEstadisticas.setModel(moE);
    }

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        txtDias.setText('');
        txtPersonas.setText('');
        cboHabitacion.setSelectedIndex(0);
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estás seguro de
                        salir...?', 'HOTEL', JOptionPane.YES_NO_
                        OPTION, JOptionPane.QUESTION_MESSAGE, null,
                        null, null);
        if (r == 0) {
            System.exit(0);
        }
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 3: **Registro de venta (usando for)**

Una librería necesita tener el control de las ventas de cuadernos escolares para el año escolar en curso. Implemente una aplicación que permita controlar la venta de cuadernos, la cual se realiza mediante la selección de un producto y una cantidad determinada. Los precios de los productos son como siguen:

Descripción del producto	Precio $
Cuadernos LayConsa	1.50
Cuadernos Justus	1.90
Cuadernos StanFord	3.50
Cuadernos David	2.50
Cuadernos College	3.00
Cuadernos Alpha	4.50

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**.
- Use el siguiente diseño de interfaz gráfica:

- Debe seleccionar un producto desde el control cuadro combinado según la tabla proporcionada en el caso.
- Debe ingresar la cantidad de productos seleccionados por el cliente en un cuadro de texto.
- Cree una clase llamada **Venta** que defina los atributos referentes a los datos de la venta.
- Debe validar todos los valores ingresados mostrando un mensaje de error si fuera necesario.
- El subtotal es el valor acumulado de la lista subtotal, para ello debe usar la estructura repetitiva **for**.
- Se deben imprimir todos los valores obtenidos en los controles **JList**.
- El descuento representa al 10% del subtotal acumulado.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion73**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmVenta**.
5. Agregue una clase llamada **Venta** en el paquete **pClases**.

6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

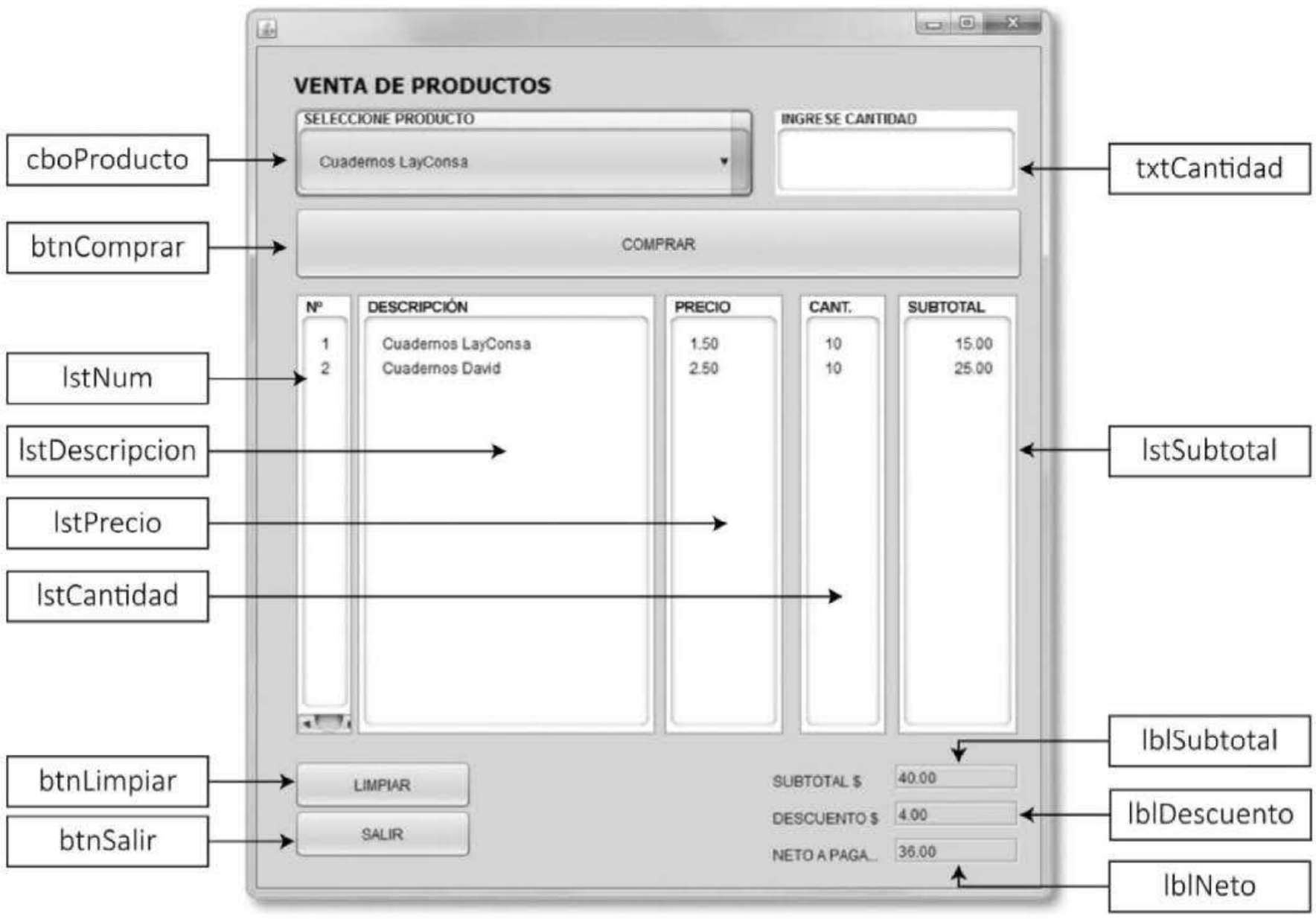


Figura 7.10 Controles del formulario de control de venta.

Resumen de controles:

N.º	Clase	Control
1	JLabel	VENTA DE PRODUCTOS
2	JCombobox	cboProducto
3	JTextField	txtCantidad
4	JLabel	lblSubtotal
5	JLabel	lblDescuento
6	JLabel	lblNeto
7	JList	lstNum
8	JList	lstDescripcion
9	JList	lstPrecio
10	JList	lstCantidad
11	JList	lstSubtotal
12	JButton	btnComprar
13	JButton	btnLimpiar
14	JButton	btnSalir

7. En la clase **Venta** coloque el siguiente código:

```
package pClases;

public class Venta {
    //Atributo
    private int codigo;
    private String descripcion;
    private int cantidad;

    //Métodos Get y Set
    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    //Métodos propios
    public double asignaPrecio() {
        switch (codigo) {
        case 0:
            return 1.5;
        case 1:
            return 1.9;
        case 2:
            return 3.5;
        case 3:
            return 2.5;
        case 4:
            return 3.0;
        default:
            return 4.5;
        }

    public double calculaSubtotal() {
        return asignaPrecio() * cantidad;
    }
}
```

8. El código del formulario **frmVenta** se muestra como sigue:

```
package pFormularios;

import java.text.DecimalFormat;
import javax.swing.DefaultListCellRenderer.UIResource;
import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import javax.swing.SwingConstants;
import pClases.Venta;

public class frmVenta extends javax.swing.JFrame {
    //Definición de modelos
    DefaultListModel moNumero - new DefaultListModel();
    DefaultListModel moDescripcion - new DefaultListModel();
    DefaultListModel moPrecio - new DefaultListModel();
    DefaultListModel moCantidad - new DefaultListModel();
    DefaultListModel moSubtotal - new DefaultListModel();

    //Formato para decimales
    DecimalFormat df;

    //Número de Producto
    int num;

    public frmVenta() {
        initComponents();
        cargaProductos();
        df - new DecimalFormat('###0.00');
    }

    //Métodos que llenan los cuadros combinados
    void cargaProductos() {
        cboProducto.addItem('Cuadernos LayConsa');
        cboProducto.addItem('Cuadernos Justus');
        cboProducto.addItem('Cuadernos StanFord');
        cboProducto.addItem('Cuadernos David');
        cboProducto.addItem('Cuadernos College');
        cboProducto.addItem('Cuadernos Alpha');
    }

    private void btnComprarActionPerformed(java.awt.event.ActionEvent evt) {
        if (valida().equals('')) {
            //Objeto de la clase Venta
            Venta objV - new Venta();

            //Enviando los valores a la clase
            objV.setCodigo(getProducto());
            objV.setDescripcion(getDescripcionProducto());
            objV.setCantidad(getCantidad());

            //Obtener los cálculos desde la clase
            double precio - objV.asignaPrecio();
            double subtotal - objV.calculaSubtotal();

            //Imprimir en las listas
            imprimirListas(precio, subtotal);

            //Ubicar el valor en el lado derecho
            UIResource posicion - new UIResource();
            posicion.setHorizontalAlignment(SwingConstants.RIGHT);
            lstSubtotal.setCellRenderer(posicion);

            limpiarControles();
```

```
        } else {
            JOptionPane.showMessageDialog(null, 'Error en ' + valida());
        }
    }

    private void btnLimpiarActionPerformed(java.awt.event.ActionEvent evt) {
        limpiarControles();

        //Limpiando los modelos
        moNumero.removeAllElements();
        moDescripcion.removeAllElements();
        moCantidad.removeAllElements();
        moPrecio.removeAllElements();
        moSubtotal.removeAllElements();

        //Limpiando los controles
        lblSubtotal.setText('');
        lblDescuento.setText('');
        lblNeto.setText('');

        num = 0;
    }

    private void btnSalirActionPerformed(java.awt.event.ActionEvent evt) {
        int r = JOptionPane.showOptionDialog(this, 'Estás seguro de
                                salir...?', 'Sistema de Ventas', JOptionPane.
                                YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE,
                                null, null, null);
        if (r == 0) {
            System.exit(0);
        }
    }

    //Método de validación
    String valida() {
        if (cboProducto.getSelectedIndex() == -1) {
            cboProducto.requestFocus();
            return 'Descripción del Producto';
        } else if (txtCantidad.getText().equals('')
                    || Integer.parseInt(txtCantidad.getText()) < 0) {
            txtCantidad.setText('');
            txtCantidad.requestFocus();
            return 'Cantidad comprada';
        } else {
            return '';
        }
    }

    //Método que limpia los controles
    void limpiarControles() {
        cboProducto.setSelectedIndex(0);
        txtCantidad.setText('');
        cboProducto.requestFocus();
    }

    //Método que calcula los totales
    void calculaTotales() {
        int n = moNumero.size();
        double subtotal = 0;

        for (int i = 0; i < n; i++) {
            subtotal+=Double.parseDouble(moSubtotal.getElementAt(i).toString());
        }
```

```
        double descuento = subtotal * 0.1;
        double neto = subtotal-descuento;

        lblSubtotal.setText(df.format(subtotal));
        lblDescuento.setText(df.format(descuento));
        lblNeto.setText(df.format(neto));
    }

    //Método para imprimir en las listas
    void imprimirListas(double precio, double subtotal) {
        num++;
        moNumero.addElement(num);
        moDescripcion.addElement(getDescripcionProducto());
        moPrecio.addElement(df.format(precio));
        moCantidad.addElement(getCantidad());
        moSubtotal.addElement(df.format(subtotal));
        calculaTotales();
        lstNum.setModel(moNumero);
        lstDescripcion.setModel(moDescripcion);
        lstPrecio.setModel(moPrecio);
        lstCantidad.setModel(moCantidad);
        lstSubtotal.setModel(moSubtotal);
    }

    //Método que devuelve el índice del producto
    int getProducto() {
        return cboProducto.getSelectedIndex();
    }

    //Método que devuelve la descripción del producto
    String getDescripcionProducto() {
        return String.valueOf(cboProducto.getSelectedItem());
    }

    //Método que devuelve la cantidad seleccionada
    int getCantidad() {
        return Integer.parseInt(txtCantidad.getText());
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 4: Control de pago (usando modificar static)

Una empresa desea tener el control de pago de sus empleados. La empresa cuenta con la siguiente información: nombres y apellidos del empleado, horas trabajadas, pago por hora según el cargo y las bonificaciones por modalidad de empleo. Véanse las siguientes tablas:

Cargo del empleado	Pago por hora
Gerente	$ 20.00
Administrativo	$10.00
Jefe	$ 8.00
Operario	$ 3.50

Modo de empleo	Bonificación %
Tiempo completo	20 %
Tiempo parcial	5 %

```
        double descuento = subtotal * 0.1;
        double neto = subtotal-descuento;

        lblSubtotal.setText(df.format(subtotal));
        lblDescuento.setText(df.format(descuento));
        lblNeto.setText(df.format(neto));
    }

    //Método para imprimir en las listas
    void imprimirListas(double precio, double subtotal) {
        num++;
        moNumero.addElement(num);
        moDescripcion.addElement(getDescripcionProducto());
        moPrecio.addElement(df.format(precio));
        moCantidad.addElement(getCantidad());
        moSubtotal.addElement(df.format(subtotal));
        calculaTotales();
        lstNum.setModel(moNumero);
        lstDescripcion.setModel(moDescripcion);
        lstPrecio.setModel(moPrecio);
        lstCantidad.setModel(moCantidad);
        lstSubtotal.setModel(moSubtotal);
    }

    //Método que devuelve el índice del producto
    int getProducto() {
        return cboProducto.getSelectedIndex();
    }

    //Método que devuelve la descripción del producto
    String getDescripcionProducto() {
        return String.valueOf(cboProducto.getSelectedItem());
    }

    //Método que devuelve la cantidad seleccionada
    int getCantidad() {
        return Integer.parseInt(txtCantidad.getText());
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 4: Control de pago (usando modificar static)

Una empresa desea tener el control de pago de sus empleados. La empresa cuenta con la siguiente información: nombres y apellidos del empleado, horas trabajadas, pago por hora según el cargo y las bonificaciones por modalidad de empleo. Véanse las siguientes tablas:

Cargo del empleado	Pago por hora
Gerente	$ 20.00
Administrativo	$10.00
Jefe	$ 8.00
Operario	$ 3.50

Modo de empleo	Bonificación %
Tiempo completo	20 %
Tiempo parcial	5 %

Se pide calcular:

- Total de empleados registrados
- Total de empleados registrados como gerentes
- Total de empleados registrados como administrativos
- Total de empleados registrados como jefes
- Total de empleados registrados como operarios
- Cantidad de empleados que ganan menos a $ 1200.00
- Cantidad de empleados que ganan entre $ 1200.00 y $ 2500.00
- Cantidad de empleados que ganas más de $ 2500.00

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos con los métodos **get** y **set**, los modificadores **static** y el método **constructor**.
- Use el siguiente diseño de interfaz gráfica:

- En un cuadro de texto, debe ingresar el nombre del empleado, así como las horas de trabajo.
- Debe seleccionar un cargo y una modalidad de trabajo a partir de controles **JCombobox**.
- Cree una clase llamada **Pago** que defina los atributos referentes a los datos del pago.
- Debe validar todos los valores ingresados mostrando un mensaje de error si fuera necesario.
- Muestre los datos del empleado en un control **JList**.
- Muestre las estadísticas en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion74**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmPago**.
5. Agregue una clase llamada **Pago** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

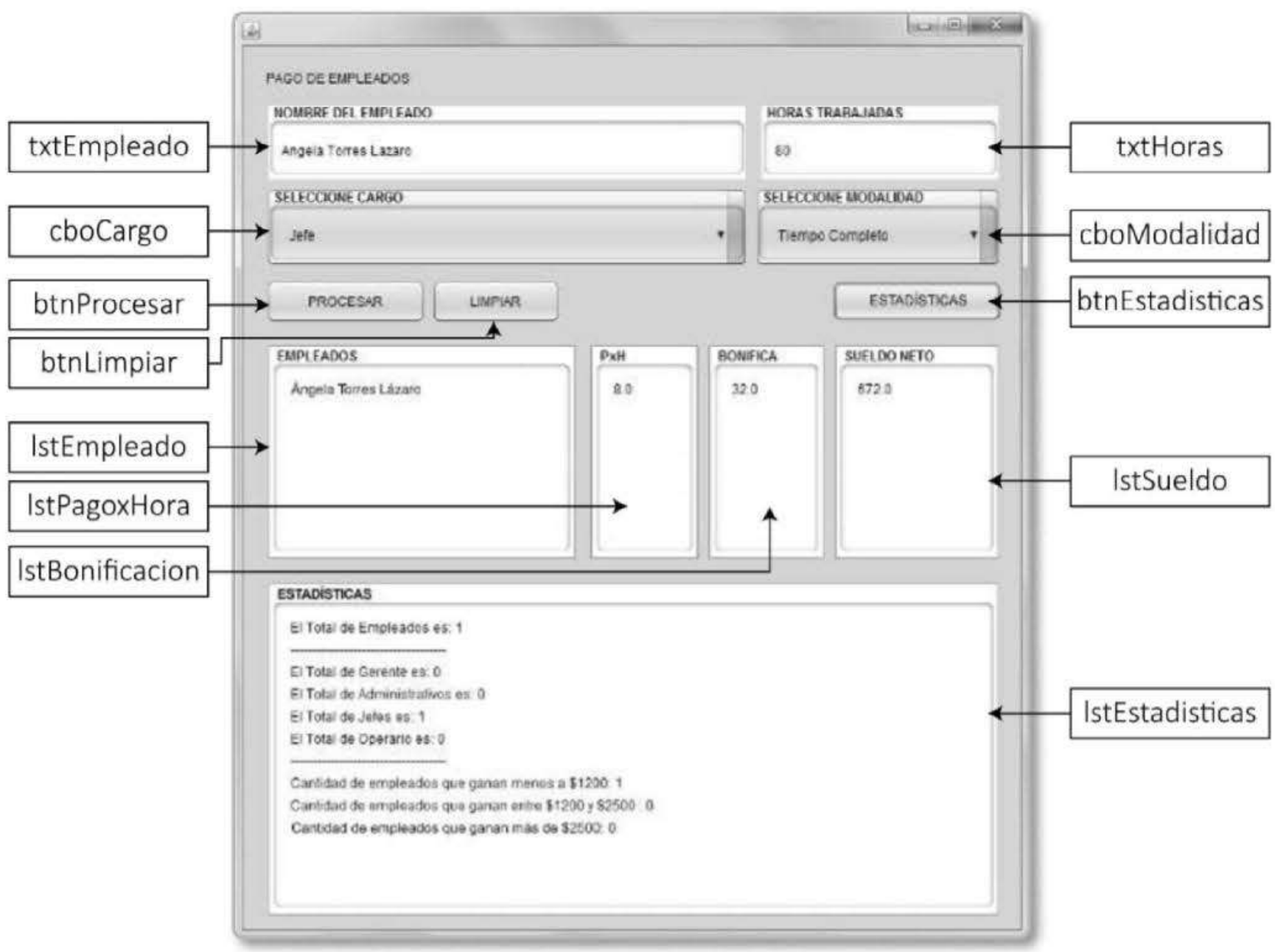


Figura 7.11 Controles del formulario de control de pago de empleados.

Resumen de controles:

N.º	Clase	Control
1	JLabel	PAGO DE EMPLEADOS
2	JTextField	txtEmpleado
3	JTextField	txtHoras
4	JCombobox	cboCargo
5	JCombobox	cboModalidad
6	JList	lstEmpleado

N.º	Clase	Control
7	JList	lstPagoxHora
8	JList	lstBonificacion
9	JList	lstSueldo
10	JList	lstEstadisticas
11	JButton	btnProcesar
12	JButton	btnLimpiar

7. En la clase **Pago** coloque el siguiente código:

```
package pClases;
public class Pago {
    //Atributos
    private String empleado;
    private String cargo;
    private String modalidad;
    private int horas;

    private static int total - 0;
    private static int tGerente - 0;
    private static int tAdministrativo - 0;
    private static int tJefe - 0;
    private static int tOperario - 0;

    private static int cMenos1200 - 0;
    private static int cEntre1200y2500 - 0;
    private static int cMas2500 - 0;

    //Definición de método constructor
    public Pago(String empleado, String cargo, String modalidad, int horas) {
        this.empleado - empleado;
        this.cargo - cargo;
        this.modalidad - modalidad;
        this.horas - horas;
        total++;
        contadorCargos();
        contadorSueldo();
    }

    private void contadorCargos() {
        switch (cargo) {
            case 'Gerente':
                tGerente++;
                break;
            case 'Administrativo':
                tAdministrativo++;
                break;
            case 'Jefe':
                tJefe++;
                break;
            default:
                tOperario++;
        }
    }
```

```
    //Método que determina el pago por hora según el cargo
    public double asignaPagoxHora() {
        switch (cargo) {
            case 'Gerente':
                return 20;
            case 'Administrativo':
                return 10;
            case 'Jefe':
                return 8;
            default:
                return 3.5;
        }
    }

    //Método que determina el pago por hora según el cargo
    public double calculaBruto() {
        return this.horas * asignaPagoxHora();
    }
    public double asignaBonificacion(double bruto) {
        switch (modalidad) {
            case 'Tiempo completo':
                return 0.2 * bruto;
            default:
                return 0.05 * bruto;
        }
    }

    //Método que determina el pago por hora según el cargo
    public double calculaSueldo() {
        return calculaBruto() + asignaBonificacion(calculaBruto());
    }

    //Método que determina el pago por hora según el cargo
    private void contadorSueldo() {
        if (calculaSueldo() < 1200) {
            cMenos1200++;
        } else if (calculaSueldo() <- 2500) {
            cEntre1200y2500++;
        } else {
            cMas2500++;
        }
    }

    //Métodos Get y Set
    public String getEmpleado() {
        return empleado;
    }

    public void setEmpleado(String empleado) {
        this.empleado - empleado;
    }

    public String getCargo() {
        return cargo;
    }

    public void setCargo(String cargo) {
        this.cargo - cargo;
    }

    public String getModalidad() {
        return modalidad;
    }
```

```
        public void setModalidad(String modalidad) {
            this.modalidad = modalidad;
        }

        public int getHoras() {
            return horas;
        }

        public void setHoras(int horas) {
            this.horas = horas;
        }

        public static int getTotal() {
            return total;
        }

        public static int gettGerente() {
            return tGerente;
        }

        public static int gettAdministrativo() {
            return tAdministrativo;
        }

        public static int gettJefe() {
            return tJefe;
        }

        public static int gettOperario() {
            return tOperario;
        }

        public static int getcMenos1200() {
            return cMenos1200;
        }

        public static int getcEntre1200y2500() {
            return cEntre1200y2500;
        }

        public static int getcMas2500() {
            return cMas2500;
        }
    }
```

8. El código del formulario **frmPago** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Pago;
public class frmPago extends javax.swing.JFrame {
    //Definición de modelos
    DefaultListModel moEmpleado = new DefaultListModel();
    DefaultListModel moPagoHora = new DefaultListModel();
    DefaultListModel moBonificacion = new DefaultListModel();
    DefaultListModel moSueldo = new DefaultListModel();
    DefaultListModel moEstadisticas = new DefaultListModel();
```

```
    //Definición del objeto de la clase Pago
    Pago objP;

    public frmPago() {
        initComponents();

        //Definicion de modelos
        llenaCargo();
        llenaModalidad();
    }

    //Llenar el combo de cargos
    void llenaCargo() {
        cboCargo.addItem('Gerente');
        cboCargo.addItem('Administrativo');
        cboCargo.addItem('Jefe');
        cboCargo.addItem('Operario');
    }

    //Llenar el combo de modalidades
    void llenaModalidad() {
        cboModalidad.addItem('Tiempo Completo');
        cboModalidad.addItem('Tiempo Parcial');
    }

    //Métodos para capturar los valores del formulario
    String getEmpleado() {
        return txtEmpleado.getText();
    }

    int getHoras() {
        return Integer.parseInt(txtHoras.getText());
    }

    String getCargo() {
        return String.valueOf(cboCargo.getSelectedItem());
    }

    String getModalidad() {
        return String.valueOf(cboModalidad.getSelectedItem());
    }

    //Validación de controles
    String valida() {
        if (txtEmpleado.getText().equals('')) {
            return 'Nombre del Empleado';
        } else if (txtHoras.getText().equals('')
                    || Integer.parseInt(txtHoras.getText()) < 0) {
            return 'Horas de trabajo';
        } else if (cboCargo.getSelectedIndex() == -1) {
            return 'Cargo del empleado';
        } else if (cboModalidad.getSelectedIndex() == -1) {
            return 'Modalidad del empleado';
        } else {
            return '';
        }
    }
```

```
    //Definición del objeto de la clase Pago
    Pago objP;

    public frmPago() {
        initComponents();

        //Definicion de modelos
        llenaCargo();
        llenaModalidad();
    }

    //Llenar el combo de cargos
    void llenaCargo() {
        cboCargo.addItem('Gerente');
        cboCargo.addItem('Administrativo');
        cboCargo.addItem('Jefe');
        cboCargo.addItem('Operario');
    }

    //Llenar el combo de modalidades
    void llenaModalidad() {
        cboModalidad.addItem('Tiempo Completo');
        cboModalidad.addItem('Tiempo Parcial');
    }

    //Métodos para capturar los valores del formulario
    String getEmpleado() {
        return txtEmpleado.getText();
    }

    int getHoras() {
        return Integer.parseInt(txtHoras.getText());
    }

    String getCargo() {
        return String.valueOf(cboCargo.getSelectedItem());
    }

    String getModalidad() {
        return String.valueOf(cboModalidad.getSelectedItem());
    }

    //Validación de controles
    String valida() {
        if (txtEmpleado.getText().equals('')) {
            return 'Nombre del Empleado';
        } else if (txtHoras.getText().equals('')
                    || Integer.parseInt(txtHoras.getText()) < 0) {
            return 'Horas de trabajo';
        } else if (cboCargo.getSelectedIndex() == -1) {
            return 'Cargo del empleado';
        } else if (cboModalidad.getSelectedIndex() == -1) {
            return 'Modalidad del empleado';
        } else {
            return '';
        }
    }
```

```
    private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
        if (valida().equals('')) {
            objP = new Pago(getEmpleado(), getCargo(), getModalidad(), getHoras());

            moEmpleado.addElement(getEmpleado());
            moPagoHora.addElement(objP.asignaPagoxHora());
            double bruto = objP.calculaBruto();
            moBonificacion.addElement(objP.asignaBonificacion(bruto));
            moSueldo.addElement(objP.calculaSueldo());

            lstEmpleado.setModel(moEmpleado);
            lstPagoxHora.setModel(moPagoHora);
            lstBonificacion.setModel(moBonificacion);
            lstSueldo.setModel(moSueldo);
            lstEstadisticas.setModel(moEstadisticas);
        } else {
            JOptionPane.showMessageDialog(null, 'El error está en ' + valida());
        }
    }

    private void btnEstadisticasActionPerformed(java.awt.event.ActionEvent evt) {
        moEstadisticas.clear();
        moEstadisticas.addElement('El Total de Empleados es: '+ objP.getTotal());
        moEstadisticas.addElement('————————————————-');
        moEstadisticas.addElement('El Total de Gerente es: '+objP.gettGerente());
        moEstadisticas.addElement('El Total de Administrativos es: ' +
                                  objP.gettAdministrativo());
        moEstadisticas.addElement('El Total de Jefes es: ' + objP.gettJefe());
        moEstadisticas.addElement('El Total de Operario es: ' +
                                  objP.gettOperario());
        moEstadisticas.addElement('————————————————-');
        moEstadisticas.addElement('Cantidad de empleados que ganan menos
                                  a $1200: '+ objP.getcMenos1200());
        moEstadisticas.addElement('Cantidad de empleados que
                                  ganan $1200 y $2500 : ' + objP.
                                  getcEntre1200y2500());
        moEstadisticas.addElement('Cantidad de empleados que ganan más de
                                  $2500: '+ objP.getcMas2500());
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

Lambda y manejo de excepciones

CAPÍTULO

8

Competencias

- Implementa métodos usando **Lambda**.
- Reconoce el manejo del bloque **try-catch**.
- Implementa aplicaciones usando el control de excepciones Java.

Contenido

8.1 Generalidades

8.2 Expresión **Lambda**

8.3 Bloque **try-catch**

8.4 Cláusula **throw**

8.5 Bloque **finally**

- Caso desarrollado 1: Registro de libros
- Caso desarrollado 2: Control de pago de empleados usando **Lambda**

8.1 Generalidades

Cuando se desarrolla una aplicación, ya sea en cualquier lenguaje de programación, siempre se pensará en implementarla con cero errores; sin embargo, muchas veces no se prevé los posibles errores que un usuario puede ocasionar en la aplicación. Esto resulta algo muy particular. Así, por ejemplo, cuando se compra un producto en una tienda comercial, siempre nos aseguran que el aparato no tendrá problemas, al menos, en varios años; no obstante, a veces no sucede así y quedamos algo decepcionados y sin ganas de volver a comprar en dicha tienda. Algo así sucede con las aplicaciones, pero la vida es así, estamos rodeados de errores. Y solo la experiencia será la que nos permita evitar, prevenir o anticipar los posibles errores que se pueden ocasionar en una aplicación. Dicho esto, no pensemos que controlar los errores sea complicado, sino más bien que el control de estos no debe quedar desapercibido por el desarrollador.

En Java, las situaciones que pueden provocar un fallo en la aplicación se las denomina *excepciones*. Este capítulo se enfocará en mostrar la mejor manera de cómo controlarlas. Las excepciones en Java son objetos de clases derivadas de la clase base **Exception**. Existen también los errores internos que son objetos de la clase **Error** que no estudiaremos. Ambas clases, **Error** y **Exception**, son clases derivadas de la clase base **Throwable**.

Java implementa el código de intento para controlar las excepciones que se pueden ocasionar en una aplicación. Este bloque de código protege la secuencialidad del script, de modo que, si se produce una excepción, esta será controlada mediante un mensaje o alguna instrucción y no permitirá devolverlo al script de la aplicación, como ocurre con cualquier excepción no controlada.

Los errores que se pueden encontrar en las aplicaciones podrían ser:

- Por divisiones entre cero.
- Al intentar acceder a elementos de arreglos con un índice mayor al declarado.
- Al acceder a una base de datos.
- En el manejo de archivos (existencia o permiso).
- Errores definidos por el usuario.

8.2 Expresión Lambda

Es una de las nuevas características que presenta Java 8. Esta nos permitirá simplificar bloques de código que implementábamos en los métodos de devolución de valor. **Lambda**, por característica, define un proceso de forma directa, lo cual nos permitirá ahorrar líneas de código y nos sumergirá en una nueva propuesta de implementación de métodos. Dichos métodos ya han sido implementados por otros lenguajes de programación.

A continuación, se muestran algunas particularidades de la definición de **Lambda**:

- Definiendo parámetros con especificación de tipo de datos y devolviendo la suma de los mismos mediante la cláusula **return**:

```
(int n1, int n2) -> { return n1 + n2; }
```

Como se trata de una sola expresión de devolución, también podría escribirse de la siguiente manera:

```
(int n1, int n2) -> n1 + n2;
```

- Definiendo parámetros sin especificar el tipo de datos y devolviendo valores sin el uso de la cláusula **return**:

```
(n1, n2) -> n1 + n2;
```

- Definiendo un solo parámetro:

```
n1 -> n1 * 2;
```

- Definiendo sin valores de entrada y devolviendo un valor:

```
() -> n1
```

- Definiendo un único parámetro sin definición de tipo, y un retorno sin valor:

```
n1 -> { imprimir(n1); }
```

¿Cómo implementar un método Lambda en Java?

Obsérvese los siguientes casos:

> Método **Lambda** que permita sumar dos números enteros.

1. Definir la interface

```
public interface Operacion {
    int calcular(int n1, int n2);
}
```

2. Definir el método Lambda

```
public Operacion calculaSuma = (n1, n2) -> n1 + n2;
```

3. Invocar al método Lambda

```
int suma = calculaSuma.calcular(100,50);
```

> Método **Lambda** que determine el costo basado en el nombre de un producto.

1. Definir la interface

```
public interface Operacion {
    double determina(String producto);
}
```

2. Definir el método Lambda

```
public Operacion asignaPrecio = (producto) -> {
    if (producto.equasl('Lavadora'))
        return 1200;
    else if (producto.equasl('Refrigeradora'))
        return 3500;
    else if (producto.equasl('Cocina'))
        return 700;
    else
        return 0;
};
```

3. Invocar al método Lambda

```
double precio = asignaPrecio.determina('Lavadora');
```

8.3 Bloque try-catch

Un bloque **try** permite proteger un script que podría generar algún tipo de error al ejecutar una aplicación.

Formato:

```
try{
    //Instrucciones
} catch(clase identificador){
    //instrucciones catch
}
```

Dentro del bloque **try** se coloca todo script que pueda generar algún tipo de error; al terminar este bloque se define un grupo de instrucciones en un bloque **catch**. Cada bloque **catch** es seguido por un paréntesis que contiene una clase y un identificador, los cuales son considerados como datos de entrada para cada sección **catch**; el tipo de clase definida dentro de cada sección **catch** depende del número de excepciones que pueden ser generadas por una misma aplicación.

He aquí un script que muestra una excepción:

```
String sSueldo='1000x';
double dSueldo=Double.parseDouble(sSueldo);
JOptionPane.showMessageDialog(null, 'El resultado es: '+dSueldo);
```

El error parece ser bien claro, pues se tiene una variable de tipo cadena con un valor inicial de 1000x que, al convertirla al tipo **double**, genera el siguiente error:

La excepción se podría controlar por medio del siguiente script:

```
try{
    String sSueldo='1000x';
    double dSueldo=Double.parseDouble(sSueldo);
    JOptionPane.showMessageDialog(null, 'El resultado es: '+dSueldo);
} catch(NumberFormatException ex){
    JOptionPane.showMessageDialog(null, ex.getMessage());
}
```

El resultado sería como se muestra en la siguiente imagen:

O se podría mostrar un mensaje genérico de la excepción con el siguiente script:

```
try{
    String sSueldo='1000x';
    double dSueldo=Double.parseDouble(sSueldo);
    JOptionPane.showMessageDialog(null, 'El resultado es: '+dSueldo);
} catch(Exception ex){
    JOptionPane.showMessageDialog(null, 'Error en la aplicación..!! ');
}
```

8.4 Cláusula throw

Es la cláusula que permite listar los tipos de excepción que un método privado puede lanzar. También se la denomina lanzamiento manual de excepción. Sus principales características, son las siguientes:

- Se usa para proteger los métodos de una clase.
- Es necesario usarla con todas las excepciones, excepto la clase **Error** y **RuntimeException**.
- Cuando se implementa un método, se tiene que especificar la cláusula **throw**; en caso contrario, se genera un error.

