



ESCUELA DE CIENCIAS BÁSICAS E INGENIERÍA

Ingeniería de Sistemas

ASIGNATURA: Lenguaje de Programación II

CORPORACIÓN UNIVERSITARIA REMINGTON
DIRECCIÓN PEDAGÓGICA

Este material es propiedad de la Corporación Universitaria Remington (CUR), para los estudiantes de la CUR en todo el país.

2011

CRÉDITOS



El módulo de estudio de la asignatura Lenguaje de Programación II del programa Ingeniería de Sistemas es propiedad de la Corporación Universitaria Remington. Las imágenes fueron tomadas de diferentes fuentes que se relacionan en los derechos de autor y las citas en la bibliografía. El contenido del módulo está protegido por las leyes de derechos de autor que rigen al país.

Este material tiene fines educativos y no puede usarse con propósitos económicos o comerciales.

AUTOR

César Augusto Jaramillo Henao
Tecnólogo en Sistemas
cesar.jaramillo@remington.edu.co

Nota: el autor certificó (de manera verbal o escrita) No haber incurrido en fraude científico, plagio o vicios de autoría; en caso contrario eximió de toda responsabilidad a la Corporación Universitaria Remington, y se declaró como el único responsable.

RESPONSABLES

Escuela de Ciencias Básicas e Ingeniería
Director Dr. Mauricio Sepúlveda

Director Pedagógico
Octavio Toro Chica
dirpedagogica.director@remington.edu.co

Coordinadora de Medios y Mediaciones
Angélica Ricaurte Avendaño
mediaciones.coordinador01@remington.edu.co

GRUPO DE APOYO

Personal de la Unidad de Medios y Mediaciones
EDICIÓN Y MONTAJE
Primera versión. Febrero de 2011.

Derechos Reservados



Esta obra es publicada bajo la licencia Creative Commons. Reconocimiento-No Comercial-Compartir Igual 2.5 Colombia.

TABLA DE CONTENIDO

1.	MAPA DE La ASIGNATURA	7
2.	CAPAS E INTERFACES	9
2.1.	Capas (Modelo Vista Controlador)	9
2.1.1.	Modelo Vista Controlador	9
2.2.	Interfaces.....	10
2.2.1.	Java Swing	10
3.	ESTRUCTURAS DE CONTROL.....	28
3.1.	Estructuras de Datos	29
3.1.1.	Colecciones.....	29
3.2.	Excepciones	46
3.3.	Hilos.....	57
3.3.1.	¿Qué son los hilos?.....	57
4.	ALMACENAMIENTO PERMANENTE.....	71
4.1.	Archivos	71
4.1.1.	Flujos	71
4.2.	Bases de Datos	87
4.2.1.	Configuración de ODBC.....	87
5.	RELACIÓN CON OTROS TEMAS	101
5.1.	Fuentes.....	101
5.1.1.	Libros	101
6.	PÁGINAS WEB	102

1. MAPA DE LA ASIGNATURA

LENGUAJE DE PROGRAMACION II

PROPÓSITO DEL MÓDULO

Lenguaje de programación II, esta orientado a personas habidas de conocimiento que desean desarrollar habilidades lógicas y de aplicaciones, estas alternativas tendrán unos requerimiento mínimos como el manejo de Lenguaje I, quien dará las bases necesarias para que este segundo procesos sea de completo agrado y se puede explotar al máximo características más profesionales del Ambiente.

OBJETIVO GENERAL

Desarrollar un conjunto de habilidades que permitan un manejo más amplio y profesional, además de complementario para el Lenguaje de programación, en el cual se verán herramientas de mayor alcance y propósitos más específicos orientados especialmente al almacenamiento de datos, esto sin descuidar procesos de trabajo de múltiples procesos y controles de procesos.

OBJETIVOS ESPECÍFICOS

- ◆ Desarrollar habilidades que permitan un manejo apropiado de los datos, los procesos y la visualización del usuario, de este modo se tendrá un manejo por capas apropiado que dará mas orden al trabajo establecido, además de tener herramientas de interfaz mas profesionales y de mejor presentación y alcance para el usuario.
- ◆ Dar a conocer procesos de gran ayuda para el manejo de los datos y los procesos que permitan mayor agilidad, rapidez y confiabilidad, permitiendo al usuario mayor eficacia a la hora de realizar procesos sensibles a los datos.
- ◆ Conocer métodos de procesos que permitan el acceso a datos en forma permanente partiendo de la base de alternativas como Archivos y las BD, que nos dan gran alcance y métodos mas eficientes para el usuario y para la empresa de hoy

Unidades

UNIDAD 1

Se basara en procesos conceptuales que permitan aplicar de forma firme a través del lenguaje, por medio de la presentación de la interfaz y la distribución adecuada de las tareas que lo componen.

UNIDAD 2

Conocer la forma apropiada para realizar tareas de manera más simple mediante herramientas del lenguaje sin descuidar aspectos lógicos, desarrollar habilidades que permitan la validación de las tareas del usuario.

UNIDAD 3

Conceptualizar y aplicar métodos de almacenamiento, partiendo de la configuración y el propósito de este, dando herramientas de fácil acceso a los datos sea con el fin de ingresar, consultar, retirar y modificar datos.

2. CAPAS E INTERFACES

OBJETIVO GENERAL

Dar a conocer los aspectos mas importantes del manejo de capas, sus características y el gran aporte que esto genera en el buen desarrollo de software buscando estar a nivel de los desarrollos profesionales de la industria, además de acompañarlo de un diseño gráfico mas detallado como es el SWING, complemento importante y mas detallado que su antecesor AWT.

Estos nuevos procesos harán crecer en retos los objetivos del lenguaje de programación II, y se podrá tener una visión general del lenguaje en cuanto alcance y metas de trabajo, permitiendo así que el trabajo cumpla con unos estándares mas regulados y con un aspecto mas defendió y profesional.

OBJETIVOS ESPECÍFICOS

- ◆ Conocer mas detalladamente las diferencias de los ambientes gráficos presentados por el lenguaje, ampliar la cantidad de posibilidades y de mejoras en diseño
- ◆ Apropiarnos del trabajo por capas que permite que el desarrollo tenga un perfil más definido, más organizado, de mejor distribución y de claridad en la documentación de los procesos.

Prueba Inicial

Desarrollar una aplicación de carácter gráfico (Applet), que capture los aspectos mas esenciales de una hoja de vida, todo el diseño debe estar conformado por objetos AWT.

2.1. Capas (Modelo Vista Controlador)

2.1.1. Modelo Vista Controlador

Posiblemente, dentro de las teorías de programación se habrás mencionado el concepto del Modelo Vista Controlador (MVC), el objetivo de este tipo de modelos es de intentar repetirse lo menos posible y de tenerlo todo organizado o sea hacer una distinción entre la lógica de toda la aplicación y presentación, con un fin claro como es la organización de las tareas.

Los Fundamentos básicos del MVC son los siguientes:

- ◆ **Modelo:** Sirve como representación específica de toda la información con la cual el aplicativo va a trabajar. La lógica de “datos” nos puede llegar a asegurar la integridad correcta de ellos y nos permitirá tener alternativas nuevas de los datos. Nos preguntaremos ¿Como lo hace?, no permitiéndonos comprar un número de unidades negativas, y también calculando si hoy puede ser el cumpleaños del usuario o también los totales, impuestos, esto es lo que se define como la lógica del negocio.
- ◆ **Vista:** Presenta el modelo con el que va a interactuar el usuario, más conocida como interfaz grafica, corresponde a toda la presentación y a los resultados obtenidos según la lógica del programa.
- ◆ **Controlador:** El controlador responde a los eventos, normalmente son acciones que el usuario invoca, implica interacción con el modelo y también en la vista.

2.2. Interfaces

2.2.1. Java Swing

Swing ha existido desde la JDK 1. Antes de la existencia de Swing, las interfaces gráficas con el usuario se realizaban mediante AWT, de quien Swing hereda todo el manejo de eventos y controles. Usualmente, para toda componente AWT existe una par Swing que la reemplaza, la clase Button de AWT es reemplazada por la clase JButton de Swing (el nombre de todas las componentes Swing comienza con "J").

Los controles Swing utilizan la misma infraestructura de AWT, incluyendo el modelo de eventos AWT, este permite que el componente reaccione a eventos tales como, eventos de teclado, mouse, etc... Es por esto, que la mayoría de los programas Swing necesitan importar dos paquetes AWT: `java.awt.*` y `java.awt.event.*`.

Como regla, los programas no deben usar componentes pesados de AWT junto a componentes Swing, ya que los componentes de AWT son siempre pintados sobre los de Swing.

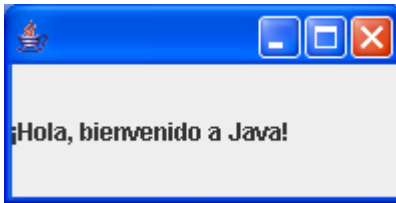
JLabel

Iniciamos por uno de los elementos gráficos más simples, el JLabel, pero ahora utilizemos una aplicación gráfica utilizando JFrame:

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

```
public class AplicacionSwing1 extends JFrame {  
  
    public static void main( String argv[] ) {  
        AplicacionSwing1 app = new AplicacionSwing1();  
    }  
  
    public AplicacionSwing1() {  
        JLabel hola = new JLabel( "¡Hola, bienvenido a Java!" );  
  
        getContentPane().add( hola,"Center" );  
        setSize( 200,100);  
        setVisible( true );  
    }  
}
```

La ejecución de esta aplicación sería:



Este tipo de aplicativos nos presenta una ventana parecida a la del applet, pero ahora no tenemos que utilizar alguna página de html o ejecutar el mismo programa poniéndole el tag <applet code = ...>

Analizando la aplicación gráfica, la manera de hacer esta aplicación es en el main se crea un objeto de la misma aplicación

```
public static void main( String argv[] ) {  
    AplicacionSwing1 app = new AplicacionSwing1();  
}
```

y se utiliza un constructor del nombre de la misma aplicación el cual crea los elementos gráficos, tal como se hacia en el init de un Applet:

```
public AplicacionSwing1() {  
  
    JLabel hola = new JLabel( "¡Hola, bienvenido a Java!" );  
  
    getContentPane().add( hola,"Center" );
```

```
setSize( 200,100);  
setVisible( true );  
}
```

El JLabel es la clase que nos permite tener más opciones que el Label normal del paquete awt.

Constructor Summary	
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(Icon image)	Creates a JLabel instance with the specified image.
JLabel(Icon image, int horizontalAlignment)	Creates a JLabel instance with the specified image and horizontal alignment.
JLabel(String text)	Creates a JLabel instance with the specified text.
JLabel(String text, Icon icon, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment.
JLabel(String text, int horizontalAlignment)	Creates a JLabel instance with the specified text and horizontal alignment.

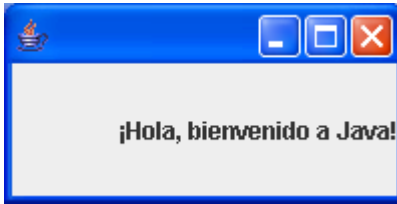
En este tipo de control podemos insertar iconos como parte de los parámetros del constructor, e inclusive podemos dar alineaciones a la etiqueta.

Una variación de esta aplicación es utilizar el constructor con alineación:

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

```
public class AplicacionSwing1 extends JFrame {  
  
    public static void main( String argv[] ) {  
        AplicacionSwing1 app = new AplicacionSwing1();  
    }  
  
    public AplicacionSwing1() {  
        JLabel hola = new JLabel( "¡Hola, bienvenido a Java!",JLabel.RIGHT);  
  
        getContentPane().add( hola,"Center" );  
        setSize( 200,100);  
        setVisible( true );  
    }  
}
```

La aplicación quedaría como:



Ahora si queremos utilizar la misma aplicación, pero con un icono, podemos tener la siguiente aplicación gráfica:

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

public class AplicacionSwing2 extends JFrame {

    public static void main( String argv[] ) {
        AplicacionSwing2 app = new AplicacionSwing2();
        app.setSize( 300,150 );
        app.setVisible( true );
    }

    public AplicacionSwing2() {
        setLayout(new GridLayout(1,1,5,5));
        Icon icon = new ImageIcon("ejercicio.gif");
        JLabel hola = new JLabel( icon, JLabel.CENTER);
        add( hola );
    }
}
```

La cual despliega el siguiente Frame:



Donde el icono “ejercicio.gif” aparece centrado.

JButton, JToggleButton, JCheckBox, JRadioButton

El JButton al igual que el Button le permite al usuario dar acciones para la ejecución de algún grupo de instrucciones.

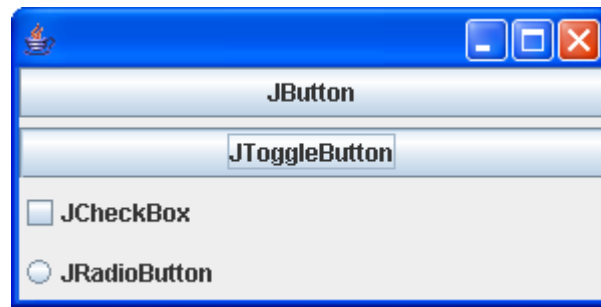
```
import java.awt.*;
import javax.swing.*;

public class AplicacionSwing3 extends JFrame {

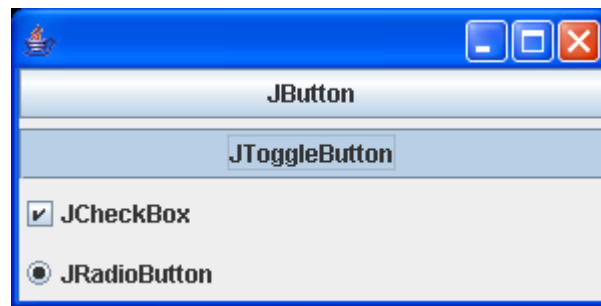
    public static void main( String argv[] ) {
        AplicacionSwing3 app = new AplicacionSwing3();
        app.setSize( 300,150 );
        app.setVisible( true );
    }

    public AplicacionSwing3() {
        setLayout(new GridLayout(4,1,5,5));
        JButton boton1 = new JButton("JButton");
        JToggleButton boton2 = new JToggleButton("JToggleButton");
        JCheckBox boton3 = new JCheckBox("JCheckBox");
        JRadioButton boton4 = new JRadioButton("JRadioButton");
        add(boton1);
        add(boton2);
        add(boton3);
        add(boton4);
    }
}
```

La ejecución de esta aplicación gráfica sería:



Al seleccionar el JToggleButton se sombree, el JCheckBox se selecciona con una marca \surd y el JRadioButton se selecciona con un punto en su interior.



Los constructores del JButton son:

Constructor Summary	
JButton ()	Creates a button with no set text or icon.
JButton (Action a)	Creates a button where properties are taken from the Action supplied.
JButton (Icon icon)	Creates a button with an icon.
JButton (String text)	Creates a button with text.
JButton (String text, Icon icon)	Creates a button with initial text and an icon.

Utilizando algunas de las instrucciones como ejemplo:

```
import java.awt.*;
```

```
Import javax.swing.*;

public class AplicacionSwing3 extends JFrame {

    public static void main( String argv[] ) {
        AplicacionSwing3 app = new AplicacionSwing3();
        app.setSize( 300,150 );
        app.setVisible( true );

    }

    public AplicacionSwing3() {
        setLayout(new GridLayout(4,1,5,5));
        Icon icon = new ImageIcon("star0.gif");
        JButton boton1 = new JButton("JButton", icon);
        JToggleButton boton2 = new JToggleButton("JToggleButton");
        JCheckBox boton3 = new JCheckBox("JCheckBox");
        JRadioButton boton4 = new JRadioButton("JRadioButton");
        add(boton1);
        add(boton2);
        add(boton3);
        add(boton4);
    }
}
```

Y la ejecución de la aplicación se vería así:



La forma de aplicar las acciones para el botón en esta aplicación gráfica es igual que con el applet, hay que darle a cada botón la facilidad de que sea utilizado con el addActionListener e implementar también esta clase y utilizar el actionPerformed para ejecutar las instrucciones.

JList, JCombo

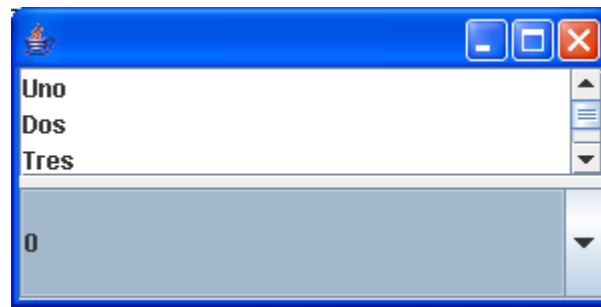
Las listas y cajas "combo" en Swing funcionan de la misma forma que lo hace el AWT, aunque sus funciones son mayores a través de algunas funciones de conveniencia que se han incorporado. Por ejemplo, JList tiene un constructor al que se puede pasar un array de objetos String para que los presente, un ejemplo es la siguiente aplicación:

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

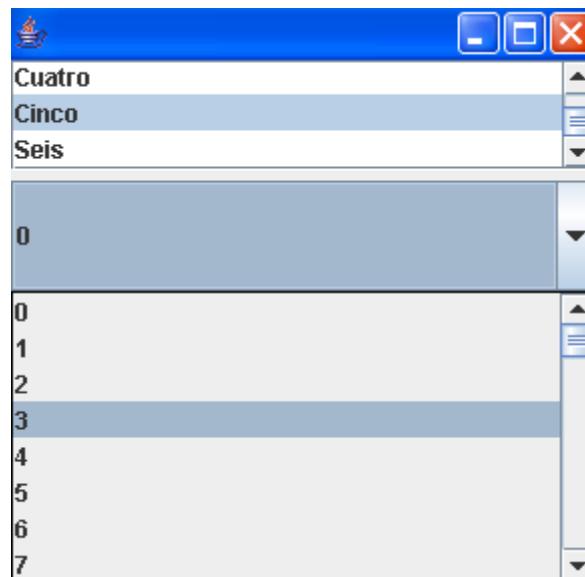
public class AplicacionSwing4 extends JFrame {
    private static String datos[] = {
        "Uno", "Dos", "Tres",
        "Cuatro", "Cinco", "Seis",
    };
    public static void main( String argv[] ) {
        AplicacionSwing4 app = new AplicacionSwing4();
        app.setSize( 300,150 );
        app.setVisible( true );
    }

    public AplicacionSwing4() {
        setLayout(new GridLayout(2,1,5,5));
        JList lista = new JList( datos );
        add( new JScrollPane( lista ) );
        JComboBox combo = new JComboBox();
        for( int i=0; i < 100; i++ )
            combo.addItem( Integer.toString( i ) );
        add( combo );
    }
}
```

La muestra de la aplicación ejecutando sería:



Donde al seleccionar la barra de desplazamiento del objeto JList podemos seleccionar alguno de los elementos de la lista, igual podemos hacer con el objeto de la clase Combo:



◆ Aplicaciones con Java Swing

En Java Swing se puede utilizar la clase JMenuBar para poder crear barras de menús y dar opciones a ejecutar, haciendo así más llamativas nuestras aplicaciones y de mayor funcionalidad cuando el proyecto es mas amplio. A continuación veamos la manera de usar el JMenuBar, viendo su constructor y algunos de sus métodos:

Constructor Summary

JMenuBar()
Creates a new menu bar.

Method Summary

JMenu	add (JMenu c) Appends the specified menu to the end of the menu bar.
void	addNotify () Overrides <code>JComponent.addNotify</code> to register this menu bar with the current keyboard manager.
AccessibleContext	getAccessibleContext () Gets the <code>AccessibleContext</code> associated with this <code>JMenuBar</code> .
Component	getComponent () Implemented to be a <code>MenuElement</code> .
Component	getComponentAtIndex (int i) Deprecated. <i>replaced by <code>getComponent(int i)</code></i>
int	getComponentIndex (Component c) Returns the index of the specified component.
JMenu	getHelpMenu () Gets the help menu for the menu bar.

El método `add` es común utilizarlo para añadir elementos del objeto `JMenu`, que sería cada menú a aparecer. El `JMenu` tiene los siguientes constructores y algunos de los métodos utilizados:

Constructor Summary

JMenu()
Constructs a new `JMenu` with no text.

JMenu(Action a)
Constructs a menu whose properties are taken from the `Action` supplied.

JMenu(String s)
Constructs a new `JMenu` with the supplied string as its text.

JMenu(String s, boolean b)
Constructs a new `JMenu` with the supplied string as its text and specified as a tear-off menu or not.

Method Summary

JMenuItem	add (Action a) Creates a new menu item attached to the specified <code>Action</code> object and appends it to the end of this menu.
Component	add (Component c) Appends a component to the end of this menu.
Component	add (Component c, int index) Adds the specified component to this container at the given position.
JMenuItem	add (JMenuItem menuItem) Appends a menu item to the end of this menu.

El objeto de la clase JMenu nos sirve para crear menús en la barra de menús del objeto JMenuBar, primero debemos crear cada objeto por separado, añadirle a cada objeto de la clase JMenu sus opciones y después estos objetos de la clase JMenu añadirlos al objeto JMenuBar, también tenemos otra clase de Java Swing que hereda de la clase JEditorPane que nos permite tener un texto el cual puede tener atributos que son representados gráficamente:

Constructor Summary	
JTextPane()	Creates a new JTextPane.
JTextPane(StyledDocument doc)	Creates a new JTextPane, with a specified document model.

Method Summary	
Style	addStyle(String nm, Style parent) Adds a new style into the logical style hierarchy.
protected EditorKit	createDefaultEditorKit() Creates the EditorKit to use by default.
AttributeSet	getCharacterAttributes() Fetches the character attributes in effect at the current location of the caret, or null.
MutableAttributeSet	getInputAttributes() Gets the input attributes for the pane.
Style	getLogicalStyle() Fetches the logical style assigned to the paragraph represented by the current position of the caret, or null.
AttributeSet	getParagraphAttributes() Fetches the current paragraph attributes in effect at the location of the caret, or null if none.

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
import javax.swing.text.*;
```

```
public class AplicacionSwing5 extends JPanel implements ActionListener {  
    private Style estiloMorado,estiloGris,estiloCeleste,estiloRojo,estiloAzul;  
    private JTextPane texto;
```

```
    public AplicacionSwing5() {  
        setLayout( new BorderLayout() );  
        JMenuBar menu = new JMenuBar();  
        JMenu estilo = new JMenu( "Estilo" );  
        menu.add( estilo );
```

```
        JMenuItem mi = new JMenuItem( "Morado" );
```

```
estilo.add( mi );
mi.addActionListener(this);
mi = new JMenuItem( "Gris" );
estilo.add( mi );
mi.addActionListener(this);
mi = new JMenuItem( "Celeste" );
estilo.add( mi );
mi.addActionListener(this);
mi = new JMenuItem( "Rojo" );
estilo.add( mi );
mi.addActionListener(this);
    mi = new JMenuItem( "Azul" );
estilo.add( mi );
mi.addActionListener( this );
add( menu, BorderLayout.NORTH );
```

```
StyleContext sc = new StyleContext();
estiloMorado = sc.addStyle( null,null );
StyleConstants.setForeground( estiloMorado,Color.magenta );
estiloGris = sc.addStyle( null,null );
StyleConstants.setForeground( estiloGris,Color.gray );
StyleConstants.setFontSize( estiloGris,24 );
estiloCeleste = sc.addStyle( null,null );
StyleConstants.setForeground( estiloCeleste,Color.cyan );
estiloRojo = sc.addStyle( null,null );
StyleConstants.setForeground( estiloRojo,Color.red );
estiloAzul = sc.addStyle( null,null );
StyleConstants.setForeground( estiloAzul,Color.blue );
```

```
DefaultStyledDocument doc = new DefaultStyledDocument(sc);
```

```
JTextPane texto = new JTextPane(doc);
add( texto, BorderLayout.CENTER );
}
```

```
public void actionPerformed( ActionEvent e ) {
    Style estilo = null;
    String color = (String) e.getActionCommand();

    if( color.equals( "Morado" ) ) {
```

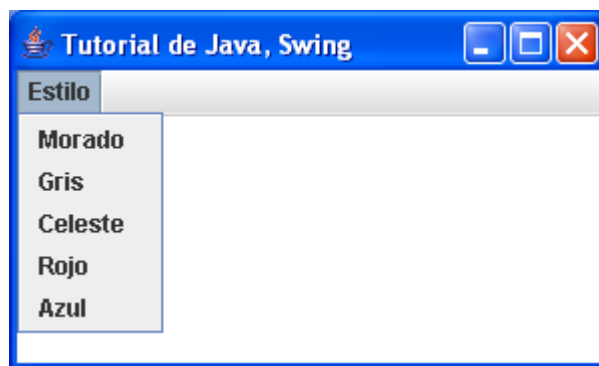
```
        estilo = estiloMorado;
    } else if( color.equals( "Celeste" ) ) {
        estilo = estiloCeleste;
    } else if( color.equals( "Gris" ) ) {
        estilo = estiloGris;
    } else if( color.equals( "Rojo" ) ) {
        estilo = estiloRojo;
    } else if( color.equals( "Azul" ) ) {
        estilo = estiloAzul;
    }
    texto.setCharacterAttributes( estilo,false );
}

public static void main( String argv[] ) {
    JFrame app = new JFrame( "Tutorial de Java, Swing" );

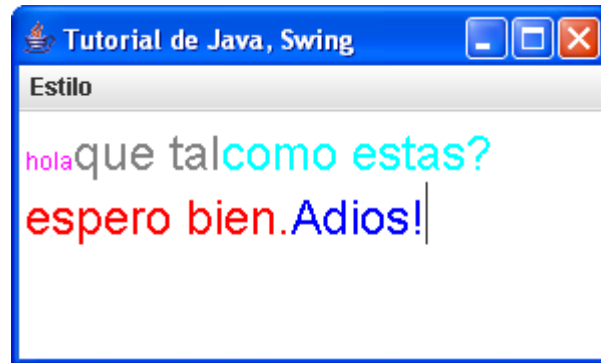
    app.addWindowListener( new WindowAdapter() {
        public void windowClosing( WindowEvent evt ){
            System.exit( 0 );
        }
    });
    app.getContentPane().add( new AplicacionSwing5(),BorderLayout.CENTER );
    app.setSize( 300,180 );

    app.setVisible( true );
}
}
```

La ejecución de la aplicación sería:



Y una vez utilizadas las opciones del menú:



Primero creamos la barra de menú y la primera de las opciones es el estilo y después el Morado y las demás, añadiéndole a las correspondientes la facilidad de ser escuchadas, como el Morado y las que siguen:

```
JMenuBar menu = new JMenuBar();  
JMenu estilo = new JMenu( "Estilo" );  
menu.add( estilo );
```

```
JMenuItem mi = new JMenuItem( "Morado" );  
estilo.add( mi );  
mi.addActionListener(this);
```

Antes de crear el objeto JTextPane, debemos definir los diferentes estilos que este objeto va a tener y eso lo hacemos con la ayuda de la clase DefaultStyledDocument la cual debe de tener todo el contexto de los estilos a utilizar con la ayuda de la clase StyleContext, como se ve:

```
StyleContext sc = new StyleContext();  
estiloMorado = sc.addStyle( null,null );  
StyleConstants.setForeground( estiloMorado,Color.magenta );  
estiloGris = sc.addStyle( null,null );
```

```
DefaultStyledDocument doc = new DefaultStyledDocument(sc);
```

```
JTextPane texto = new JTextPane(doc);
```

JTable

JTable es la clase en principio, se creó para constituir un interfaz ligado a bases de datos a través del Java Database Connectivity (JDBC), y así evitar la complejidad para el manejo de los datos, proporcionando flexibilidad al programador.

La JTable nos permite manejar información como si fuera una hoja electrónica de cálculo.

Los constructores de la JTable son:

Constructor Summary	
JTable()	Constructs a default JTable that is initialized with a default data model, a default column model, and a default selection model.
JTable(int numRows, int numColumns)	Constructs a JTable with numRows and numColumns of empty cells using DefaultTableModel.
JTable(Object[][] rowData, Object[] columnNames)	Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.
JTable(TableModel dm)	Constructs a JTable that is initialized with dm as the data model, a default column model, and a default selection model.
JTable(TableModel dm, TableColumnModel cm)	Constructs a JTable that is initialized with dm as the data model, cm as the column model, and a default selection model.
JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)	Constructs a JTable that is initialized with dm as the data model, cm as the column model, and sm as the selection model.
JTable(Vector rowData, Vector columnNames)	Constructs a JTable to display the values in the Vector of Vectors, rowData, with column names, columnNames.

Una manera sencilla de usar la JTable es tomando ayuda de la TableModel, que nos ayuda a modelar la tabla, y por otro lado la clase JScrollPane permite añadirle una barra de desplazamiento en ambas partes de la tabla tanto horizontal como vertical, un ejemplo sencillo de cómo se manejaría esto sería crear el objeto dataModel como se ve a continuación:

```
TableModel dataModel = new AbstractTableModel() {
    public int getColumnCount() {
        return 10;
    }
    public int getRowCount() {
        return 10;
    }
    public Object getValueAt(int row, int col) {
        return new Integer(row*col);
    }
};
JTable table = new JTable(dataModel);
JScrollPane scrollpane = new JScrollPane(table);
```

Un ejemplo de aplicación gráfica que genera una JTable es el siguiente:

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

class AplicacionSwing6 extends JPanel {
    private JTable tabla;
    private JScrollPane panelScroll;
    private String tituloColumna[];
    private String datoColumna[][];

    public AplicacionSwing6() {
        setLayout( new BorderLayout() );
        // se crean las columnas con su titulo por separado
        tituloColumna = new String[8];

        for( int i=0; i < 8; i++ ) {
            tituloColumna[i] = "Col: "+i;
        }
        datoColumna = new String[100][8];
        // se cargan los valores en cada celda para lo que queremos que aparezca
        for( int i=0; i < 100; i++ ) {
            for( int j=0; j < 8; j++ ) {
                datoColumna[i][j] = "" + j + "," + i;
            }
        }
        // se crea una instancia del componente Swing
        tabla = new JTable( datoColumna,tituloColumna );
        // se cambia la presentación a la tabla
        tabla.setShowHorizontalLines( false );
        tabla.setRowSelectionAllowed( true );
        tabla.setColumnSelectionAllowed( true );
        // se cambia el color de la zona seleccionada (rojo/blanco)
        tabla.setSelectionForeground( Color.white );
        tabla.setSelectionBackground( Color.red );
        // la tabla se añade a un objeto de la clase JScrollPane
```

```
// para que tenga la facilidad de barras deslizadoras
panelScroll = new JScrollPane( tabla );
add( panelScroll, BorderLayout.CENTER );
}

public static void main( String args[] ) {
JFrame ventana = new JFrame( "Utilizando JTable" );

ventana.addWindowListener( new WindowAdapter() {
    public void windowClosing( WindowEvent evt ){
        System.exit( 0 );
    }
});
ventana.getContentPane().add( new AplicacionSwing6(),BorderLayout.CENTER );
ventana.setSize( 300,180 );

ventana.setVisible( true );
}
}
```

La aplicación se vería de la siguiente manera:



Col: 0	Col: 1	Col: 2	Col: 3	Col: 4	Col: 5	Col: 6	Col: 7
0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5
0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6
0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7

Para utilizar o hacer algo con los datos, tendríamos que revisar si el dato o la columna o el renglón fueron seleccionados o editados, utilizando los métodos de la clase.