Formato de implementación de un método:

```
Visibilidad tipoDatos nombreMetodo(Parametros) throws nombreExcepcion{
    //Instrucciones
}
```

Formato de implementación de clase heredada de la clase **Exception**:

```
public class nombreExcepcion extends Exception{
    public nombreExcepcion(){
        super('Mensaje1');
    }
    public nombreExcepcion(tipo valor){
        super(valor);
    }
}
```

A continuación, se muestra un ejemplo de cómo validar un número entero usando **throw**:

Paso 1. Se implementa la clase **validaNota** con dos métodos constructores; el primero mostrará el mensaje "Fuera del Rango".

```
public class validaNota extends Exception{
    public validaNota(){
        super('Error al convertir');
    }
    public validaNota(String valor){
        super(valor);
    }
}
```

Paso 2. Se implementa el método **validaNumero** que invoca a la clase **validaNota** por medio de la cláusula **throws**.

```
String validaNumero(int nota) throws validaNota{
    if (nota<0 || nota>20)
        throw new validaNota();
    return 'Numero valido';
}
```

Paso 3. Se implementa el bloque **try-catch**, el cual puede colocarse dentro del botón de acción de un formulario.

```
try {
    int nota=22;
    String valida=validaNumero(nota);
    JOptionPane.showMessageDialog(null, valida);
} catch (validaNota ex) {
    JOptionPane.showMessageDialog(null, ex);
}
```

8.5 Bloque finally

Es un bloque que permite ejecutar un conjunto de instrucciones, ocurra o no una excepción controlada con **try-catch**. Normalmente es usado para liberar recursos de la aplicación.

Formato:

```
try{
    //Instrucciones
} catch(clase identificador){
    //Instrucciones catch
} finally {
    //Instrucciones Finally
}
```

CASO DESARROLLADO 1: Registro de libros

Implemente una aplicación que permita registrar la siguiente información de los libros en una biblioteca: nombre del libro, tipo de editorial (A, B o C), clase de libro (Programación, Análisis, Diseño), año de edición, número de páginas y costo del libro. La aplicación deberá mostrar los siguientes resultados:

- Número de libros de Análisis de la editorial B.
- Nombre del libro con el año de edición más reciente.
- Nombre de la editorial que tiene el libro con el menor número de páginas.
- Nombre del libro que tenga el mayor costo.

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos.
- Use el siguiente diseño de interfaz gráfica:

- Debe ingresar el nombre del libro, año de edición, número de páginas y el costo en controles de cuadros de textos.
- Debe seleccionar un tipo de editorial, clase de libro a partir de controles **JCombobox**.
- Cree una clase llamada **Libro** que defina los atributos referentes al registro de libro.
- Muestre los datos del registro de libros en un control **JList**.
- Muestre las estadísticas en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion81**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmRegistro**.
5. Agregue una clase llamada **Libro** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

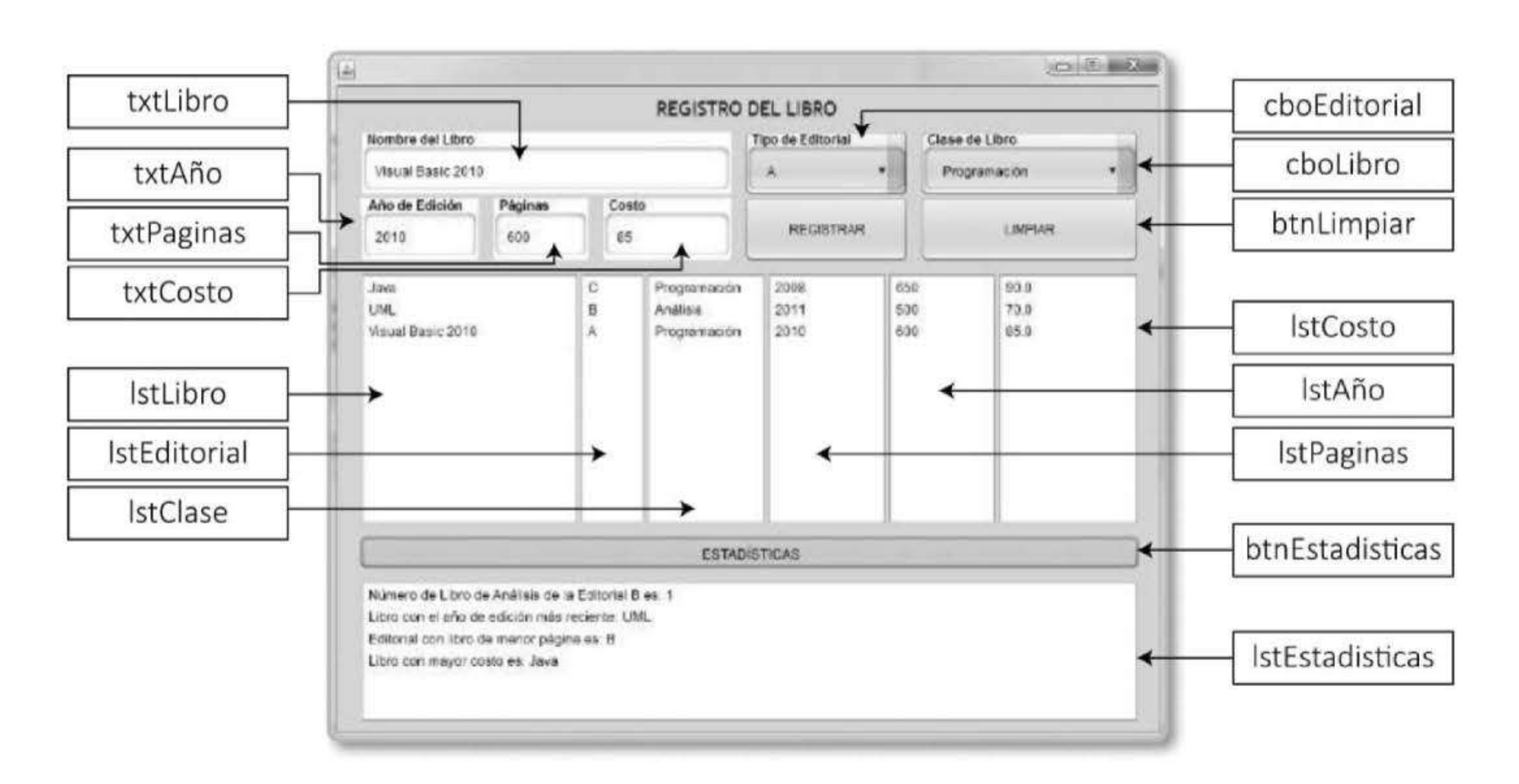


Figura 8.1 Controles del formulario de control de registro de libros.

Resumen de controles:

N.º	Clase	Control
1	JLabel	REGISTRO DE LIBRO
2	JTextField	txtLibro
3	JTextField	txtAño
4	JTextField	txtPaginas
5	JTextField	txtCosto

N.º	Clase	Control
6	JCombobox	cboEditorial
7	JCombobox	cboLibro
8	JList	lstLibro
9	JList	lstEditorial
10	JList	lstClase
11	JList	lstAño
12	JList	lstPaginas
13	JList	lstCosto
14	JList	lstEstatisticas
15	JButton	btnRegistrar
16	JButton	btnLimpiar
17	JButton	btnEstadisticas

7. En la clase **Libro** coloque el siguiente código:

```
package pClases;
public class Libro {
    //Atributos de la clase Libro
    private String nombre;
    private String editorial;
    private String clase;
    private int año;
    private int paginas;
    private double costo;

    //Variables de clase
    private static int tAnalisisB;

    //Implementación del método constructor
    public Libro(String nombre,String editorial,String clase, int año,int
paginas,double costo){
        this.nombre=nombre;
        this.editorial=editorial;
        this.clase=clase;
        this.año=año;
        this.paginas=paginas;
        this.costo=costo;

        conteos();
    }

    //Métodos GET/SET de la clase
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getEditorial() {
        return editorial;
    }
```

```
    public void setEditorial(String editorial) {
        this.editorial = editorial;
    }

    public String getClase() {
        return clase;
    }

    public void setClase(String clase) {
        this.clase = clase;
    }

    public int getAño() {
        return año;
    }

    public void setAño(int año) {
        this.año = año;
    }

    public int getPaginas() {
        return paginas;
    }

    public void setPaginas(int paginas) {
        this.paginas = paginas;
    }

    public double getCosto() {
        return costo;
    }

    public void setCosto(double costo) {
        this.costo = costo;
    }

    public int getTAnalisis(){
        return tAnalisisB;
    }

    //Método que permite contar según el conteo establecido en el caso
    void conteos(){
        if (getClase().equals('Analisis') && getEditorial().equals('B'))
            tAnalisisB++;
    }
}
```

8. El código del formulario **frmRegistro** se muestra como sigue:

```
package pFormularios;

//Importación de librerías necesarias en la aplicación
import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.Libro;

public class frmRegistro extends javax.swing.JFrame {
    //Declarando el objeto objLi de la clase Libro
    Libro objLi;

    //Declarando los modelos por cada lista
    DefaultListModel moLibro,moEditorial,moClase,moAño, moPaginas,moCosto;
    DefaultListModel moEstadisticas;
```

```
    public frmRegistro() {
        initComponents();

        //Invocando a los métodos que llenan los controles JCombo
        llenaEditorial();
        llenaClase();

        //Invocando al método que crea los objetos de tipo DefaultList
        cargaModelos();
    }

    //Método que permite crear los objetos de la clase DefaultListModel
    //Además asigna los controles JList con su modelo correspondiente
    void cargaModelos(){
        moLibro=new DefaultListModel();
        moEditorial=new DefaultListModel();
        moClase=new DefaultListModel();
        moAño=new DefaultListModel();
        moPaginas=new DefaultListModel();
        moCosto=new DefaultListModel();

        moEstadisticas=new DefaultListModel();

        lstLibro.setModel(moLibro);
        lstEditorial.setModel(moEditorial);
        lstClase.setModel(moClase);
        lstAño.setModel(moAño);
        lstPaginas.setModel(moPaginas);
        lstCosto.setModel(moCosto);

        lstEstadisticas.setModel(moEstadisticas);
    }

    //Método que muestra los valores seleccionados en las listas por medio
    //de sus modelos correspondientes
    void llenaModelos(){
        moLibro.addElement(objLi.getNombre());
        moEditorial.addElement(objLi.getEditorial());
        moClase.addElement(objLi.getClase());
        moAño.addElement(objLi.getAño());
        moPaginas.addElement(objLi.getPaginas());
        moCosto.addElement(objLi.getCosto());
    }

    //Método que llena el control cboEditorial de las Editoriales
    void llenaEditorial(){
        cboEditorial.addItem('A');
        cboEditorial.addItem('B');
        cboEditorial.addItem('C');
    }

    //Método que llena el control cboClase de las Clases especificadas
    //en el caso
    void llenaClase(){
        cboClase.addItem('Programación');
        cboClase.addItem('Análisis');
        cboClase.addItem('Diseño');
    }
```

```
    private void btnRegistrarActionPerformed(…) {
        try{
            //Creando el objeto de la clase Libro y enviando los valores
            objLi - new Libro(getLibro(),getEditorial(),getClase(),
                              getAño(),getPaginas(),getCosto());

            //Llena los datos obtenidos en las listas por medio de los
            //modelos
            llenaModelos();
        } catch(Exception ex){
            JOptionPane.showMessageDialog(null, 'Error en la Aplicacion '
                        + ex.getMessage());
        }
    }
    //Métodos que obtienen los valores ingresados y seleccionados por el usuario.
    String getLibro(){
        return txtLibro.getText();
    }
    String getEditorial(){
        return String.valueOf(cboEditorial.getSelectedItem());
    }
    String getClase(){
        return String.valueOf(cboClase.getSelectedItem());
    }
    int getAño(){
        return Integer.parseInt(txtAño.getText());
    }
    int getPaginas(){
        return Integer.parseInt(txtPaginas.getText());
    }
    double getCosto(){
        return Double.parseDouble(txtCosto.getText());
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 2: Control de pago de empleados usando Lambda

Implemente una aplicación que permita determinar los montos del salario de un empleado. La información que se debe ingresar es el nombre completo del empleado, el cargo o categoría y las horas de trabajo mensual. Debemos tener en cuenta las dos siguientes tablas de valor:

La asignación del costo por hora se establece según la categoría del empleado, quedando de la siguiente manera:

Categoría	Costo por hora
Jefe	$ 50.00
Administrativo	$ 30.00
Operario	$ 20.00
Servicio	$ 10.00

Los descuentos a su salario se establecen bajo el siguiente criterio:

Salario	Porcentaje de descuento
Inferior o igual a 750	7 %
Entre 751 y 1250	12 %
Entre 1251 y 2500	15 %
Superior a 2500	17 %

La aplicación deberá mostrar los siguientes resultados:

- Los datos ingresados por el empleado.
- El costo por hora según su categoría.
- El monto de salario.
- El monto de descuento según la tabla anterior.
- El monto neto a cobrar por el empleado.

Consideraciones:

- Desarrolle la aplicación usando la metodología orientada a objetos.
- Use el siguiente diseño de interfaz gráfica:

- En cuadros de textos, debe ingresar el nombre del empleado y las horas trabajadas.
- Debe seleccionar un tipo de categoría a partir del control **JCombobox**.
- Al seleccionar una categoría, automáticamente debe mostrarse el costo por hora.
- Cree una clase llamada **Pago** que defina las interfaces y los métodos **Lambda** que necesita la aplicación.
- Muestre los datos del control de pago de empleados en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjAplicacion82**.
3. Agregue los paquetes **pFormularios** y **pClases**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmPagoEmpleados**.
5. Agregar una clase llamada **Pago** en el paquete **pClases**.
6. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

Figura 8.2 Diseño del formulario de control de pago de empleados.

Resumen de controles:

N.º	Clase	Control
1	JLabel	CONTROL DE PAGO DE EMPLEADOS-LAMBDA
2	JLabel	lblCosto
3	JTextField	txtEmpleado
4	JTextField	txtHoras
5	JCombobox	cboCategoria
6	JList	lstR
7	JButton	btnRegistrar

7. En la clase **Pago** coloque el siguiente código:

```
package pClases;
public class Pago {

    /* Declaración de interfaces */
    public interface OperacionesCadena {
        double calcular(String cadena);
    }

    public interface OperacionesParametro {
        double calcular(double decimal);
    }

    public interface OperacionesDosParametros {
        double calcular(double decimal1, double decimal2);
    }
    /*Fin de las Interfaces*/

    //Método que calcula el costo por hora
    public OperacionesCadena calculaCosto = (categoria) -> {
        if (categoria.equals('Jefe')) {
            return 50;
        } else if (categoria.equals('Administrativo')) {
            return 30;
        } else if (categoria.equals('Operario')) {
            return 20;
        } else {
            return 10;
        }
    };

    //Método que calcula el salario
    public OperacionesDosParametros calculaSalario=(horas, costo)->horas*costo;

    //Método que calcular el descuento
    public OperacionesParametro calculaDescuento = (salario) -> {
        if (salario <= 750) {
            return 0.07 * salario;
        } else if (salario > 750 && salario <= 1250) {
            return 0.12 * salario;
        } else if (salario > 1250 && salario <= 2500) {
            return 0.15 * salario;
        } else {
            return 0.17 * salario;
        }
    };

    //Método que calcula el neto
    public OperacionesDosParametros calculaNeto=(salario,descuento)->salario-descuento;
}
```

8. El código del formulario **frmPagoEmpleados** se muestra como sigue:

```
package pFormularios;

import javax.swing.DefaultListModel;
import pClases.Pago;
public class frmPagoEmpleados extends javax.swing.JFrame {
    //Declaración GLOBAL
    DefaultListModel moR;
    Pago objP=new Pago();
```

```
    public frmPagoEmpleados() {
        initComponents();
        llenaCategorias();

        //Objeto de modelo y enlace al JList
        moR=new DefaultListModel();
        lstR.setModel(moR);
    }

    //Método que llena las catetorías
    void llenaCategorias(){
        cboCategoria.addItem('Jefe');
        cboCategoria.addItem('Administrativo');
        cboCategoria.addItem('Operario');
        cboCategoria.addItem('Servicio');
    }

    private void cboCategoriaItemStateChanged(java.awt.event.ItemEvent evt) {
        String categoria=String.valueOf(cboCategoria.getSelectedItem());
        double costo=objP.calculaCosto.calcular(categoria);
        lblCosto.setText(String.format('%.2f', costo));
    }

    private void btnRegistrarActionPerformed(java.awt.event.ActionEvent evt) {

        //Capturar los valores del formulario
        String empleado = txtEmpleado.getText();
        String categoria=String.valueOf(cboCategoria.getSelectedItem());
        int horas=Integer.parseInt(txtHoras.getText());

        //Obtener los valores resultantes
        //Obtenidos desde funciones Lambda
        double costo=objP.calculaCosto.calcular(categoria);
        double salario=objP.calculaSalario.calcular(horas, costo);
        double descuento= objP.calculaDescuento.calcular(salario);
        double neto=objP.calculaNeto.calcular(salario, descuento);

        //Imprimir los valores resultantes
        moR.addElement('EMPLEADO: '+empleado);
        moR.addElement('CATEGORÍA: '+categoria);
        moR.addElement('HORAS TRABAJADAS: '+horas);
        moR.addElement('————————————————————');
        moR.addElement('COSTO POR HORA: $'+String.format('%.2f',costo));
        moR.addElement('MONTO SALARIO: $'+String.format('%.2f',salario));
        moR.addElement('MONTO DESCUENTO: $' +
                        String.format('%.2f',descuento));
        moR.addElement('MONTO NETO: $'+String.format('%.2f',neto));
    }
}
```

9. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CAPÍTULO

9

Arreglos

Competencias

- Comprende los conceptos básicos de los arreglos y poder diferenciarlos por las dimensiones.
- Implementa aplicaciones matriciales de una y dos dimensiones, logrando así manipular información masiva en una misma aplicación.

Contenido

9.1 Generalidades

9.2 Arreglos

9.3 Arreglo unidimensional

9.4 Arreglo bidimensional

- Caso desarrollado 1: Listado de números básico
- Caso desarrollado 2: Listado de números usando clase
- Case desarrollado 3: Certamen de belleza
- Caso desarrollado 4: Matriz de números enteros

9.1 Generalidades

Las variables que hemos usado en todas las aplicaciones, hasta el momento, son de tipo simple, es decir, son de una sola caja de memoria y solo pueden contener un valor cada vez. Igualmente, los tipos de datos vistos hasta ahora, como lo son entero (**int**) y real (**double**, **float**), también son considerados simples, puesto que solo pueden almacenar un valor a la vez; es decir, existe una relación de uno a uno entre la variable y el elemento que es capaz de almacenar. En cambio, un dato de tipo estructurado, como los arreglos, pueden almacenar más de un elemento a la vez, con la condición de que todos los elementos deben ser del mismo tipo.

Se exponen, a continuación, los siguientes casos:

- Leer una lista de calificaciones de un aula de clases.
- Encontrar la media aritmética de todas las calificaciones.
- Listar qué alumnos tienen la calificación más alta.
- Ordenar la lista de calificaciones en orden ascendente o descendente.

Ahora, si se quiere registrar 100 o 200 calificaciones sin usar arreglo, tendríamos que declarar 100 o 200 variables diferentes, como, por ejemplo, calificacion1, calificacion2... calificacion200; de esta manera estaríamos separando 200 variables, es decir, 200 espacios de memoria diferentes para almacenar las calificaciones. Sin embargo, esta no sería la forma más adecuada de registrar las calificaciones, ya que se usan muchas variables.

En Java se tendría el siguiente script para declarar las calificaciones:

```
int calificacion1, calificacion1, calificacion1, …,calificacion200;
```

- Calculando la media aritmética:

```
double mediaArit - (calificacion1+calificacion2+….calificacion200)/200;
```

- Imprimir las calificaciones en un modelo:

```
DefaultListModel moCalificaciones - new DefaultListModel();
moCalificaciones.addElement(calificacion1);
moCalificaciones.addElement(calificacion2);
moCalificaciones.addElement(calificacion3);
…
moCalificaciones.addElement(calificacionN);
```

El equivalente al script anterior, usando arreglo, sería de la siguiente forma:

```
int calificación[] - new int(200);
```

- Calculando la media aritmética:

```
double mediaArit;
for(int i=0;i<200;i++){
    mediaArit += calificacion[i];
}
mediaArit = mediaArit/200;
```

- Imprimir las calificaciones en un modelo:

```
DefaultListModel moCalificaciones = new DefaultListModel();

for(int i=0;i<200;i++){
    moCalificaciones.addElement(calificación[i]);
}
```

Ventajas y desventajas de usar arreglos

- Si se conoce la posición dentro del arreglo del elemento que se quiere consultar, la consulta toma un tiempo constante.
- Se pueden utilizar para implementar otras estructuras de datos como pilas, colas, tablas hash, etc.
- El tamaño de un arreglo es fijo, por lo que, si no se conoce de antemano el número máximo de elementos a almacenar, pueden ocurrir problemas si el espacio reservado es menor del que se necesita.
- Insertar elementos de manera ordenada es muy lento.
- Buscar un elemento en un arreglo desordenado es muy lento.

9.2 Arreglos

Un *arreglo* es una colección de datos del mismo tipo; dichos datos se almacenan en posiciones consecutivas de memoria y reciben un nombre de variable común. Es decir, varios datos estarán controlados por una sola variable. Para hacer referencia a un determinado elemento del arreglo se debe utilizar el nombre del arreglo acompañado del índice, el cual especifica la posición relativa en que se encuentra el elemento.

Los arreglos se pueden dividir en:

- Unidimensionales, también llamados *vectores*.

0	1000.00
1	1500.00
2	1200.00
3	1050.00
4	2500.00
5	3000.00

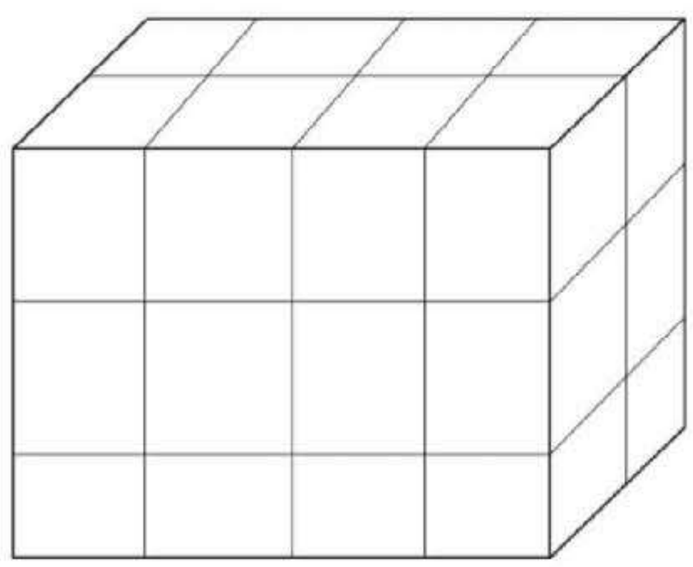
0	1	2	3	4	5
1000.00	1500.00	1200.00	1050.00	2500.00	3000.00

- Bidimensionales, también llamados *matrices* o simplemente *tablas*.

(0,0) 100	(0,1) 50	(0,2) 75
(1,0) 42	(1,1) 30	(1,2) 80
(2,0) 82	(2,1) 51	(2,2) 12

0,0	100
0,1	50
0,2	75
1,0	42
1,1	30
1,2	80
2,0	82
2,1	51
2,2	12

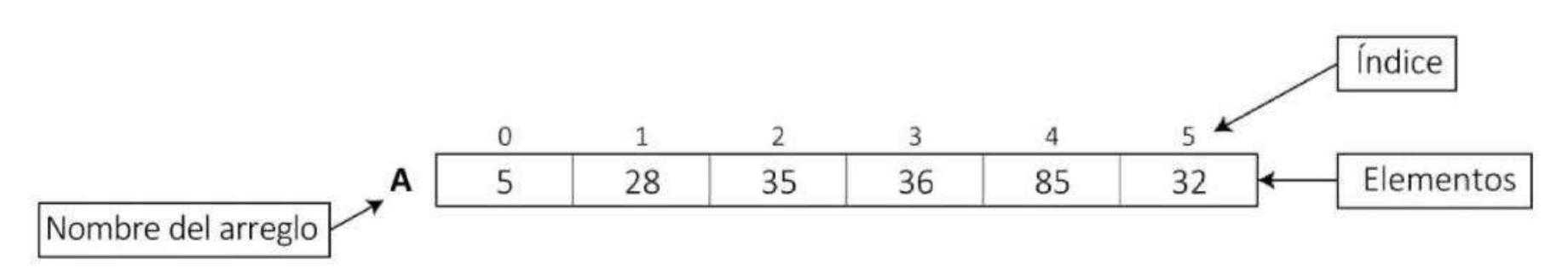
- Multidimensionales, cuando tienen de tres a más dimensiones.



9.3 Arreglo unidimensional

Un *arreglo unidimensional* es un conjunto de elementos almacenados consecutivamente, uno a continuación del otro, donde cada elemento conserva su propio espacio con el mismo nombre para todos los elementos. Se le denomina unidimensional porque tiene un solo índice.

Forma gráfica:



Donde:

- El nombre del arreglo es asignado dependiendo de los elementos que contiene el arreglo. Por ejemplo, un arreglo de números (*número*) o un arreglo de edades (*edad*).
- El índice de un arreglo tiene como punto de inicio la posición cero, y como última posición la cantidad total de elementos menos uno (N-1). Para recorrer por cada uno de los elementos de un arreglo se necesita una estructura repetitiva como **for** o **while**.
- Cada elemento es asignado a una posición independiente del arreglo.

Veamos:

```
A[0] - 5
A[1] - 28
A[2] - 35
A[3] - 36
A[4] - 85
A[5] - 32
```

- **Formato de declaración de un arreglo unidimensional**

```
tipodeDatos nombreArreglo[];
tipoDato[] nombreArreglo;
```

Donde:

- **TipodeDatos.** Es el tipo de datos especificado para los valores que se almacenarán en el arreglo unidimensional. Hay que tener en cuenta que todos los elementos serán del mismo tipo.
- **nombreArreglo.** Es el nombre asignado al arreglo.

Obsérvese los siguientes casos de declaración de un arreglo unidimensional:

Declarar el arreglo nombres del empleado, número de hijos y pago mensual para el registro de N empleados en una empresa.

Forma 1: **tipoDato** nombreArreglo[];

```
String nombres[];
int nHijos[];
double pagoMensual[];
```

Forma 2: **tipoDato**[] nombreArreglo;

```
String[] nombres;
int[] nHijos;
double[] pagoMensual;
```

- **Formato de creación del objeto de tipo arreglo**

```
nombreArreglo - new tipoDatos[tamaño]
```

Donde:

- **NombreArreglo.** Se especifica el nombre del arreglo ya declarado.
- **tipoDatos.** Es el mismo tipo de datos especificado en la declaración.
- **tamaño.** Se especifica el total de elementos que puede contener el arreglo; hay que considerar que aquí no se especifica el máximo tamaño, sino el total de elementos.

Obsérvese el siguiente caso:

Crear los objetos de tipo arreglo para los nombres del empleado, número de hijos y pago mensual de 100 empleados en una empresa.

```
nombres = new String[100];
nHijos = new int[100];
pagoMensual = new double[100];
```

- **Formato para la declaración y creación del objeto de tipo arreglo en una línea**

```
tipoDatos nombreArreglo[] = new tipoDatos[tamaño]
```

Donde:

- **NombreArreglo.** Se especifica el nombre del arreglo a declarar y crear.
- **tipoDatos.** Es el tipo de datos especificado en la declaración y creación del objeto.
- **tamaño.** Se especifica el total de elementos que puede contener el arreglo.

Obsérvese el siguiente caso:

Declarar y crear los objetos de tipo arreglo para los nombres del empleado, número de hijos y pago mensual de 100 empleados en una empresa.

```
String nombres[] = new String[100];
int nHijos[] = new int[100];
double pagoMensual[] = new double[100];
```

9.4 Arreglo bidimensional

Un *arreglo bidimensional* es considerado un vector de vectores o un conjunto de elementos, todos del mismo tipo, en el cual el orden de los componentes es significativo; por lo tanto, necesitan dos índices, allí radica su principal característica.

Forma gráfica:

(0,0) 100	(0,1) 50	(0,2) 75
(1,0) 42	(1,1) 30	(1,2) 80
(2,0) 82	(2,1) 51	(2,2) 12

Los elementos se registran en una posición de doble índice, donde el primer índice es la posición de la fila, mientras el segundo indica la posición de la columna; si esto es correcto, para poder registrar

sobre ella se necesitará una estructura de repetición doble. Se debe considerar que internamente el arreglo bidimensional trabaja de la siguiente forma:

0,0	100
0,1	50
0,2	75
1,0	42
1,1	30
1,2	80
2,0	82
2,1	51
2,2	12

Es decir, se visualiza como un arreglo unidimensional de valores, pero aún se mantiene el doble índice, factor indicativo del arreglo bidimensional.

Formato de declaración de un arreglo bidimensional

```
tipoDato nombreArreglo[][];
tipoDato[][] nombreArreglo;
```

Donde:

- **TipodeDatos.** Es el tipo de datos especificado para los valores que se almacenarán en el arreglo bidimensional. Hay que tener en cuenta que todos los elementos serán del mismo tipo.
- **nombreArreglo.** Es el nombre asignado al arreglo.

  Hay que tener en cuenta que, a diferencia del arreglo unidimensional que solo asignaba un corchete, en el bidimensional se asignan dos bloques de corchetes: el primero para las filas y el segundo para las columnas.

  Obsérvese los siguientes casos de declaración de un arreglo bidimensional:

  Declarar el arreglo bidimensional, números de tipo entero y pagos de tipo double.

  Forma 1: tipoDato nombreArreglo[][];

```
int números[][];
double pagos[][];
```

  Forma 2: tipoDato[][] nombreArreglo;

```
int[][] números;
double[][] pagos;
```

Formato de declaración de un arreglo bidimensional

```
nombreArreglo = new tipoDato[tamañoF][tamañoC]
```

Donde:

- **TipodeDatos.** Es el tipo de datos especificado en la declaración del arreglo.
- **nombreArreglo.** Es el nombre asignado al arreglo.
- **tamañoF.** Determina el máximo tamaño en filas declarado por el usuario.
- **tamañoC.** Determina el máximo tamaño en columnas declarado por el usuario.

Obsérvese el siguiente caso:

Crear el arreglo bidimensional, números de tipo entero y pagos de tipo double ya declarados.

```
números = new int[10][10];
pagos = new double[10][10];
```

Operaciones sobre un arreglo bidimensional

- **Ingreso de datos.** Para este caso se implementará un script que permita llenar un arreglo bidimensional de 3x3 de números enteros.

```
for(int i=0;i<3;i++)
     for(int j=0;j<3;j++)
        N[i][j]=Integer.parseInt(JOptionPane. showInputDialog(this,
                                    'Ingrese un número'))
```

- **Ingreso de datos aleatorios a una matriz.** Para este caso se implementará un script que permita llenar un arreglo bidimensional de 3x3 con números enteros de dos cifras en forma aleatoria de números enteros.

```
for(int i=0;i<3;i++)
    for(int j=0;j<3;j++)
        N[i][j]=(int)((99-10+1)*Math.random()+10);
```

- **Recorrer por los datos.** Para este caso se enviará la matriz N[][] al modelo **moNumeros**.