EJERCICIOS DE AUTOEVALUACIÓN

1. Desarrollar un Modelo – Vista – Control para el manejo de una nomina, crear solo el esquema de cómo sería la distribución y los procesos de este.
2. Diseñar una calculadora utilizando Swing

Prueba Final

Crear un aplicativo en Java que cree un menú que contenga tres opciones, una de figura geométrica, una de color y una de cantidad, según los seleccionados que muestre en pantalla de la figura seleccionada, el número específico de estos y en el color seleccionado.

Actividad

Realizar una lectura del tema MVC, que permita definir nuevos aspectos y determinar lo siguiente ¿la Programación por capas esta ligada directamente a un BD?

3. ESTRUCTURAS DE CONTROL

OBJETIVO GENERAL

Conocer las estructuras de control que el lenguaje aporta a los ya definidos procesos lógicos, estos permitirán la creación de una codificación más simple y sencilla, que darán mayor flexibilidad en el trabajo de todas las herramientas, además de permitir nuevas alternativas de trabajo.

Complementario a este tema se verán muy enfocado el tema de excepciones que nos darán la opción de controlar las diferentes tareas ejecutadas por el usuario y así no tener un aspecto lógico solo, sino que sea mas complejo y completo al saber que la información estará debidamente controlada, esto ayudado por casos como el manejo recursivo, de multiprocesos, que harán de el trabajo mas dinámico y ágil, permitiendo que los resultados sean mas adecuados y en el tiempo que se requieran.

OBJETIVOS ESPECÍFICOS

- ◆ Conocer las herramientas de estructuras que permitan realizar de forma óptica el aprovechamiento de los recursos del sistema.
- ◆ Experimentar con las excepciones a la información y los procesos tanto del sistema como propios.
- ◆ Aplicar el manejo de hilos que permitirán al usuario ver cambios significativos en el tiempo de los procesos y arrojo de resultados.

Prueba Inicial

Realizar una estructura de datos matricial, que permita manipular un cuadrado mágico, en la forma tradicional lógica de este tema.

3.1. Estructuras de Datos

3.1.1. Colecciones

Java tiene matrices para almacenar grupos de datos de tipo similar, que son muy útiles para modelos simples de acceso a datos. Las Colecciones o enumeraciones ofrecen una manera más amplia y orientada a objetos para almacenar conjuntos de datos de tipo igual.

Si se ha utilizado en Java un arreglo, ya sea de datos primitivos o de Objetos, ha sido necesaria la definición inicial del número de elementos a crear en este arreglo, para realizar el recorrido desde la primera posición que tiene un valor (ya sea un dato primitivo o un objeto) hasta la última posición del vector o matriz.

Si en el momento que estamos utilizando un arreglo, ya sea en un applet o en una aplicación, requerimos que el arreglo se amplie, esto sería posible solamente si hacemos una creación de otro arreglo con una dimensión mayor, pasando todos los elementos del arreglo anterior al arreglo nuevo, recrear el arreglo viejo y volver a hacer toda la asignación de elementos, para que así todos los elementos vuelvan a estar en el mismo arreglo, y este pueda reutilizarse sin mayor problema en todos los métodos del applet o aplicación.

Este planteamiento es muy extenso y complejo, un ejemplo, tomando en cuenta que arreglo1 es nuestro arreglo de enteros que queremos engrandecer (al doble por ejemplo):

```
int arreglo2[] = new int[arreglo1.length];
for (int i=0; i< arreglo1.length; i++) {
    arreglo2[i] = arreglo1[i];
}
int arreglo1[] = new int[arreglo2.length * 2];
for (int i=0; i< arreglo2.length; i++) {
    arreglo1[i] = arreglo2[i];
}
```

Ya teneindo la posibilidad de utilizar el arreglo1 pero ahora con el doble de capacidad que antes.

Como vemos este proceso no es util, ya que requiere tiempo de ejecución y requiere estar revisando si ya se completo.

Las colecciones manejan este tipo de anomalías en forma transparente, ya que una colección tiene definida una capacidad inicial y Java crea de nuevo la colección para hacerla más dinámica, según sea lo requerido en un momento dado del proceso.

La clase Vector

Una de las clases más utilizadas y comunes en la clase Vector es una matriz ampliable de referencia a objeto. Internamente, un Vector implementa un crecimiento para minimizar la reasignación y el espacio perdidos, esto lo hace dinámico. Los objetos se pueden almacenar al final de un Vector utilizando el método `addElement()` o en un índice dado mediante el método `insertElement()`. Se puede almacenar una matriz de objetos en un Vector utilizando el método `copyInto()`. Una vez se ha almacenado un conjunto de objetos en un Vector, se puede utilizar para buscar un elemento en concreto utilizando los métodos `contains()`, `indexOf()` o `lastIndexOf()`. También se puede extraer un objeto de una posición específica de un Vector utilizando los métodos `elementAt()`, `firstElement()` y `lastElement()`.

Los constructores de la clase Vector (perteneciente al paquete `java.util`) son:

Constructor Summary	
<code>Vector()</code>	Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
<code>Vector(Collection<? extends E> c)</code>	Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
<code>Vector(int initialCapacity)</code>	Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
<code>Vector(int initialCapacity, int capacityIncrement)</code>	Constructs an empty vector with the specified initial capacity and capacity increment.

Al crear un objeto de la clase Vector, se puede no definir parámetros, en cuyo caso inicia con 10 elementos, que inicie con una cierta capacidad, que empiece con elementos de alguna otra colección, etc.

Entre algunos de los métodos de la clase están:

Method Summary	
boolean	<u>add</u> (<u>E</u> o) Appends the specified element to the end of this Vector.
void	<u>add</u> (int index, <u>E</u> element) Inserts the specified element at the specified position in this Vector.
boolean	<u>addAll</u> (<u>Collection</u> <? extends <u>E</u> > c) Appends all of the elements in the specified Collection to the end of this Vector, in the order that they returned by the specified Collection's Iterator.
boolean	<u>addAll</u> (int index, <u>Collection</u> <? extends <u>E</u> > c) Inserts all of the elements in the specified Collection into this Vector at the specified position.

Para añadir elementos al objeto, dichos elementos deben ser objetos, es decir suponiendo que tenemos Vector vec = new Vector(); anteriormente definido, no podemos hacer:

```
vec. add(3);
```

para poder añadir un 3 como objeto debemos utilizar la ayuda de la clase Integer:

```
// esto ya añade un objeto entero al objeto de la clase Vector
```

```
vec. add( new Integer(3) );
```

También hay otros métodos a saber:

<u>E</u>	<u>get</u> (int index) Returns the element at the specified position in this Vector.
int	<u>hashCode</u> () Returns the hash code value for this Vector.
int	<u>indexOf</u> (Object elem) Searches for the first occurrence of the given argument, testing for equality using the equals method.
int	<u>indexOf</u> (Object elem, int index) Searches for the first occurrence of the given argument, beginning the search at index, and testing for equality using the equals method.
void	<u>insertElementAt</u> (E obj, int index) Inserts the specified object as a component in this vector at the specified index.
boolean	<u>isEmpty</u> () Tests if this vector has no components.
<u>E</u>	<u>lastElement</u> () Returns the last component of the vector.
int	<u>lastIndexOf</u> (Object elem) Returns the index of the last occurrence of the specified object in this vector.
int	<u>lastIndexOf</u> (Object elem, int index) Searches backwards for the specified object, starting from the specified index, and returns an index to it.
<u>E</u>	<u>remove</u> (int index) Removes the element at the specified position in this Vector.
boolean	<u>remove</u> (Object o) Removes the first occurrence of the specified element in this Vector. If the Vector does not contain the element, it is unchanged.
boolean	<u>removeAll</u> (Collection<?> c) Removes from this Vector all of its elements that are contained in the specified Collection.
void	<u>removeAllElements</u> () Removes all components from this vector and sets its size to zero.
boolean	<u>removeElement</u> (Object obj) Removes the first (lowest-indexed) occurrence of the argument from this vector.

El método get nos ayuda a obtener un objeto específico, dando como parámetro el índice, también existe el remove y el removeAll, el método remove borra un objeto específico, mientras que el removeAll borra todos los elementos del vector.

<u>E</u>	<u>set</u> (int index, <u>E</u> element) Replaces the element at the specified position in this Vector with the specified element.
void	<u>setElementAt</u> (<u>E</u> obj, int index) Sets the component at the specified index of this vector to be the specified object.
void	<u>setSize</u> (int newSize) Sets the size of this vector.
int	<u>size</u> () Returns the number of components in this vector.
<u>List<E></u>	<u>subList</u> (int fromIndex, int toIndex) Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
<u>Object[]</u>	<u>toArray</u> () Returns an array containing all of the elements in this Vector in the correct order.
<T> T[]	<u>toArray</u> (T[] a) Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
<u>String</u>	<u>toString</u> () Returns a string representation of this Vector, containing the String representation of each element.
void	<u>trimToSize</u> () Trims the capacity of this vector to be the vector's current size.

Veamos el applet, este sería como el definido abajo:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.util.Vector;

// <applet width="400" height="200" code="AppletArreglos1"></applet>

public class AppletArreglo3 extends Applet implements ActionListener{
    Label l1, l2;
    Button b1, b2,b3,b4;
    TextField t1;
    TextArea ta1;
    Vector vector;
    Panel p1, p2,p3;

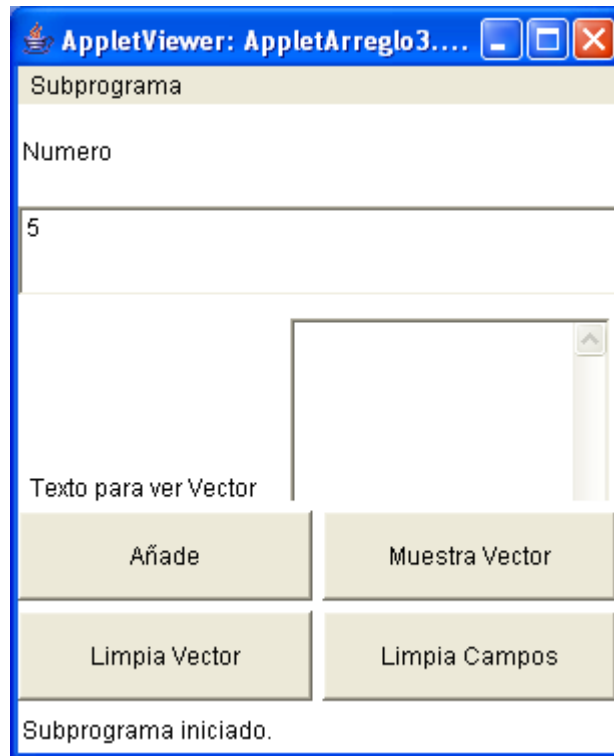
    public AppletArreglo3() {
        l1 = new Label("Numero");
        l2 = new Label("Texto para ver Vector");
        t1 = new TextField(8);
        ta1 = new TextArea(10,20);
        b1 = new Button("Añade");
        b2 = new Button("Muestra Vector");
```

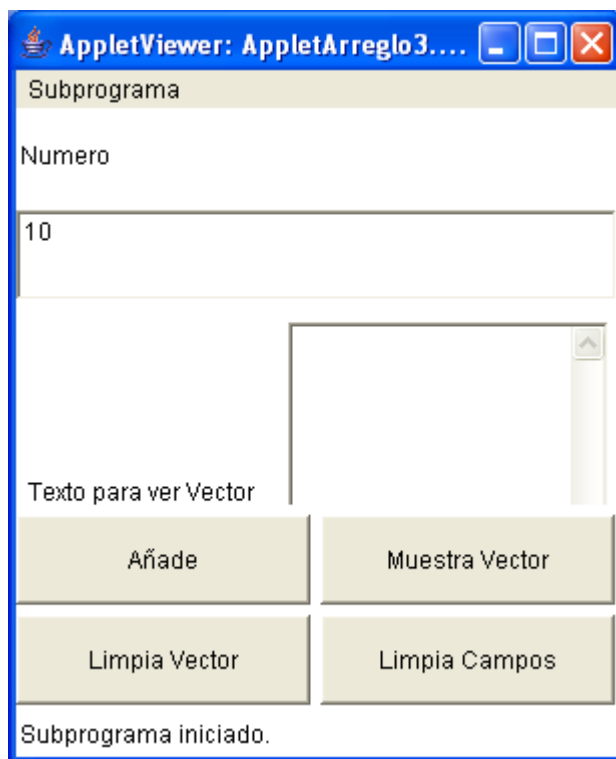
```
b3 = new Button("Limpia Vector");
b4 = new Button("Limpia Campos");
p1 = new Panel(new GridLayout(2,1,5,5));
p2 = new Panel(new FlowLayout());
p3 = new Panel(new GridLayout(2,2,5,5));
setLayout(new GridLayout(3,1,5,5));
p1.add(l1);
p1.add(t1);
p2.add(l2);
p2.add(ta1);
p3.add(b1);
p3.add(b2);
p3.add(b3);
p3.add(b4);
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
b4.addActionListener(this);
vector = new Vector();
add(p1);
add(p2);
add(p3);
}
```

```
public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == b1) {
        vector.add(new Integer(t1.getText()));
        t1.setText("");
    }
    if (ae.getSource() == b2) {
        ta1.setText("");
        for (int i=0; i < vector.size(); i++) {
            ta1.append("" + vector.get(i).toString() + "\n");
        }
    }
    if (ae.getSource() == b3) {
        vector = new Vector();
    }
    if (ae.getSource() == b4) {
        t1.setText("");
    }
}
```

```
        ta1.setText("");  
    }  
}
```

Para revisar que hace el applet veamos algunos ejemplos de su ejecución:





AppletViewer: AppletArreglo3....

Subprograma

Numero

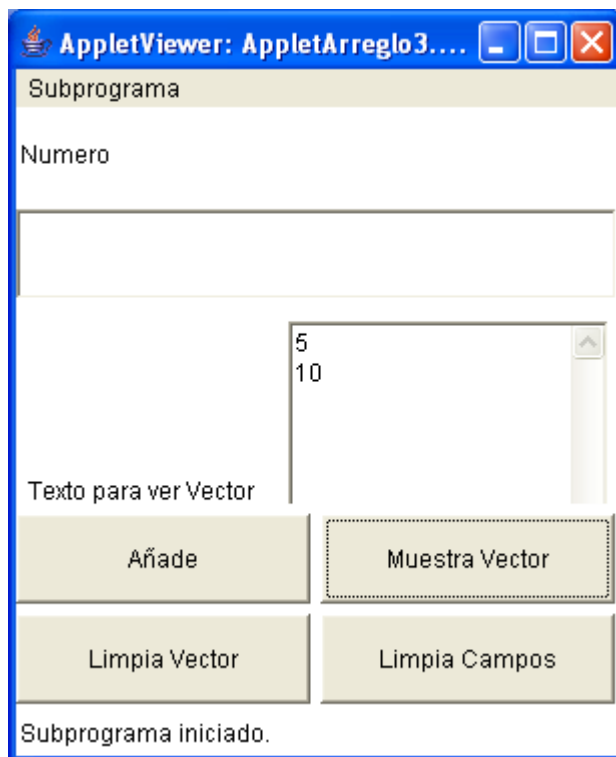
10

Texto para ver Vector

Añade Muestra Vector

Limpia Vector Limpia Campos

Subprograma iniciado.



AppletViewer: AppletArreglo3....

Subprograma

Numero

5
10

Texto para ver Vector

Añade Muestra Vector

Limpia Vector Limpia Campos

Subprograma iniciado.

Observemos como para crear el objeto utilizamos la instrucción:

```
arreglo = new Vector();
```

Para definir un nuevo elemento utilizamos la instrucción:

```
arreglo.add(new Integer(t1.getText()));
```

es importante aquí hacer notar que el método `getText()` nos regresa un objeto de la clase `String`, en la clase `Integer` podemos observar que se puede crear un objeto con un `String`, y entonces estamos añadiendo un objeto al objeto `arreglo` de la clase `Vector`.

Para recorrer todo el arreglo podemos utilizar un ciclo que empieza en cero, al igual que un arreglo, un objeto de la clase `Vector` inicia en cero y puedes utilizar el cero como índice, es por eso que para recorrer el objeto utilizamos:

```
for (int i=0; i < arreglo.size(); i++) {  
    ta1.append("" + arreglo.get(i) + "\n");
```

en cada renglón del objeto `ta1` (`TextArea`) añadimos el valor del objeto, veamos que es lo mismo utilizar esto que tomar el método `toString` (definido en toda clase) para pasar el valor `String` del objeto.

```
for (int i=0; i < arreglo.size(); i++) {  
    ta1.append("" + arreglo.get(i).toString() + "\n");
```

Esto debería ser la mejor manera de definirlo, pero se debe asegurar que la clase que estamos utilizando tenga el método `toString` definido, analicemos que hace Java.

Cuando Java encuentra:

```
arreglo.get(i).toString()
```

primero utiliza el método `get()` pasándole el valor de `i`, y eso hace que se regrese un objeto de la clase `Integer`, ya que es lo que le estamos añadiendo, posteriormente `arreglo.get(i)` es realmente un objeto de la clase `Integer`, entonces al utilizar el método `toString()` lo que estamos haciendo es hacer uso del método `toString()` de la clase `Integer`.

Stack

Un `Stack`, pila, es una subclase de `Vector` que implementa una pila simple del tipo `FIFO` (`First In First Out`, primero en entrar, primero en salir). Además de los métodos comunes de la clase padre, `Stack` utiliza el método `push()`, que ubica objetos en la parte superior de la pila y el método `pop()`

que elimina y devuelve el objeto superior de la pila. También tiene del método peek() para obtener el objeto superior de la pila, pero no retirarlo. El método empty() devolverá true si no hay nada en la pila. El método search() comprobará si existe un objeto en la pila y devolverá el número de llamadas al método pop() que se necesitarán realizar para que dicho objeto se quede en la parte superior de la pila, o -1 si el objeto pasado como parámetro no se encuentra.

```
import java.util.Stack;
import java.util.EmptyStackException;

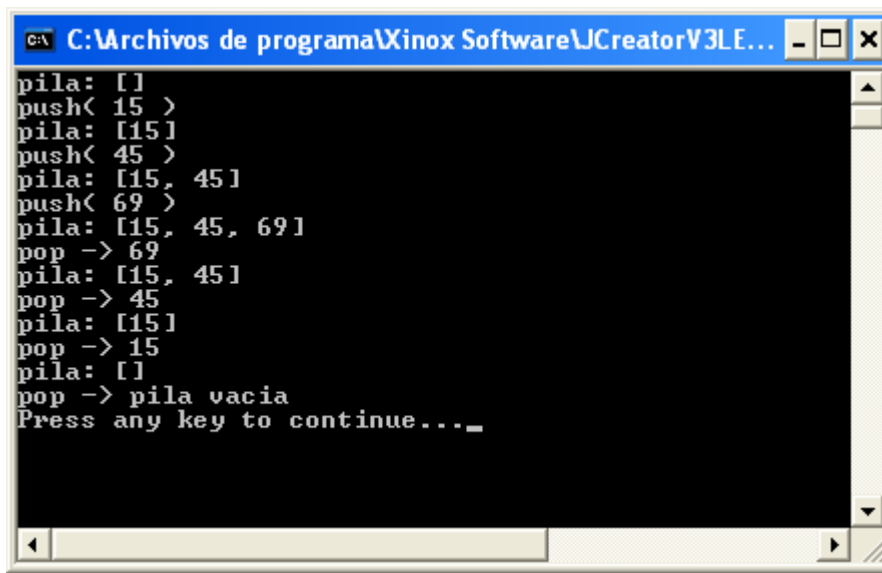
public class AplicacionColeccion1 {
    static void hacePush( Stack st,int a ) {
        st.push( new Integer( a ) );
        System.out.println( "push( "+a+" )" );
        System.out.println( "pila: "+st );
    }

    static void hacePop( Stack st ) {
        System.out.print( "pop -> " );
        Integer a = (Integer)st.pop();
        System.out.println( a );
        System.out.println( "pila: "+st );
    }

    public static void main( String args[] ) {
        Stack st = new Stack();
        System.out.println( "pila: "+st );
        hacePush( st,15 );
        hacePush( st,45 );
        hacePush( st,69 );
        hacePop( st );
        hacePop( st );
        hacePop( st );

        try {
            hacePop( st );
        } catch( EmptyStackException e ) {
            System.out.println( "pila vacia" );
        }
    }
}
```

La ejecución de la aplicación anterior es:



```
pila: []
push< 15 >
pila: [15]
push< 45 >
pila: [15, 45]
push< 69 >
pila: [15, 45, 69]
pop -> 69
pila: [15, 45]
pop -> 45
pila: [15]
pop -> 15
pila: []
pop -> pila vacia
Press any key to continue..._
```

Observemos como se van añadiendo los elementos en la pila, y vemos como cada elemento nuevo se va añadiendo después, pero cada vez que se saca un objeto de la pila, se toma el último añadido en ella.

Es importante reconocer que la pila sigue siendo un objeto que hereda de la clase Vector, por lo cual si utilizamos algún método de esta clase se comportará como tal, veamos ahora el mismo ejemplo, pero añadiendo algunas instrucciones para utilizar los métodos de la clase Vector:

```
import java.util.Stack;
import java.util.EmptyStackException;

public class AplicacionColeccion1 {
    static void hacePush( Stack st,int a ) {
        st.push( new Integer( a ) );
        System.out.println( "push( "+a+" )" );
        System.out.println( "pila: "+st );
    }

    static void hacePop( Stack st ) {
        System.out.print( "pop -> " );
        Integer a = (Integer)st.pop();
        System.out.println( a );
        System.out.println( "pila: "+st );
    }
}
```

```
public static void main( String args[] ) {  
    Stack st = new Stack();  
    System.out.println( "pila: "+st );  
    hacePush( st,15 );  
    hacePush( st,45 );  
    hacePush( st,69 );  
    for (int i=0; i<st.size(); i++) {  
        System.out.println("Elemento (" + i + " ) = "+ st.get(i) );  
    }  
    hacePop( st );  
    hacePop( st );  
    hacePop( st );  
  
    try {  
        hacePop( st );  
    } catch( EmptyStackException e ) {  
        System.out.println( "pila vacia" );  
    }  
}
```

Ahora si revisamos un ejemplo en el que utilizamos el objeto Vector, pero añadiendo objetos de la clase Cuenta, en este ejemplo vemos que ahora la clase Cuenta tiene el método toString() necesario para poder desplegar las cuentas a la pantalla.

Clase Cuenta

```
public class Cuenta {  
    private String nombre; // nombre del cliente  
    private int numero; // numero de la cuenta  
    private double saldo; // saldo de la cuenta  
  
    // método para construir una cuenta vacía  
    public Cuenta() {  
        nombre = "";  
        numero = 0;  
        saldo = 0.0d;  
    }  
  
    // método para construir una cuenta con valores
```

```
public Cuenta(int numero, String nombre, double saldo) {
    this.nombre = nombre;
    this.numero = numero;
    this.saldo = saldo;
}

// método que te da el nombre de la cuenta
public String obtenNombre() {
    return nombre;
}

// método que te da el número de la cuenta
public int obtenNumero() {
    return numero;
}

// método que te da el saldo de una cuenta
public double obtenSaldo() {
    return saldo;
}

// método que sirve para cambiar el valor del nombre
public void cambiaNombre(String nombre) {
    this.nombre = nombre;
}

// método que sirve para cambiar el valor del saldo
public void cambiaNumero(int numero) {
    this.numero = numero;
}

// método que sirve para cambiar el valor del saldo
public void cambiaSaldo(double saldo) {
    this.saldo = saldo;
}

// metodo para depositar
public void deposita(double cantidad) {
    cambiaSaldo(obtenSaldo() + cantidad);
}
```

```
// metodo para retirar
public boolean retira(double cantidad) {
    if (cantidad <= obtenSaldo()) {
        cambiaSaldo(obtenSaldo() - cantidad);
        return true;
    }
    return false;
}

// metodo para regresar un String como objeto
public String toString() {
    return obtenNumero() + " " + obtenNombre() + " " + obtenSaldo();
}
}
```

Clase AppletArreglo4

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import java.util.Vector;

// <applet width="400" height="200" code="AppletArreglo4"></applet>

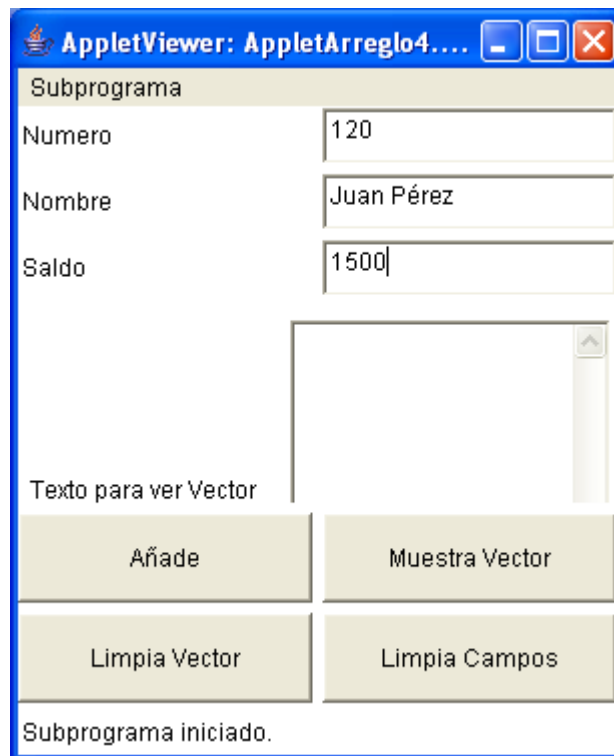
public class AppletArreglo4 extends Applet implements ActionListener{
    Label l1, l2, l3, l4;
    Button b1, b2, b3, b4;
    TextField t1, t2, t3;
    TextArea ta1;
    Vector vector;
    Panel p1, p2, p3;

    public AppletArreglo4() {
        l1 = new Label("Numero");
        l2 = new Label("Nombre");
        l3 = new Label("Saldo");
        l4 = new Label("Texto para ver Vector");
        t1 = new TextField(8);
        t2 = new TextField(20);
```

```
t3 = new TextField(12);
ta1 = new TextArea(10,20);
b1 = new Button("Añade");
b2 = new Button("Muestra Vector");
b3 = new Button("Limpia Vector");
b4 = new Button("Limpia Campos");
p1 = new Panel(new GridLayout(3,2,5,5));
p2 = new Panel(new FlowLayout());
p3 = new Panel(new GridLayout(2,2,5,5));
setLayout(new GridLayout(3,1,5,5));
p1.add(l1);
p1.add(t1);
p1.add(l2);
p1.add(t2);
p1.add(l3);
p1.add(t3);
p2.add(l4);
p2.add(ta1);
p3.add(b1);
p3.add(b2);
p3.add(b3);
p3.add(b4);
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
b4.addActionListener(this);
vector = new Vector();
add(p1);
add(p2);
add(p3);
}

public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == b1) {
        int numero = Integer.parseInt(t1.getText());
        String nombre = t2.getText();
        double saldo = Double.parseDouble(t3.getText());
        vector.add(new Cuenta(numero, nombre, saldo));
        t1.setText("");
        t2.setText("");
    }
}
```

```
t3.setText("");
}
if (ae.getSource() == b2) {
    ta1.setText("");
    for (int i=0; i < vector.size(); i++) {
        ta1.append("" + ((Cuenta) vector.get(i)).toString() + "\n");
    }
}
if (ae.getSource() == b3) {
    vector = new Vector();
}
if (ae.getSource() == b4) {
    t1.setText("");
    ta1.setText("");
}
}
}
```



Añadiendo otro elemento

AppletViewer: AppletArreglo4....

Subprograma

Numero: 75

Nombre: Luisa Lankenau

Saldo: 1000

Texto para ver Vector

Añade Muestra Vector

Limpia Vector Limpia Campos

Subprograma iniciado.

Desplegando todo el vector.

AppletViewer: AppletArreglo4....

Subprograma

Numero:

Nombre:

Saldo:

120 Juan Pérez 1500.0
75 Luisa Lankenau 1000

Texto para ver Vector

Añade Muestra Vector

Limpia Vector Limpia Campos

Subprograma iniciado.

Se sabe que la manera de tomar un objeto del vector es con el método `get()`, el cual regresa un objeto, pero en este caso el objeto debe ser cambiado de tipo usando el “cast” de Java para usar un objeto de la clase `Cuenta`, entonces usamos `(Cuenta)` esto hace que lo que esta al lado derecho se le cambie el tipo a `Cuenta`, ahora ya con este nuevo objeto de la clase `Cuenta` podemos utilizar el método `toString()`, es por eso que decimos

```
( ( Cuenta) vector.get(i) ) . toString()
```

Del lado izquierdo tenemos encerrado un objeto de la clase `Cuenta`, entonces ponemos punto `(.)` y al lado derecho utilizamos el método `toString()` que en este caso será el que tomemos de la clase `Cuenta`, si no utilizamos esto de esta manera, no podremos tomar los datos del vector.

No se puede usar solamente `vector.get(i).toString()` porque el `toString()` lo tomaría de la clase `Object` que es de la cual heredan todas las clases y como este método en esa clase no se sabe que regresa, por esta situación el proceso no funcionara.

3.2. Excepciones

Las excepción se utiliza en Java cuando algo salió mal, o cuando sabemos que por algún motivo algo puede salir mal, esto tiene muchas causas y muchas alternativas de prevención, esto también lo conocemos como validación, y se pueden tomar herramientas del sistema o se pueden crear a la necesidad de las circunstancias.

Una excepción es entonces un evento que sucede con algún error, pero es algo que no debiera pasar, resulta en la lógica de la clase en la que ocurre, esta lógica puede estar destinada a la solución de una situación particular, pero nunca estaremos fuera de alcance de Los errores, como puede ser el ingreso de un valor con decimales donde solo debería ir un entero o de un carácter o símbolo donde debería ir un numero.