```
for(int i=0;i<3;i++)
    for(int j=0;j<3;j++)
        moNumeros.addElement(N[i][j]);
```

- **Operar sobre una matriz.** Para este caso se acumularán los valores contenidos en la matriz N.

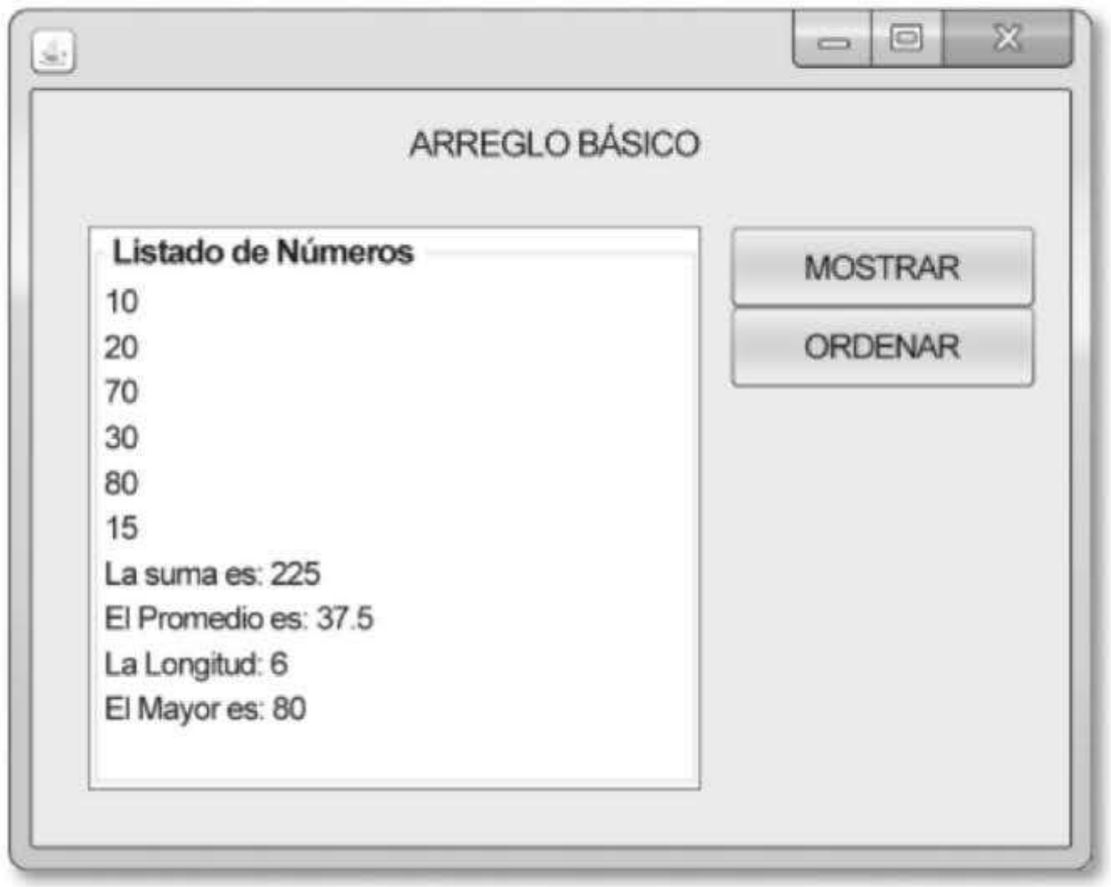
```
int s=0;
    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
            s+=N[i][j]);
```

CASO DESARROLLADO 1: **Listado de números básico**

Implemente una aplicación que permita registrar un arreglo unidimensional básico de 6 elementos numéricos enteros y que al final determine la suma, el promedio, la longitud y el mayor elemento registrado en el arreglo. Adicionalmente, implemente un método que permita ordenar los elementos del arreglo en forma ascendente.

Consideraciones:

- Use el siguiente diseño de interfaz gráfica:



- A través del botón **Mostrar** debe listar los números solicitados por la aplicación.
- Muestre los valores en un control **JList**.

Solución:

1. Seleccione la ruta **File > New > Java Project**.
2. Asigne el siguiente nombre al proyecto: **pjBasico**.
3. Agregue los paquetes **pFormularios**.
4. Agregue un **JFrame** dentro del paquete **pFormularios** y asígnele el nombre **frmBasico**.

5. Diseñe la siguiente interfaz y asigne nombres a los controles como se muestra en la imagen:

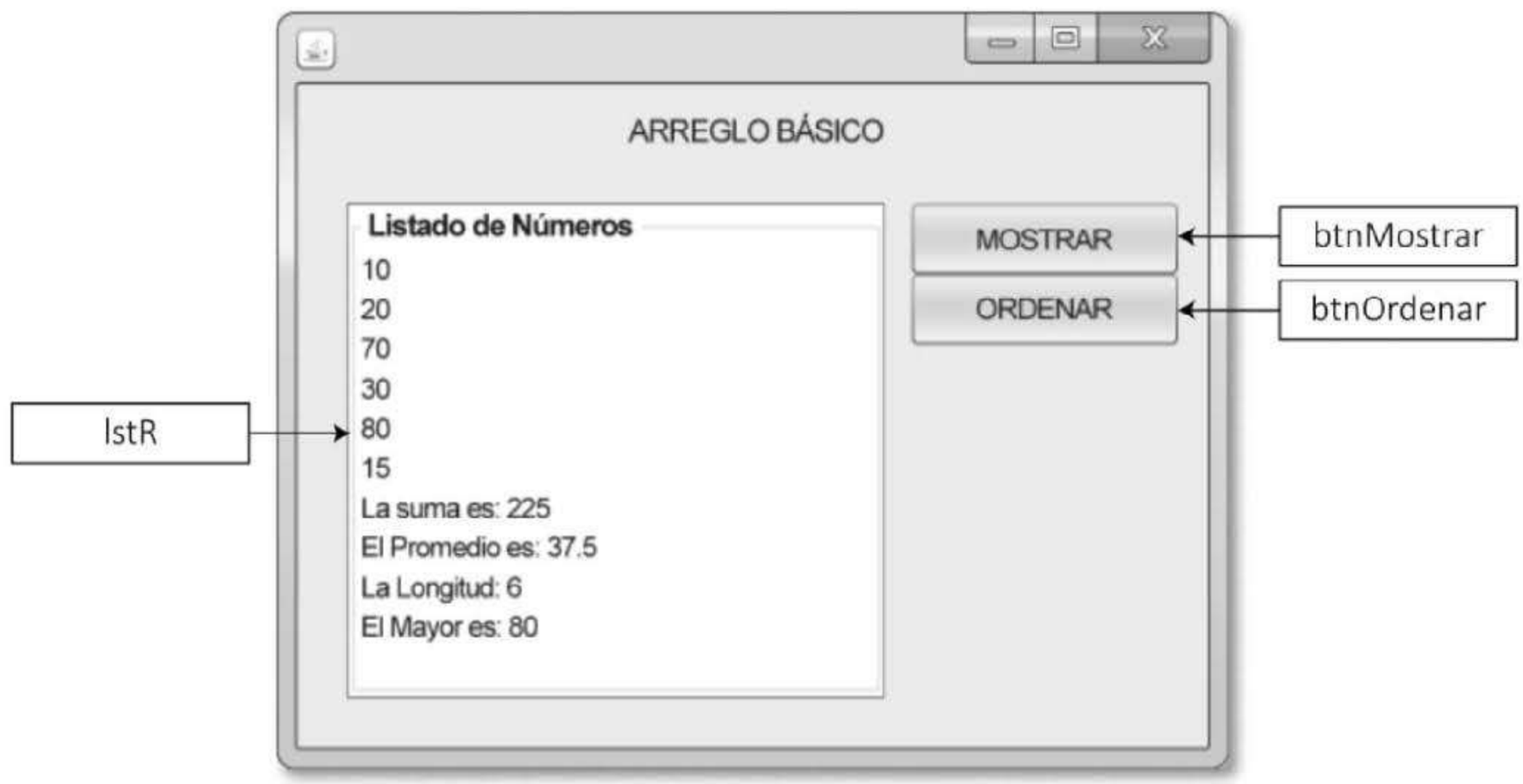


Figura 9.1 Controles del formulario de listado de arreglo.

Resumen de controles:

N.°	Clase	Control
1	JLabel	ARREGLO BÁSICO
2	JList	lstR
3	JButton	btnMostrar
4	JButton	btnOrdenar

6. El código del formulario **frmBasico** se muestra como sigue:

```
package pFormularios;
import javax.swing.DefaultListModel;
public class frmBasico extends javax.swing.JFrame {
    //Declaración del arreglo unidimensional
    int a[]-new int[6];
    public frmBasico() {
        initComponents();
    }
    void ordenaAscendente(){
        int temp;
        for(int i-0;i<a.length;i++)
            for(int j-0;j<a.length;j++)
                if (a[i]<a[j]){
                    temp - a[i];
                    a[i]-a[j];
                    a[j]-temp;
                }
        DefaultListModel moNumeros-new DefaultListModel();
        for(int i-0;i<-5;i++){
            moNumeros.addElement(a[i]);
        }
        lstR.setModel(moNumeros);
    }
```

```
    int determinaMayor(){
        int mayor=Integer.MIN_VALUE;
        for(int i=0;i<a.length;i++){
            if(a[i]>mayor)
                mayor=a[i];
        }
        return mayor;
    }
    double calculaPromedio(int suma){
        return (suma*1.0/a.length);
    }
    int calculaSuma(){
        int s=0;
        for(int i=0;i<=5;i++){
            s+=a[i];
        }
        return s;
    }
    private void btnMostrarActionPerformed(java.awt.event.ActionEvent evt) {
        //1
        a[0]=10;
        a[1]=20;
        a[2]=70;
        a[3]=30;
        a[4]=80;
        a[5]=15;
        //2
        int suma=calculaSuma();
        double promedio=calculaPromedio(suma);
        int mayor=determinaMayor();
        //3
        DefaultListModel moNumeros=new DefaultListModel();
        for(int i=0;i<=5;i++){
            moNumeros.addElement(a[i]);
        }
        moNumeros.addElement('La suma es: '+suma);
        moNumeros.addElement('El Promedio es: '+promedio);
        moNumeros.addElement('La Longitud: '+a.length);
        moNumeros.addElement('EL Mayor es: '+mayor);
        lstR.setModel(moNumeros);
    }
}
```

7. Ejecute la aplicación. Una vez colocado el código en los botones, presione la combinación de teclas **<Shift>** + **<F6>**.

CASO DESARROLLADO 2: **Listado de números usando clase**

Implemente una aplicación que permita registrar un arreglo unidimensional básico de 6 elementos numéricos enteros y que al final determine la suma, el promedio, la longitud y el mayor elemento registrado en el arreglo. Adicionalmente, implemente un método que permita ordenar los elementos del arreglo en forma ascendente.

Debe implementar la clase **ArregloUnidimensional** que permita controlar todos los métodos de la aplicación.

Consideraciones:

- Cree un arreglo para los números dentro de la clase **arregloNumeros**.
- Implemente los siguientes métodos dentro de la clase **arregloNumeros**:
  - El método **constructor**, que se encargará de crear el arreglo numérico de 6 números. Inicialice los valores del arreglo con los siguientes números: 10, 20, 70, 30, 80 y 15.
  - El método **tamaño**, que devolverá el número de elementos registrados en el arreglo.
  - El método **ordenaAscendente**, que se encargará de ordenar los números registrados en el arreglo en forma ascendente.
  - El método **determinaMayor**, que se encargará de obtener el mayor elemento del arreglo.
  - El método **calculaPromedio**, que se encargará de promediar los números del arreglo.
  - El método **calculaSuma**, que se encargará de sumar todos los elementos del arreglo.
  - El método **devuelveValor**, que se encargará de devolver el elemento contenido en el arreglo de uno en uno.
- Use el siguiente diseño de interfaz:

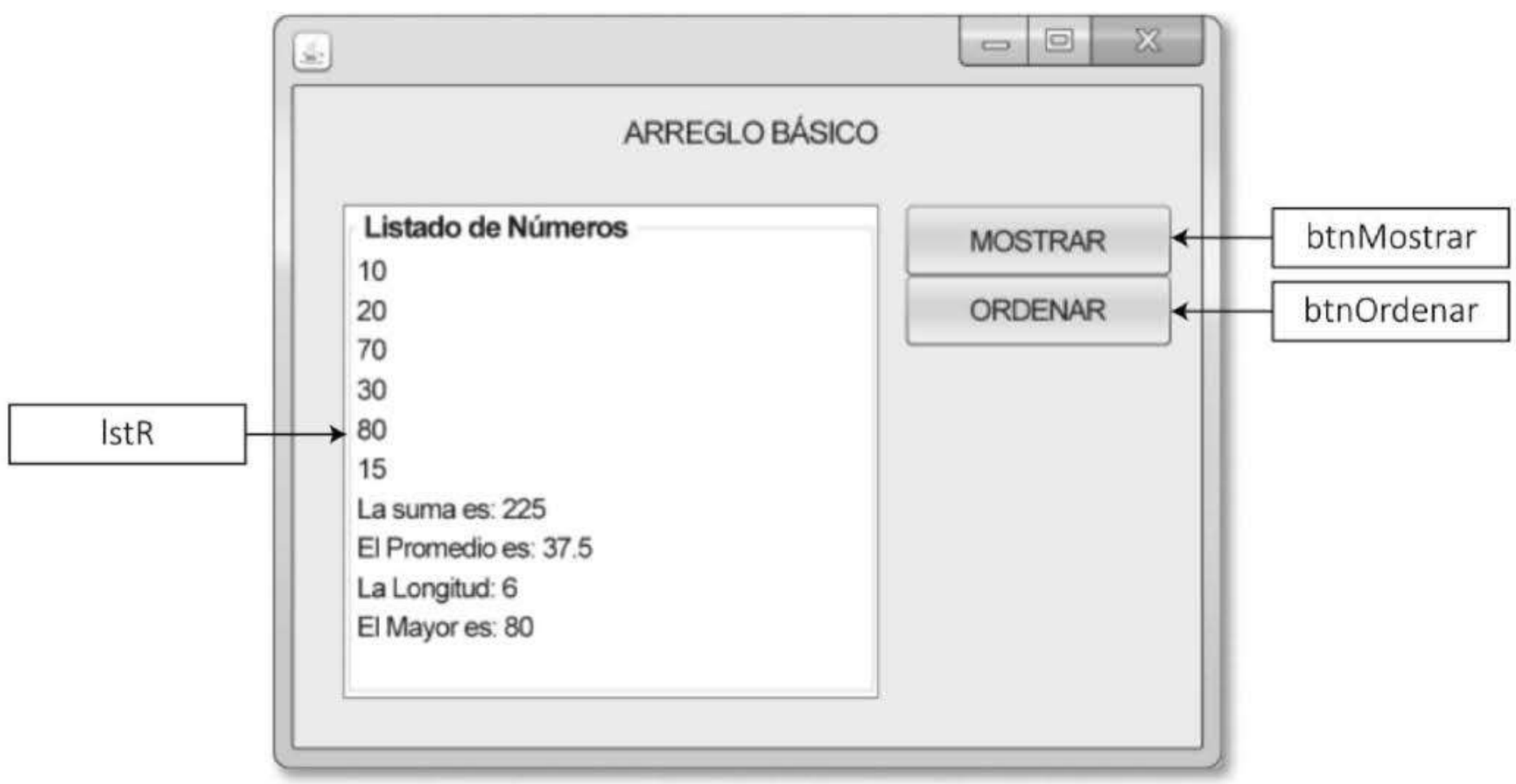
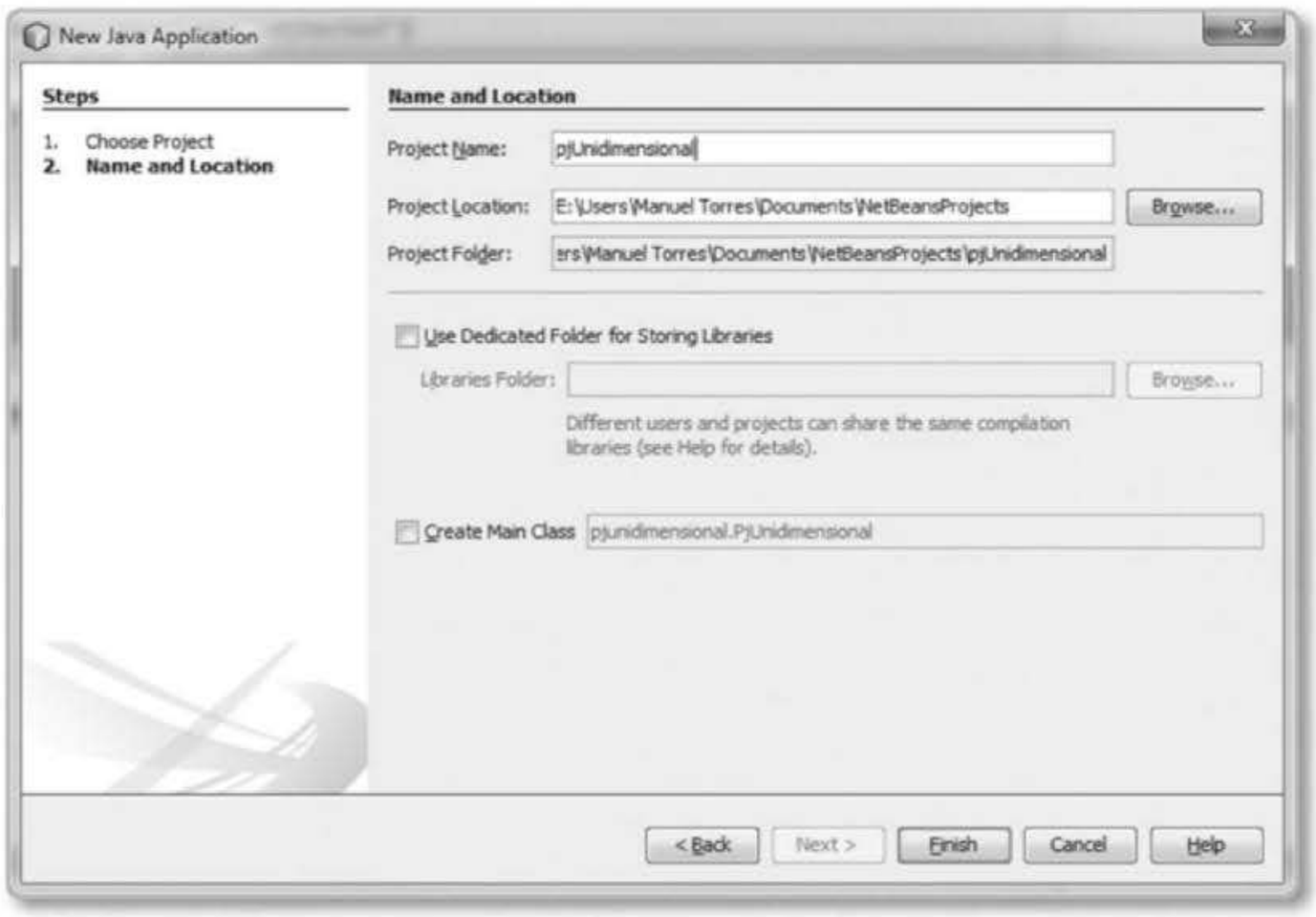


Figura 9.2 Formulario de arreglo básico.

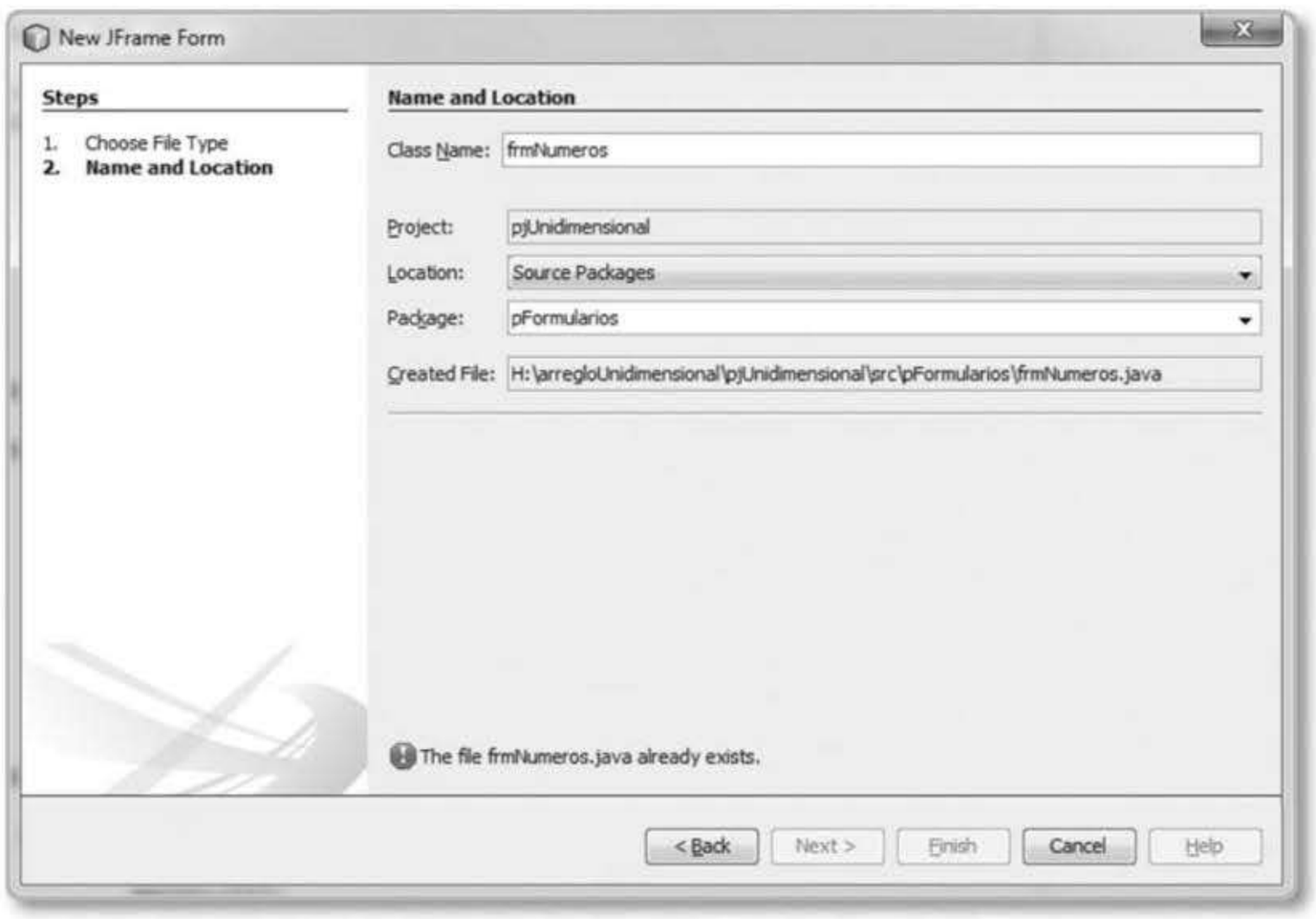
Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjUnidimensional**.



2. Agregue el paquete **pFormularios** al proyecto **pjUnidimensional**.

3. Agregue la clase **frmNumeros** al paquete **pFormularios**; para ello, haga clic derecho sobre el paquete **pFormularios**. Luego siga la ruta **New > JFrame Form**.



4. Asigne un nombre a cada uno de los controles como se muestra en la figura 9.2; para ello, haga clic derecho sobre el objeto y seleccione **Change Variable Name...**
5. Después de asignar nombres a los objetos, asegúrese de que los controles sean los correctos; para ello, visualice el panel **Navigator**, el cual debe mostrarse como la figura 9.3.

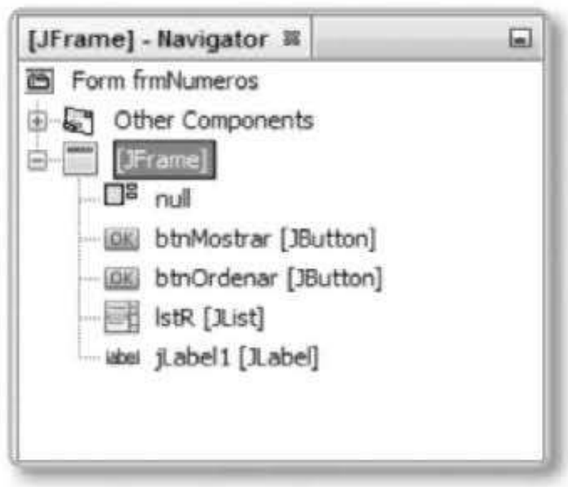


Figura 9.3 Listado de controles desde la ventana Navigator.

6. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.
7. Al **Frame** active el **Form Size Police** con el valor **Generate Resize Code**.
8. Asigne las siguientes propiedades a los controles:

btnMostrar	**Text**	MOSTRAR
btnOrdenar	**Text**	ORDENAR
lstR	**Border**	TitledBorder=Listado de Números
	Model	Dejar vacío

El siguiente script muestra el contenido de la clase **arregloNumeros**:

```
package pClases;

public class arregloNumeros {
    //Atributos
    private int a[];

    //método constructor
    public arregloNumeros(){
        a=new int[6];
        a[0]=10;
        a[1]=20;
        a[2]=70;
        a[3]=30;
        a[4]=80;
        a[5]=15;
    }

    public int tamaño(){
        return a.length;
    }
```

```
    public void ordenaAscendente(){
        int temp;
        for(int i=0;i<tamaño();i++)
            for(int j=0;j<tamaño();j++)
                if (a[i]<a[j]){
                    temp = a[i];
                    a[i]=a[j];
                    a[j]=temp;
                }
    }

    public int determinaMayor(){
        int mayor=Integer.MIN_VALUE;
        for(int i=0;i<tamaño();i++){
            if(a[i]>mayor)
                mayor=a[i];
        }
        return mayor;
    }

    public double calculaPromedio(int suma){
        return (suma*1.0/tamaño());
    }

    public int calculaSuma(){
        int s=0;
        for(int i=0;i<tamaño();i++){
            s+=a[i];
        }
        return s;
    }

    //Devuelve el elemento del arreglo segun la posición
    public int devuelveValor(int pos){
        return a[pos];
    }
}
```

Se declara como atributo privado a[] para el arreglo de tipo entero, se implementa el método **constructor** en el cual se asigna, como capacidad máxima del arreglo, 6 elementos numéricos y luego los valores. Hay que tener en cuenta que el trabajo del método **constructor** es inicializar los valores; por tanto, al iniciar la aplicación e instanciar a la clase **arregloNumeros**, el arreglo a[] ya debe tener asignado los valores numéricos.

El método **tamaño** determina el total de elementos que tiene asignado hasta el momento el arreglo; es decir, devolverá el valor 6, puesto que dentro del método **constructor** se asignó 6 valores a los espacios establecidos en el arreglo.

El método **ordenaAscendente** permite ordenar los elementos del arreglo en forma ascendente, aplicando el método de la burbuja.

El método **determinaMayor** permite devolver el mayor elemento registrado en el arreglo.

El método **calculaPromedio** permite calcular el promedio de los valores; el arreglo para esto necesita como parámetro el valor de la suma.

El método **calculaSuma** determina la suma de los valores dentro del arreglo.

Finalmente, el método **devuelveValor** permite devolver elemento por elemento dependiendo de la posición de este, para lo cual se necesita como parámetro la posición del elemento que se

desea obtener; considere que, si este método es invocado dentro de una estructura repetitiva, se puede obtener todos los elementos al mismo tiempo, ya que el solo invocar al método devolver un solo valor.

El siguiente script muestra las librerías y declaración de las variables globales de la aplicación:

```
package pFormularios;
import javax.swing.DefaultListModel;
import pClases.arregloNumeros;

public class frmNumeros extends javax.swing.JFrame {
    arregloNumeros aNum = new arregloNumeros();

    public frmNumeros() {
        initComponents();
    }
}
```

La variable de clase **aNum** permite tener acceso a todos los métodos y atributos públicos de la clase **arregloNumeros**.

El siguiente script muestra las instrucciones del botón **btnMostrar**:

```
private void btnMostrarActionPerformed(java.awt.event.ActionEvent evt) {
    int suma=aNum.calculaSuma();
    double promedio=aNum.calculaPromedio(suma);
    int mayor=aNum.determinaMayor();

    //enviarlo a la lista
    DefaultListModel moNumeros=new DefaultListModel();
    for(int i=0;i<=5;i++){
        moNumeros.addElement(aNum.devuelveValor(i));
    }
    moNumeros.addElement('La suma es: '+suma);
    moNumeros.addElement('El Promedio es: '+promedio);
    moNumeros.addElement('La Longitud: '+aNum.tamaño());
    moNumeros.addElement('EL Mayor es: '+mayor);
    lstR.setModel(moNumeros);
}
```

Empezamos con el botón de mostrar para obtener los valores de la suma, promedio y el mayor elemento por medio del objeto **aNum**, que es la instancia de la clase **arregloNumeros**.

Luego se envían los valores del arreglo al control **lstR**; para esto, debemos recorrer por todos los elementos del arreglo, por eso implementamos la estructura repetitiva **for**, cuyo valor inicial es cero y el valor tope de repeticiones es menor o igual a 5; pero también se pudo implementar de la siguiente forma:

```
for (int i=0;i<aNum.tamaño();i++)
```

Recordemos que dentro de la clase **arregloNumeros** se implementó el método **devuelveValor**, que tenía la misión de devolver uno a uno los elementos del arreglo, dependiendo del valor del índice; en este caso, la variable **i** mediante la estructura **for** tiene todas las posiciones del arreglo, por tanto, **i** se vuelve el parámetro para la obtención de los valores.

El siguiente script muestra las instrucciones del botón **btnOrdenar**:

```
private void btnOrdenarActionPerformed(java.awt.event.ActionEvent evt) {
    aNum.ordenaAscendente();

    //enviarlo a la lista
    DefaultListModel moNumeros=new DefaultListModel();
    for(int i=0;i<=5;i++){
        moNumeros.addElement(aNum.devuelveValor(i));
    }
    lstR.setModel(moNumeros);
}
```

CASO DESARROLLADO 3: **Certamen de belleza**

Implemente una aplicación que permita registrar un número de candidatas de un certamen de belleza, el cual necesita determinar la ganadora del concurso y por cuántos puntos lo logró. Hay que tener en cuenta que son 3 los criterios del jurado que al final se reflejan en los puntajes, este puntaje tiene un rango de 0 a 100 puntos.

Consideraciones:

- Cree dos arreglos unidimensionales: uno para el nombre de la candidata y otro para el promedio de los puntajes dentro de una clase llamada arreglo.
- Implemente los siguientes métodos:
  - El método **constructor**, que se encargará de crear los arreglos e inicializar la posición del primer elemento de los arreglos en cero.
  - El método **registraCandidata**, que permitirá registrar los datos y el promedio de una determinada candidata.
  - El método **devuelveNombre**, que permitirá devolver el nombre de la candidata.
  - El método **devuelvePromedio**, que permitirá devolver el promedio obtenido por una determinada candidata.
  - El método **totalCandidatas**, que se encargará de devolver el total de candidatas registradas en la aplicación.
  - El método **puntajeAlto**, que se encargará de devolver el puntaje más alto obtenido por una candidata.
  - El método **ganadora**, que se encargará de devolver quién es la ganadora del certamen.

- Use el siguiente diseño de interfaz:

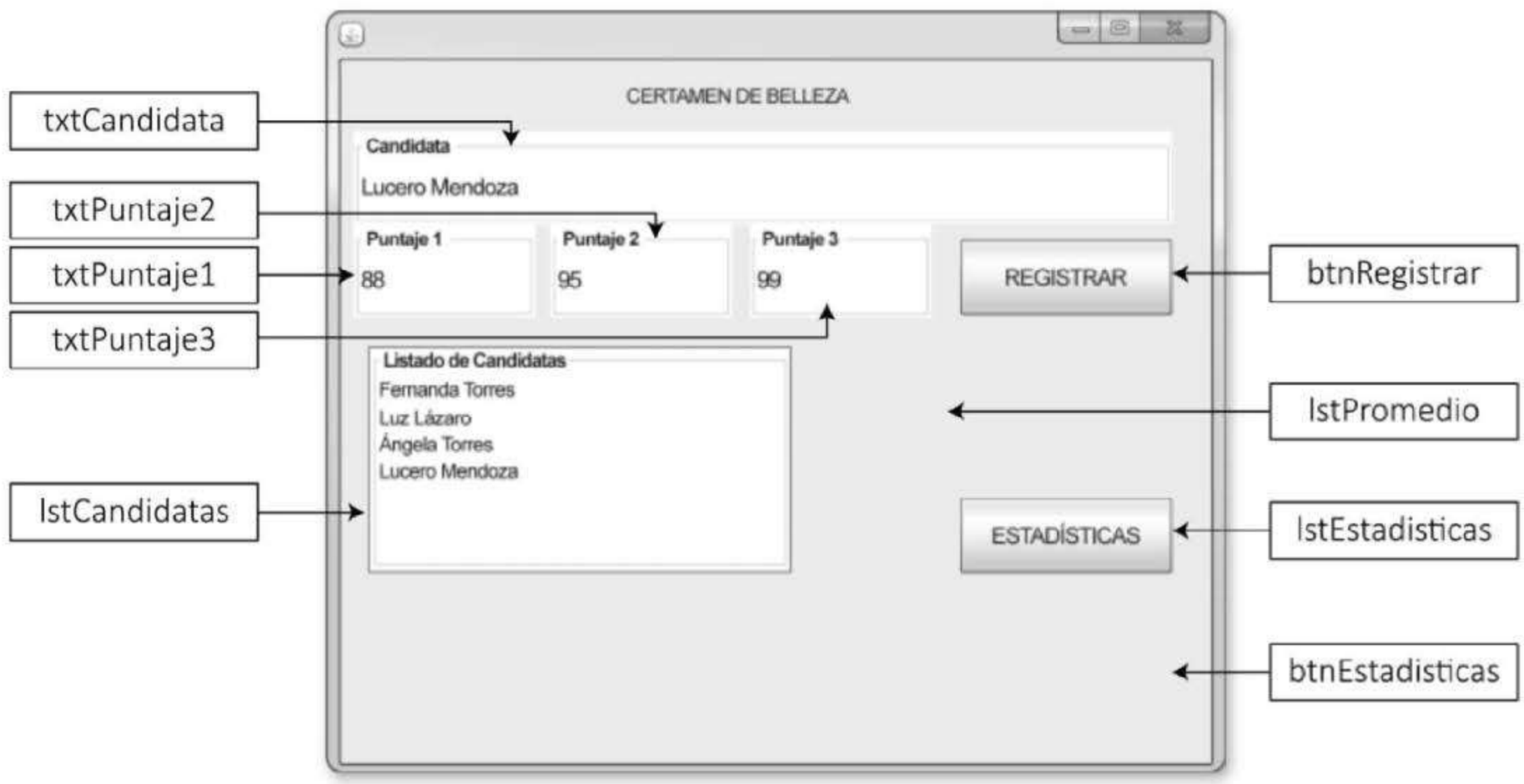


Figura 9.4 Formulario

Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjCertamen**.
2. Agregue el paquete **pFormularios** al proyecto **pjCertamen**.
3. Agregue la clase **frmCertamen** al paquete **pFormularios**.
4. Asigne un nombre a cada uno de los controles como se muestra en la figura 9.4; para ello, haga clic derecho sobre el objeto y seleccionar **Change Variable Name...**
5. Después de asignar nombres a los objetos, asegúrese de que los controles sean los correctos; con ese fin, visualice el panel **Navigator**, el cual debe mostrarse como la figura 9.5.

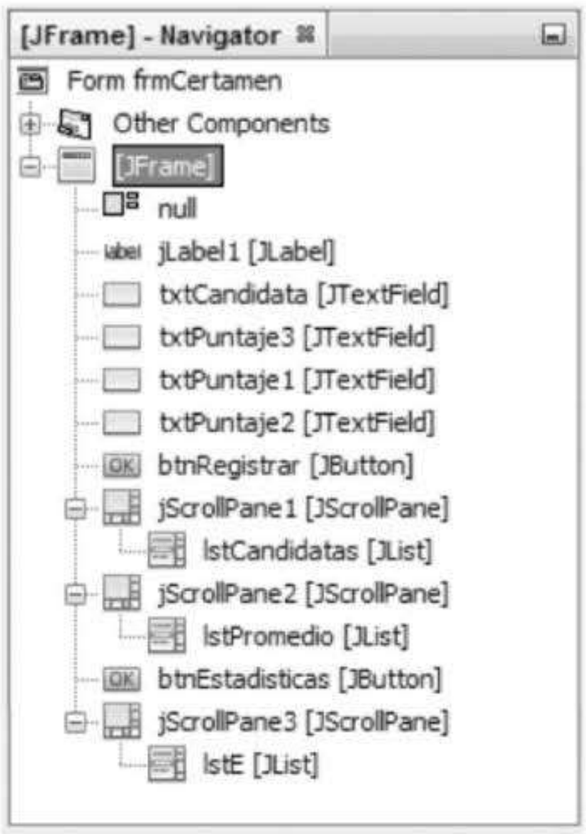


Figura 9.5 Listado de controles desde la ventana Navigator.

6. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.
7. Al **Frame** active el **Form Size Police** con el valor **Generate Resize Code**.

El siguiente script muestra el contenido de la clase **arreglo**:

```
package pClases;
public class arreglo {
    //Atributos de clase
    private String nombre[];
    private double promedio[];
    private int pos=0;
    private int posicion;

    //constructor
    public arreglo(){
        nombre=new String[100];
        promedio=new double[100];
        pos=0;
    }

    public void registraCandidata(String nombre,double promedio){
        this.nombre[pos]=nombre;
        this.promedio[pos]=promedio;
        pos++;
    }

    public String devuelveNombre(int pos){
        return nombre[pos];
    }

    public double devuelvePromedio(int pos){
        return promedio[pos];
    }

    public int totalCandidatas(){
        return pos;
    }

    public double puntajeAlto(){
        double mayor=Double.MIN_VALUE;
        for(int i=0;i<totalCandidatas();i++){
            if (promedio[i]>mayor){
                mayor=promedio[i];
                posicion=i;
            }
        }
        return mayor;
    }

    public String ganadora(){
        return nombre[posicion];
    }
}
```

Se declaran los atributos de la clase nombre y promedio de tipo Arreglo. **Pos** es el atributo encargado de posicionarse dentro de los índices del arreglo; y **posición** tendrá la misión de obtener la posición de la candidata ganadora, es decir, al momento de encontrar a la ganadora, ubica en qué posición del arreglo se encuentra este valor, el cual se guardará en la variable **posición**, ya que otro método solicitará dicho valor para determinar el nombre de la ganadora.

El método **constructor** crea los objetos de tipo Arreglo con una capacidad de 100 elementos, es decir, tendremos como máximo 100 candidatas que se podrían registrar en la aplicación. Además, se inicializa la variable **pos** en cero, ya que es el punto de inicio de un arreglo.

El método **registraCandidata** permite enviar los nombres y el promedio de una determinada candidata en la posición actual de los arreglos. Al final, se aumenta en uno la variable **pos**, ya que así se posicionará en un valor más el índice del arreglo para la siguiente candidata.

Los métodos **devuelveNombre** y **devuelvePromedio** permiten devolver los valores registrados según la posición del índice.

El método **totalCandidatas** devuelve el total de participantes registradas hasta el momento.

El método **puntajeAlto** permite devolver cuál es el puntaje más alto y, a la vez, registrar en qué posición del arreglo se encontró dicho valor, el cual lo guardará dentro de la variable **posición**.

El método **ganadora** permite devolver el nombre de la candidata ganadora según la posición obtenida en el método **puntajeAlto**.

El siguiente script muestra las librerías y la declaración de las variables globales de la aplicación:

```
package pFormularios;

import java.text.DecimalFormat;
import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.arreglo;

public class frmCertamen extends javax.swing.JFrame {
    arreglo aCan - new arreglo();

    DefaultListModel moCandidatas - new DefaultListModel();
    DefaultListModel moPromedio - new DefaultListModel();
    DefaultListModel moEstadisticas - new DefaultListModel();

    DecimalFormat df;

    public frmCertamen() {
        initComponents();
        df - new DecimalFormat('#0.00');
    }
}
```

Usaremos la librería **DecimalFormat** para redondear los puntajes obtenidos. También se declara en forma global el objeto **aCan** de la clase arreglo. Luego se definen los modelos a usar en la aplicación y finalmente se declara **df** como objeto de la clase **DecimalFormat**.

En el método **constructor** de la clase **frmCertamen** se declara el objeto **df** como instancia de la clase **DecimalFormat** con un formato de solo 2 decimales.

El siguiente script muestra los métodos de la clase **frmCertamen**:

```
String getnombre(){
return txtCandidata.getText();
    }

double getPuntaje1(){
    return Double.parseDouble(txtPuntaje1.getText());
}
```

```
    double getPuntaje2(){
        return Double.parseDouble(txtPuntaje2.getText());
    }

    double getPuntaje3(){
        return Double.parseDouble(txtPuntaje3.getText());
    }

    double calculaPromedio(){
        return (getPuntaje1()+getPuntaje2()+getPuntaje3())/3.0;
    }

    void imprimir(){
        moCandidatas.clear();
        moPromedio.clear();
        moEstadisticas.clear();
        for(int i=0;i<aCan.totalCandidatas();i++){
            moCandidatas.addElement(aCan.devuelveNombre(i));
            moPromedio.addElement(df.format(aCan.devuelvePromedio(i)));
        }
        lstCandidatas.setModel(moCandidatas);
        lstPromedio.setModel(moPromedio);
    }

    void limpiar(){
        txtCandidata.setText('');
        txtPuntaje1.setText('');
        txtPuntaje2.setText('');
        txtPuntaje3.setText('');
        txtCandidata.requestFocus();
    }

    String valida(){
        if (txtCandidata.getText().equals('')){
            txtCandidata.requestFocus();
            return 'Nombre de la candidata';
        } else if(txtPuntaje1.getText().equals('') ||
                  Double.parseDouble(txtPuntaje1.getText())<0 ||
                  Double.parseDouble(txtPuntaje1.getText())>100){
            txtPuntaje1.setText('');
            txtPuntaje1.requestFocus();
            return 'Puntaje 1';
        }else if(txtPuntaje2.getText().equals('') ||
                  Double.parseDouble(txtPuntaje2.getText())<0 ||
                  Double.parseDouble(txtPuntaje2.getText())>100){
            txtPuntaje2.setText('');
            txtPuntaje2.requestFocus();
            return 'Puntaje 2';
        }else if(txtPuntaje3.getText().equals('') ||
                  Double.parseDouble(txtPuntaje3.getText())<0 ||
                  Double.parseDouble(txtPuntaje3.getText())>100){
            txtPuntaje3.setText('');
            txtPuntaje3.requestFocus();
            return 'Puntaje 3';
        } else
            return '';
    }
```

El siguiente script muestra las instrucciones del botón **btnRegistrar**:

```
private void btnRegistrarActionPerformed(…) {
    try{
        if (valida().equals('')){
            String candidata = getnombre();
            double puntaje1=getPuntaje1();
            double puntaje2=getPuntaje2();
            double puntaje3=getPuntaje3();
            aCan.registraCandidata(candidata, calculaPromedio());
            imprimir();
            limpiar();
        }else
            JOptionPane.showMessageDialog(null,'Error en '+valida());
    } catch(Exception ex){
        JOptionPane.showMessageDialog(null,'Error en la Aplicación');
    }
}
```

Dentro del botón **btnRegistrar** se implementa un **try-catch** para prevenir algunas excepciones que se pudieran ocasionar en la aplicación. Luego se valida la entrada de los valores a la aplicación; si todo es correcto, se capturan dichos valores y son enviados a la clase por medio del objeto **aCan**. En el caso de que el método **valida** devuelva una respuesta, esta será impresa.

El siguiente script muestra las instrucciones del botón **btnEstadisticas**:

```
private void btnEstadisticasActionPerformed(…) {
    int total=aCan.totalCandidatas();
    moEstadisticas.clear();
    moEstadisticas.addElement('El Total de Candidatas es: ' + total);
    moEstadisticas.addElement('La ganadora es: '+aCan.ganadora());
    moEstadisticas.addElement('El Máximo puntaje es: ' +
                              df.format(aCan.puntajeAlto()));
    lstE.setModel(moEstadisticas);
}
```

CASO DESARROLLADO 4: **Matriz de números enteros**

Implemente una aplicación que genere un arreglo bidimensional de números enteros de dos cifras, en forma aleatoria, de acuerdo a una cantidad de filas y columnas ingresadas por el usuario. Su tamaño como máximo será de 3×3.

Consideraciones:

- Declare como atributos privados la cantidad de filas (**f**), la cantidad de columnas (**c**) y un arreglo bidimensional **N**.
- Cree la clase **ArregloBiNumeros** con los siguientes métodos:
  - El método **constructor**, que defina la cantidad de filas y columnas de la matriz y, a la vez, genere los números aleatorios de la matriz.
  - El método que devuelva la cantidad de filas definidas para la matriz.
  - El método que devuelva la cantidad de columnas definidas para la matriz.

- El método **obtener**, que devuelva el elemento del arreglo matricial dependiendo de la posición de la fila y la columna.
- El método **generar**, que permita generar automáticamente valores numéricos de 2 cifras.
- El método **sumaFila**, que permita sumar una determinada fila ingresada por el usuario. Se utiliza **JOptionPane.showInputDialog()**.
- El método **sumaColumna**, que permita sumar una determinada columna ingresada por el usuario. Se utiliza **JOptionPane.showInputDialog()**.

■ Use el siguiente diseño de interfaz:

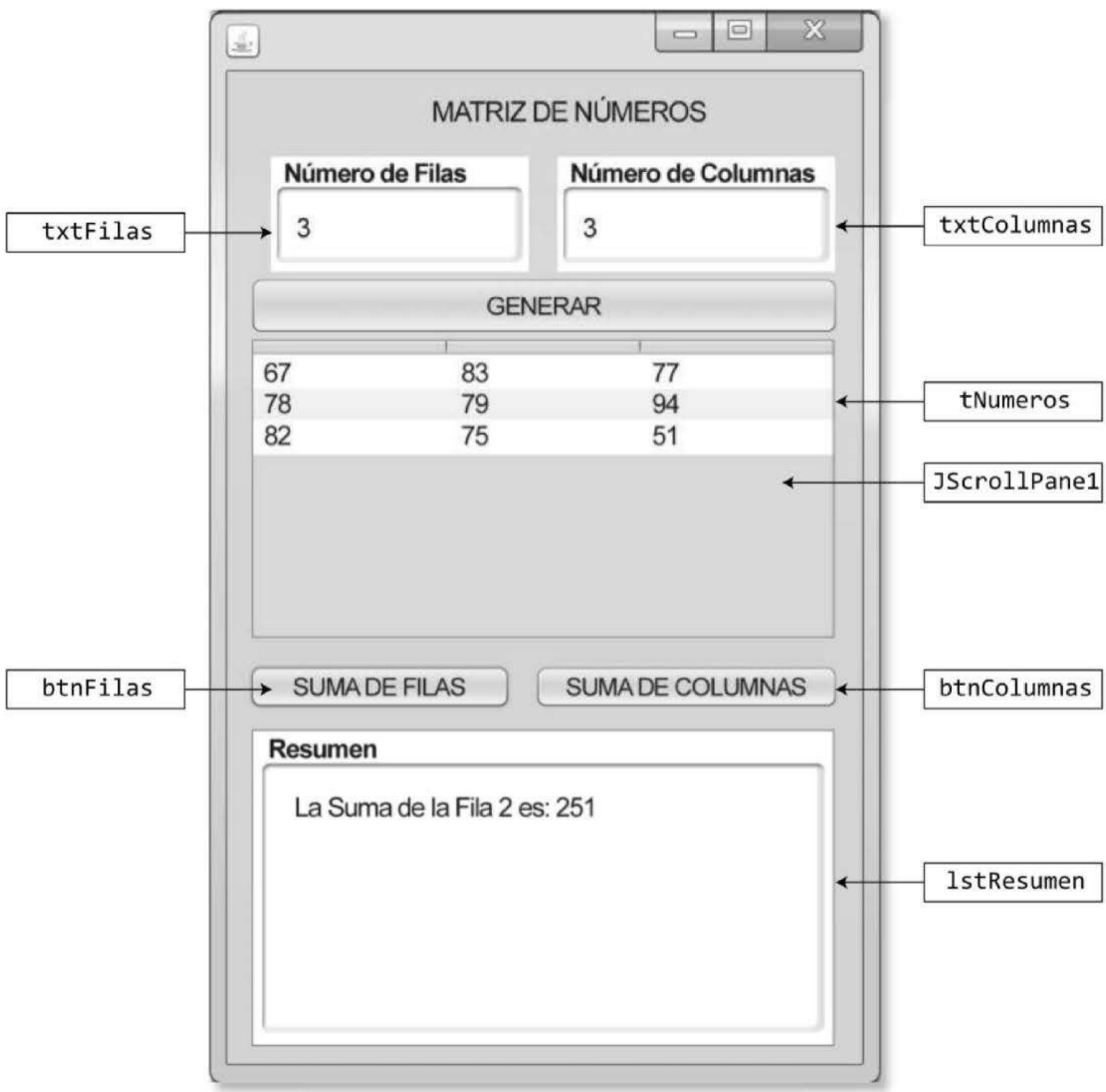
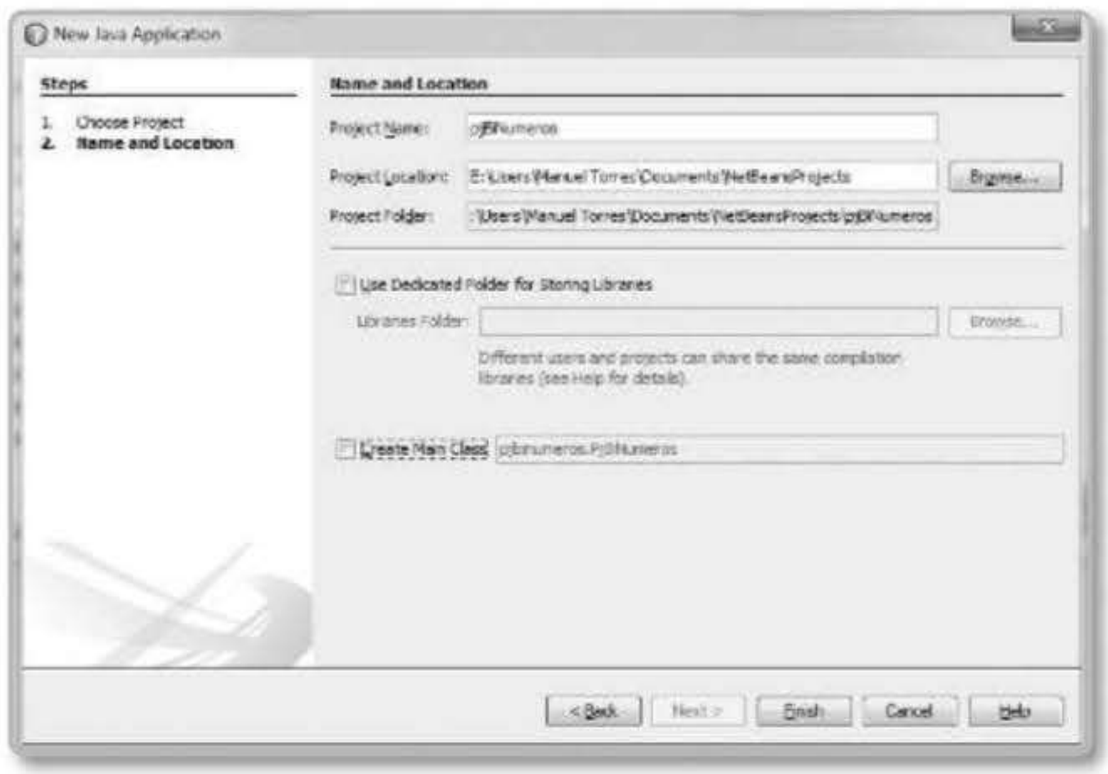


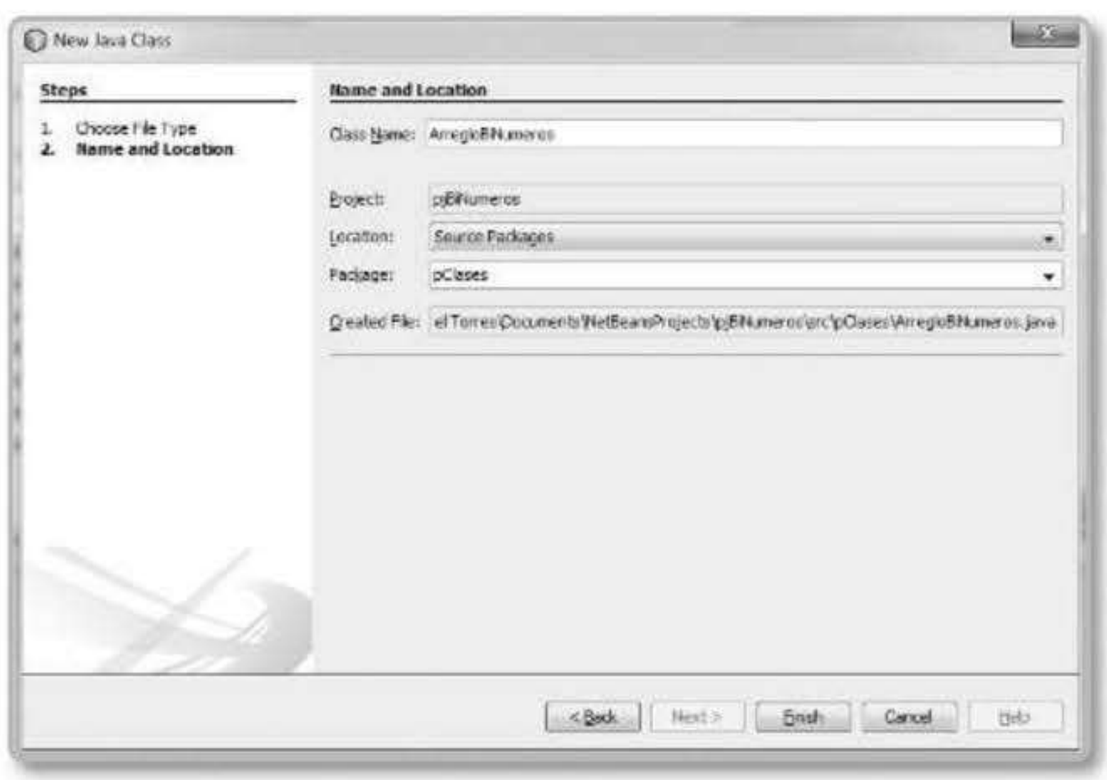
Figura 9.6 Formulario de matriz de números.

Solución:

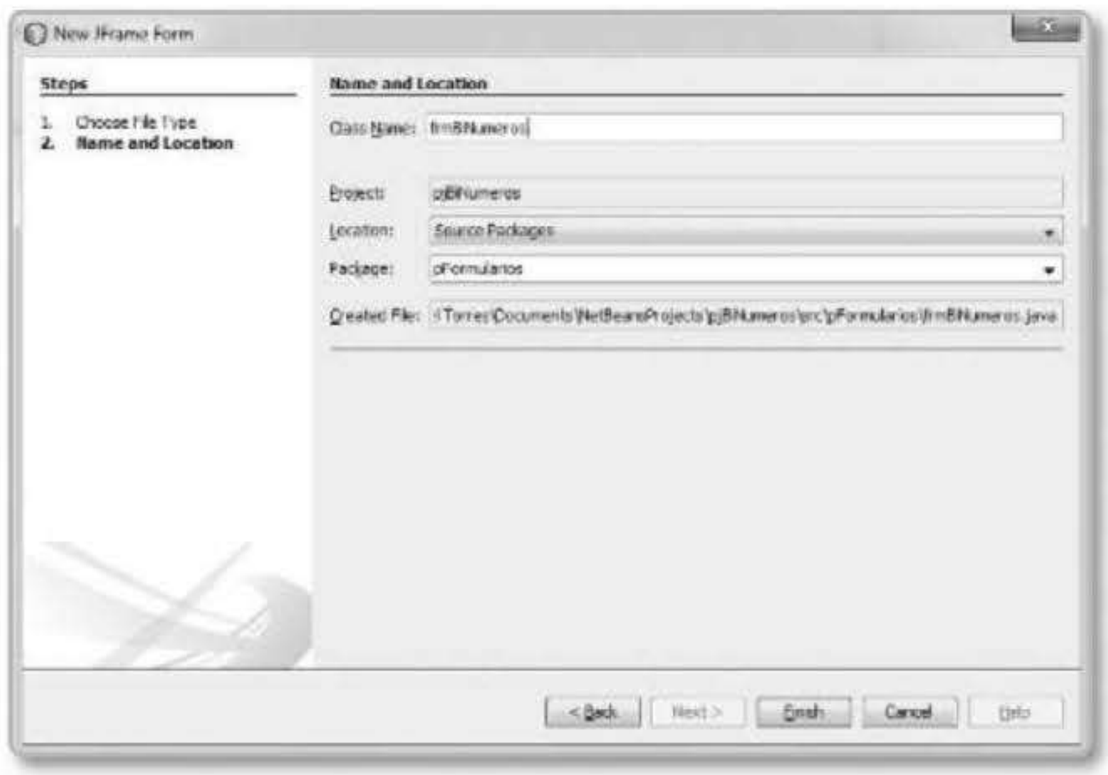
1. Cree un nuevo proyecto en NetBeans llamado **pjBiNumeros**.



2. Agregue el paquete **pFormularios** al proyecto **pjBiNumeros**.
3. Agregue el paquete **pClases** al proyecto **pjBiNumeros**.
4. Agregue la clase **ArregloBiNumeros** al paquete **pClases**.



5. Agregue la clase **frmBiNumeros** al paquete **pFormularios**.



6. Asigne un nombre a cada uno de los controles como se muestra en la figura 9.6; para ello, haga clic derecho sobre el objeto y seleccione **Change Variable Name...**

7. Después de asignar nombres a los objetos, asegúrese de que los controles sean los correctos; para ello, visualice el panel **Navigator**, el cual debe mostrarse como la figura 9.7.

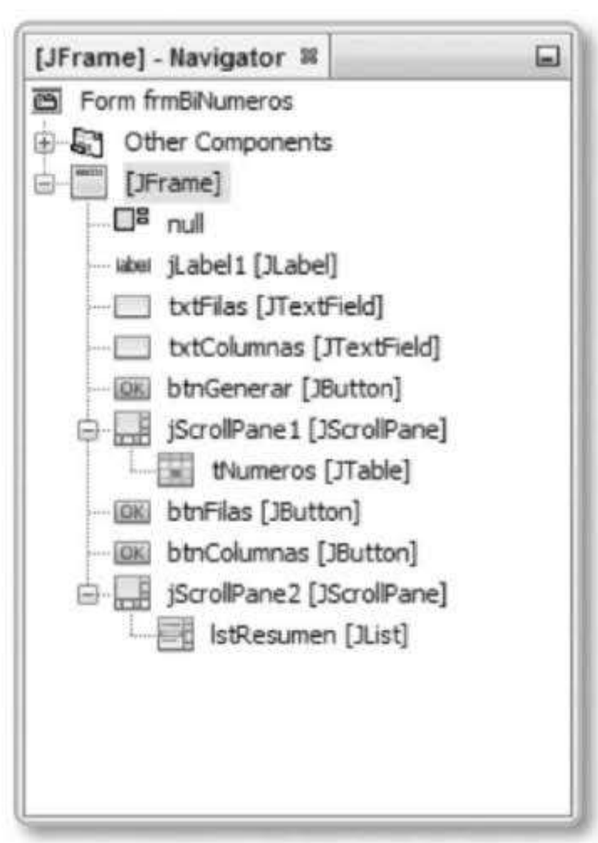


Figura 9.7 Listado de controles desde la ventana Navigator.

8. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.

9. Seleccione el formulario y en la propiedad **Form Size Police** modifique el valor encontrado por **Generate Resize Code**.

El siguiente script muestra el contenido de la clase **ArregloBiNumeros**:

```
package pClases;
public class ArregloBiNumeros {
    private int f,c;
    private int n[][];

    //1
    public ArregloBiNumeros(int f, int c){
        this.f=f;
        this.c=c;
        n=new int[f][c];
        generar();
    }

    //2
    public int getFilas(){
        return f;
    }
    public int getColumnas(){
        return c;
    }

    //3
    public int obtener(int posF, int posC){
        return n[posF][posC];
    }

    //4
    private void generar(){
        for(int i=0;i<f;i++)
            for(int j=0;j<c;j++)
                n[i][j]=(int)((99-10+1)*Math.random()+10);
    }

    //5
    public int sumaFila(int f){
        int s=0;
        int i=f-1;
        for(int j=0;j<c;j++)
            s+=n[i][j];
        return s;
    }

    //6
    public int sumaColumna(int c){
        int s=0;
        int j=c-1;
        for(int i=0;i<f;i++)
            s+=n[i][j];
        return s;
    }
}
```

En la clase **ArregloBiNumeros** se declaran las variables globales **f** para las filas, **c** para las columnas y **n**[][] para la matriz bidimensional.

En el punto uno se implementa el método **constructor** de la clase **ArregloBiNumeros**, que tiene como parámetros a **f** y **c** que serán enviados desde el formulario con valores para la construcción de la matriz. Dentro del método **constructor** se asignan dichos valores a los atributos privados **f** y **c**, respectivamente; luego se crea el objeto **n** con la capacidad en filas y columnas de **f** y **c**, que se reciben como parámetros del exterior. Finalmente, se invoca al método **generar** que permitirá llenar la matriz con elementos aleatorios.

En el punto dos se implementa el método **getFilas** y **getColumnas**, que permitirán devolver el total de filas y columnas implementadas en el arreglo bidimensional.

En el punto tres se implementa el método **obtener**, que permitirá devolver los valores contenidos en el arreglo bidimensional; esto dependerá de la posición de la fila y la columna, que serán obtenidas por el parámetro del método; en el formulario, este método tendrá que ser invocado dentro de una estructura repetitiva, donde **posF** representa a **i** de filas y **posC** representa a **j** de columnas.

En el punto cuatro se implementa el método **generar**, que permite asignar a la matriz valores numéricos en forma aleatoria, para ello se usa la función **random** de la clase **Math**.

En el punto cinco se implementa el método **sumaFila**, que permite sumar una de las filas del arreglo bidimensional; la fila considerada para la suma dependerá del usuario, y este valor sería recibido como parámetro (**f**) por el método **sumaFila**.

En el punto seis se implementa el método **sumaColumna**, que permite sumar los valores de una determinada columna del arreglo bidimensional; la columna será evaluada según el valor del parámetro (**c**).

El siguiente script muestra las librerías y declaración de las variables globales de la clase **frmBiNumeros**:

```
package pFormularios;

import javax.swing.DefaultListModel;
import javax.swing.JOptionPane;
import pClases.ArregloBiNumeros;

public class frmBiNumeros extends javax.swing.JFrame {

    DefaultListModel moRespuesta=new DefaultListModel();
    ArregloBiNumeros aN;

    public frmBiNumeros() {
        initComponents();
        lstResumen.setModel(moRespuesta);
    }
}
```

Dentro de la clase **frmBiNumeros** se declara como variables globales al objeto **moRespuesta**, que tendrá la misión de asimilar la respuesta de la suma de filas o columnas en el control

lstResumen. También se declara la variable **aN** de la clase **ArregloBiNumeros**. Finalmente, dentro del método **constructor** de la clase **frmBiNumeros**, se asigna al control **lstResumen** el modelo **moRespuesta**.

El siguiente script muestra los métodos de la clase **frmBiNumeros**:

```
public int getFilas(){
    return Integer.parseInt(txtFilas.getText());
}

public int getColumnas(){
    return Integer.parseInt(txtColumnas.getText());
}

void listar(){
    limpiar();
    int x;
    for(int i=0;i<getFilas();i++){
        for(int j=0;j<getColumnas();j++){
            x=aN.obtener(i,j);
            tNumeros.setValueAt(x, i, j);
        }
    }
}

void limpiar(){
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            tNumeros.setValueAt('', i, j);
        }
    }
}
void mensaje(String s){
    JOptionPane.showMessageDialog(null, s);
}
```

El método **getFilas** y **getColumnas** permitirá obtener el número de filas y columnas que el usuario desea implementar.

El método **listar** permite enviar los valores obtenidos desde la clase **ArregloBiNumeros** hacia el control **tNumeros**; recordemos que tenemos un método llamado **obtener** desde la clase **ArregloBiNumeros**, que permite devolver los valores del arreglo, cada elemento del arreglo bidimensional es asignado a una variable local **x**, y esta se asignará al control **tNumeros** mediante una estructura repetitiva.

El método **limpiar** permitirá limpiar los valores contenidos en el control **tNumeros**; ahora bien, como el máximo número de filas y columnas es 3×3, la estructura **for** también debe recorrer a 3×3 para poder limpiar el control.

El método **mensaje** permite mostrar un cuadro de mensaje desde cualquier parte de la aplicación.

El siguiente script muestra las instrucciones del botón **btnGenerar**:

```
private void btnGenerarActionPerformed(java.awt.event.ActionEvent evt) {
    try{
        moRespuesta.clear();

        //Capturando los valores ingresados en el Formulario
        int filas=getFilas();
        int columnas=getColumnas();

        //Verificando la cantidad de filas y columnas
        if (filas>0 && columnas>0 && filas<=3 && columnas<=3){
            aN=new ArregloBiNumeros(filas, columnas);
            listar();
        }else
            JOptionPane.showMessageDialog(null,'Error de Datos');
    }catch(Exception ex){
        JOptionPane.showMessageDialog(null, 'Error de Ingreso de Datos');
    }
}
```

El siguiente script muestra las instrucciones del botón **btnFilas**:

```
private void btnFilasActionPerformed(java.awt.event.ActionEvent evt) {
    int f=Integer.parseInt(JOptionPane.showInputDialog (this,'Ingrese un
                                        numero de Fila: '));

    if (f>getFilas())
        mensaje('El número de Filas es superior a la matriz');
    else
        moRespuesta.addElement('La Suma de la Fila ' + f + 'es: ' +
                                    aN.sumaFila(f));
}
```

Se declara la variable local **f** y se le asignará un número de filas que el usuario ingresará mediante un cuadro de diálogo de entrada. Luego se compara si dicho valor no sobrepasa el límite de la matriz; si es así, se envía un mensaje al usuario; de lo contrario, se invoca al método **sumaFila** de la clase **ArregloBiNumeros** mediante el objeto **An**.

El siguiente script muestra las instrucciones del botón **btnColumnas**:

```
private void btnColumnasActionPerformed(java.awt.event.ActionEvent evt) {
    int c=Integer.parseInt(JOptionPane.showInputDialog(this,'Ingrese un
                                        numero de Columna: '));

    if (c>getColumnas())
        mensaje('El número de Columnas es superior a la matriz');
    else
        moRespuesta.addElement('La Suma de la Columna ' + c + 'es: ' +
                                    aN.sumaColumna(c));
}
```

Se declara la variable local **c** y se le asignará un número de columnas que el usuario ingresará mediante un cuadro de diálogo de entrada. Luego se compara si dicho valor no sobrepasa el límite de la matriz; si es así, se envía un mensaje al usuario; de lo contrario, se invoca al método **sumaColumna** de la clase **ArregloBiNumeros** mediante el objeto **An**.

Vector de objetos y ArrayList

CAPÍTULO

10

Competencias

- Reconoce un vector de objetos.
- Emplea vectores de objetos en aplicaciones comerciales.

Contenido

10.1 Vector de objetos

10.2 Clase ArrayList

- Caso desarrollado 1: Mantenimiento de empleados (vector de objetos)
- Caso desarrollado 2: Mantenimiento de facturas (**ArrayList**)
- Caso desarrollado 3: Mantenimiento de productos (**ArrayList** y **Lambda**)

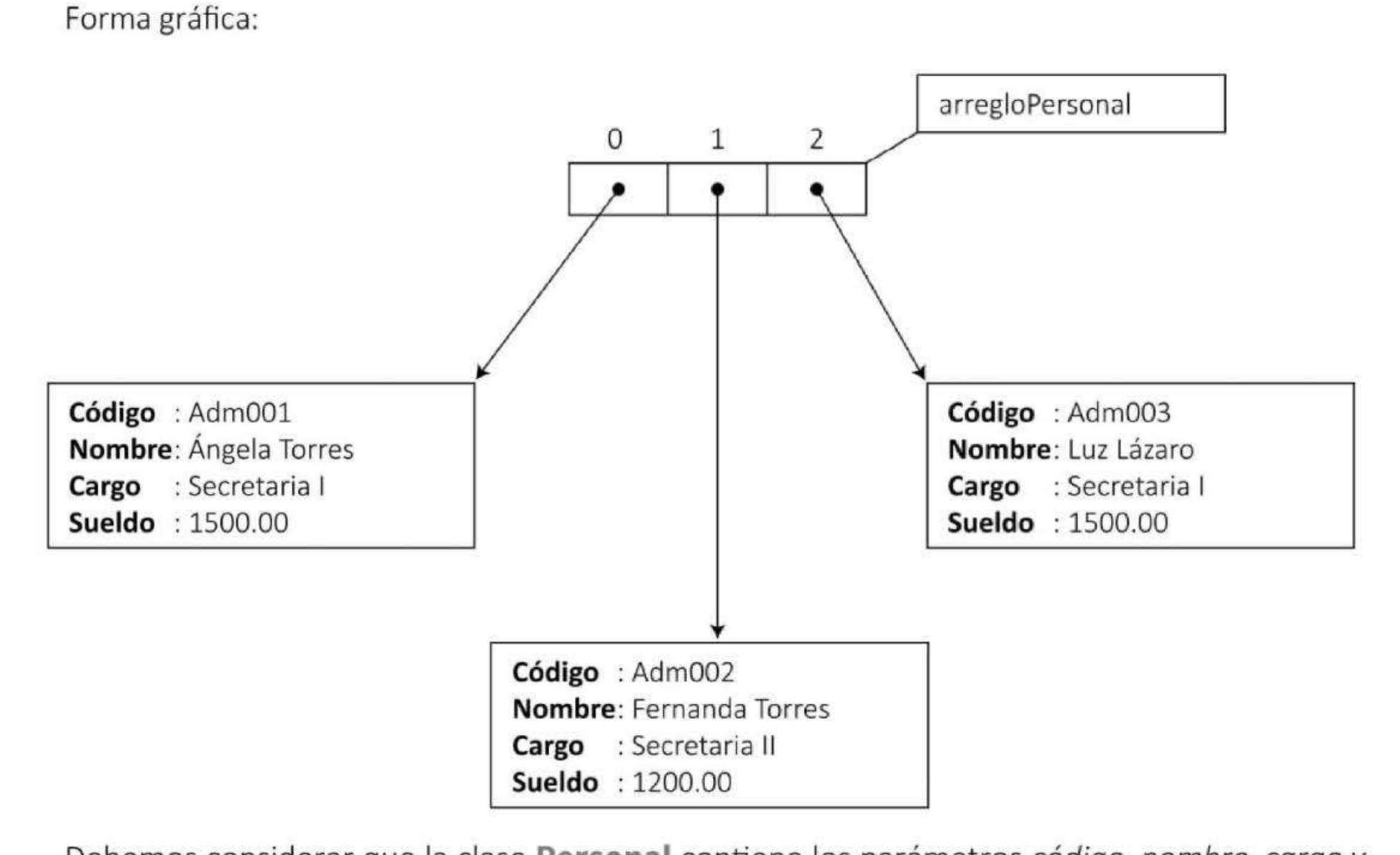
10.1 Vector de objetos

Los *vectores de objetos* son colecciones de tipos de datos definidos por el programador o que forman parte de la API JAVA y que son referenciados desde el arreglo.

Si tenemos una clase llamada **Personal**, podemos crear muchos registros del personal, pero se perderían los valores. Ahora bien, si implementamos un arreglo de la clase **Personal**, estaríamos en la capacidad de almacenar muchos registros del personal dentro de un vector y no perderíamos la información anterior al último registro.

Las aplicaciones que se implementarán en los casos desarrollados serán de mantenimiento de registros: agregar, modificar, eliminar y listar, así como sucede cuando se trabaja con un motor de base de datos.

Forma gráfica:



Debemos considerar que la clase **Personal** contiene los parámetros *código, nombre, cargo* y *sueldo*, y que **arregloPersonal** es un arreglo unidimensional de tipo **Personal**, y que al registrar al primer personal será en la posición 0 del **arregloPersonal**.

Formato de declaración del vector de objetos:

```
NombreClase[][] nombreVector;
```

Donde:

- **NombreClase.** Es el nombre de la clase implementada previamente a la declaración del arreglo.
- **nombreVector.** Es el nombre asignado al vector de objetos. Hay que tener en cuenta que no debe ser igual al nombre de la clase.

Formato de creación del vector de objetos:

```
nombreVector = new NombreClase[Numero];
```

Donde:

- **nombreVector.** Es el nombre del vector declarado previamente.
- **NombreClase.** Es el nombre de la clase de la cual se va a crear el vector.
- **Numero.** Define el número total de elementos a registrar en el vector de objetos.

10.2 Clase ArrayList

La clase **ArrayList** permite tener un control de los datos realizando el mismo trabajo de un vector de objetos; esta clase dispone de diversos métodos para manipular una colección de objetos dinámicamente.

Formato para crear un **ArrayList**:

```
ArrayList <nombre_clase> objeto = new ArrayList < nombre_clase > ();
```

Donde:

- **ArrayList.** Es la representación de la clase **ArrayList**.
- **nombre_clase.** Es el nombre de la clase que se creará en el vector.
- **Objeto.** Es el nombre asignado al **ArrayList**.

Por ejemplo, tenemos:

```
ArrayList <Producto> prod = new ArrayList <Producto> ();
```

Métodos que componen la clase ArrayList

Métodos	Descripción

add(Object)	Agrega un elemento al final.
	`public void adicionar(Producto x){`
	`  prod.add(x);`
	`}`

add(**int**, Object)	Agrega un elemento en la posición especificada en el primer parámetro. `prod.add(0,x);`
clear()	Elimina todos los elementos. `prod.clear();`

get(**int**)	Devuelve el elemento de la posición especificada.
	`public Producto obtener(int pos){`
	`  return prod.get(pos);`
	`}`

Métodos	Descripción

`indexOf(Object)`	Devuelve el índice del elemento especificado, de no encontrarlo devuelve –1.
	`public int posicion(Producto x){`
	`  return prod.indexOf(x);`
	`}`

`remove(int)`	Elimina el elemento de la posición especificada.
	`public void eliminar(int x){`
	`  prod.remove(x);`
	`}`

`remove(Object)`	Elimina el elemento especificado.
	`public void eliminar(Producto x){`
	`  prod.remove(x);`
	`}`

`set(int, Object)`	Reemplaza el elemento de la posición especificada en el primer parámetro por el elemento del segundo parámetro.
	`public void modificar(int pos, Producto x){`
	`  prod.set(pos,x);`
	`}`

`size()`	Devuelve la cantidad de elementos agregados.
	`public int tamaño(){`
	`  return prod.size();`
	`}`

CASO DESARROLLADO 1: **Mantenimiento de empleados (vector de objetos)**

Implemente una aplicación que permita realizar el mantenimiento de los empleados de una empresa, para lo cual se deben crear tres clases dentro de sus respectivos paquetes: la clase **Empleado** (clase que controla los atributos privados del empleado y los métodos **set** y **get**), la clase **ArregloEmpleados** (clase que maneja el vector de objetos) y la clase **frmMantenimiento** (clase que interactúa con el usuario a través de la GUI).

Consideraciones:

Dentro de la clase **Empleado** se deben implementar:

- Los atributos privados del empleado como **código**, **nombre**, **horas** y **tarifa**.
- El método **constructor** que inicialice los atributos.
- Los métodos **get** de todos los atributos.
- Los métodos **set** de todos los atributos.

Dentro de la clase **ArregloEmpleados** se deben considerar los siguientes aspectos:

- Declare como privado el arreglo unidimensional **aEmp** de tipo **Empleado** y el **índice**.
- Implemente los siguientes métodos:
  - El método **constructor**, que inicialice el vector de empleados en 100 elementos y un índice en cero.
  - El método **adiciona**, que se encargará de registrar un empleado en el arreglo e incrementará el índice por cada empleado registrado.
  - El método **getTamaño**, que devolverá el tamaño del vector, es decir, el total de empleados registrados.
  - El método **obtener**, que devolverá la matriz de elementos de acuerdo a la posición en el arreglo.
  - El método **eliminar**, que dependiendo de la posición del código buscado deberá eliminar a todo un registro del arreglo de objetos.
  - El método **buscar**, que será el encargado de comparar si el código ingresado existe en la matriz o no, deberá devolver, dependiendo de lo anterior, el índice de la posición del elemento encontrado; en caso contrario, devolverá –1.

Use el siguiente diseño de interfaz:

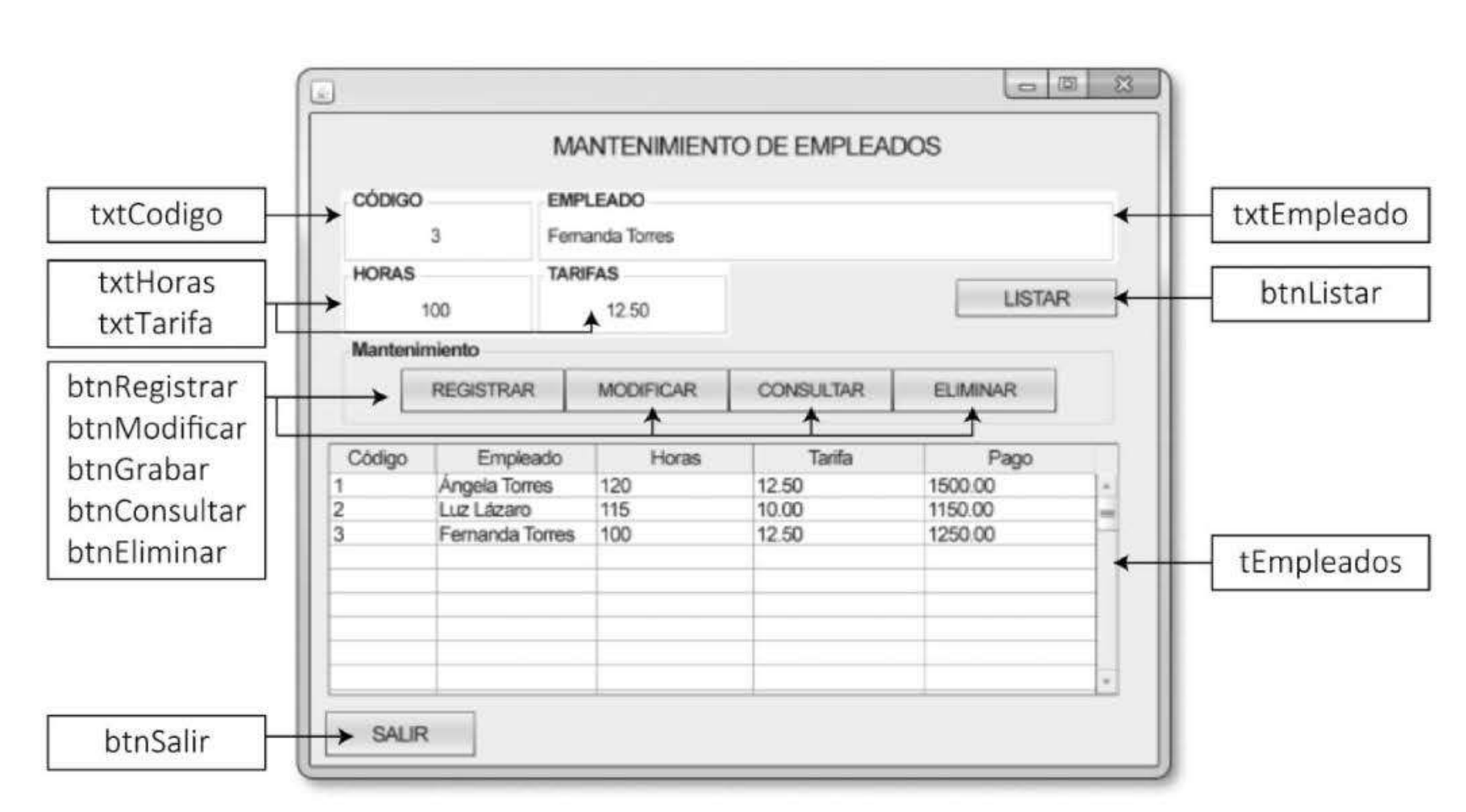


Figura 10.1 Formulario de mantenimiento de empleados.

Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjEmpresa**.
2. Agregue el paquete **pFormularios** al proyecto **pjEmpresa**.
3. Agregue el paquete **pClases** al proyecto **pjEmpresa**.

4. Agregue la clase **Empleado** al paquete **pClases**.
5. Agregue la clase **ArregloEmpleados** al paquete **pClases**.
6. Agregue la clase **frmMantenimiento** al paquete **pFormularios**.
7. Asigne un nombre a cada uno de los controles como se muestra en la figura 10.1; para ello, haga clic derecho sobre el objeto y seleccione **Change Variable Name...**
8. Después de asignar nombres a los objetos, asegúrese de que los controles sean los correctos; para ello, visualice el panel **Navigator**, el cual debe mostrarse como la figura 10.2.

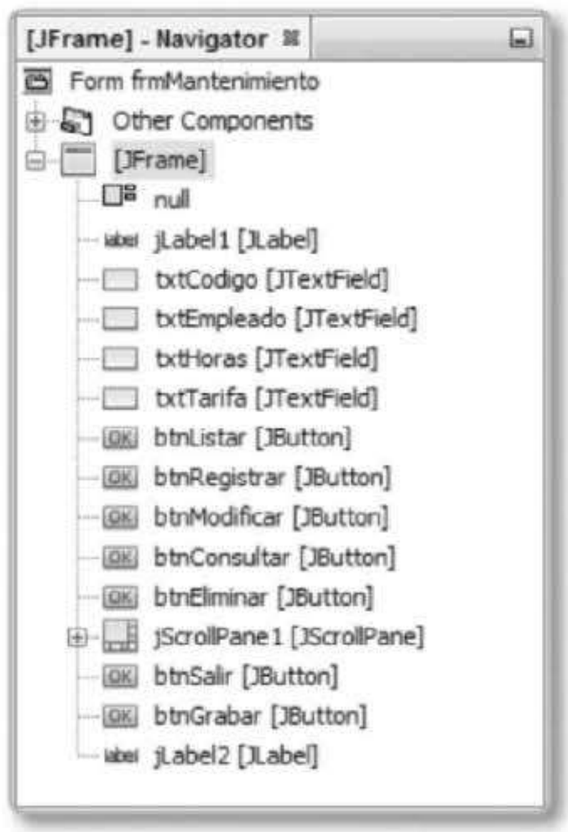


Figura 10.2 Listado de controles desde la ventana Navigator.

9. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.
10. Seleccione el formulario y en la propiedad **Form Size Police** modifique el valor encontrado por **Generate Resize Code**.

El siguiente script muestra el contenido de la clase **Empleado**:

```
package pClases;
public class Empleado {
    //Atributos de la clase
    private int código;
    private String nombre;
    private int horas;
    private double tarifa;

    //Método Constructor
    public Empleado(int codigo,String nombre, int horas,double tarifa){
        this.codigo=codigo;
        this.nombre=nombre;
        this.horas=horas;
        this.tarifa=tarifa;
    }

    //Métodos Get y Set de cada atributo
    public int getCodigo() {
        return codigo;
    }
```

```
    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getHoras() {
        return horas;
    }

    public void setHoras(int horas) {
        this.horas = horas;
    }

    public double getTarifa() {
        return tarifa;
    }

    public void setTarifa(double tarifa) {
        this.tarifa = tarifa;
    }
}
```

El siguiente script muestra el contenido de la clase **ArregloEmpleados**:

```
package pClases;
import javax.swing.JOptionPane;

public class ArregloEmpleados {
    //Atributos de la clase ArregloEmpleados
    private Empleado aE[];
    private int índice;

    //Método Constructor
    public ArregloEmpleados(){
        aE=new Empleado[100];
        índice=0;
    }

    //Método que registra los valores de un empleado
    //en el arreglo de empleados
    public void adiciona(Empleado objEmp){
        aE[índice]=objEmp;
        índice++;
    }

    //Método que devuelve los datos del Empleado desde el arreglo
    public Empleado devolver(int pos){
        return aE[pos];
    }

    //Método que devuelve el tamaño del arreglo de Empleados
    public int getTamaño(){
        return índice;
    }
```

```
    //Método que busca un empleado por su código
    public int buscar(int codigo){
        for(int i=0;i<getTamaño();i++)
            if (codigo==aE[i].getCodigo())
                return i;
        return -1;
    }
    //Método que permite eliminar un empleado
    public void eliminar(int codigo){
        int pos=buscar(codigo);
        for(int i=pos;i<getTamaño()-1;i++){
            aE[i]=aE[i+1];
        }
        indice--;
    }
}
```

El siguiente script muestra las librerías y variables globales de la clase **frmMantenimiento**:

```
package pFormularios;
import java.text.DecimalFormat;
import javax.swing.JOptionPane;
import pClases.ArregloEmpleados;
import pClases.Empleado;

public class frmMantenimiento extends javax.swing.JFrame {
    //Crear el objeto de la clase ArregloEmpleado
    ArregloEmpleados aEmp = new ArregloEmpleados();

    //Declaración de la variable df
    DecimalFormat df;

    public frmMantenimiento() {
        initComponents();

        //Asignación del formato de decimales para dinero
        df=new DecimalFormat('##0.00');
    }
}
```

El siguiente script muestra los métodos de la clase **frmMantenimiento**:

```
//Método que captura los valores ingresados por el usuario
int getCodigo(){
    return Integer.parseInt(txtCodigo.getText());
}

String getEmpleado(){
    return txtEmpleado.getText();
}

int getHoras(){
    return Integer.parseInt(txtHoras.getText());
}

double getTarifa(){
    return Double.parseDouble(txtTarifa.getText());
}
```

```
    //Método que limpia la tabla de Empleados
    void limpiaTabla(){
        for(int i=0;i<50;i++){
            tEmpleado.setValueAt('', i, 0);
            tEmpleado.setValueAt('', i, 1);
            tEmpleado.setValueAt('', i, 2);
            tEmpleado.setValueAt('', i, 3);
            tEmpleado.setValueAt('', i, 4);
        }
    }

    //Método que lista los valores a la tabla
    void listar(){
        limpiaTabla();
        for(int i=0;i<aEmp.getTamaño();i++){
            tEmpleado.setValueAt(aEmp.devolver(i).getCodigo(), i, 0);
            tEmpleado.setValueAt(aEmp.devolver(i).getNombre(), i, 1);
            tEmpleado.setValueAt(aEmp.devolver(i).getHoras(), i, 2);
            tEmpleado.setValueAt(df.format(aEmp.devolver(i).getTarifa()), i, 3);
            tEmpleado.setValueAt(df.format(aEmp.devolver(i).getTarifa() *
                                    aEmp.devolver(i).getHoras()), i, 4);
        }
    }

    //Método que valida el ingreso de los datos del usuario
    String valida(){
        if (txtCodigo.getText().equals('') ||
            Integer.parseInt(txtCodigo.getText())<0){
            txtCodigo.requestFocus();
            return 'Código del Empleado';
        } else if (txtEmpleado.getText().equals('')){
            txtEmpleado.requestFocus();
            return 'Nombre del Empleado';
        } else if (txtHoras.getText().equals('') ||
                   Integer.parseInt(txtHoras.getText())<0){
            txtHoras.requestFocus();
            return 'Hora de trabajo';
        } else if (txtTarifa.getText().equals('') ||
                   Double.parseDouble(txtTarifa.getText())<0){
            txtHoras.requestFocus();
            return 'Monto de Tarifa';
        } else
        return '';
    }

    //Método que limpia todas las cajas de texto
    void limpiarControles(){
        txtCodigo.setText('');
        txtEmpleado.setText('');
        txtTarifa.setText('');
        txtHoras.setText('');
    }
```

El siguiente script muestra las instrucciones del botón **btnRegistrar**:

```
private void btnRegistrarActionPerformed(java.awt.event.ActionEvent evt) {
    try{
        if (valida().equals('')){
            //Creando un objeto de la clase Empleado
            Empleado objEmp = new Empleado(getCodigo(), getEmpleado(),
                                           getHoras(),getTarifa());

            //Invocar al método registrar de la clase ArregloEmpleados
            aEmp.adiciona(objEmp);
            limpiarControles();
            listar();
        } else {
            JOptionPane.showMessageDialog(null,'Error en ' + valida());
        }
    } catch(Exception ex){
        JOptionPane.showMessageDialog(null, 'Error en la aplicación..!!');
    }
}
```

El siguiente script muestra las instrucciones del botón **btnConsultar**:

```
private void btnConsultarActionPerformed(…) {
    try {
        limpiaTabla();
        //Solicitar el código del empleado a consultar
        int codigo=Integer.parseInt(JOptionPane.showInputDialog(null,'Ingrese
                                    código a buscar'));

        //El código es enviado al método buscar
        int pos=aEmp.buscar(código);
        if (pos==-1)
            JOptionPane.showMessageDialog(null,'Código no existe');
        else{
            //Se muestran los valores en el control tEmpleado
            tEmpleado.setValueAt(aEmp.devolver(pos).getCodigo(), 0, 0);
            tEmpleado.setValueAt(aEmp.devolver(pos).getNombre(), 0, 1);
            tEmpleado.setValueAt(aEmp.devolver(pos).getHoras(), 0, 2);
            tEmpleado.setValueAt(df.format(aEmp.devolver(pos).getTarifa()), 0, 3);

            tEmpleado.setValueAt(df.format(aEmp.devolver(pos).getTarifa()*
                                           aEmp.devolver(pos).
                                           getHoras()), 0, 4);
            }
    } catch(Exception ex){
        JOptionPane.showMessageDialog(null,'Error en la aplicación..!!');
        }
}
```

El siguiente script muestra las instrucciones del botón **btnListar**:

```
private void btnListarActionPerformed(java.awt.event.ActionEvent evt) {
    listar();
}
```

El siguiente script muestra las instrucciones del botón **btnEliminar**:

```
private void btnEliminarActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        //Solicitando el código del empleado a eliminar
        int codigo=Integer.parseInt(JOptionPane.showInputDialog(null,
                                    'Ingrese código a eliminar: '));

        //Invocando al método eliminar del arreglo
        aEmp.eliminar(código);
        JOptionPane.showMessageDialog(null,'Empleado eliminado...!!');
    } catch(Exception ex){
        JOptionPane.showMessageDialog(null,'Error en la aplicación..!!');
    }
}
```

El siguiente script muestra las instrucciones del botón **btnModificar**:

```
private void btnModificarActionPerformed(…) {
    try {
        int codigoB=Integer.parseInt(JOptionPane.showInputDialog(this,
                                    'Ingrese Codigo a Modificar:'));

        //Buscar si el código del empleado Existe en el Arreglo
        int pos=aEmp.buscar(codigoB);
        if (pos!=-1){
            //Devuelve los datos del Empleado encontrado
            Empleado x = aEmp.devolver(pos);

            //Habilita el botón btnGrabar y oculta el botón btnModificar
            btnGrabar.setVisible(true);
            btnModificar.setVisible(false);

            //Se muestra el código del empleado y se bloquea
            txtCodigo.setText(''+codigoB);
            txtCodigo.setEditable(false);

            //Se envían los valores desde el arreglo a los controles
            txtEmpleado.setText(x.getNombre());
            txtHoras.setText(''+x.getHoras());
            txtTarifa.setText(''+x.getTarifa());
            txtEmpleado.requestFocus();
        } else {
            //Mensaje de error cuando el código del empleado sea errado
            JOptionPane.showMessageDialog(null,'Código no existe');
        }
    } catch(Exception ex){
        JOptionPane.showMessageDialog(null, 'Error en la aplicación..!!');
    }
}
```

El siguiente script muestra las instrucciones del botón **btnGrabar**:

```
private void btnGrabarActionPerformed(java.awt.event.ActionEvent evt) {
    try{
        //buscando el código del empleado a modificar
        int pos=aEmp.buscar(getCodigo());

        //La posición es enviada al arreglo para devolver
        //el empleado buscado
        Empleado x=aEmp.devolver(pos);

        //Actualizando los valores del empleado
        x.setNombre(getEmpleado());
        x.setHoras(getHoras());
        x.setTarifa(getTarifa());
        JOptionPane.showMessageDialog(null, 'Empleado actualizado
                    correctamente', 'Confirmación',
                    JOptionPane.INFORMATION_MESSAGE);
        btnModificar.setVisible(true);
        btnGrabar.setVisible(false);
    }catch(Exception ex){
        JOptionPane.showMessageDialog(null,'Error al actualizar...!!',
                    'Error', JOptionPane.ERROR_MESSAGE);
    }
}
```

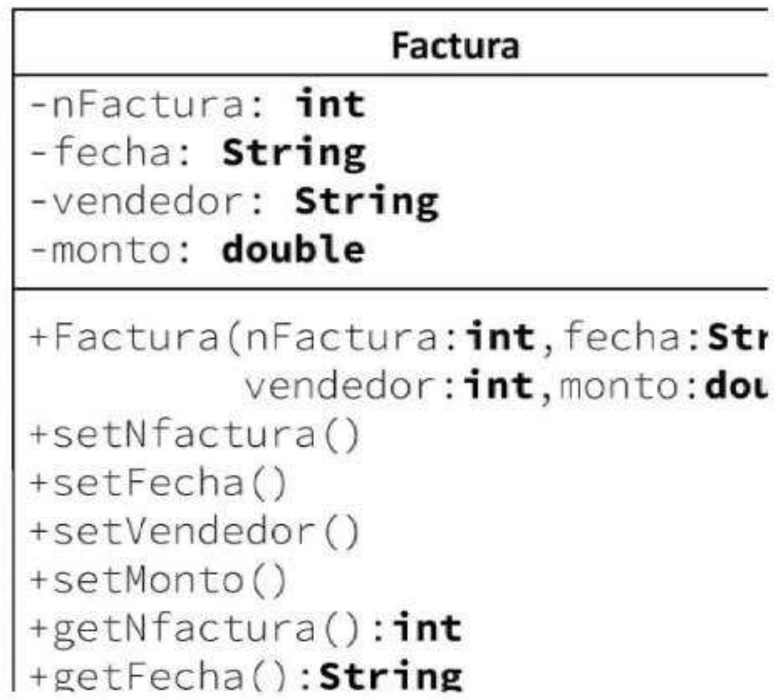
CASO DESARROLLADO 2: **Mantenimiento de facturas (ArrayList)**

Implemente una aplicación que permita controlar las facturas de una tienda comercial, para lo cual se deben crear tres clases dentro de sus paquetes respectivos: la clase **Factura** (clase que controla los atributos privados de la factura), la clase **ArregloFacturas** (clase que maneja el vector de tipo Factura) y la clase **frmVenta** (clase que interactúa con el usuario a través de la GUI).

Consideraciones:

Dentro de la clase **Factura** se deben implementar:

- Los atributos privados para **Nfactura** (int), **fecha** (String), **vendedor** (String) y **monto** (double).
- El método **constructor** que inicialice los atributos.
- Los métodos **get** de todos los atributos.
- Los métodos **set** de todos los atributos.



Dentro de la clase **ArregloFacturas** se deben considerar los siguientes aspectos:

- Declare como atributo privado el arreglo unidimensional **fact** de tipo **ArrayList** y el **índice.**
- Implemente los siguientes métodos:
  - El método **constructor**, que inicialice el arreglo de **fact** de tipo **ArrayList**.
  - El método **agregar**, que se encargará de registrar una factura en el **ArrayList**.
  - El método **getTamaño**, que devolverá el tamaño del **ArrayList**.
  - El método **obtener**, que devolverá todos los datos registrados de la factura de acuerdo a la posición en el **ArrayList**.
  - El método **buscar**, , que será el encargado de comparar si el **número de Factura** ingresado existe en la matriz o no, deberá devolver, dependiendo de lo anterior, el **objeto de la Factura** encontrado; en caso contrario, devolverá **null.** Aquí se debe utilizar la estructura **for** de la siguiente forma:

```
for(Clase varReferencia : NombreDelArrayList){
    return varReferencia;
}
```

Ej: Si tenemos la Clase **Personal** y quisiéramos buscarlo por su código de tipo **int**, dentro del **ArrayList** Per, ingresamos lo siguiente:

```
public Personal buscar(int codigo){
    for( Personal p : Per ) {
        if ( p.getCodigo() -- codigo )
            return p;
    }
    return null;
}
```

  - El método **eliminar**que, dependiendo del tipo, deberá eliminar el objeto analizado del **ArrayList**.

ArregloFactura
-fact: **ArrayList <Factura>** -indice: **int**
+ArregloFacturas() +getTamaño(): **int** +agregar(f:**Factura**) +obtener(pos:**int**): **Factura** +buscar(num:**int**): **Factura** +eliminar(x:**Factura**)

Adicionalmente, se deben implementar en la Clase **ArregloFacturas** los siguientes métodos:

- El método que retorne el nombre del empleado que tiene la venta más alta.
- En el formulario diseñe las opciones de **mantenimiento** como **registrar** (para un nuevo registro de factura generando el número de factura de forma correlativa), **consultar** (que permite buscar una determinada factura), **modificar** (que permite modificar el nombre del vendedor y

el monto registrado, mas no el número ni la fecha, ya que estos son autogenerados) y **eliminar** (que de acuerdo a un determinado objeto deberá eliminar un registro de factura del **ArrayList**).

Use el siguiente diseño de interfaz:

Figura 10.3 Formulario de registro de ventas.

Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjArrayList**.
2. Agregue el paquete **pFormularios** al proyecto **pjArrayList**.
3. Agregue el paquete **pClases** al proyecto **pjArrayList**.
4. Agregue la clase **Factura** al paquete **pClases**.
5. Agregue la clase **ArregloFacturas** al paquete **pClases**.
6. Agregue la clase **frmMantenimiento** al paquete **pFormularios**.
7. Asigne un nombre a cada uno de los controles como se muestra en la figura 10.3; para ello, haga clic derecho sobre el objeto y seleccione **Change Variable Name...**

8. Después de asignar nombres a los objetos, asegúrese de que los controles sean los correctos; para ello, visualice el panel **Navigator**, el cual debe mostrarse como la figura 10.4.

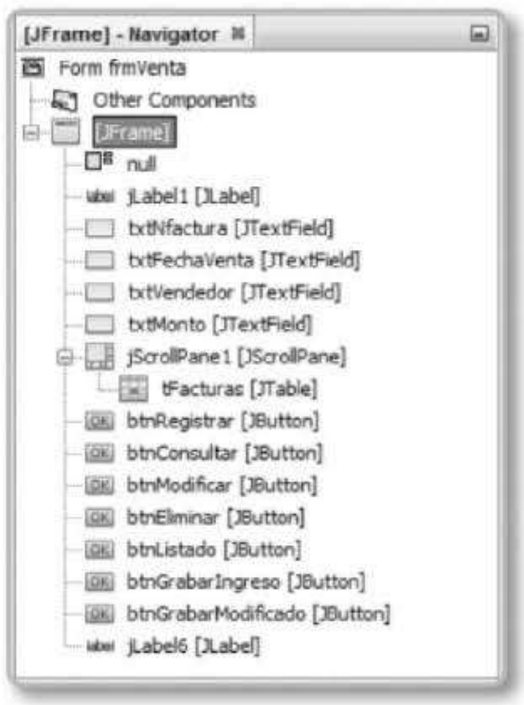


Figura 10.4 Listado de controles desde la ventana Navigator.

9. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.
10. Seleccione el formulario y en la propiedad **Form Size Police** modifique el valor encontrado por **Generate Resize Code**.

El siguiente script muestra el contenido de la clase **Factura**:

```
package pClases;

public class Factura {
    private int nFactura;
    private String fecha;
    private String vendedor;
    private double monto;

    //Método constructor
    public Factura(int nFactura,String fecha, String vendedor,double monto)
{
        this.nFactura=nFactura;
        this.fecha=fecha;
        this.vendedor=vendedor;
        this.monto=monto;
    }

    //Métodos set
    public void setNfactura(int nFactura){
        this.nFactura=nFactura;
    }

    public void setFecha(String fecha){
        this.fecha=fecha;
    }

    public void setVendedor(String vendedor){
        this.vendedor=vendedor;
    }

    public void setMonto(double monto){
        this.monto=monto;
    }
```

```
        //Métodos get
        public int getNfactura(){
            return nFactura;
        }

        public String getfecha(){
            return fecha;
        }

        public String getVendedor(){
            return vendedor;
        }

        public double getMonto(){
            return monto;
        }
    }
```

El siguiente script muestra el contenido de la clase **ArregloFacturas**:

```
package pClases;
import java.util.ArrayList;

public class ArregloFacturas {
    private ArrayList <Factura> fact;
    private int indice;

    //Método constructor
    public ArregloFacturas(){
        fact=new ArrayList<Factura>();
    }

    //Método que devuelve el tamaño actual del vector
    public int getTamaño(){
        return fact.size();
    }

    //Método que permite agregar una factura al vector
    public void agregar(Factura F){
        fact.add(F);
    }

    //Método que devuelve el objeto factura
    public Factura obtener(int pos){
        return fact.get(pos);
    }

    //Método que busca una factura
    public Factura buscar(int num){
        for (Factura f: fact)
            if (f.getNfactura()==num)
                return f;
        return null;
    }

    //Método que elimina una factura
    public void eliminar(Factura x){
        fact.remove(x);
    }
}
```

El siguiente script muestra las librerías y variables globales de la clase **frmMantenimiento**:

```
package pFormularios;
import java.util.GregorianCalendar;
import javax.swing.JOptionPane;
import javax.swing.table.TableColumn;
import pClases.ArregloFacturas;
import pClases.Factura;

public class frmVenta extends javax.swing.JFrame {
    ArregloFacturas f=new ArregloFacturas();
    int num=0;

    public frmVenta() {
        initComponents();
        DefinirAnchos();
        asignaFecha();
        habilitaCajas(false);
        btnGrabarIngreso.setVisible(false);
        btnGrabarModificado.setVisible(false);
    }
}
```

El siguiente script muestra los métodos de la clase **frmMantenimiento**:

```
//Método que define el ancho de las columnas de la tabla
void DefinirAnchos(){
    TableColumn columna;
    columna=tFacturas.getColumnModel().getColumn(0);
    columna.setPreferredWidth(30);
    columna=tFacturas.getColumnModel().getColumn(1);
    columna.setPreferredWidth(50);
    columna=tFacturas.getColumnModel().getColumn(2);
    columna.setPreferredWidth(150);
    columna=tFacturas.getColumnModel().getColumn(3);
    columna.setPreferredWidth(70);
    tFacturas.getTableHeader().setReorderingAllowed(false);
    tFacturas.getTableHeader().setResizingAllowed(false);
}

//Método que bloquea y desbloquea los controles JTextField
void habilitaCajas(boolean opcion){
    txtNfactura.setEditable(opcion);
    txtVendedor.setEditable(opcion);
    txtMonto.setEditable(opcion);
    txtFechaVenta.setEditable(opcion);
}

//Método que limpia los controles JTextField
void limpiaCajas(){
    txtNfactura.setText('');
    txtVendedor.setText('');
    txtFechaVenta.setText('');
    txtMonto.setText('');
}
```

```
    //Método que limpia el control tFacturas
    void limpiaMatriz(){
        for(int i=0;i<10;i++){
            tFacturas.setValueAt('', i, 0);
            tFacturas.setValueAt('', i, 1);
            tFacturas.setValueAt('', i, 2);
            tFacturas.setValueAt('', i, 3);
        }
    }

    //Método que genera el número de factura
    public int generaNumero(){
        num++;
        return num;
    }

    //Métodos que capturan los valores ingresados por el usuario
    public int getNumFact(){
        return Integer.parseInt(txtNfactura.getText());
    }

    public String getFecha(){
        return txtFechaVenta.getText();
    }

    public String getVendedor(){
        return txtVendedor.getText();
    }

    public double getMonto(){
        return Double.parseDouble(txtMonto.getText());
    }

    //Método que lista las facturas en el control tFacturas
    void listar(){
        if (f.getTamaño()>0){
            for(int i=0;i<f.getTamaño();i++){
                Factura fact=f.obtener(i);
                tFacturas.setValueAt(fact.getNfactura(),i,0);
                tFacturas.setValueAt(fact.getfecha(), i, 1);
                tFacturas.setValueAt(fact.getVendedor(), i, 2);
                tFacturas.setValueAt(fact.getMonto(), i, 3);
            }
        } else {
            JOptionPane.showMessageDialog(this, 'No hay facturas registradas',
                        'Confirmación',JOptionPane.INFORMATION_MESSAGE);
            limpiaMatriz();
        }
    }

    //Método que muestra la fecha actual en el control txtFechaVenta
    void asignaFecha(){
        GregorianCalendar cal = new GregorianCalendar();
        txtFechaVenta.setText(cal.get(cal.DAY_OF_MONTH) + '/' + cal.MONTH +
                              '/' +cal.get(cal.YEAR));
    }
```

El siguiente script muestra las instrucciones del botón **btnRegistrar**:

```
private void btnRegistrarActionPerformed(…) {
    txtNfactura.setText('' + generaNumero());
    asignaFecha();
    txtVendedor.requestFocus();

    habilitaCajas(true);
    txtVendedor.setEditable(true);
    txtMonto.setEditable(true);

    txtVendedor.setText('');
    txtMonto.setText('');

    btnRegistrar.setVisible(false);
    btnGrabarIngreso.setVisible(true);
}
```

El siguiente script muestra las instrucciones del botón **btnConsultar**:

```
private void btnConsultarActionPerformed(…) {
    try{
        limpiaCajas();
        limpiaMatriz();
        int buscoFactura=Integer.parseInt(JOptionPane.showInputDialog(null,
                                        'Ingrese un número de Factura:'));

        //objeto fact que busca el número de factura en el ArrayList f
        Factura fact=f.buscar(buscoFactura);
        if (fact!=null){
            tFacturas.setValueAt(fact.getNfactura(), 0, 0);
            tFacturas.setValueAt(fact.getfecha(), 0, 1);
            tFacturas.setValueAt(fact.getVendedor(), 0, 2);
            tFacturas.setValueAt(fact.getMonto(), 0, 3);
        }else
            JOptionPane.showMessageDialog(null,'Factura NO encontrada',
                        'Confirmación',JOptionPane.ERROR_MESSAGE);
    } catch(Exception ex){
        JOptionPane.showMessageDialog(null,'Error de Entrada de Datos',
                            'Confirmación',JOptionPane.ERROR_MESSAGE);
    }
}
```

El siguiente script muestra las instrucciones del botón **btnModificar**:

```
private void btnModificarActionPerformed(…) {
    try{
        limpiaCajas();
        limpiaMatriz();

        btnModificar.setVisible(false);
        btnGrabarModificado.setVisible(true);

        int buscoFactura=Integer.parseInt(JOptionPane.
        showInputDialog(null,'Ingrese un numero de Factura:'));
```

```
        //objeto fact que busca el número de factura en el ArrayList f
        Factura fact=f.buscar(buscoFactura);
        if (fact!=null){
            tFacturas.setValueAt(fact.getNfactura(), 0, 0);
            tFacturas.setValueAt(fact.getfecha(), 0, 1);
            tFacturas.setValueAt(fact.getVendedor(), 0, 2);
            tFacturas.setValueAt(fact.getMonto(), 0, 3);

            txtNfactura.setText(''+fact.getNfactura());
            txtFechaVenta.setText(fact.getfecha());
            txtVendedor.setText(fact.getVendedor());
            txtMonto.setText(''+fact.getMonto());

            habilitaCajas(true);
            txtNfactura.setEditable(false);
            txtFechaVenta.setEditable(false);
        }else
            JOptionPane.showMessageDialog(null,'Factura NO encontrada',
                        'Confirmación', JOptionPane.ERROR_MESSAGE);
    }catch(Exception ex){
        JOptionPane.showMessageDialog(null,'Factura NO encontrada',
                    'Confirmación',JOptionPane.ERROR_MESSAGE);
        btnModificar.setVisible(true);
        btnGrabarModificado.setVisible(false);
    }
}
```

El siguiente script muestra las instrucciones del botón **btnEliminar**:

```
private void btnEliminarActionPerformed(…) {
    try{
        int buscoFactura=Integer.parseInt(JOptionPane.showInputDialog(null,
                                'Ingrese un numero de Factura a Eliminar:'));

        //objeto fact que busca el número de factura en el ArrayList f
        Factura fact=f.buscar(buscoFactura);
        if (fact!=null){
            f.eliminar(fact);
            JOptionPane.showMessageDialog(null,'Factura Eliminada',
                        'Confirmación',JOptionPane.INFORMATION_MESSAGE);
            listar();
        } else
            JOptionPane.showMessageDialog(null, 'NO existe el N{umero de
                        Factura ingresada', 'Confirmacion',
                        JOptionPane.INFORMATION_MESSAGE);
    }catch(Exception ex){
        JOptionPane.showMessageDialog(null, 'NO existe el Número de
                                Factura ingresada', 'Confirmación',
                                JOptionPane.INFORMATION_MESSAGE);
    }
}
```

El siguiente script muestra las instrucciones del botón **btnListar**:

```
private void btnListadoActionPerformed(java.awt.event.ActionEvent evt) {
    listar();
}
```

El siguiente script muestra las instrucciones del botón **btnGrabarIngreso**:

```
private void btnGrabarIngresoActionPerformed(…) {
    try{
        habilitaCajas(false);
        btnRegistrar.setVisible(true);
        btnGrabarIngreso.setVisible(false);
        Factura fact=new Factura(getNumFact(), getFecha(),
                                 getVendedor(), getMonto());
        f.agregar(fact);
        listar();
        JOptionPane.showMessageDialog(null, 'Factura ingresada
                                 correctamente', 'Confirmación', JOptionPane.
                                 INFORMATION_MESSAGE);
    }catch(Exception ex){
        JOptionPane.showMessageDialog(null, 'Error de Ingreso de Datos',
                                 'Error',JOptionPane.ERROR_MESSAGE);
        num--;
    }
}
```

El siguiente script muestra las instrucciones del botón **btnGrabarModificado**:

```
private void btnGrabarModificadoActionPerformed(…) {
    try{
        Factura fact=f.buscar(getNumFact());
        fact.setVendedor(getVendedor());
        fact.setMonto(getMonto());
        JOptionPane.showMessageDialog(null, 'Factura Modificada
                                 Correctamente', 'Confirmación',
                                 JOptionPane.INFORMATION_MESSAGE);
        listar();
    }catch(Exception ex){
        JOptionPane.showMessageDialog(null, 'Ocurrió un error al
                                 intentar Grabar','Confirmación',
                                 JOptionPane.INFORMATION_MESSAGE);
    }
    btnGrabarModificado.setVisible(false);
    btnModificar.setVisible(true);
}
```

CASO DESARROLLADO 3: **Mantenimiento de productos (ArrayList y Lambda)**

Implemente una aplicación que permita realizar el mantenimiento de registros de los productos en una tienda, para lo cual se deben crear tres clases en sus paquetes respectivos: la clase **Producto** (clase que controla los atributos privados de los productos), la clase **ArregloProductos** (clase que maneja el vector de tipo Producto) y la clase **frmProducto** (clase que interactúa con el usuario a través de la GUI).

Consideraciones:

Dentro de la clase **Producto** se deben implementar:

- Los atributos privados para **codigo** (int), **descripcion** (String), **precio** (double), **categoria** (String) y **stock** (int).
- El método **constructor** que inicialice los atributos.
- Los métodos **get** de todos los atributos.
- Los métodos **set** de todos los atributos.