Normalmente Java detecta algunos errores en los que pudiera suceder una excepción y nos pide que declaremos el lugar en el que puede ocurrir, si es que no la manejaremos, entonces un programa puede estar creado para procesar las excepciones en tres formas distintas:

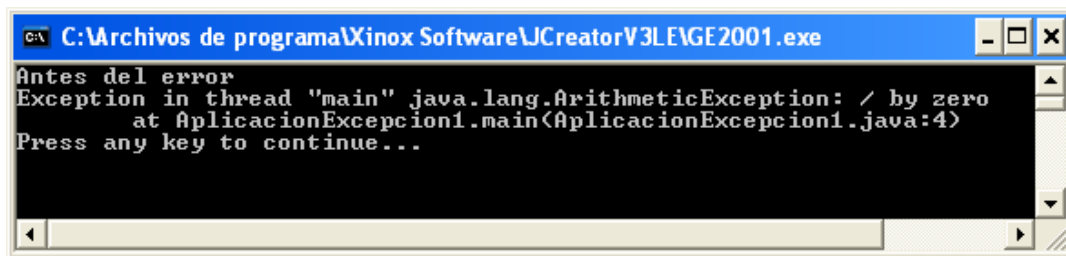
- ◆ No manejarla
- ◆ manejar la excepción cuando ocurre
- ◆ manejar la excepción en algún otro punto del programa

```
public class AplicacionExcepcion1 {
```

```
public static void main(String args[]) {  
    System.out.println("Antes del error");  
    System.out.println("Division por cero = " + 3 / 0);  
    System.out.println("Despues del error");  
}  
}
```

En el ejemplo anterior podemos ver como primero se despliega un mensaje “Antes del error”, después se despliega un cálculo en el que sabemos que habrá un error y finalmente se despliega otro mensaje “Después del error”.

Al ejecutar la aplicación anterior observamos lo que sucede:



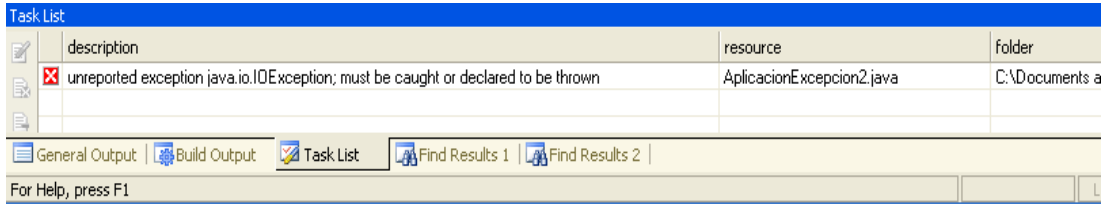
Vemos como se despliega el mensaje “Antes del error”, pero después observamos el error de ejecución denominado en Java una excepción, la Arithmetic Exception, esta excepción ocurre cuando tratamos de dividir un valor por cero, por eso mismo no se alcanza a ver el mensaje “Después del error”, ya que al haber error de ejecución, se termina de usar la aplicación.

Hay excepciones las cuales el compilador las marca desde un inicio y nos pide que nos protejamos para eso, por ejemplo la siguiente aplicación fue escrita para pedir un número al usuario y obtener el cuadrado de ese número:

```
import java.io.*;
```

```
public class AplicacionExcepcion1 {  
    public static void main(String args[]) {  
  
        BufferedReader in =  
            new BufferedReader(new InputStreamReader(System.in));  
  
        int n;  
        System.out.println("Da el numero");  
        n = Integer.parseInt(in.readLine());  
        System.out.println("El cuadrado del numero = " + n*n); }  
}
```

Pero al compilar esta aplicación observamos que no compila, el compilador manda el siguiente mensaje:



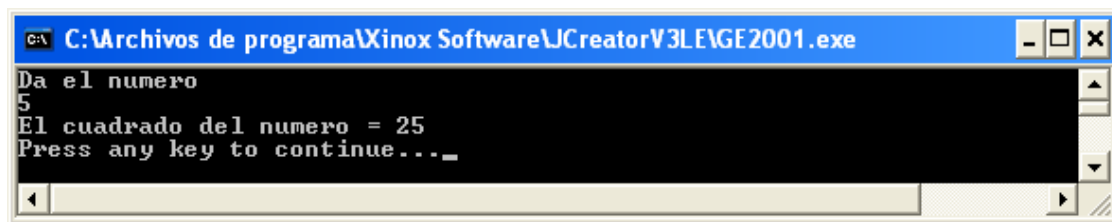
Este mensaje nos indica que al momento de tratar de leer un dato, puede haber un error de ejecución el cual debe ser capturado para que la aplicación no presente error. Pero que tipo de error puede ocurrir, pues si se supone que se pide un número y el usuario no da un número, sino un nombre, entonces habrá problemas con la aplicación y se finaliza.

La aplicación se compila entonces marcando la excepción que puede lanzar de la siguiente manera:

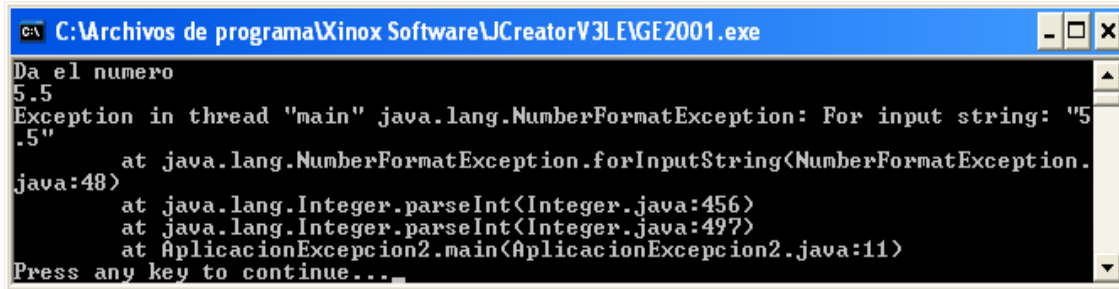
```
import java.io.*;
```

```
public class AplicacionExcepcion2 {  
    public static void main(String args[]) throws IOException {  
  
        BufferedReader in =  
            new BufferedReader(new InputStreamReader(System.in));  
  
        int n;  
        System.out.println("Da el numero");  
        n = Integer.parseInt(in.readLine());  
        System.out.println("El cuadrado del numero = " + n*n);  
    }  
}
```

Vemos que al declarar lo que está en repintado, la aplicación si compila, es decir, ahora si podemos ejecutarla tranquilamente, por ejemplo:



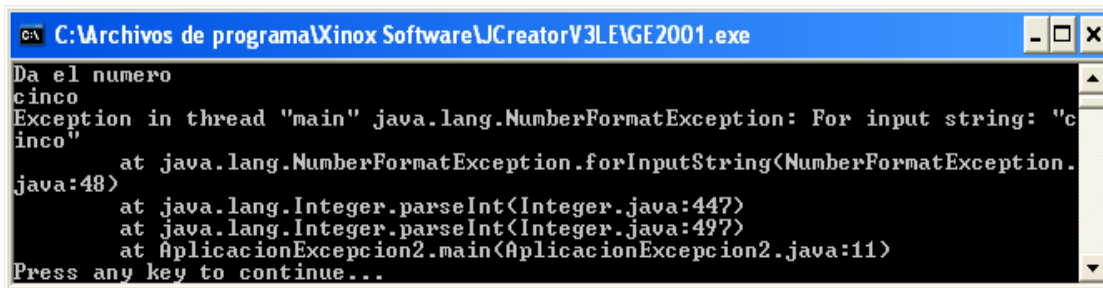
si analizamos, la aplicación todavía puede fallar, es decir, ¿qué pasa si en lugar de un número entero le damos un número real?



```
C:\Archivos de programa\Xinox Software\JCreatorV3\LENGE2001.exe
Da el numero
5.5
Exception in thread "main" java.lang.NumberFormatException: For input string: "5.5"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:456)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AplicacionExcepcion2.main(AplicacionExcepcion2.java:11)
Press any key to continue...
```

Vemos como da una excepción llamada NumberFortmatException, con el valor de 5.5

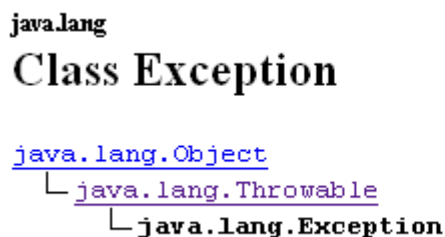
Por otra parte, ¿qué pasa si le damos letras o símbolos en lugar de números?



```
C:\Archivos de programa\Xinox Software\JCreatorV3\LENGE2001.exe
Da el numero
cinco
Exception in thread "main" java.lang.NumberFormatException: For input string: "cinco"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AplicacionExcepcion2.main(AplicacionExcepcion2.java:11)
Press any key to continue...
```

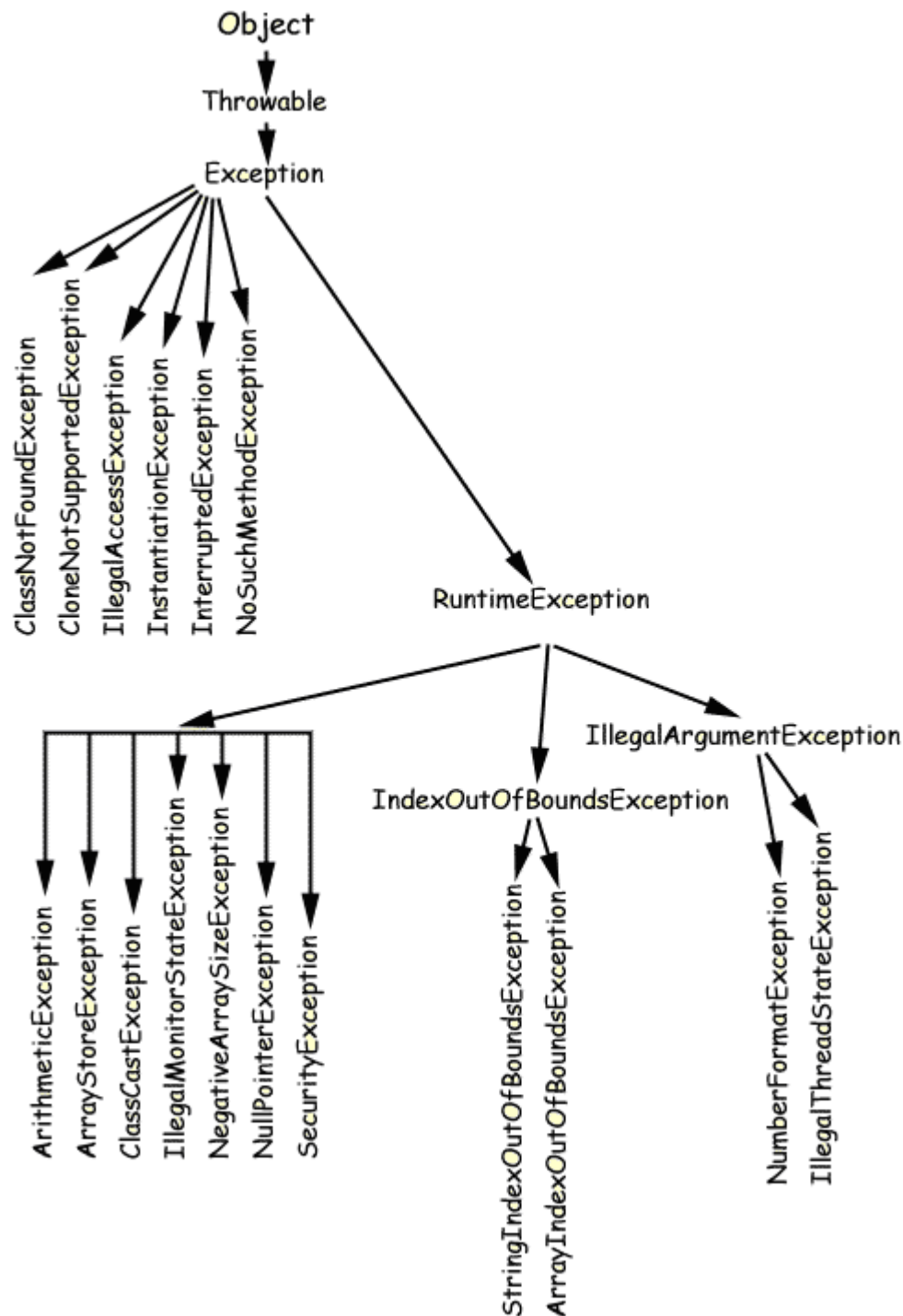
Vemos como también nos da el error de excepción NumberFormatException, con el valor cinco.

Con esto se observa que las excepciones NumberFormatException y la de ArithmeticException, las cuales son clases que están definidas en Java que hablan de errores, todas las clases de excepción se desprenden de la clase Throwable. Se pensaría que Exception es la superclase de todas las clases de excepciones, pero no es así, la super clase es Throwable. La clase Throwable tiene dos subclases: una de ellas es Error y la otra Exception. Hay en total nueve subclases de la clase Exception ya predefinidas, y cada una de ellas, a su vez, tiene numerosas subclases:



Y a su vez esta clase Throwable descende de la clase Object.

Las excepciones se pueden visualizar como se muestra a continuación:



¿Cómo es que se pueden arreglar estos errores de Ejecución?

La única manera de hacer que el software no tenga errores, es utilizando la instrucción try/catch, esta es la instrucción en Java que nos permiten detectar y corregir estos errores, observemos el formato del try/catch:

```
try
{
    instrucciones que pueden lanzar una excepción
}
catch (Excepción1 e1)
{
    instrucciones para el error 1
}
catch (Excepción2 e2)
{
    instrucciones para el error 2
}
finally // bloque opcional
{
    instrucciones que se hacen se haya encontrado error o no
}
```

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque y sigue el flujo de control por el bloque finally (si lo hay) y concluye el control de la excepción.

Si ninguna de las cláusulas catch coincide con la excepción que se ha producido, entonces se ejecutará el código de la cláusula finally. Lo que ocurre en este caso, es exactamente lo mismo que si la instrucción que lanza la excepción no se encontrase encerrada en el bloque try.

Ejemplo:

```
import java.io.*;

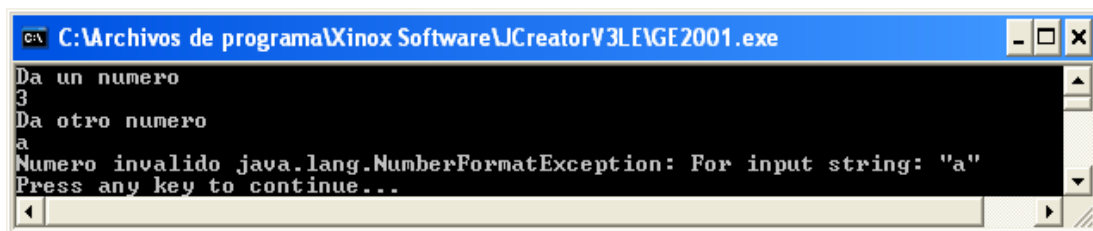
public class AplicacionExcepcion3 {
    public static void main(String args[]) throws IOException {

        try {
```

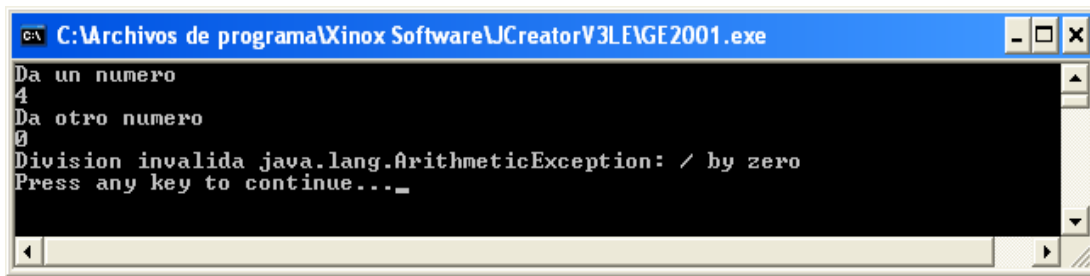
```
        BufferedReader in =  
            new BufferedReader(new InputStreamReader(System.in));  
  
        int n1, n2, n;  
        System.out.println("Da un numero");  
        n1 = Integer.parseInt(in.readLine());  
        System.out.println("Da otro numero");  
        n2 = Integer.parseInt(in.readLine());  
        System.out.println("La division = " + n1/n2);  
    }  
    catch (NumberFormatException nfe) {  
        System.out.println("Numero invalido " + nfe.toString());  
    }  
    catch (ArithmeticException ae) {  
        System.out.println("Division invalida " + ae.toString());  
    }  
}
```

Observamos de esta aplicación que las instrucciones completas están en el try, y de estas instrucciones pueden suceder dos diferentes errores de excepción, la que tengamos un número que es inválido o que tengamos una división por cero. En este ejemplo se observa que la parte de finally de la instrucción try/catch no fue usado, porque es opcional.

Pero se pudo haber utilizado, para desplegar algún mensaje de despedida o alguna otra instrucción:



Aquí observamos como al tratar de dar el segundo número obtuvimos mensaje de error.



Aquí nos damos cuenta que al dar el cero y tratar de hacer la división nos da el error de la excepción aritmética.

Si deseamos que el programa no terminara podemos utilizar un ciclo para repetir mientras que el usuario haya dado datos incorrectos, haciendo uso de una variable booleana:

```
import java.io.*;

public class AplicacionExcepcion4 {
    public static void main(String args[]) throws IOException {

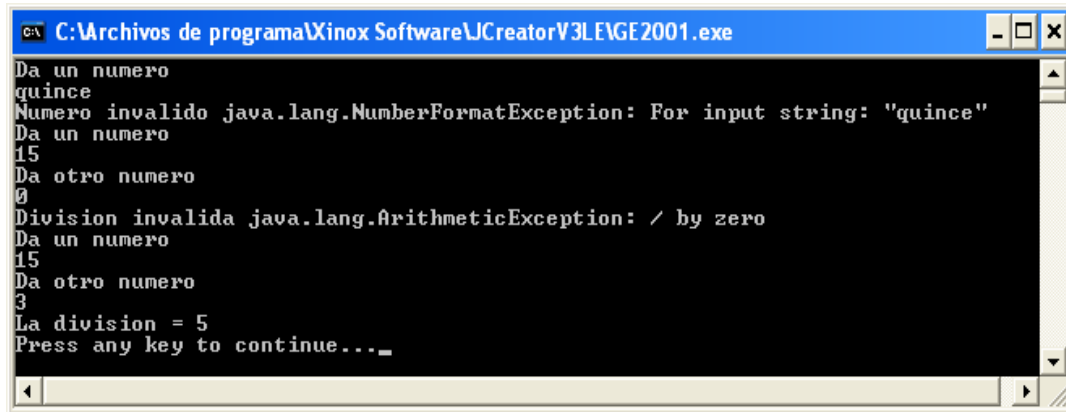
        boolean error = true; //asumimos que hay error

        while (error) {
            try {
                BufferedReader in =
                    new BufferedReader(new InputStreamReader(System.in));

                int n1, n2, n;
                System.out.println("Da un numero");
                n1 = Integer.parseInt(in.readLine());
                System.out.println("Da otro numero");
                n2 = Integer.parseInt(in.readLine());
                System.out.println("La division = " + n1/n2);
                error = false; // al llegar aqui no hubo error
            }
            catch (NumberFormatException nfe) {
                System.out.println("Numero invalido " + nfe.toString());
            }
            catch (ArithmeticException ae) {
                System.out.println("Division invalida " + ae.toString());
            }
        }
    }
}
```

```
}  
}
```

Al revisar la aplicación ejecutándola se puede ver como podemos volver a dar los datos hasta que funciona:



Revisando Excepciones Existentes

Entre algunas de las excepciones ya definidas en el sistema más conocidas están:

NullPointerException

Se produce cuando se intenta acceder a una variable o método antes de ser definido:

```
public class Ejemplo {  
    String hola;  
  
    public static void main(String[] args) {  
        System.out.println(hola);  
    }  
}
```

IncompatibleClassChangeException

El intento de cambiar una clase afectada por referencias en otros objetos, específicamente cuando esos objetos todavía no han sido recompilados.

ClassCastException

El intento de convertir un objeto a otra clase que no es válida.

y = (ClaseA)x; // donde x no puede ser de tipo ClaseA

NegativeArraySizeException

Puede ocurrir si hay un error aritmético al cambiar el tamaño de un arreglo.

OutOfMemoryException

No debería producirse nunca, pero sucede con el intento de crear un objeto con el operador new y este ha fallado por falta de memoria. Y siempre tendría que haber memoria suficiente porque el garbage collector se encarga de proporcionarla al ir liberando objetos que no se usan y devolviendo memoria al sistema.

NoClassDefFoundException

Se hizo referencia a una clase que el sistema es incapaz de encontrar.

ArrayIndexOutOfBoundsException

Es la excepción que más frecuentemente se produce. Se genera al intentar acceder a un elemento de un arreglo más allá de los límites definidos inicialmente para ese arreglo.

Ejemplo:

```
int arreglo[] = new int[5];  
arreglo[5] = 100; // no puede ser ya que solo existen del cero al cuatro
```

◆ Creación de propias Excepciones

Un desarrollador puede crear excepciones propias en Java, las cuales pueden ser utilizadas en diferentes aplicaciones.

Las Excepciones se pueden crear, es decir puede haber excepciones definidas por el usuario para que sean utilizadas en algunas aplicaciones en las que podemos incurrir en ellas, esto puede ser definido en alguna compañía, o por un mismo programador que desarrolle software en el que se desee reutilizar código.

Una manera fácil de manejar excepciones nuevas sería la de siempre hacerlas subclases de Throwable, como se muestra a continuación:

```
public class MiExcepcion extends Throwable {  
  
    public MiExcepcion() {  
        System.out.println("Se arrojó excepción mía");  
    }  
}
```

```
}
```

Con esto, las excepciones de MiExcepcion pueden ser lanzadas, declaradas y atrapadas como en el siguiente ejemplo:

```
import java.io.*;
```

```
public class AplicacionExcepcion5 {
```

```
    public static void metodo(String s) throws MiExcepcion {  
        if ("feliz".equals(s)) {  
            System.out.println("Son iguales.");  
        } else {  
            throw new MiExcepcion(); //se lanza  
        }  
    }  
}
```

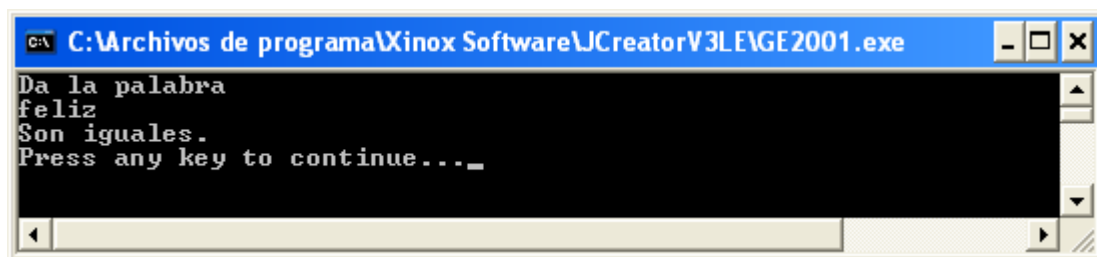
```
    public static void main(String[] args) throws IOException,  
    MiExcepcion {
```

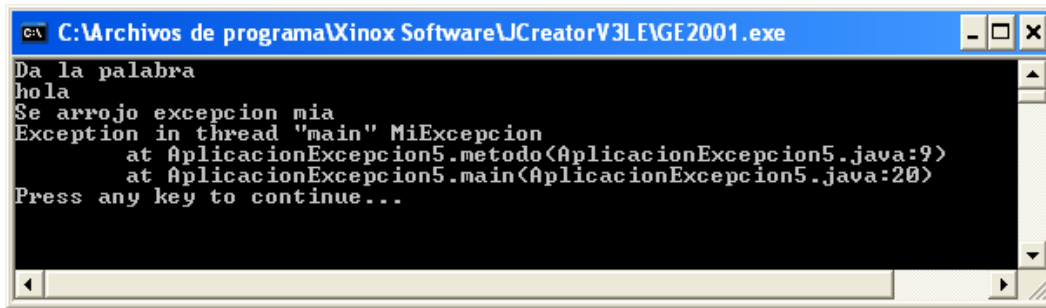
```
        BufferedReader in =  
            new BufferedReader(new InputStreamReader(System.in));
```

```
        String s;  
        System.out.println("Da la palabra");  
        s = in.readLine();  
        metodo(s);  
    }
```

```
}
```

En este ejemplo se observa que si la palabra dada por el usuario coincide con “feliz” entonces se desplegará el mensaje “son iguales”, y la aplicación terminará normalmente, por otro lado si la palabra no concuerda con “feliz”, entonces se lanzará la excepción MiExcepcion y el mensaje se arrojó excepción mía será desplegado, veamos la ejecución en los dos sentidos:





```
C:\Archivos de programa\Xinox Software\JCreatorV3\EJGE2001.exe
Da la palabra
hola
Se arrojo excepcion mia
Exception in thread "main" MiExcepcion
    at AplicacionExcepcion5.metodo(AplicacionExcepcion5.java:9)
    at AplicacionExcepcion5.main(AplicacionExcepcion5.java:20)
Press any key to continue...
```

Throw

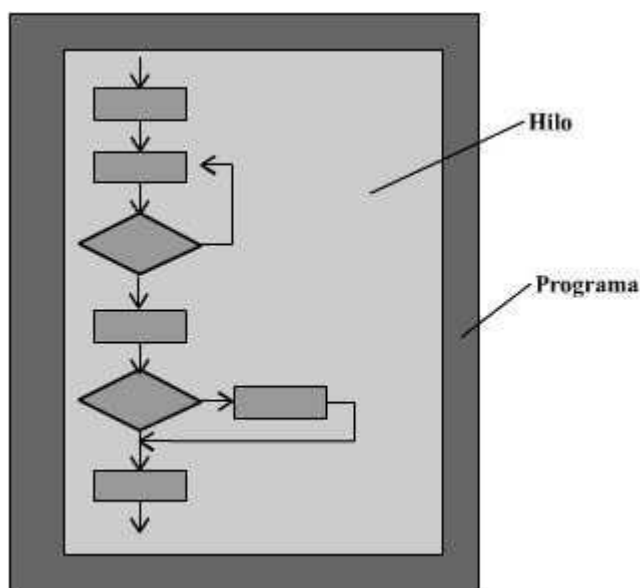
La instrucción throw puede ser utilizada para lanzar una excepción ya sea propia o del sistema, se podría decir throw ArithmeticException, así como se puso throw MiExcepcion, claro que no tendría razón que lanzáramos una excepción, lo importante es atrapar el error, es decir tener el try/catch correspondiente para cada una de las excepciones que podamos llegar a tener en nuestra aplicación o applet.

3.3. Hilos

3.3.1. ¿Qué son los hilos?

Es un proceso de ejecución también llamado proceso liviano como contraparte a multiproceso que usa programas que se denominan procesos pesados, un hilo es un flujo secuencial de instrucciones pero que está dentro de un programa, no es un programa independiente, es decir, un hilo forma parte de un programa.

- ◆ Contexto de ejecución, proceso liviano (lightweight).
- ◆ Un flujo secuencial de instrucciones dentro de un programa .



◆ Multiproceso vs Multihilos

Como se menciona un hilo es un proceso liviano, para entender esto se necesita hacer la separación en lo que es multiproceso y lo que es multihilo.

En multiproceso se tienen dos o más programas independientes que se ejecutan en forma paralela, esta ejecución paralela no es tal, en el caso de que tengamos un procesador en una máquina, cada programa tiene su propio espacio de memoria, su propio conjunto de variables, sus propios recursos, el control para ver que programa se está ejecutando en determinado momento lo tiene el sistema operativo, el programador no tiene ninguna injerencia al respecto.

En programación multihilo, que lo podemos definir como dos o más tareas ejecutándose en forma paralela dentro de un programa, estas tareas comparten los recursos del programa es decir las variables definidas para el programa son compartidas por las dos o más tareas que se están ejecutando, esto trae una responsabilidad para el programador por que el control para el acceso de recursos, lo tiene el programa y esto es un punto delicado que se vera más adelante. Así es que debe de quedar claro lo que es programación multiproceso y programación multihilo, la programación multihilo deja más responsabilidad al programador y tenemos que saber ciertas cosas para que se realice en forma adecuada.

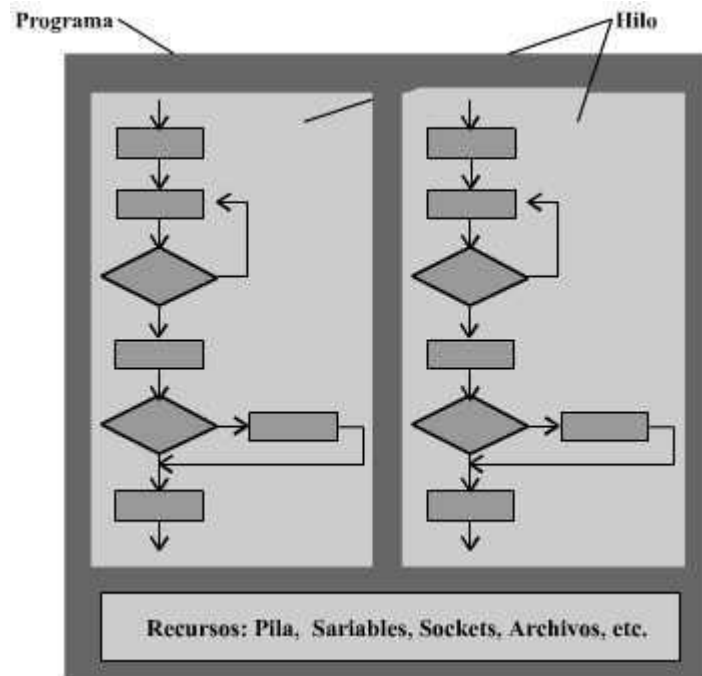
◆ Multiproceso:

- ◆ Dos o más programas (procesos) independientes ejecución en forma "paralela". Cada proceso tiene su propio espacio de memoria, su propio conjunto de variables, su propia pila, etc.

El control lo tiene el sistema operativo.

◆ **Multihilo:**

Dos o más "tareas" ejecutándose en forma "paralela" dentro de un programa.
Comparten los recursos del programa.
El control lo tiene el programa.



- ◆ ¿Por qué y cómo usar hilos?
- ◆ ¿Por qué usar hilos?
- ◆ Hacen un mejor uso de los recursos Se pueden ejecutar en el fondo (background) mientras se espera por entrada del usuario.
- ◆ Evitan bloqueo del programa
- ◆ Ejecutar tareas mientras se está leyendo de disco o se espera por una conexión de red.
- ◆ Programas más adecuados
- ◆ Desplegar barra de progreso mientras se realiza alguna tarea tardada.
- ◆ Programación más elegante y "fácil"
- ◆ Más estructuración en la programación
- ◆ "Imprescindible" para animaciones

El método run()

- ◆ Método de la clase Thread, es el que realiza el trabajo, el que se llama para que arranque un hilo su ejecución.
- ◆ Este método es el que se tiene que redefinir (override) para que tu hilo haga algo.

- ◆ El método run en la clase Thread no hace nada
- ◆ Cuando un objeto tiene en ejecución el método run(), ese hilo está compitiendo con los demás hilos activos por recursos del sistema (CPU).
- ◆ Cuando un objeto termina la ejecución de su método run(), el hilo se "muere".