```
Producto
-codigo: int
-descripcion: String
-precio: double
-categoria: String
-stock: int
+Producto(codigo:int,descripcion:String,
          precio:double,categoria:String,
          stock:int)
+getCodigo():int
+setCodigo(int codigo)
+getDescripcion():String
+setDescripcion(String descripcion)
+getPrecio():double
+setPrecio(double precio)
+getCategoria():String
+setCategoria(String categoria)
+getStock():int
+setStock(int stock)
```

Dentro de la clase **ArregloProductos** se deben considerar los siguientes aspectos:

- Declare como atributo privado el arreglo unidimensional **prod** de tipo **ArrayList.**
- Implemente los siguientes métodos:
  - El método **constructor**, que inicialice el arreglo de **prod** de tipo **ArrayList**.
  - El método **agregar**, que se encargará de registrar un producto en el **ArrayList**.
  - El método **totalProductos**, que devolverá el tamaño del ArrayList. Use métodos **Lambda**.
  - El método **obtener**, que devolverá todos los datos registrado del producto de acuerdo a la posición en el **ArrayList**.

- El método **buscarProducto**, que será el encargado de comparar si el **código de producto** ingresado existe en la matriz o no, deberá devolver, dependiendo de lo anterior, el **objeto del producto** encontrado; en caso contrario, devolverá **null**.
- El método **eliminar**, que dependiendo del tipo de objeto deberá eliminarlo del **ArrayList**.

ArregloProductos
-prod: **ArrayList <Producto>**
+ArregloProducto() +totalProductos():Lambda +agregaProducto(objP: **Producto**) +obtener(): **Producto** +buscarProducto(cod:**int**):**Producto** +eliminaProducto(objP: **Producto**)

Finalmente, en el formulario diseñe las opciones de **mantenimiento** como **grabar** (para un nuevo registro de producto generando el código de forma correlativa), **buscar** (que permite buscar un determinado producto), **modificar** (que permite modificar el nombre y los datos del producto) y **eliminar** (que de acuerdo a un determinado objeto deberá eliminar un registro de producto del **ArrayList**).

Use el siguiente diseño de interfaz:

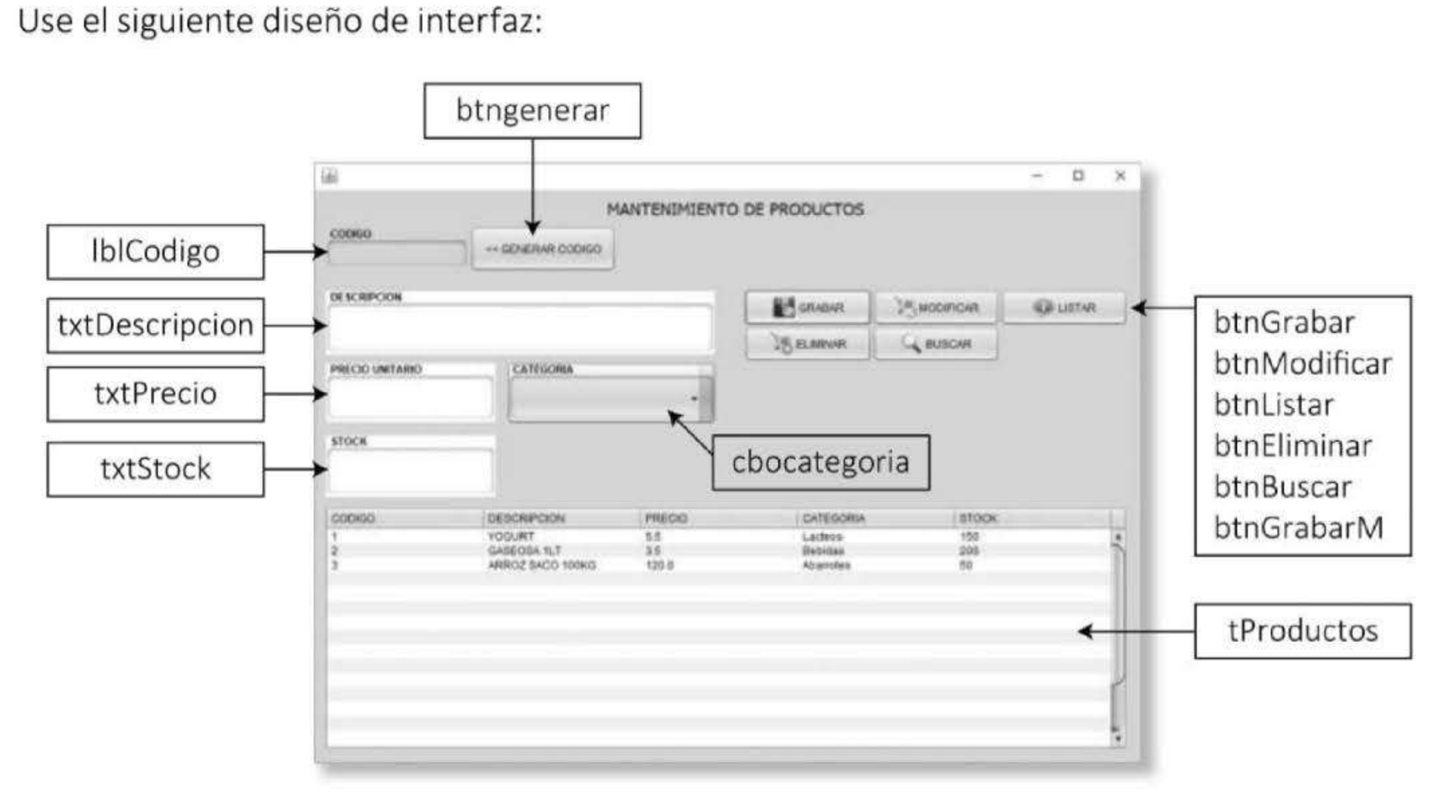


Figura 10.5 Interfaz del mantenimiento de productos.

Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjMantenimientoProductos**.
2. Agregue el paquete **pFormularios** al proyecto **pjArrayList**.
3. Agregue el paquete **pClases** al proyecto **pjArrayList**.
4. Agregue el paquete **pImagenes** al proyecto.
5. Agregue la clase **Producto** al paquete **pClases**.

- El método **buscarProducto**, que será el encargado de comparar si el **código de producto** ingresado existe en la matriz o no, deberá devolver, dependiendo de lo anterior, el **objeto del producto** encontrado; en caso contrario, devolverá **null**.
- El método **eliminar**, que dependiendo del tipo de objeto deberá eliminarlo del **ArrayList**.

ArregloProductos
-prod: **ArrayList <Producto>**
+ArregloProducto() +totalProductos():Lambda +agregaProducto(objP: **Producto**) +obtener(): **Producto** +buscarProducto(cod:**int**):**Producto** +eliminaProducto(objP: **Producto**)

Finalmente, en el formulario diseñe las opciones de **mantenimiento** como **grabar** (para un nuevo registro de producto generando el código de forma correlativa), **buscar** (que permite buscar un determinado producto), **modificar** (que permite modificar el nombre y los datos del producto) y **eliminar** (que de acuerdo a un determinado objeto deberá eliminar un registro de producto del **ArrayList**).

Use el siguiente diseño de interfaz:

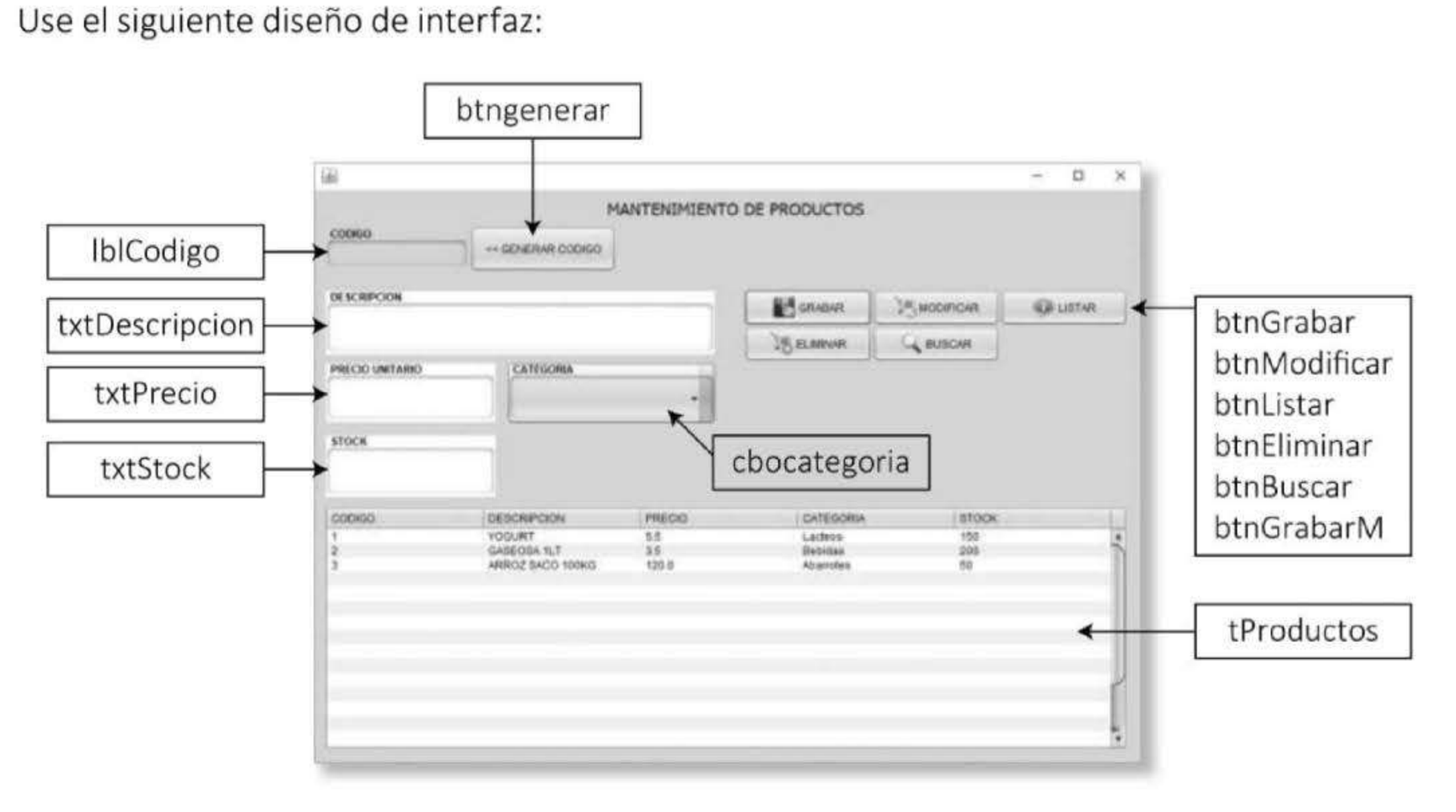


Figura 10.5 Interfaz del mantenimiento de productos.

Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjMantenimientoProductos**.
2. Agregue el paquete **pFormularios** al proyecto **pjArrayList**.
3. Agregue el paquete **pClases** al proyecto **pjArrayList**.
4. Agregue el paquete **pImagenes** al proyecto.
5. Agregue la clase **Producto** al paquete **pClases**.

6. Agregue la clase **ArregloProductos** al paquete **pClases**.
7. Agregue la clase **JFrameForm** y llámelo **frmProducto** dentro del paquete **pFormularios**.
8. Asigne un nombre a cada uno de los controles como se muestra en la figura 10.5; para ello, haga clic derecho sobre el objeto y seleccione **Change Variable Name...**
9. Al finalizar el proyecto, asegúrese de que el explorador de proyectos se muestre como la figura 10.6.

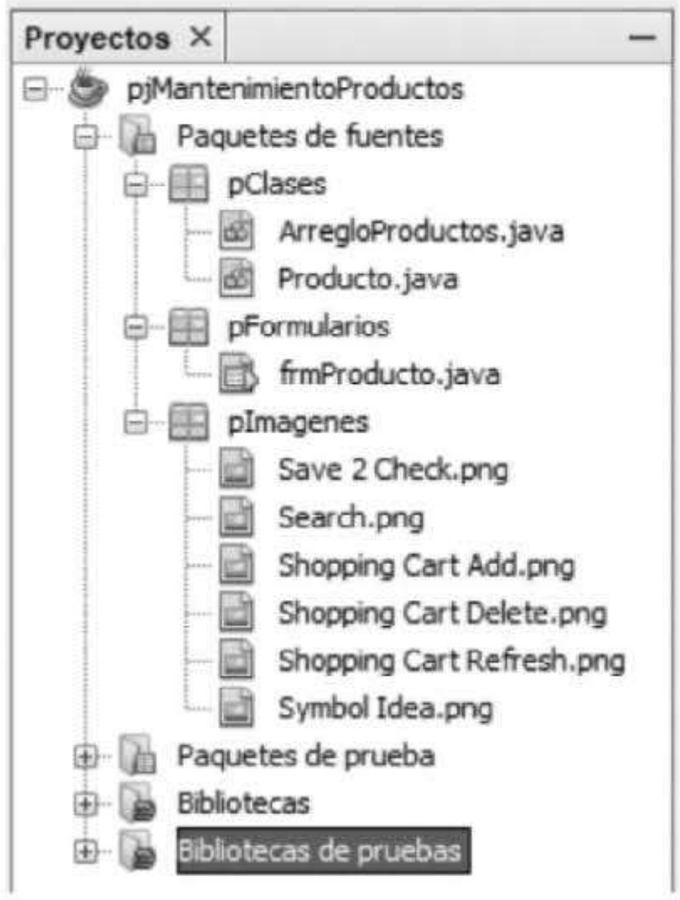


Figura 10.6 Explorador de proyectos.

10. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.
11. Seleccione el formulario y en la propiedad **Form Size Police** modifique el valor encontrado por **Generate Resize Code**.

El siguiente script muestra el contenido de la clase **Producto**:

```
package pClases;
public class Producto {
    private int codigo;
    private String descripcion;
    private double precio;
    private String categoria;
    private int stock;

    //Definir el constructor
    public Producto(int codigo, String descripcion, double precio,
                    String categoria, int stock) {
        this.codigo = codigo;
        this.descripcion = descripcion;
        this.precio = precio;
        this.categoria = categoria;
        this.stock = stock;
    }
```

```
    //Definir los métodos Get y Set
    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    public String getCategoria() {
        return categoria;
    }

    public void setCategoria(String categoria) {
        this.categoria = categoria;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

El siguiente script muestra el contenido de la clase **ArregloProductos**.

```
package pClases;
import java.util.ArrayList;

public class ArregloProductos {
    public interface Operacion{
        int determinar();
    }

    //Definir el arreglo de productos
    ArrayList<Producto> prod;

    //Definir el método constructor
    public ArregloProductos(){
        prod = new ArrayList<Producto>();
    }
```

```
        //Método Lambda que devuelve el total de productos registrados
        public Operacion totalProductos=()-> prod.size();

        //Método para agregar un nuevo producto
        public void agregaProducto(Producto objP){
            prod.add(objP);
        }

        //Método para obtener elementos del arreglo
        public Producto obtener(int pos){
            return prod.get(pos);
        }

        //Método para buscar un producto desde el arreglo
        public Producto buscarProducto(int cod){
            for(Producto p:prod){
                if (p.getCodigo()==cod)
                    return p;
            }
            return null;
        }

        //Método para eliminar un producto del arreglo
        public void eliminaProducto(Producto objP){
            prod.remove(objP);
        }
    }
```

El siguiente script muestra el contenido del formulario **frmProducto**:

```
package pFormularios;

import javax.swing.JOptionPane;
import pClases.ArregloProductos;
import pClases.Producto;

public class frmProducto extends javax.swing.JFrame {
    ArregloProductos aProductos = new ArregloProductos();
    int n=1;
    public frmProducto() {
        initComponents();
        //Ocultar los botones de grabación
        btnGrabarM.setVisible(false);

        //Llenar de productos
        cboCategoria.addItem('Bebidas');
        cboCategoria.addItem('Lacteos');
        cboCategoria.addItem('Abarrotes');
        cboCategoria.addItem('Golosinas');
    }

    //Método que genera código de producto
    void generaCodigo() {
        lblCodigo.setText('' + n);
    }
```

```
    private void btnGrabarAActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            int codigo = Integer.parseInt(lblCodigo.getText());
            String descripcion = txtDescripcion.getText();
            String categoria = String.valueOf(cboCategoria.getSelectedItem());
            double precio = Double.parseDouble(txtPrecio.getText());
            int stock = Integer.parseInt(txtStock.getText());

            //Objeto de la clase Producto
            Producto objP = new Producto(código, descripcion, precio,
                                         categoria, stock);

            aProductos.agregaProducto(objP);

            //Aumenta el número de productos registrados
            n++;

            listar();
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, 'Ocurrió un error al
                        intentar agregar', 'Confirmación',
                        JOptionPane.INFORMATION_MESSAGE);
        }
        limpiarCajas();
    }

    private void btnModificarActionPerformed(java.awt.event.ActionEvent evt) {
        //Ingresar el código a buscar
        int cod = Integer.parseInt(JOptionPane.showInputDialog(null,
                                   'Ingrese código de producto:'));

        //Buscar el producto
        Producto objP = aProductos.buscarProducto(cod);

        //Mostrar lo encontrado
        if (objP != null) {
            lblCodigo.setText('' + objP.getCodigo());
            txtDescripcion.setText('' + objP.getDescripcion());
            txtPrecio.setText('' + objP.getPrecio());
            cboCategoria.setSelectedItem('' + objP.getCategoria());
            txtStock.setText('' + objP.getStock());
        }

        //Gestión de botones
        btnModificar.setVisible(false);
        btnGrabarM.setVisible(true);
    }

    private void btnGrabarMActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            //Busca el producto a modificar
            Producto pro = aProductos.buscarProducto(Integer.
                                     parseInt(lblCodigo.getText()));

            //Envia la nueva información
            pro.setDescripcion(txtDescripcion.getText());
            pro.setPrecio(Double.parseDouble(txtPrecio.getText()));
            pro.setStock(Integer.parseInt(txtStock.getText()));
            pro.setCategoria(String.valueOf(cboCategoria.getSelectedItem()));
            JOptionPane.showMessageDialog(null, 'Producto modificada
                        correctamente', 'Confirmación',
                        JOptionPane.INFORMATION_MESSAGE);
```

```
            //Muestra lo grabado
            listar();
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, 'Ocurrió un error al
                        intentar modificar', 'Confirmación',
                        JOptionPane.INFORMATION_MESSAGE);
        }

        //Gestión de botones
        btnGrabarM.setVisible(false);
        btnModificar.setVisible(true);
        limpiarCajas();
    }

    private void btnEliminarActionPerformed(java.awt.event.ActionEvent evt) {
        limpiaTabla();

        //Seleccionar el código de producto a eliminar
        int cod = Integer.parseInt(JOptionPane.showInputDialog(null,
                                    'Ingrese código de producto:'));

        //Buscar el producto en el arreglo
        Producto objP = aProductos.buscarProducto(cod);
        //Una vez encontrado se procede a eliminar
        if (objP != null) {
            aProductos.eliminaProducto(objP);
            JOptionPane.showMessageDialog(null, 'Producto eliminado
                        correctamente', 'Confirmacion',
                        JOptionPane.INFORMATION_MESSAGE);
            listar();
        }
    }

    private void btnListarActionPerformed(java.awt.event.ActionEvent evt) {
        limpiaTabla();
        limpiarCajas();
        listar();
    }

    private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
        //Seleccionar el código a buscar
        int cod = Integer.parseInt(JOptionPane.showInputDialog(null,
                                    'Ingrese código de producto:'));

        limpiaTabla();

        //Buscar el producto en el arreglo
        Producto objP = aProductos.buscarProducto(cod);

        //Mostrar el producto
        if (objP != null) {
            tProductos.setValueAt(objP.getCodigo(), 0, 0);
            tProductos.setValueAt(objP.getDescripcion(), 0, 1);
            tProductos.setValueAt(objP.getPrecio(), 0, 2);
            tProductos.setValueAt(objP.getCategoria(), 0, 3);
            tProductos.setValueAt(objP.getStock(), 0, 4);

            lblCodigo.setText('' + objP.getCodigo());
            txtDescripcion.setText('' + objP.getDescripcion());
            txtPrecio.setText('' + objP.getPrecio());
            cboCategoria.setSelectedItem('' + objP.getCategoria());
            txtStock.setText('' + objP.getStock());
        }
    }
```

```
    private void btnGenerarActionPerformed(java.awt.event.ActionEvent evt) {
        limpiarCajas();
        generaCodigo();
    }

    //Método que lista los productos almacenados en el arreglo
    void listar() {
        for (int i = 0; i < aProductos.totalProductos.determinar(); i++) {
            Producto objP = aProductos.obtener(i);
            tProductos.setValueAt(objP.getCodigo(), i, 0);
            tProductos.setValueAt(objP.getDescripcion(), i, 1);
            tProductos.setValueAt(objP.getPrecio(), i, 2);
            tProductos.setValueAt(objP.getCategoria(), i, 3);
            tProductos.setValueAt(objP.getStock(), i, 4);
        }
    }

    //Método que limpia la tabla
    void limpiaTabla() {
        for (int i = 0; i < 20; i++) {
            tProductos.setValueAt('', i, 0);
            tProductos.setValueAt('', i, 1);
            tProductos.setValueAt('', i, 2);
            tProductos.setValueAt('', i, 3);
            tProductos.setValueAt('', i, 4);
        }
    }

    //Método que limpia las cajas
    void limpiarCajas() {
        lblCodigo.setText('');
        txtDescripcion.setText('');
        txtPrecio.setText('');
        txtStock.setText('');
        cboCategoria.setSelectedIndex(-1);
    }
}
```

Archivos de texto

CAPÍTULO

11

Competencias

- Reconoce el trabajo de los archivos de texto.
- Implementa una aplicación de mantenimiento con datos registrados en un archivo de texto.

Contenido

11.1 Configuración del JDK

11.2 Librerías a utilizar para el manejo de archivos

11.3 Clases y métodos para el manejo y control de archivos de texto

- Caso desarrollado 1: Mantenimiento de estudiantes
- Caso desarrollado 2: Mantenimiento de productos

11.1 Configuración del JDK

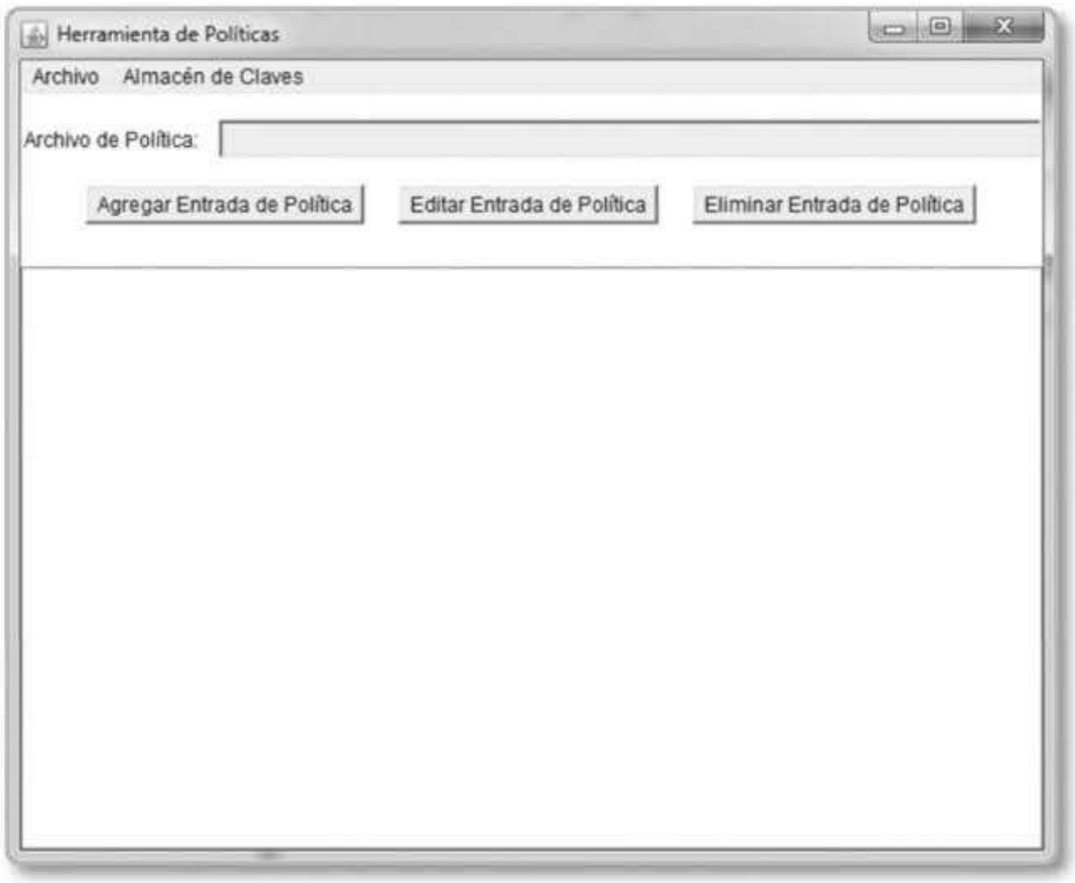
El JDK deberá estar preparado para manipular archivos de texto, es decir, para habilitar la escritura, la sobreescritura y la eliminación. En el caso de NetBeans, este tiene habilitado todos estos servicios, pero, igualmente, es necesario conocer cómo deberán ser habilitados estos servicios en otros IDE. A continuación se muestran los pasos para configurar la manipulación de archivos desde el JDK.

Pasos para configurar el JDK:

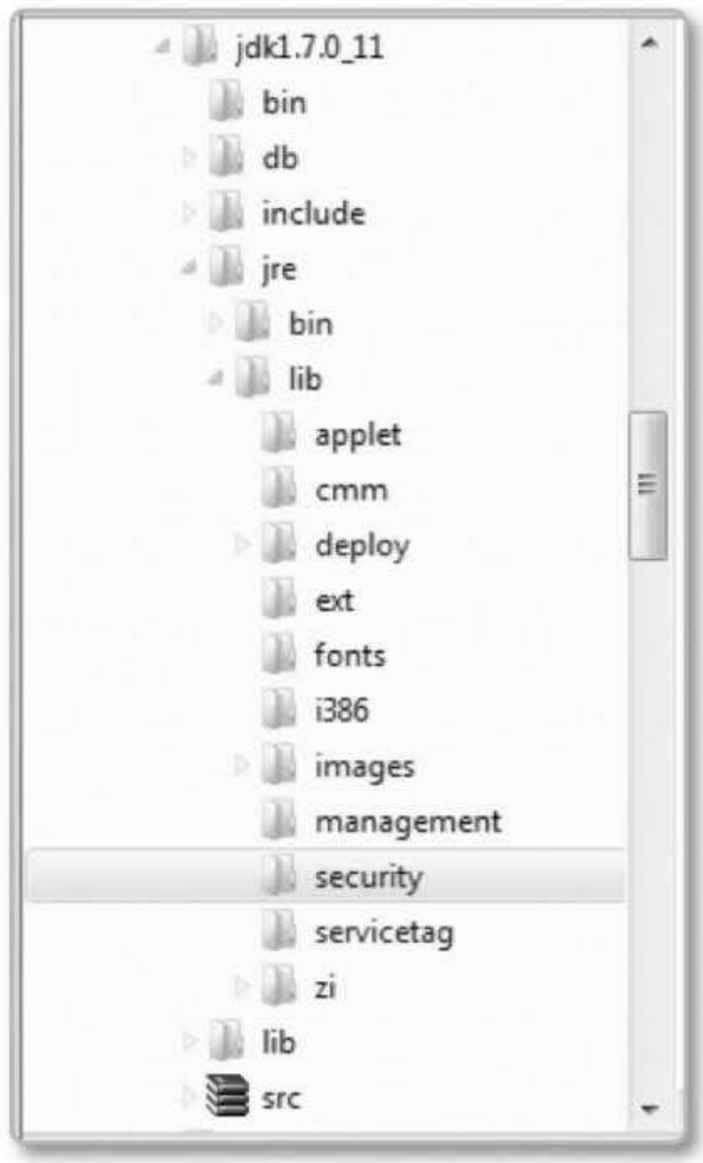
1. Diríjase a **C:\Program Files\Java\jdk1.7.0_5\bin**.

La imagen anterior muestra los archivos que contiene la carpeta **bin** del JDK de Java.

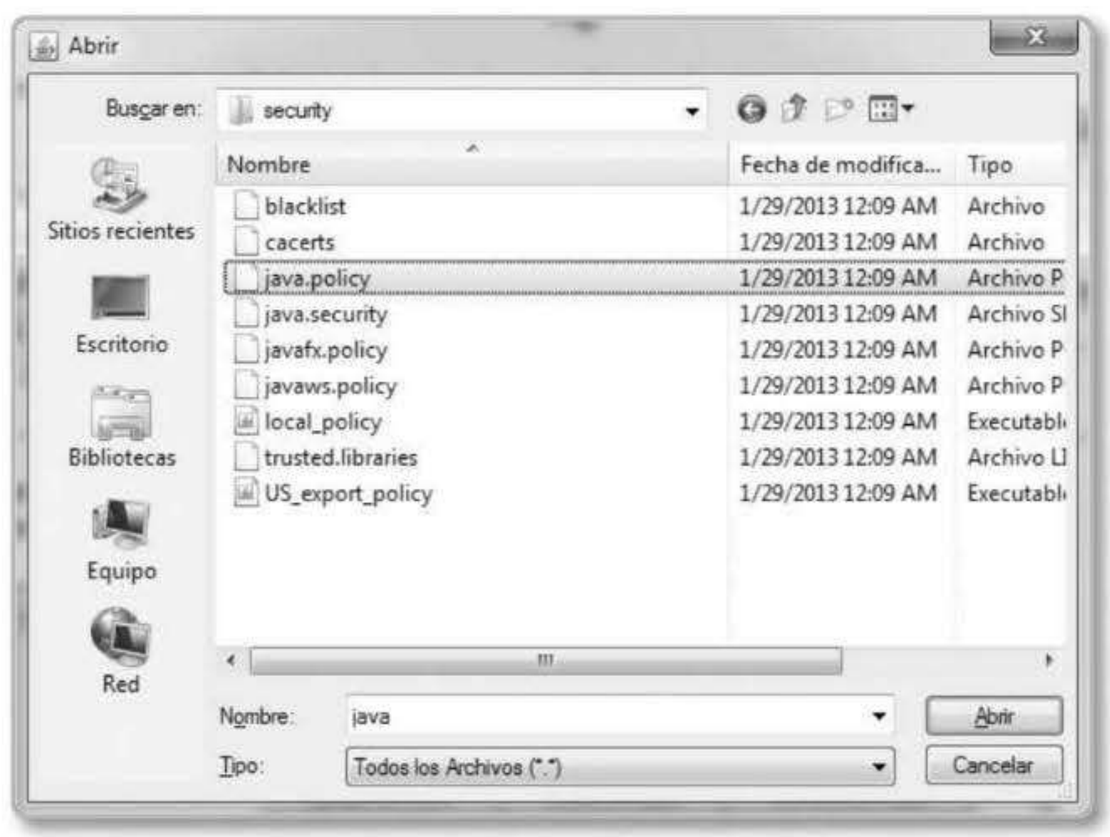
2. Seleccione y ejecute el archivo **policytool**; tenga en cuenta que, si usted se encuentra en Windows 7 u otro sistema operativo superior, tendrá que ejecutar el archivo en modo Administrador; para ello deberá hacer clic derecho sobre el archivo **policytool** y seleccionar **Ejecutar como Administrador**.