- ◆ **Dos maneras de crear un hilo de ejecución**
- ◆ Extender la clase Thread, reprogramando el método run () que está vacío. La manera natural de crear hilos.
- ◆ Implementar la interfaz Runnable, programando el método run (). En caso de que los objetos que se quieren ejecutar en un hilo pertenezcan a una clase que extiende a otra

Antes de ver nuestros primeros ejemplos, veamos un poco más de teoría, hay dos maneras de crear un hilo:

Una es extender la clase Thread, como ya dijimos la clase Thread tiene un método run que es el que tienes que reprogramar,

La otra es implementar la interfaz Runnable, la interfaz Runnable tiene solamente un método definido que es el método run,

de las dos maneras que tenemos de crear un hilo, la primera, es decir extendiendo la clase Thread es la que más se utiliza, la segunda es simplemente en caso de que tengamos una clase donde queremos implementar la ejecución de hilos de esa clase ya extienda a otra, sería imposible que extendiéramos la clase Thread ya que en Java no se permite la herencia múltiple en este caso ayuda el hecho de tener una interfaz, por eso nos dan las dos maneras de crear hilos extendiendo la clase Thread o implementando la interfaz Runnable, extender la clase Thread es la manera natural de crear hilos pero en caso de que sea imprescindible podemos usar la interfaz Runnable.

◆ **Arranque de un hilo**

Por último, el arranque de hilos se realiza en dos pasos, primero creas un objeto de la clase Thread y en segundo lugar llamas al método run del objeto que acabas de crear. Un detalle es que la llamada al método run no es directa, lo que tienes que hacer es llamar al método start, este llamará al método run. Además hay algunos métodos auxiliares que nos permiten una programación más adecuada que usaremos en el ejemplo de la siguiente película, el método sleep que es de la clase Thread en una de sus versiones, permite un parámetro que es long, lo que hace esto es que el hilo el cual llama al método run deja de competir por el CPU durante la cantidad de milisegundos especificada en el parámetro, después de que ese tiempo pase, el hilo volverá a competir por el CPU, esto no garantiza que el hilo se comenzará a ejecutar inmediatamente, simplemente va a competir por el CPU con los otros hilos activos, al llamar a start,

automáticamente hay dos hilos ejecutándose, el que arrancó con start y obviamente el que ejecutó la llamada, entonces cuando utilizas hilos, automáticamente siempre tienes por lo menos dos hilos compitiendo por el CPU. Veamos un ejemplo para aterrizar todos estos conceptos que hemos visto rápidamente.

◆ **El arranque de hilos se realiza en dos pasos:**

- ◆ Crear un objeto de la clase Thread.
- ◆ Llamar al método run () del objeto creado.

La llamada al método run () no se hace directamente, sino que se llama al método start () del objeto, esto causa que la máquina virtual llame a run().

El método sleep(long x) de la clase thread, ocasiona que el hilo deje de compartir por los recursos durante x milisegundos, después de ese tiempo, comienza a pelear por recursos otra vez.

Al llamar a start, automáticamente hay dos hilos corriendo, el que se arrancó con start () y el que ejecutó la llamada.

- ◆ Ejemplo
- ◆ Primer ejemplo

```
public class Hilos1{
    public static void main(String[] args){
        Hilo1 h1 = new Hilo1("Uno");
        Hilo1 h2 = new Hilo1("Dos");
        h1.start(); h2.start();
    }
}

class Hilo1 extends Thread{
    String s;
    public Hilo1(String s){
        this.s = s;
    }
    public void run(){
        for(int i=0; i<10; i++){
            System.out.println(s+" "+i);
            try{
                sleep(Math.round(Math.random()*1000));
            }catch(InterruptedException e){}
```

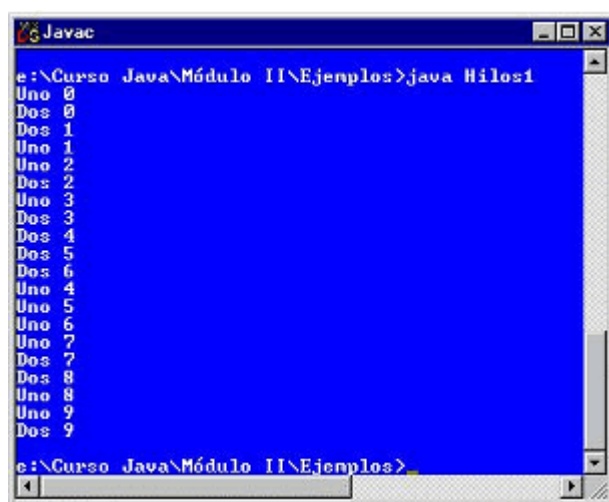
```
}  
}  
}
```

En este programa, tenemos definidas dos clases, la clase Hilos1 que es la clase principal, la cual va a crear en este caso dos hilos que arrancarán su ejecución y la clase Hilo1 que es donde programaremos el hilo a ejecutar, como puedes ver la ejecución de la clase Hilo1 extiende a Thread, como extiende a Thread, automáticamente hereda el método run pero como sabemos el método run no hace nada, ahí es donde nosotros programamos lo que queremos que se ejecute en un hilo.

El constructor de Hilo1 recibe como parámetro un string que nos va a servir para identificar que hilo se está ejecutando. Ahora veamos la programación de run, es simplemente un ciclo for que mediante una variable de control i, se ejecutará 10 veces; dentro de su ejecución simplemente va a imprimir el string que le mandamos de parámetro al constructor y después el valor de la variable i, inmediatamente después de haber ejecutado la impresión en pantalla, vamos a poner al hilo a dormir, esto lo hacemos con el método sleep, si analizas el parámetro de sleep que tiene que ser un valor de tipo long realmente es un número aleatorio entre 0 y 1 multiplicado por 1000, ¿que quiere decir esto? Que el hilo se irá a dormir por una fracción de un segundo, el valor de esta fracción será aleatorio.

Algo importante que puedes ver es que sleep está dentro de un bloque try-catch porque así lo requiere la clase Thread o mejor dicho el método sleep de la clase Thread, este método sleep arroja la interrupción InterruptedException la cual tienes que atrapar cuando utilizas sleep.

Entonces como vemos la clase Hilo1 simplemente ejecutará un ciclo durante 10 veces y dentro de ese ciclo imprimirá un letrero y dejará de competir por el CPU por una fracción de segundo.



```
Javac  
e:\Curso Java\Módulo II\Ejemplos>java Hilos1  
Uno 0  
Dos 0  
Dos 1  
Uno 1  
Uno 2  
Dos 2  
Uno 3  
Dos 3  
Dos 4  
Dos 5  
Dos 6  
Uno 4  
Uno 5  
Uno 6  
Uno 7  
Dos 7  
Dos 8  
Uno 8  
Uno 9  
Dos 9  
e:\Curso Java\Módulo II\Ejemplos>
```

- ◆ El mismo ejemplo, pero implementando Runnable

```
public class Hilos2{
    public static void main(String[] args){
        Hilo1 h1 = new Hilo1("Uno");
        Hilo1 h2 = new Hilo1("Dos");
        Thread t1 = new Thread(h1);
        Thread t2 = new Thread(h2);
        t1.start();
        t2.start();
    }
}

class Hilo1 implements Runnable{
    String s;
    public Hilo1(String s){
        this.s = s;
    }
    public void run(){
        for(int i=0; i<10; i++){
            System.out.println(s+" "+i);
            try{
                Thread.sleep(Math.round(Math.random()*1000));
            }catch(InterruptedException e){}
        }
    }
}
```

- ◆ Un ejemplo
- ◆ Primer ejemplo

```
public class Hilos1{
    public static void main(String[] args){
        Hilo1 h1 = new Hilo1("Uno");
        Hilo1 h2 = new Hilo1("Dos");
        h1.start(); h2.start();
    }
}
```

```
class Hilo1 extends Thread{  
    String s;  
  
    public Hilo1(String s){  
        this.s = s;  
    }  
    public void run(){  
        for(int i=0; i<10; i++){  
            System.out.println(s+" "+i);  
            try{  
                sleep(Math.round(Math.random()*1000));  
            }catch(InterruptedException e){}  
        }  
    }  
}
```

En este programa, tenemos definidas dos clases, la clase Hilos1 que es la clase principal, la cual va a crear en este caso dos hilos que arrancarán su ejecución y la clase Hilo1 que es donde programaremos el hilo a ejecutar, como puedes ver la ejecución de la clase Hilo1 extiende a Thread, como extiende a Thread, automáticamente hereda el método run pero como sabemos el método run no hace nada, ahí es donde nosotros programamos lo que queremos que se ejecute en un hilo.

El constructor de Hilo1 recibe como parámetro un string que nos va a servir para identificar que hilo se está ejecutando. Ahora veamos la programación de run, es simplemente un ciclo for que mediante una variable de control i, se ejecutará 10 veces; dentro de su ejecución simplemente va a imprimir el string que le mandamos de parámetro al constructor y después el valor de la variable i, inmediatamente después de haber ejecutado la impresión en pantalla, vamos a poner al hilo a dormir, esto lo hacemos con el método sleep, si analizas el parámetro de sleep que tiene que ser un valor de tipo long realmente es un número aleatorio entre 0 y 1 multiplicado por 1000, ¿que quiere decir esto? Que el hilo se irá a dormir por una fracción de un segundo, el valor de esta fracción será aleatorio.

Algo importante que puedes ver es que sleep está dentro de un bloque try-catch porque así lo requiere la clase Thread o mejor dicho el método sleep de la clase Thread, este método sleep arroja la interrupción InterruptedException la cual tienes que atrapar cuando utilizas sleep.

Entonces como vemos la clase Hilo1 simplemente ejecutará un ciclo durante 10 veces y dentro de ese ciclo imprimirá un letrero y dejará de competir por el CPU por una fracción de segundo.

Ahora vamos a la clase principal Hilos 1, aquí como puedes ver estamos creando dos ejemplares (objetos) o dos instancias de la clase Hilos1, a la primera le mandamos el parámetro string "Uno" y a la segunda el parámetro string "Dos", el hecho de crear dos instancias no es todo, además debemos de arrancar los hilos, y esto se hace mandando ejecutar el método run, pero como ya vimos el método run no se ejecuta directamente, tienes que llamar a start que son las dos últimas instrucciones del método main, cuando ves h1.start() y h2.start() en ese momento comienza la ejecución de dos hilos, técnicamente lo que comienza es una competencia por el CPU. En la parte inferior de la filmina puedes ver la salida de una de tantas ejecuciones de este programa, como puedes ver no hay un orden en el cual se ejecutan los hilos, ambos están compitiendo por el CPU o están en estado detenidos por la llamada sleep o un tiempo aleatorio.



```
Java
e:\Curso Java\Módulo II\Ejemplos>java Hilos1
Uno 0
Dos 0
Dos 1
Uno 1
Uno 2
Dos 2
Uno 3
Dos 3
Dos 4
Dos 5
Dos 6
Uno 4
Uno 5
Uno 6
Uno 7
Dos 7
Dos 8
Uno 8
Uno 9
Dos 9
e:\Curso Java\Módulo II\Ejemplos>
```

◆ El mismo ejemplo, pero implementando Runnable

```
public class Hilos2{
    public static void main(String[] args){
        Hilo1 h1 = new Hilo1("Uno");
        Hilo1 h2 = new Hilo1("Dos");
        Thread t1 = new Thread(h1);
        Thread t2 = new Thread(h2);
        t1.start();
        t2.start();
    }
}

class Hilo1 implements Runnable{
```

```
String s;  
  
public Hilo1(String s){  
    this.s = s;  
}  
public void run(){  
    for(int i=0; i<10; i++){  
        System.out.println(s+" "+i);  
        try{  
            Thread.sleep(Math.round(Math.random()*1000));  
        }catch(InterruptedException e){}  
    }  
}
```

Ahora veamos el mismo ejemplo pero implementando runnable es decir por alguna circunstancia, nuestra clase Hilo1 no va poder extender a Thread en ese caso necesitamos que implemente runnable y cuando implementamos una interfaz nos comprometemos a programar todos los métodos de esa interfaz, en este caso la interfaz runnable solamente tiene run.

Si te fijas, la clase Hilo1 es idéntica a la clase Hilo1 del ejemplo pasado simplemente hemos sustituido extends por implement runnable, ahora veamos la clase principal que se llama Hilos2, también es muy similar a la anterior, el único detalle es que cuando creamos una instancia o un ejemplar de la clase Hilo1 todavía no hemos creado un thread que es el primer paso antes de mandar a ejecutar el método run entonces las dos instrucciones que continúan, son las que se encargan de crear instancias de la clase Thread , en el ejemplo pasado eso era automático porque al crear una instancia de Hilo1 automáticamente creabas una instancia de Thread, el otro detalle es que cuando creas una instancia de Thread tienes que asociarle un hilo o una clase que implemente runnable, en este caso los dos objetos, h1 y h2 son los que se asocian a los threads t1 y t2 respectivamente.

- ◆ Y por último para arrancar los hilos simplemente mandas llamar start en cada uno de los ejemplares empleados anteriormente, y como puedes ver en la parte inferior de la filmina está una salida de la ejecución de este programa que es similar a la anterior obviamente el orden de ejecución va a ser diferente por que ambos hilos compiten por recursos y no asegura un orden.
- ◆ Algo importante del archivo Hilos1.java y el archivo Hilos2.java se encuentran en los archivos auxiliares que bajarás de la red (ver ejercicio), es que cuando ejecutes Hilos1 compílalo previamente y cuando ejecutes Hilos2 vuelve a compilar los dos, la razón es que ambos archivos tienen definida una clase Hilo1, lo que pasaría si no recompilas Hilos2

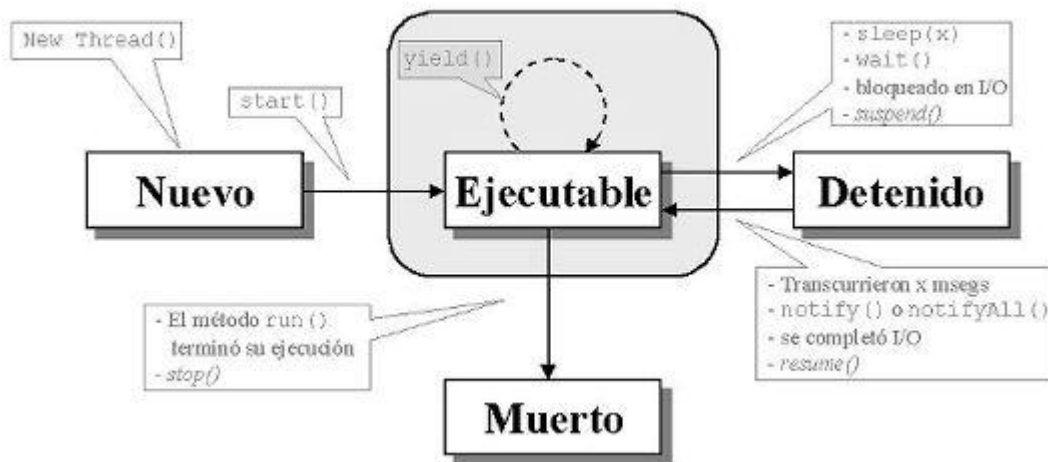
antes de ejecutarlo y previamente compilaste Hilos1, la clase que se compiló es Hilo1 pero que extiende a Thread y no Hilo1 que implementa runnable, esto está hecho en forma intencional para que veas que se generan dos clases en el momento de compilación a pesar de que es solamente un archivo fuente. Así es que por favor antes de ejecutar cada uno de los programas compílalos.



```
Java
e:\Curso Java\Módulo II\Ejemplos>java Hilos2
Uno 0
Des 0
Uno 1
Des 1
Uno 2
Des 2
Uno 3
Des 3
Uno 4
Des 4
Uno 5
Des 5
Uno 6
Des 6
Uno 7
Des 7
Uno 8
Des 8
Uno 9
Des 9
e:\Curso Java\Módulo II\Ejemplos>
```

◆ Estado de un Hilo

Un hilo puede estar en uno de cuatro estados, cuando acabas de crearlo es un hilo nuevo pero que no está en ejecución ni está compitiendo por recursos del CPU, el siguiente estado se denomina ejecutable, no en ejecución, ejecutable indica que el hilo está compitiendo por el CPU contra los otros hilos activos, solamente un hilo estará en ejecución pero puede haber muchos hilos en estado ejecutable que en cualquier momento pueden tomar posesión del CPU, el tercer estado posible es detenido, ahí simplemente el hilo no está compitiendo por el CPU y esto lo podíamos ver en el ejemplo pasado cuando ejecutábamos el método sleep y por último el estado muerto, esto sucede cuando el hilo termina de ejecutar el método run.