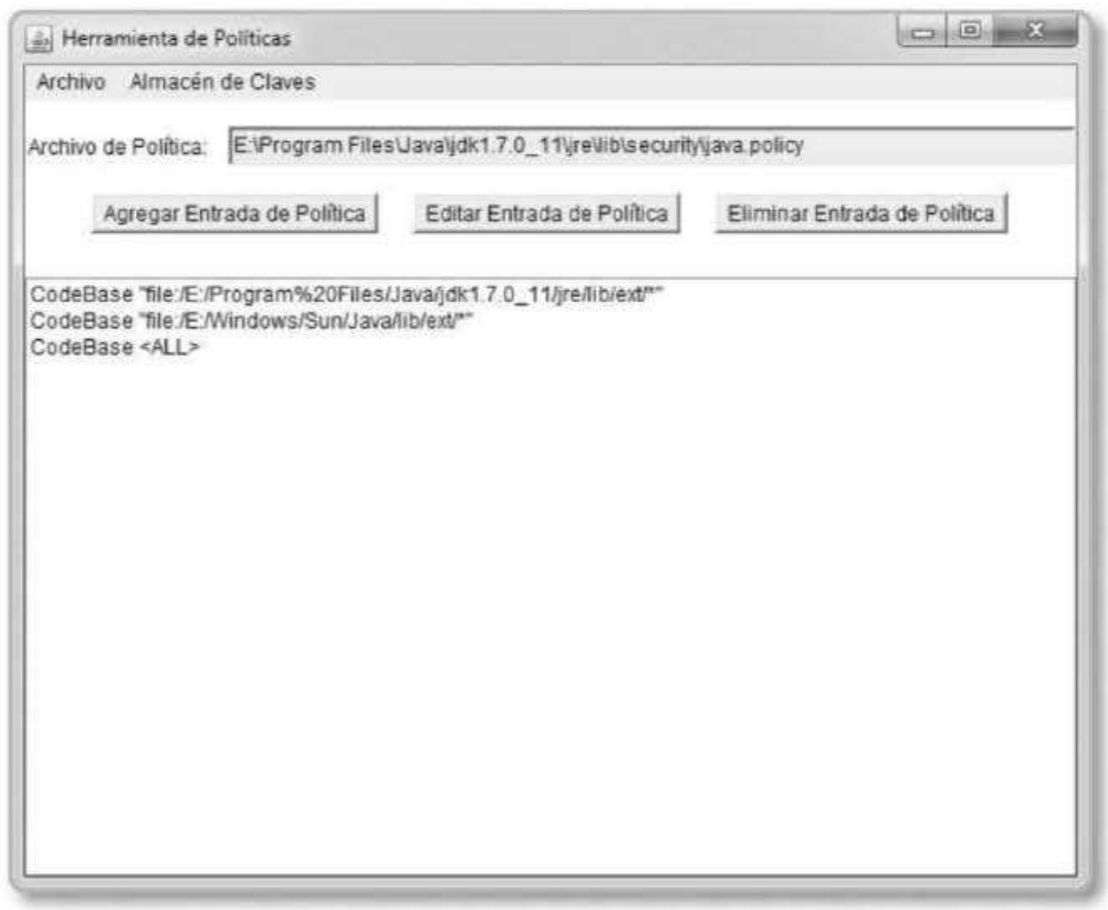
3. Seleccione **Archivo > Abrir** de la ventana **Herramienta de Políticas**. Busque y ejecute el archivo **java.policy** en la siguiente ruta:



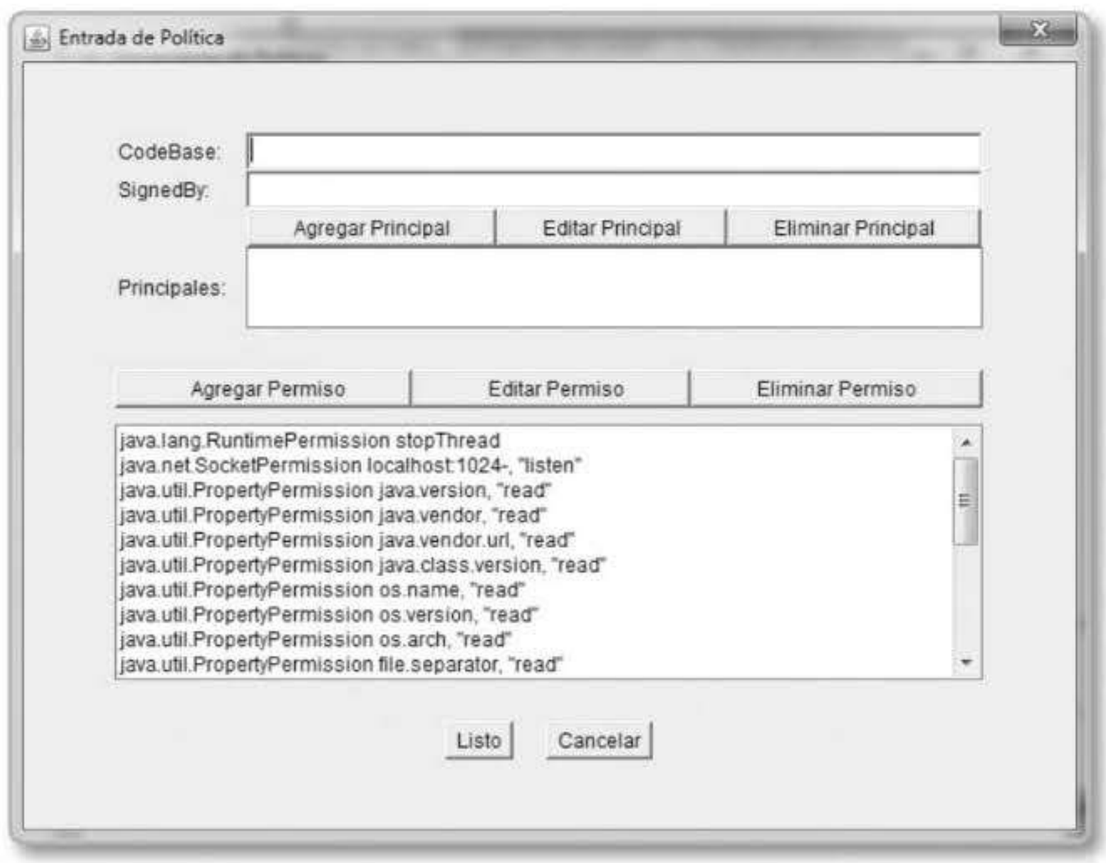
Si cuenta con una versión distinta de JDK, no debe preocuparse, puesto que la ruta que se indica es la misma en todas las versiones.



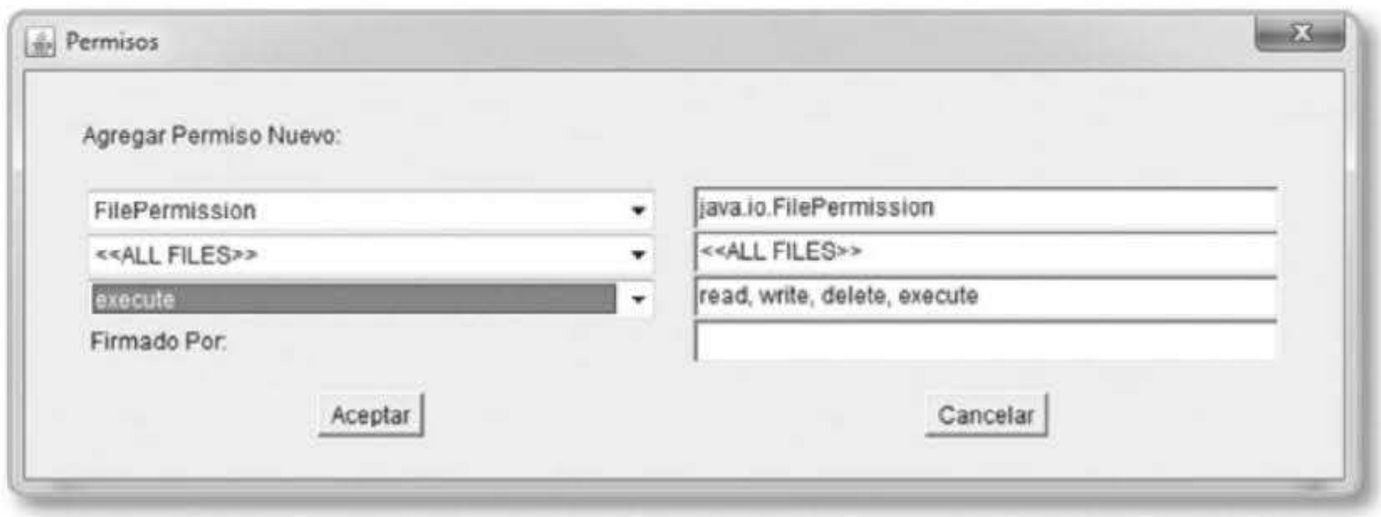
Una vez abierto el archivo **java.policy**, se muestra la siguiente ventana:



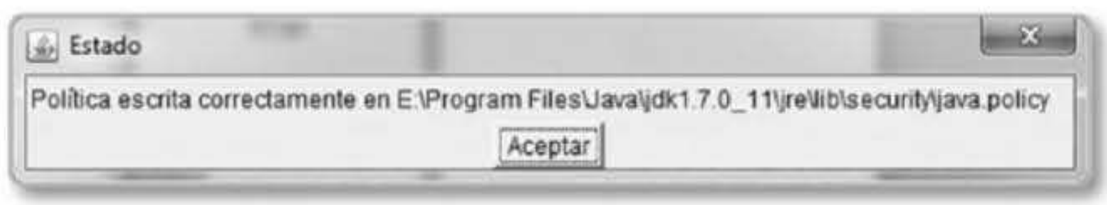
4. Seleccione **CodeBase <ALL>**, luego haga clic en **Editar entrada de Política**.
5. Seleccione **Agregar Permiso** de la ventana **Entrada de Política**.



6. Seleccione:
   a) Permiso : **FilePermission**
   b) Nombre de destino : **<<ALL FILES>>**
   c) Acciones : **read, write, delete, execute** (en este caso tendrá que seleccionar las 4 acciones)



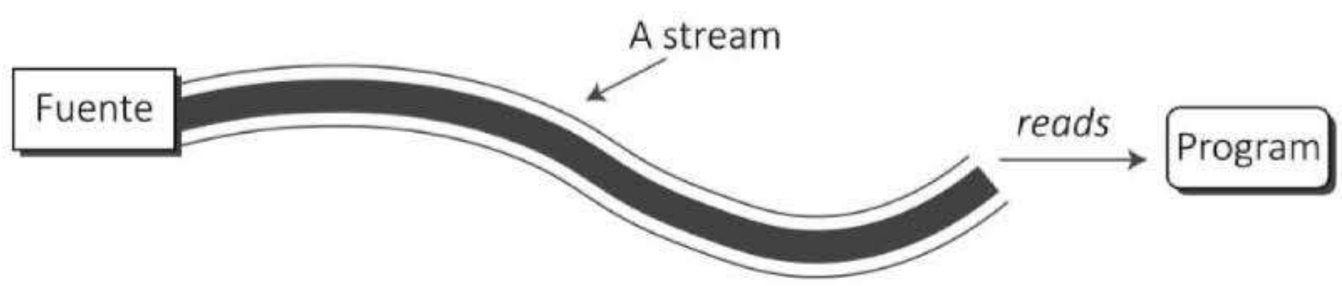
7. Seleccione **Aceptar > Listo**.
8. Seleccione **Archivo > Guardar**.
9. Para constatar que todos los cambios han sido registrados de forma correcta, deberá aparecer el siguiente mensaje:



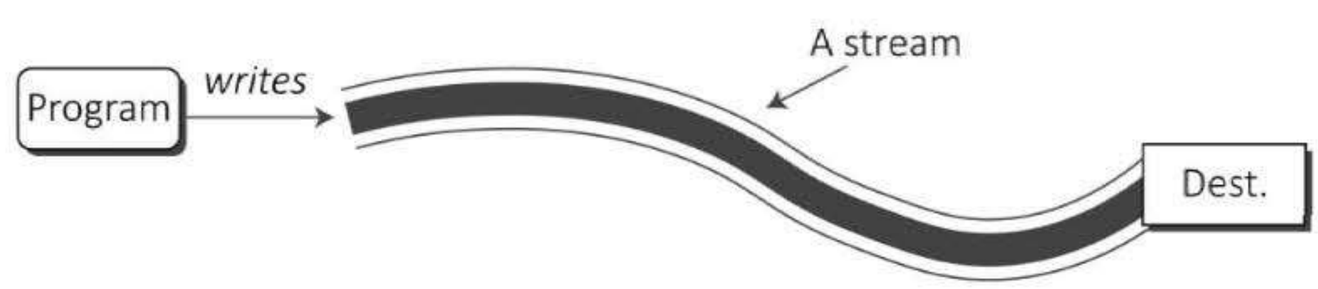
11.2 Librerías a utilizar para el manejo de archivos

Frecuentemente los programas necesitan traer información desde una fuente externa o enviar información a una fuente externa. La información puede estar en cualquier parte: en un fichero, en un disco, en algún lugar de la red, en la memoria o en otro programa. También puede ser de cualquier tipo: objetos, caracteres, imágenes o sonidos.

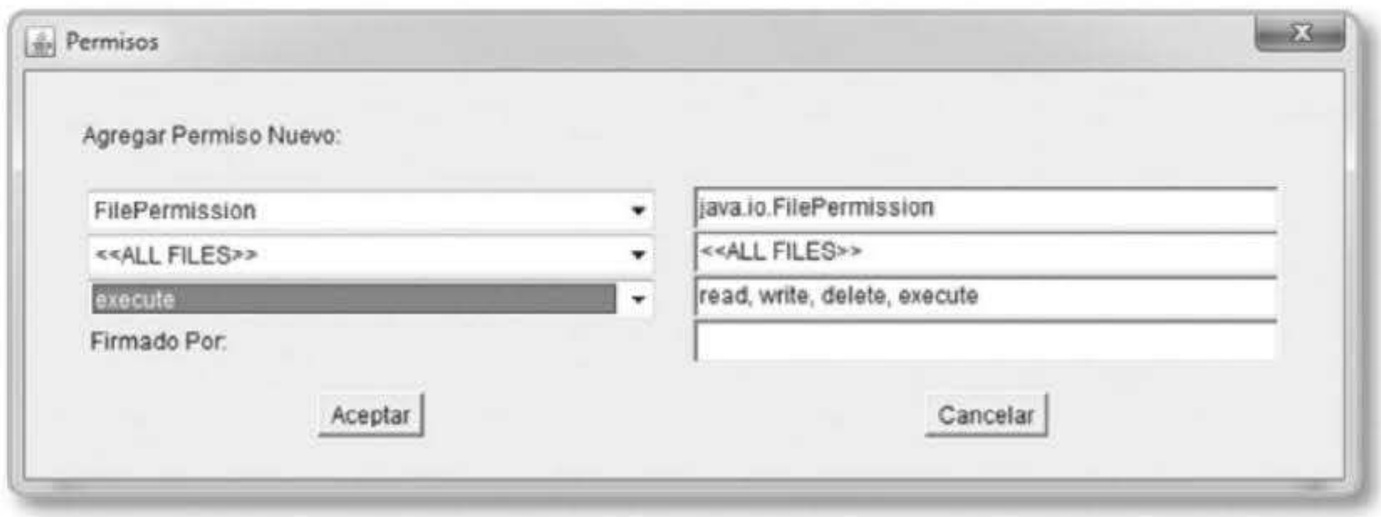
Para traer la información, un programa abre un *stream* sobre una fuente de información (un fichero, memoria, un socket) y lee la información serialmente de esta forma:



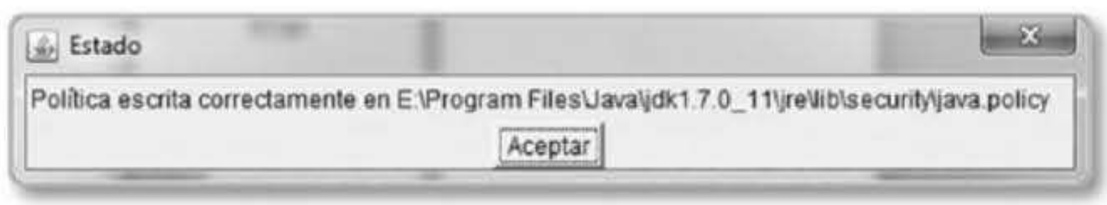
Similarmente, un programa puede enviar información a un destino externo abriendo un *stream* sobre un destino y escribiendo la información serialmente de esta forma:



6. Seleccione:
   a) Permiso : **FilePermission**
   b) Nombre de destino : **<<ALL FILES>>**
   c) Acciones : **read, write, delete, execute** (en este caso tendrá que seleccionar las 4 acciones)



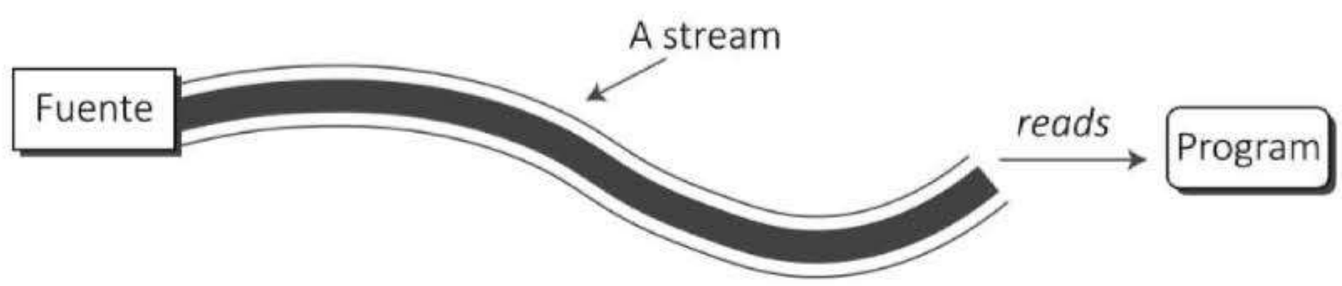
7. Seleccione **Aceptar > Listo**.
8. Seleccione **Archivo > Guardar**.
9. Para constatar que todos los cambios han sido registrados de forma correcta, deberá aparecer el siguiente mensaje:



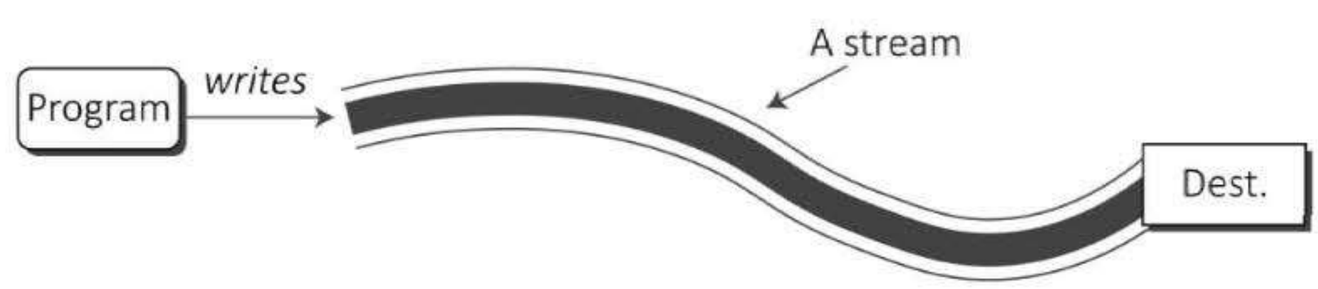
11.2 Librerías a utilizar para el manejo de archivos

Frecuentemente los programas necesitan traer información desde una fuente externa o enviar información a una fuente externa. La información puede estar en cualquier parte: en un fichero, en un disco, en algún lugar de la red, en la memoria o en otro programa. También puede ser de cualquier tipo: objetos, caracteres, imágenes o sonidos.

Para traer la información, un programa abre un *stream* sobre una fuente de información (un fichero, memoria, un socket) y lee la información serialmente de esta forma:



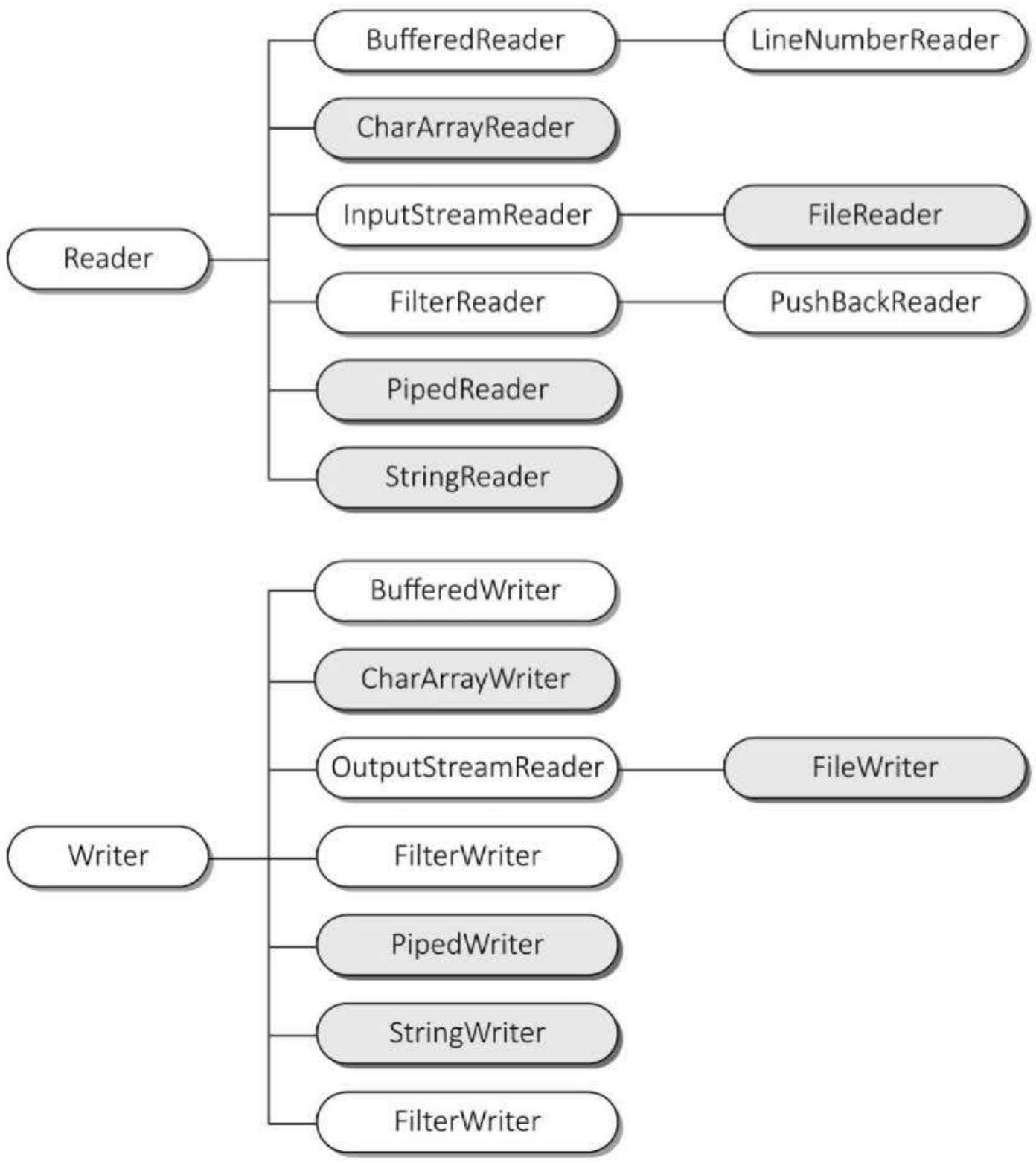
Similarmente, un programa puede enviar información a un destino externo abriendo un *stream* sobre un destino y escribiendo la información serialmente de esta forma:



El paquete *java.io* contiene una colección de clases *stream* que soportan estos algoritmos para leer y escribir. Estas clases están divididas en dos árboles, basándose en los tipos de datos (caracteres o bytes) sobre los que opera.

Sin embargo, algunas veces es más conveniente agrupar las clases basándose en su propósito en vez que en los tipos de datos que lee o escribe. Así, podemos agrupar los *streams* dependiendo de si leen o escriben lados en las "profundidades" o procesan la información que está siendo leída o escrita.

11.3 Clases y métodos para el manejo y control de archivos de texto



BufferedReader	La clase **BufferedReader** deriva de la clase **Reader**. Esta clase añade un buffer para realizar una lectura eficiente de caracteres. Dispone del método **readLine** que permite leer una línea de texto y devolverla como **String**.
Declaración de una variable tipo bufferedReader	bufferedReader br;
Invocación al método constructor	`br=new BufferedReader(new FileReader(archivotexto));`

FileReader	Esta clase tiene métodos que nos permiten leer caracteres. Sin embargo, suele ser habitual querer las líneas completas, bien porque nos interesa la línea completa, bien para poder analizarla luego y extraer campos de ella. **FileReader** no contiene métodos que nos permitan leer líneas completas, pero sí **BufferedReader**.

Crear un BufferedReader a partir de un FileReader	`File archivo = new File ('C:\\archivo.txt');`
	`FileReader fr = new FileReader (archivo);`
	`BufferedReader br = new BufferedReader(fr);`

Observaciones	■ Una vez abierto el archivo, br apunta a la primera cadena de bits. ■ El método **readLine()** asociado a br captura una cadena de bits y salta a la siguiente línea de bits. Dicha cadena es convertida seguidamente al tipo de datos requerido por la aplicación. ■ Cuando no existe más cadena que mostrar, br apunta a null. ■ El método **close()** cierra el acceso al archivo.
Declaración final	BuffereReader br=new BufferedReader(new FileReader(archivo))

PrintWriter	Es la clase que permite escribir los datos de la memoria hacia un archivo de texto (output). Lo primero que tenemos que hacer es crear una variable de tipo **PrintWriter**.
Declaración de una variable tipo PrintWriter	PrintWriter pw;
Invocación al Método constructor	`Pw = new PrintWriter(new FileWriter(archivo));`

FileWriter	Esta clase se encarga de abrir el archivo en modo de escritura. Es decir, si el archivo contiene información, esta se pierde. Si el archivo no existe, lo crea.
Observaciones	■ Una vez abierto el archivo, pw apunta al inicio. ■ El método **println(data)** asociado a pw graba como cadena de bits la data indicada y genera un salto de línea en el archivo. ■ El método **close()** cierra el acceso al archivo.
Declaración	`PrintWriter pw = new PrintWriter(new FileWriter(archivo));`

CASO DESARROLLADO 1: **Mantenimiento de alumnos**

Implemente una aplicación que permita realizar el mantenimiento de los registros de un estudiante, en la cual se ingresará un código, nombres, ciclo y pensión.

Consideraciones:

- Debe crear tres clases:
  - Clase **Estudiante**, que permitirá definir los atributos del estudiante, como su código, nombres, ciclo de estudio y el monto de la pensión.
  - Clase **ArregloEstudiantes**, que permitirá almacenar a los registros del estudiante en un vector de objetos.
  - Clase **frmMantenimiento**, que permitirá interactuar con el usuario para el control de los registros del estudiante.
- Debe crear siete botones:
  - Botón **Ingresar.** Tiene por objetivo habilitar las cajas de texto para un nuevo registro de estudiante; además, genera un código al estudiante de forma automática.
  - Botón **Consultar.** Tiene por objetivo solicitar un código de estudiante y mostrar los datos del estudiante para buscarlo en el vector. En caso no lo encuentre, muestra un mensaje de error.
  - Botón **Modificar.** Tiene por objetivo solicitar un código de estudiante para buscarlo en el vector y mostrar sus datos en los controles **JTextField**. Al modificar se habilitará el botón **grabar**.
  - Botón **Eliminar.** Tiene por objetivo solicitar un código de estudiante para eliminarlo del vector de estudiantes.
  - Botón **Listar.** Permite listar los registros contenido en el vector de estudiantes.
  - Botón **GrabarI.** Permite grabar los valores del estudiante que se ingresan por primera vez.
  - Botón **GrabarM.** Permite grabar las modificaciones realizadas sobre un registro de estudiante.
- Use la siguiente interfaz gráfica:

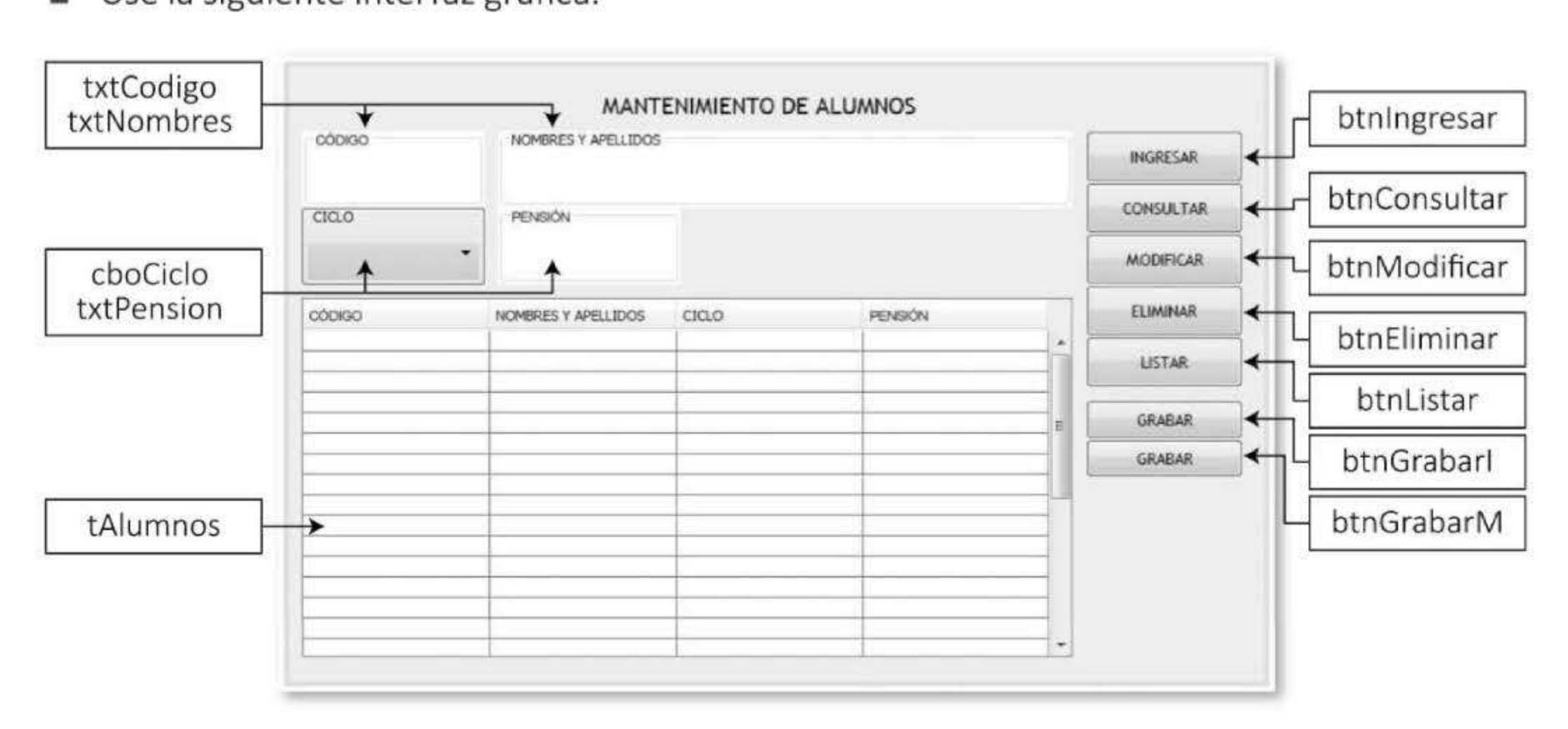
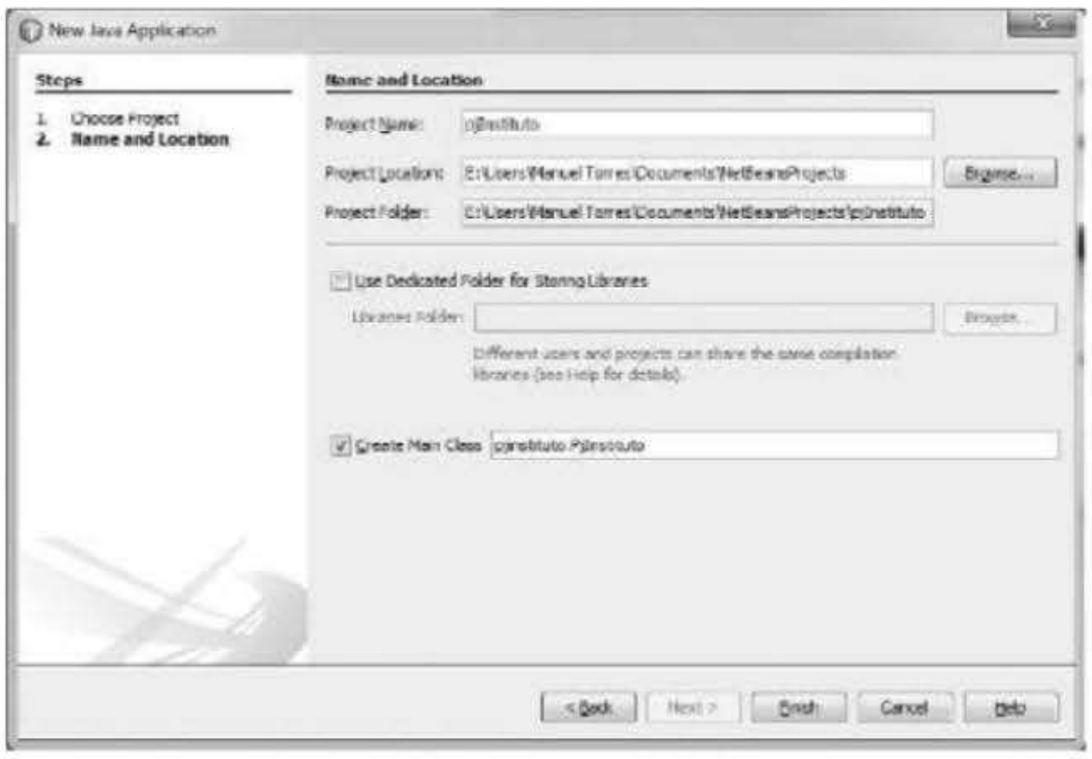


Figura 11.1 Formulario de mantenimiento de alumnos.

Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjInstituto**.



2. Agregue el paquete **pFormularios** al proyecto **pjInstituto**.
3. Agregue el paquete **pClases** al proyecto **pjInstituto**.
4. Agregue la clase **Estudiante** al paquete **pClases**.
5. Agregue la clase **ArregloEstudiantes** al paquete **pClases**
6. Agregue la clase **frmMantenimiento** al paquete **pFormularios.**
7. Asigne un nombre a cada uno de los controles como se muestra en la figura 11.1; para ello, haga clic derecho sobre el objeto y seleccione **Change Variable Name...**
8. Después de asignar nombres a los objetos, asegúrese de que los controles sean los correctos; para ello, visualice el panel **Navigator**, el cual debe mostrarse como la figura 11.2.

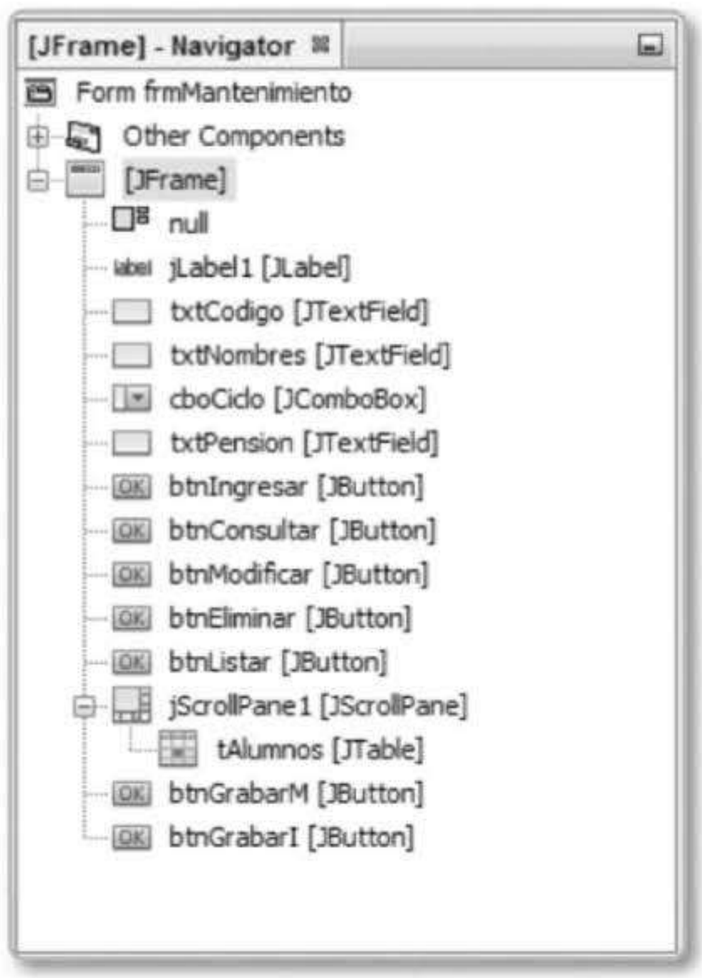


Figura 11.2 Listado de controles desde la ventana Navigator.

9. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.
10. Seleccione el formulario y en la propiedad **Form Size Police** modifique el valor encontrado por **Generate Resize Code**.

El siguiente script muestra el contenido de la clase **Estudiante**:

```
package pClases;

public class Estudiante {
    //Atributos privados de la clase Estudiante
    private int codigo,ciclo;
    private String nombre;
    private double pension;

    //Método constructor
    public Estudiante(int codigo,String nombre, int ciclo,double pensión){
        this.codigo=código;
        this.nombre=nombre;
        this.ciclo=ciclo;
        this.pension=pensión;
    }

    //Métodos GET y SET de los atributos de la clase Estudiante
    public int getCodigo(){
        return codigo;
    }
    public String getNombre(){
        return nombre;
    }
    public int getCiclo(){
        return ciclo;
    }
    public double getPension(){
        return pension;
    }
    public void setCodigo(int codigo){
        this.codigo=codigo;
    }
    public void setNombre(String nombre){
        this.nombre=nombre;
    }
    public void setCiclo(int ciclo){
        this.ciclo=ciclo;
    }
    public void setPension(double pension){
        this.pension=pensión;
    }
}
```

El siguiente script muestra el contenido de la clase **ArregloEstudiantes**:

```
package pClases;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;
```

```
public class ArregloEstudiantes {
    private ArrayList <Estudiante> aEst;

    /*
    Método constructor que crea el objeto aEst de tipo ArrayList
    y carga el contenido del archivo estudiantes.txt
    */
    public ArregloEstudiantes(){
        aEst=new ArrayList<Estudiante>();
        cargar();
    }

    /*Método que adiciona un objeto de tipo Estudiante al ArrayList*/
    public void adicionar(Estudiante e){
        aEst.add(e);
    }

    /*Método que devuelve los elementos contenidos en el arreglo
    según una posición especificada
    */
    public Estudiante obtener(int pos){
        return aEst.get(pos);
    }

    /*Método que busca un estudiante desde el arreglo y devuelve
    el objeto de tipo estudiante, si no lo encuentra devolverá NULL*/
    public Estudiante buscar(int cod){
        for(int i=0;i<aEst.size();i++){
            if (cod==aEst.get(i).getCodigo())
                return aEst.get(i);
        }
        return null;
    }

    /*Método que devuelve el total de estudiantes registrados*/
    public int getTamaño(){
        return aEst.size();
    }

    /*Método que permite eliminar un estudiante del arreglo mediante
    el objeto de tipo Estudiante*/
    public void eliminar(Estudiante e){
        aEst.remove(e);
    }

    /*Método que permite abrir el archivo Estudiante.txt
    y obtener los valores contenidos almacenándolos en variables
    locales*/
    public void cargar(){
        try{
            File archivo = new File('Estudiantes.txt');
            if (archivo.exists()){
                BufferedReader br=new BufferedReader(
                new FileReader('Estudiantes.txt'));
                String linea;
                while((linea=br.readLine())!=null){
                    //Definición del separador de valores en Estudiantes.txt
                    StringTokenizer st=new StringTokenizer(linea,',');
                    int cod=Integer.parseInt(st.nextToken().trim());
                    String nom=st.nextToken().trim();
                    int cic=Integer.parseInt(st.nextToken().trim());
                    double pen=Double.parseDouble(st.nextToken().trim());
```

```
                    //Crear un nuevo registro de estudiante obtenido del
                    //archivo Estudiantes.txt
                    Estudiante x=new Estudiante(cod,nom,cic,pen);
                    adicionar(x);
                }
                br.close();
            } else
                JOptionPane.showMessageDialog(null, 'El archivo de texto
                                                    no existe');
        } catch(Exception x){
            JOptionPane.showMessageDialog(null, 'Se produjo un Error '+x);
            }
        }

        /*Método que permite grabar los nuevos valores de un estudiante*/
        public void grabar(){
            try{
                PrintWriter pw=new PrintWriter(
                new FileWriter('Estudiantes.txt'));
                for(int i=0;i<getTamaño();i++){
                    pw.println(obtener(i).getCodigo()+',' + obtener(i).getNombre()
                    + ',' + obtener(i).getCiclo() + ',' + obtener(i).getPension());
                }
                pw.close();
                JOptionPane.showMessageDialog(null,'Operación Exitosa');
            }catch(Exception ex){
                JOptionPane.showMessageDialog(null,'Grabación Exitosa', 'Aviso',
                                              JOptionPane.ERROR_MESSAGE);
            }
        }
    }
```

El siguiente script muestra las librerías y la declaración de las variables globales de la clase **frmMantenimiento**:

```
package pFormularios;

import pClases.ArregloEstudiantes;
import javax.swing.JOptionPane;
import javax.swing.table.TableColumn;
import pClases.Estudiante;

public class frmMantenimiento extends javax.swing.JFrame {
    //Objeto de la clase ArregloEstudiantes
    ArregloEstudiantes a=new ArregloEstudiantes();

    public frmMantenimiento() {
        initComponents();
        //Define el ancho de las columnas de la tabla
        defineAnchoTabla();

        //Llenar el control cboCiclo
        llenaCombo();

        //Permite mostrar los registros contenidos en el archivo
        //Estudiantes.txt
        listar();
        btnGrabarI.setVisible(false);
        btnGrabarM.setVisible(false);
        habilitaCajas(false);
    }
}
```

El siguiente script muestra los métodos de la clase **frmMantenimiento**:

```
//Método que define los anchos de la tabla tAlumnos
void defineAnchoTabla(){
    TableColumn columna;
    columna=tAlumnos.getColumnModel().getColumn(0);
    columna.setPreferredWidth(50);
    columna=tAlumnos.getColumnModel().getColumn(1);
    columna.setPreferredWidth(150);
    columna=tAlumnos.getColumnModel().getColumn(2);
    columna.setPreferredWidth(30);
    columna=tAlumnos.getColumnModel().getColumn(3);
    columna.setPreferredWidth(40);
    tAlumnos.getTableHeader().setReorderingAllowed(false);
    tAlumnos.getTableHeader().setResizingAllowed(false);
}

//Método que llena los ciclos en el control cboCiclo
void llenaCombo(){
    cboCiclo.addItem('1');
    cboCiclo.addItem('2');
    cboCiclo.addItem('3');
    cboCiclo.addItem('4');
    cboCiclo.addItem('5');
    cboCiclo.addItem('6');
}

//Método que permite habilitar o inhabilitar el acceso
//a los controles JTextField
void habilitaCajas(boolean opcion){
    txtCodigo.setEditable(opcion);
    txtNombres.setEditable(opcion);
    txtPension.setEditable(opcion);
    cboCiclo.setEnabled(opcion);
}

//Método que permite limpiar los controles JTextField
void limpiaCajas(){
    txtCodigo.setText('');
    txtNombres.setText('');
    txtPension.setText('');
    cboCiclo.setSelectedIndex(0);
}

//Método que permite limpiar la tabla tAlumnos
void limpiaMatriz(){
    for(int i=0;i<10;i++){
        tAlumnos.setValueAt('', i, 0);
        tAlumnos.setValueAt('', i, 1);
        tAlumnos.setValueAt('', i, 2);
        tAlumnos.setValueAt('', i, 3);
    }
}

//Método que permite autogenerar un código del estudiante
//con un máximo de 4 cifras por número autogenerado.
public int autogeneraCodigo(){
    int cod=(int)((9999-1000+1)*Math.random()+1000);
    return cod;
}
```

```
//Método que permite listar los registros de los estudiantes
//al control tAlumnos
void listar(){
    if (a.getTamaño()>0){
        for(int i=0;i<a.getTamaño();i++){
            Estudiante e=a.obtener(i);
            tAlumnos.setValueAt(e.getCodigo(), i, 0);
            tAlumnos.setValueAt(e.getNombre(), i, 1);
            tAlumnos.setValueAt(e.getCiclo(), i, 2);
            tAlumnos.setValueAt(e.getPension(), i, 3);
        }
    }
}

//Métodos Get
public int getCodigo(){
    return Integer.parseInt(txtCodigo.getText());
}

public String getNombre(){
    return txtNombres.getText();
}

public double getPension(){
    return Double.parseDouble(txtPension.getText());
}

public int getCiclo(){
    return cboCiclo.getSelectedIndex()+1;
}
```

El siguiente script muestra las instrucciones del botón **btnConsultar**:

```
private void btnConsultarActionPerformed(….)
{
    try{
        limpiaCajas();
        limpiaMatriz();

        int codigo=Integer.parseInt(JOptionPane.showInputDialog(this,
                                    'Ingrese código a Buscar:'));

        Estudiante e=a.buscar(codigo);
        if (e!=null){
            tAlumnos.setValueAt(e.getCodigo(), 0, 0);
            tAlumnos.setValueAt(e.getNombre(), 0, 1);
            tAlumnos.setValueAt(e.getCiclo(), 0, 2);
            tAlumnos.setValueAt(e.getPension(), 0, 3);
        } else
            JOptionPane.showMessageDialog(this, 'Código de estudiante NO
                                                Existe');
    } catch(Exception ex){
        JOptionPane.showMessageDialog(this,'Error de Datos');
    }
}
```

El siguiente script muestra las instrucciones del botón **btnIngresar**:

```
private void btnIngresarActionPerformed(java.awt.event.ActionEvent evt) {
    int código=autogeneraCodigo();
    txtCodigo.setText(''+código);
    habilitaCajas(true);
    txtCodigo.setEditable(false);
    txtNombres.requestFocus();
    btnGrabarI.setVisible(true);
    btnIngresar.setVisible(false);
}
```

El siguiente script muestra las instrucciones del botón **btnModificar**:

```
private void btnModificarActionPerformed(…)
{
    int codigo=Integer.parseInt(JOptionPane.showInputDialog(this,
                                'Ingrese código de Estudiante a
                                Modificar:'));
    habilitaCajas(true);
    limpiaMatriz();
    txtCodigo.setEditable(false);

    Estudiante e=a.buscar(código);
    if (e != null){
        txtCodigo.setText('' + e.getCodigo());
        txtNombres.setText(e.getNombre());
        cboCiclo.setSelectedIndex(e.getCiclo()-1);
        txtPension.setText('' + e.getPension());

        txtNombres.requestFocus();
        btnGrabarM.setVisible(true);
        btnModificar.setVisible(false);
    } else
        JOptionPane.showMessageDialog(this, 'Código de estudiante NO
                                        Existe');
}
```

El siguiente script muestra las instrucciones del botón **btnEliminar**:

```
private void btnEliminarActionPerformed(java.awt.event.ActionEvent evt) {
    try{
        limpiaCajas();
        limpiaMatriz();
        int codigo=Integer.parseInt(JOptionPane.showInputDialog(this,
                                    'Ingrese código a Eliminar:'));
        Estudiante e=a.buscar(codigo);
        if (e!=null){
            a.eliminar(e);
            a.grabar();
            listar();
        }
    }catch(Exception ex){
        JOptionPane.showMessageDialog(this, 'Código de estudiante NO Existe');
    }
}
```

El siguiente script muestra las instrucciones del botón **btnListar**:

```
private void btnListarActionPerformed(java.awt.event.ActionEvent evt) {
    listar();
}
```

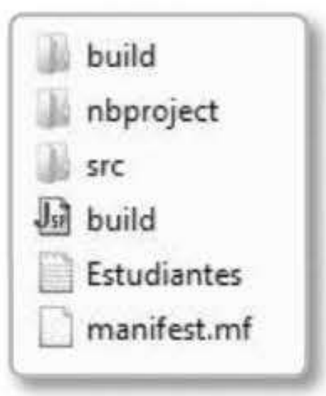
El siguiente script muestra las instrucciones del botón **btnGrabarM**:

```
private void btnGrabarMActionPerformed(java.awt.event.ActionEvent evt) {
    Estudiante e=a.buscar(getCodigo());
    if (e != null){
        e.setNombre(getNombre());
        e.setCiclo(getCiclo());
        e.setPension(getPension());

        a.grabar();
        listar();
    }
    limpiaCajas();
    habilitaCajas(false);

    btnModificar.setVisible(true);
    btnGrabarM.setVisible(false);
}
```

El siguiente script muestra las instrucciones del botón **btnGrabarI**:

```
private void btnGrabarIActionPerformed(java.awt.event.ActionEvent evt) {
    Estudiante e=a.buscar(getCodigo());
    if (e == null){
        e=new Estudiante(getCodigo(), getNombre(), getCiclo(),
                         getPension());

        a.adicionar(e);
        a.grabar();
        listar();
    }
    limpiaCajas();
    habilitaCajas(false);

    btnIngresar.setVisible(true);
    btnGrabarI.setVisible(false);
}
```

Nota. El archivo de texto **Estudiantes.txt** debe encontrarse dentro de la carpeta del proyecto, como lo muestra la siguiente imagen:



CASO DESARROLLADO 2: **Mantenimiento de productos**

Implemente una aplicación que permita realizar el mantenimiento de los registros de productos en una tienda, en la cual se ingresará un código, descripción, categoría, precio y stock de productos.

Consideraciones:

Deberá crear dos paquetes con las siguientes distribuciones de clases:

- **Paquete pClases**
  - **Clase Producto.** Permitirá definir los atributos del producto, como su código, descripción, categoría, stock y precio.
  - **Clase ArregloProductos.** Permitirá almacenar los registros de los productos en un vector de objetos; asimismo, contiene métodos que permitirán grabar y obtener información a partir de un archivo llamado **productos.txt**.
- **Paquete pFormularios**
  - **Formulario frmListadoProductos.** Permitirá listar los productos registrados en el archivo **productos.txt**.
  - **Formulario frmNuevoProducto.** Permitirá registrar los datos de un nuevo producto en el archivo **productos.txt**.
  - **Formulario frmBuscaProducto.** Permitirá buscar un determinado producto obtenido desde el archivo **productos.txt**.
  - **Formulario frmActualizaProducto.** Inicialmente buscará un producto desde el archivo de texto y una vez encontrado se podrá modificar sus datos.
  - **Formulario frmEliminaProducto.** Inicialmente buscará un determinado producto por su código y una vez encontrado se podrá eliminar del archivo de texto.
  - **Formulario frmMenu.** Permitirá mostrar todos los formularios.

Use la siguiente interfaz gráfica:

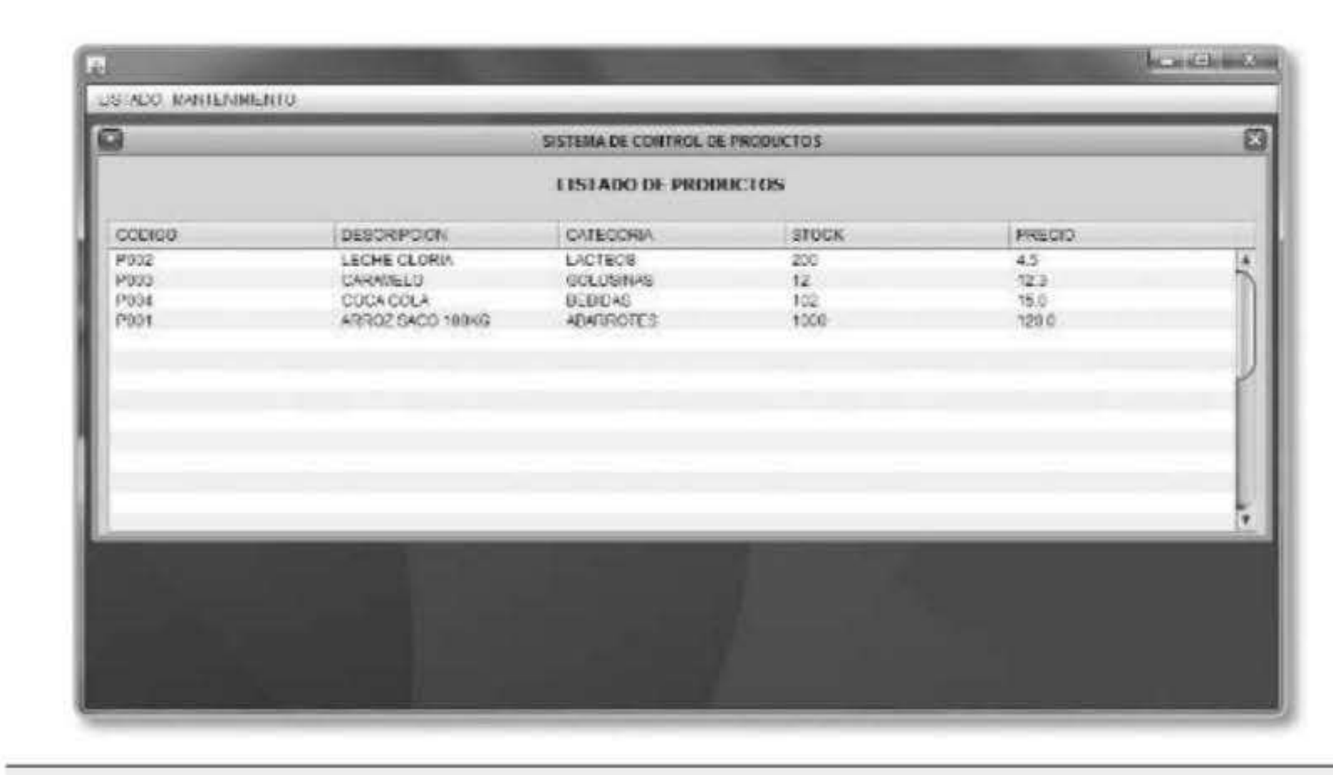


Figura 11.3 Formulario de listado de productos.

Figura 11.4 Formulario de registro del nuevo producto.

Figura 11.5 Formulario de búsqueda del producto.

Figura 11.6 Formulario de modificación de datos del producto.

Figura 11.7 Formulario de eliminación de un producto.

Solución:

1. Cree un nuevo proyecto en NetBeans llamado **pjMantenimientoProducto**.
2. Agregue el paquete **pFormularios** al proyecto **pjMantenimientoProducto**.
3. Agregue el paquete **pClases** al proyecto **pjMantenimientoProducto**.
4. Agregue la clase **Producto** al paquete **pClases**.
5. Agregue la clase **ArregloProductos** al paquete **pClases**.
6. Agregue todos los formularios al paquete **pFormularios**.
7. Asigne un nombre a cada uno de los controles que se mostrarán en cada formulario; para ello, haga clic derecho sobre el objeto y seleccione **Change Variable Name...**
8. Al finalizar el proyecto, asegúrese de que el explorador de proyectos se muestre de la siguiente manera:

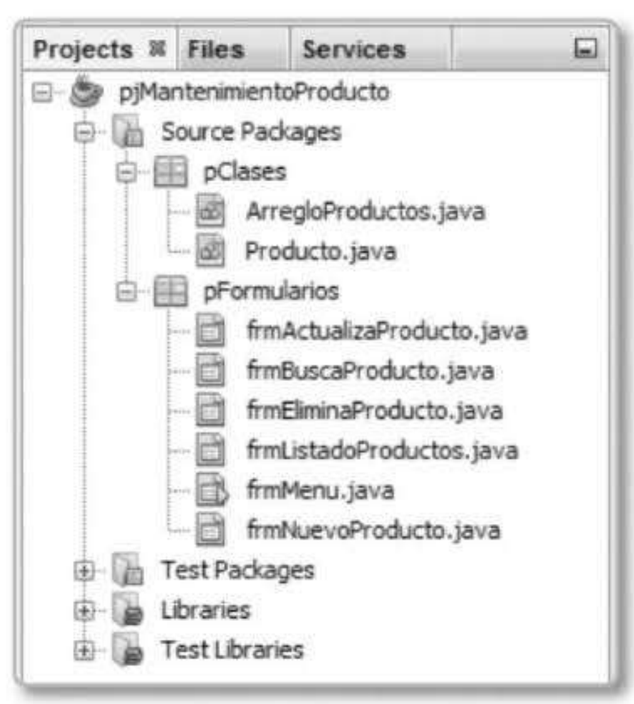


Figura 11.8 Listado de controles desde la ventana Navigator.

9. Asigne **Null Layout** al **setLayout** del **Frame** haciendo clic derecho sobre el **Frame**.
10. Seleccione el formulario y en la propiedad **Form Size Police** modifique el valor encontrado por **Generate Resize Code**.

El siguiente script muestra el contenido de la clase **Producto**:

```
package pClases;

public class Producto {
    private String codigo;
    private String descripcion;
    private String categoria;
    private int stock;
    private double precio;

    //Constructor
    public Producto(String codigo, String descripcion, String categoria,
                    int stock, double precio) {
        this.codigo - código;
        this.descripcion - descripcion;
        this.categoria - categoria;
        this.stock - stock;
        this.precio - precio;
    }

    //Método de get y set
    public String getCodigo() {
        return codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo - codigo;
    }

    public String getDescripcion() {
        return descripcion;
    }

    public void setDescripcion(String descripcion) {
        this.descripcion - descripcion;
    }

    public String getCategoria() {
        return categoria;
    }

    public void setCategoria(String categoria) {
        this.categoria - categoria;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock - stock;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio - precio;
    }
}
```

El siguiente script muestra el contenido de la clase **ArregloProductos**:

```
package pClases;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;

public class ArregloProductos {
    //Definir un ArrayList
    private ArrayList<Producto> aProductos;

    //Método constructor
    public ArregloProductos() {
        aProductos = new ArrayList<Producto>();
        cargar();
    }
    //Obtiene información de los productos ya grabados
    void cargar() {
        try {
            File archivo = new File('productos.txt');
            if (archivo.exists()) {
                //Leyendo el contenido del archivo
                BufferedReader br = new BufferedReader(
                                        new FileReader('productos.txt'));
                String línea;

                //Hay valores que sacar del archivo
                while ((linea = br.readLine()) != null) {
                    StringTokenizer st = new StringTokenizer(linea, ',');

                    String codigo = st.nextToken().trim();
                    String descripcion = st.nextToken().trim();
                    String categoria = st.nextToken().trim();
                    int stock = Integer.parseInt(st.nextToken().trim());
                    double precio=Double.parseDouble(st.nextToken().trim());

                    Producto p = new Producto(código, descripcion,
                                    categoria, stock, precio);
                    adicionar(p);
                }
                br.close();
            }
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, 'No hay nada que mostrar');
        }
    }

    //Método que permite GRABAR en un registro de productos
    public void grabar() {
        try {
            PrintWriter pw=new PrintWriter(new FileWriter('productos.txt'));
            for (int i = 0; i < getTamaño(); i++) {
                pw.println(obtener(i).getCodigo() + ',' +
                obtener(i).getDescripcion() + ',' + obtener(i).getCategoria()
                + ',' + obtener(i).getStock() + ',' + obtener(i).getPrecio());
            }
            pw.close();
            JOptionPane.showMessageDialog(null, 'Grabación correcta');
```

El siguiente script muestra el contenido de la clase **ArregloProductos**:

```
package pClases;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;

public class ArregloProductos {
    //Definir un ArrayList
    private ArrayList<Producto> aProductos;

    //Método constructor
    public ArregloProductos() {
        aProductos = new ArrayList<Producto>();
        cargar();
    }
    //Obtiene información de los productos ya grabados
    void cargar() {
        try {
            File archivo = new File('productos.txt');
            if (archivo.exists()) {
                //Leyendo el contenido del archivo
                BufferedReader br = new BufferedReader(
                                        new FileReader('productos.txt'));
                String línea;

                //Hay valores que sacar del archivo
                while ((linea = br.readLine()) != null) {
                    StringTokenizer st = new StringTokenizer(linea, ',');

                    String codigo = st.nextToken().trim();
                    String descripcion = st.nextToken().trim();
                    String categoria = st.nextToken().trim();
                    int stock = Integer.parseInt(st.nextToken().trim());
                    double precio=Double.parseDouble(st.nextToken().trim());

                    Producto p = new Producto(código, descripcion,
                                    categoria, stock, precio);
                    adicionar(p);
                }
                br.close();
            }
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, 'No hay nada que mostrar');
        }
    }

    //Método que permite GRABAR en un registro de productos
    public void grabar() {
        try {
            PrintWriter pw=new PrintWriter(new FileWriter('productos.txt'));
            for (int i = 0; i < getTamaño(); i++) {
                pw.println(obtener(i).getCodigo() + ',' +
                obtener(i).getDescripcion() + ',' + obtener(i).getCategoria()
                + ',' + obtener(i).getStock() + ',' + obtener(i).getPrecio());
            }
            pw.close();
            JOptionPane.showMessageDialog(null, 'Grabación correcta');
```

```
        } catch (Exception ex) {
            }
        }
        //Método que envía la información al arreglo
        public void adicionar(Producto p) {
            aProductos.add(p);
        }
        //Método para obtener información del producto
        public Producto obtener(int pos) {
            return aProductos.get(pos);
        }
        //Método para buscar un producto por su código
        public Producto buscar(String codigo) {
            for (int i = 0; i < aProductos.size(); i++) {
                if (codigo.equals(aProductos.get(i).getCodigo())) {
                    return aProductos.get(i);
                }
            }
            return null;
        }
        //Método que determina el tamaño del arreglo
        public int getTamaño() {
            return aProductos.size();
        }
        //Método que elimina un producto
        public void eliminar(Producto p) {
            aProductos.remove(p);
        }
    }
```

El siguiente script muestra el contenido de la clase **frmListadoProductos**:

```
package pFormularios;

import pClases.ArregloProductos;
import pClases.Producto;

public class frmListadoProductos extends javax.swing.JInternalFrame {
    ArregloProductos a = new ArregloProductos();

    public frmListadoProductos() {
        initComponents();
        listarProductos();
    }

    //Método que lista los productos
    void listarProductos(){
        for(int i=0;i<a.getTamaño();i++){
            Producto objP = a.obtener(i);
            tProductos.setValueAt(objP.getCodigo(), i, 0);
            tProductos.setValueAt(objP.getDescripcion(), i, 1);
            tProductos.setValueAt(objP.getCategoria(), i, 2);
            tProductos.setValueAt(objP.getStock(), i, 3);
            tProductos.setValueAt(objP.getPrecio(), i, 4);
        }
    }
}
```

El siguiente script muestra el contenido de la clase **frmNuevoProducto**:

```
package pFormularios;
import javax.swing.JOptionPane;
import pClases.ArregloProductos;
import pClases.Producto;
public class frmNuevoProducto extends javax.swing.JInternalFrame {
    public frmNuevoProducto() {
        initComponents();
    }
    private void btnGrabarActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            //Capturando los valores del formulario
            String código = txtCodigo.getText();
            String descripcion = txtDescripcion.getText();
            String categoria = String.valueOf(cboCategoria.getSelectedItem());
            int stock = Integer.parseInt(txtStock.getText());
            double precio = Double.parseDouble(txtPrecio.getText());
            //Objeto de la clase Producto
            Producto objP = new Producto(código, descripcion, categoria, stock, precio);
            //Objeto de la clase Arreglo de Productos
            ArregloProductos a = new ArregloProductos();
            //Adicionar el producto al arreglo
            a.adicionar(objP);
            //Grabar en el archivo de texto
            a.grabar();
            JOptionPane.showMessageDialog(null, 'GRABACIÓN EXITOSA ....!!');
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, ex.getMessage());
        }
    }
}
```

El siguiente script muestra el contenido de la clase **frmBuscaProducto**:

```
package pFormularios;
import pClases.ArregloProductos;
import pClases.Producto;
public class frmBuscaProducto extends javax.swing.JInternalFrame {
    public frmBuscaProducto() {
        initComponents();
    }
    private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
        //Capturando el código a buscar
        String código = txtCodigo.getText();
        //Objeto de la clase ArregloProducto
        ArregloProductos a = new ArregloProductos();
        //Buscar el producto
        Producto objP = a.buscar(código);
        //Mostrar los datos del producto encontrado
        txtDescripcion.setText(objP.getDescripcion());
        txtCategoria.setText(objP.getCategoria());
        txtStock.setText('' + objP.getStock());
        txtPrecio.setText('' + objP.getPrecio());
    }
}
```

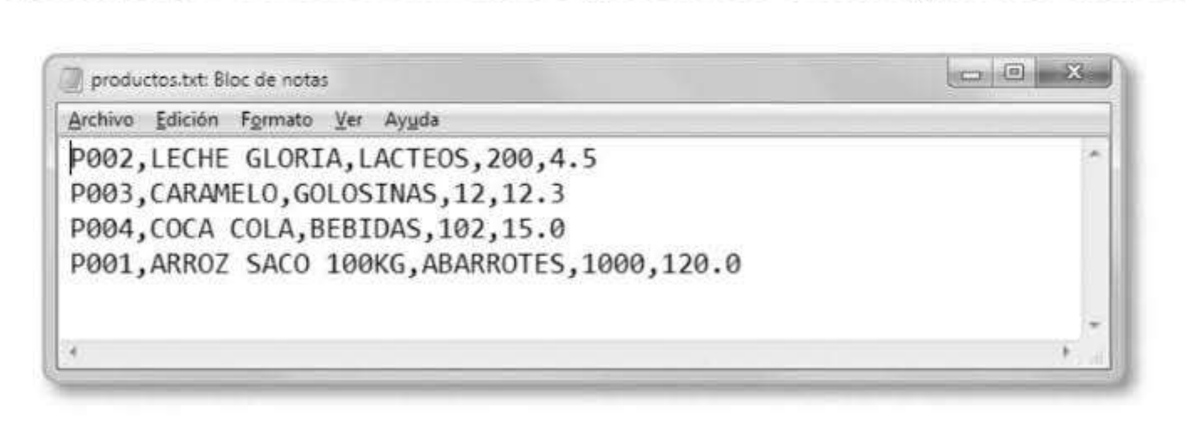
El siguiente script muestra el contenido de la clase **frmActualizaProducto**:

```
package pFormularios;

import javax.swing.JOptionPane;
import pClases.ArregloProductos;
import pClases.Producto;

public class frmActualizaProducto extends javax.swing.JInternalFrame {
    //Objeto de la clase ArregloProducto
    ArregloProductos a = new ArregloProductos();

    public frmActualizaProducto() {
        initComponents();
    }

    private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            //Capturando el código a buscar
            String código = txtCodigo.getText();

            //Buscar el producto
            Producto objP = a.buscar(código);

            //Mostrar los datos del producto encontrado
            txtDescripcion.setText(objP.getDescripcion());
            cboCategoria.setSelectedItem(objP.getCategoria());
            txtStock.setText('' + objP.getStock());
            txtPrecio.setText('' + objP.getPrecio());
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, 'Error en la búsqueda');
        }
    }

    private void btnModificarActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            //Obtener el código a modificar
            String código = txtCodigo.getText();
            Producto objP = a.buscar(código);

            //Enviar la modificación
            objP.setDescripcion(txtDescripcion.getText());

            objP.setCategoria(String.valueOf(cboCategoria.getSelectedItem()));
            objP.setStock(Integer.parseInt(txtStock.getText()));
            objP.setPrecio(Double.parseDouble(txtPrecio.getText()));

            //Grabar las modificaciones
            a.grabar();
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, 'Error al actualizar datos..!!!');
        }
    }
}
```

El siguiente script muestra el contenido de la clase **frmEliminaProducto**:

```
package pFormularios;

import javax.swing.JOptionPane;
import pClases.ArregloProductos;
import pClases.Producto;

public class frmEliminaProducto extends javax.swing.JInternalFrame {
    //Objeto de la clase ArregloProducto
    ArregloProductos a = new ArregloProductos();

    public frmEliminaProducto() {
        initComponents();
    }

    private void txtEliminarActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            //Obtener el código del producto a eliminar
            String código = txtCodigo.getText();

            //Buscar el producto por su código
            Producto objP = a.buscar(código);

            //Eliminar y grabar en el arreglo de productos
            a.eliminar(objP);
            a.grabar();
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, "Error al buscar");
        }
    }
    private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            //Capturando el código a buscar
            String código = txtCodigo.getText();

            //Buscar el producto
            Producto objP = a.buscar(código);

            //Mostrar los datos del producto encontrado
            txtDescripcion.setText(objP.getDescripcion());
            cboCategoria.setSelectedItem(objP.getCategoria());
            txtStock.setText("" + objP.getStock());
            txtPrecio.setText("" + objP.getPrecio());
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null, "Error al buscar");
        }
    }
}
```

Nota. El archivo de texto **productos.txt** debe mostrarse de la siguiente manera:



Referencias bibliográficas

Bryant, J. (2012). *Java 7 for Absolute Beginners*. Nueva York: Apress.

Deitel, P. y Deitel, H. (2012). *Cómo programar en Java*. 9.ª edición. México D.F.: Pearson Educación.

Horton's, I. (2011). *Begining Java. Java 7 Edition*. Indianápolis: John Wiley & sons.

Scharger, S. *et al*. (2007). *Professional Java*. 6.ª edición. Indianápolis: Wiley Publishing.

Impreso en los talleres gráficos de

EDITORIAL
MACRO®

Surquillo

Febrero 2018