En la figura puedes ver gráficamente los cuatro estados y la manera de irse a cada uno de ellos. El estado nuevo es cuando creas una instancia o un ejemplar de la clase Thread o de alguna clase que extienda a Thread. Para que el hilo que creaste pase a estado ejecutable necesitas llamar al método start y el hilo estará en estado ejecutable que es compitiendo por el CPU, hasta que sucedan algunas cosas, por un lado si mandas llamar al método sleep automáticamente se irá al estado de detenido, cuando sale de este estado de detenido es cuando hayan transcurrido la cantidad de milisegundos que especificaste en el método sleep en ese momento el hilo vuelve a ser ejecutable, es decir sigue compitiendo por el CPU con los otros hilos activos, además si llamas al método wait se irá a estado de detenido y saldrá de este si ejecutas notify o notifyAll (esto no lo cubriremos en esta lección), otra causa es que el hilo haga una operación de entrada-salida en ese momento se bloquea hasta que se complete la operación de entrada salida y por último es llamar al método suspend en ese momento el hilo se va a estado de detenido y se queda ahí hasta que se haga una llamada al método resume éstos dos métodos están puestos en cursivas porque están desaprobados, es decir, no es recomendado su uso, la especificación técnica del por qué está fuera de los alcances de este curso pero simplemente es necesario tener en cuenta que la utilización de estos dos métodos está desaprobada y no es conveniente hacerla.

El cuarto estado, cuando nuestro hilo se muere, llegamos a él ejecutando el método run y terminando su ejecución, la otra causa puede ser que ejecutemos el método stop pero este método también está desaprobado así es que no lo utilizaremos, entonces la manera de terminar con nuestros hilos o matarlos es, simplemente que termine la ejecución del método run en forma natural.

En los dos ejemplos anteriores nuestro método run era simplemente un for que se ejecutaba 10 veces, al terminar la ejecución del run automáticamente el hilo se moría. Y por último, cuando se está en estado ejecutable y se ejecuta el método yield el hilo cede el CPU a algún hilo de igual prioridad y comienza a competir por el CPU con los otros hilos, entonces el hecho de llamar yield

es simplemente dar la posibilidad a que otros hilos de igual prioridad puedan ejecutarse y que un hilo no adquiriera el CPU por tiempo indefinido.

- ◆ new Thread (...)
- ◆ Creado, pero todavía no comienza ejecución
- ◆ start ()

- ◆ crea los recursos necesarios para su ejecución, lo deja listo para que la máquina virtual lo ponga a correr llamando al método run (), compete por los recursos del sistema.
- ◆ Bloqueado por I/O, sleep () o wait ()

- ◆ La máquina virtual puede seleccionar otro hilo para ejecución, para salir de este estado la acción debe de corresponder la acción que la bloqueó.
- ◆ **Muerto**

Terminó run (), una vez muerto un hilo no puede volver a ejecución

Resumiendo, tenemos cuatro estados posibles de un hilo, nuevo, recién creado pero no comienza su ejecución después de ejecutar start se crean todos los recursos necesarios para hacer la ejecución, ejecutable comienza a competir por el CPU con otros hilos, tercer estado, detenido llegamos a él cuando el hilo se bloquea con una operación de entrada salida se invoca al método sleep o el método wait en ese momento la máquina virtual selecciona otro hilo para ejecutarlo y por último el estado muerto que para nosotros simplemente será el término de la ejecución del método run, es importante saber que una vez que un hilo está muerto no se puede volver a ejecutar, es decir no puede mandar llamar start otra vez. ¿Qué es lo que tienes que hacer?, es crear otra instancia de tu clase para que se cree un nuevo hilo.

EJERCICIOS DE AUTOEVALUACIÓN

Crear un aplicativo que lea un valor y determine si el valor es primo o no, utilizar excepciones para el ingreso y validación de los procesos

Crear un Aplicativo de lea para un rango de valores y determine cuales son primos y cuales no, utilizar hilos para realizar el proceso en menor tiempo.

Prueba Final

Crear un aplicativo que lea cedula, nombre y salario, determinar hilos y excepciones para determinar que el aplicativo cumple unas condiciones mínimas

Actividad

Crear un aplicativo que lea datos para un vector, crear excepciones propias que permitan determinar la validación correcta del aplicativo.

4. ALMACENAMIENTO PERMANENTE

OBJETIVO GENERAL

Generar procesos de identificación y creación que permitan el almacenamiento permanente de datos, aplicados a los archivos y las Bases de Datos, esto con el fin de realizar procesos mas acordes con la industria informática, de este modo se tendrá un gran camino recorrido que permitirá un mayor alcance y menos limitaciones de acceder a la información previamente almacenada.

OBJETIVOS ESPECÍFICOS

- ◆ Conocer los aspectos más relevantes del manejo de archivos, donde podremos consultar, visualizar y manipular la información.
- ◆ Dar a conocer las características mas importantes de la configuración y aplicación de las BD

Prueba Inicial

Crear un modelo de datos que identifique la información que se requiere para realizar una matricula de una institución académica, realizar los procesos que se deben tener presente para la sistematización de este.

4.1. Archivos

4.1.1. Flujos

Un flujo es el sistema que nos ayuda a realizar la comunicación en Java, utilizando el paquete ya implementado java.io cuyo fin es guardar y tomar la información en cada uno de los diversos dispositivos de almacenamiento.

Se puede decir que un flujo es recipiente en donde podemos leer o escribir bytes. De un extremo nosotros ponemos algo y del otro extremo del tubo puede estar cualquier dispositivo un teclado, un monitor, un archivo, un objeto de Java, etc.

Todos los flujos que aparecen en Java englobados en el paquete `java.io`, pertenecen a dos clases abstractas comunes: `java.io.InputStream` para los flujos de Entrada (aquellos de los que podemos leer) y `java.io.OutputStream` para los flujos de salida (aquellos en los que podemos escribir).

Java tiene un conjunto de Clases y métodos ya establecidos para captar la información de los flujos de entrada y de salida por los dispositivos estándar.

En el caso de los flujos de entrada tiene `System.in`, el cual suele recibir los datos de teclado, utilizando el método `read()` para leer los caracteres.

Para los flujos de salida se utiliza `System.out` y los datos se envían a pantalla, utilizando el método `print()` o `println()` cuya diferencia es que con el `print` la información se manda tal cual al buffer de salida, sin saltar de línea, pero debemos utilizar el método `flush()` para saltar de línea, en cambio con el `println` se manda el buffer de salida directamente a la pantalla y se salta de línea.

Existe un flujo de datos para los errores y éste es el `System.err`, el cual envía la salida también directamente a la pantalla, pero si se desea se puede redireccionar, de manera que se separe el dispositivo de salida del dispositivo de la salida con error.

La manera en la que en Java se toma la información de entrada es asociando al flujo estándar de entrada la creación de un objeto de `InputStreamReader`, el cual es utilizado a su vez para la creación de un objeto de la clase `BufferedReader`, de esta manera lo que viene del teclado se envuelve entre clases para de pasar de bits a bytes y luego a datos que pueden ser leídos.

Al utilizar el objeto de la clase `BufferedReader` tenemos el método `readLine()` el cual lee un conjunto de bytes del buffer de entrada hasta detectar el fin de línea.

La manera en la que Java saca la información a salida es utilizando la clase `PrintWriter` tomando el objeto de salida `System.out`, para crear el objeto de la clase `PrintWriter`. Los métodos que se utilizan son el `print` y `println`.

Las clases de Streams, Readers y Writers en java ven la entrada y salida como una secuencia de bytes. Los streams de bajo nivel más comunes son:

- ◆ `FileInputStream(String pathname)`
- ◆ `FileInputStream(File file)`
- ◆ `FileOutputStream(String pathname)`
- ◆ `FileOutputStream(File file)`

Una vez que un stream de entrada ha sido construido, pueden llamarse métodos para leer un simple byte, o una porción de un arreglo de bytes.

A continuación un ejemplo que lee bytes de un archivo.

```
byte b;  
byte bytes[] = new byte[100];  
byte morebytes[] = new byte[50];  
try {  
    FileInputStream fis = new FileInputStream("nombre_del_archivo");  
    b=(byte)fis.read(); // lee un byte  
    fis.read(bytes); //llena el arreglo  
    fis.read(morebytes, 0, 20); //lee 20 elementos  
} catch (IOException e) {}
```

Es conveniente leer bytes de un dispositivo de entrada y escribir a un dispositivo de salida. Sin embargo, normalmente lo que se desea leer y escribir no son bytes sino información tal como enteros o cadenas de caracteres (int o String), etc.

Java cuenta con manejo de streams de alto nivel. Los más comunes son:

```
DataInputStream(InputStream instream)  
DataOutputStream(OutputStream outstream)
```

Un ejemplo de cómo grabar en un archivo utilizando un DataOutputStream sería:

```
try {  
    //Construye la cadena de salida  
    FileOutputStream fos = new FileOutputStream("nombre_archivo");  
    DataOutputStream dos = new DataOutputStream(fos);  
  
    //lee  
    dos.writeDouble(123.456);  
    dos.writeInt(55);  
    dos.writeUTF("Mary tiene un pequeño borreguito");  
  
    //cierra  
    dos.close();  
    fos.close();  
} catch (IOException e) {}
```

Un ejemplo que muestra como leer los datos que el anterior ejemplo dejó, utilizando un `DataInputStream` sería:

```
try {  
    //Construye la cadena de entrada  
    FileInputStream fis = new FileInputStream("nombre_archivo");  
    DataInputStream dis = new DataInputStream(fis);  
  
    //lee  
    double d = dis.readDouble();  
    int i = dis.readInt();  
    String s = dis.readUTF();  
  
    //cierra  
    dis.close();  
    fis.close();  
} catch (IOException e) {}
```

Puedes probar implementar estas instrucciones cada conjunto en una diferente aplicación y ver lo que hacen, para que percibas como es que se genera el archivo y luego lo leas desde la otra aplicación.

Otros manejadores de streams de alto nivel:

`BufferedInputStream` y `BufferedOutputStream`, manejan internamente un buffer de manera que los bytes puedan ser escritos y leídos en bloques, optimizando el proceso de entrada/salida.

`BufferedReader(Reader reader)`

`PrintStream`: Esta clase maneja texto o primitivas de datos. Las primitivas de datos se convierten a representaciones de carácter. El `System.out` y `System.err` que se utiliza en las aplicaciones de consola son ejemplos de esta clase.

`PrintStream(OutputStream out)`

Al igual que los streams de entrada y salida: Los readers y writers de bajo nivel se comunican con dispositivos, mientras que los de alto nivel se comunican con los de bajo nivel. La diferencia es que los readers y writers se orientan exclusivamente al manejo de caracteres Unicode.

Un ejemplo de reader de bajo nivel es el `FileReader`:

◆ `FileReader(String pathname)`

◆ FileReader(File file)

Algunos métodos para lectura que provee la superclase Reader son:
int read() throws IOException.

Regresa el siguiente valor entero del carácter (16 bits: 0 to 65535), -1 si no hay más caracteres.
int read(char[] cbuf) throws IOException.
Llena el arreglo con los caracteres leídos, regresa el número de caracteres que se leyeron.

Un ejemplo de writer de bajo nivel es el FileWriter:

- ◆ FileWriter(String pathname)
- ◆ FileWriter(File file)

BufferedReader y BufferedWriter. Estas clases tienen buffers internos de manera que los datos pueden ser leídos o escritos en bloques. Son similares a BufferedInputStream y BufferedOutputStream, pero mientras éstos manejan bytes, BufferedReader y BufferedWriter se maneja con caracteres.

Constructores para estas clases:

BufferedReader(Reader in)
BufferedWriter(Writer out)

InputStreamReader y OutputStreamWriter. Estas clases convierten entre streams de bytes y secuencias de caracteres Unicode. Provee el puente entre la conversión de los bytes y un sistema (charset). Ejemplo:

```
BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
```

PrintWriter. Similar a PrintStream, pero escribe caracteres en vez de bytes. Ejemplo.
PrintWriter stdErr = new PrintWriter(System.out,true);

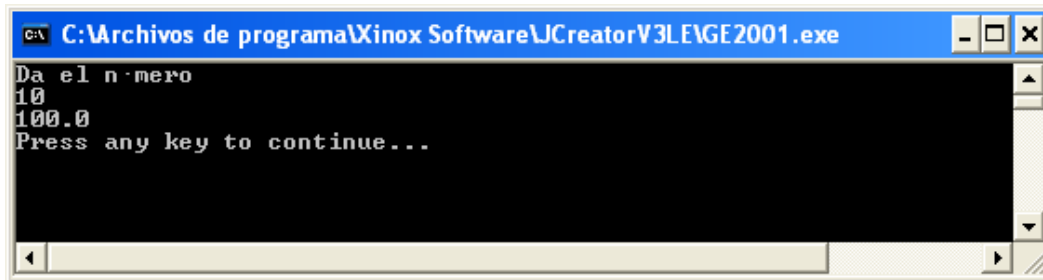
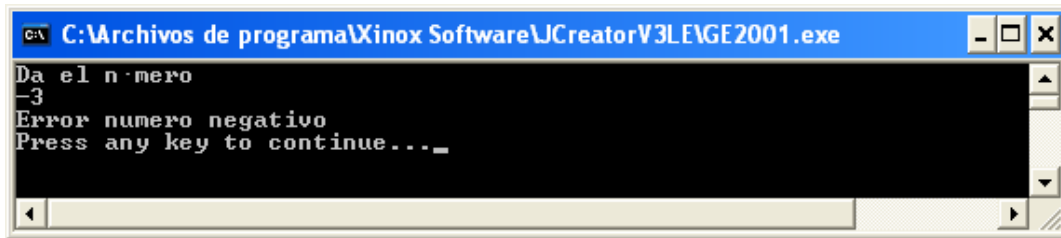
A continuación tenemos un ejemplo muy sencillo:

```
import java.io.*;
```

```
public class AplicacionStreams {  
    public static void main(String[] args) throws IOException {  
        BufferedReader ent = new BufferedReader(new InputStreamReader(System.in));  
        PrintWriter sal = new PrintWriter(System.out, true);
```

```
PrintWriter salErr = new      PrintWriter(System.err, true);
salErr.println("Da el número");
int numero = Integer.parseInt(ent.readLine());
if (numero < 0) {
    salErr.println("Error numero negativo");
}
else {
    sal.println("" + (Math.pow(numero,2)));
}
}
```

La aplicación se pudiera visualizar como los ejemplos siguientes:



Un constructor para FileWriter:

```
FileWriter(String fileName)
```

Se usan archivos para almacenar datos que se quieren conservar después de que el programa que los generó ha terminado su ejecución.

Los archivos se guardan en dispositivos de almacenamiento secundario (discos ó cintas magnéticas).

El uso de archivos es una de las capacidades más importantes que debe tener un lenguaje para apoyar aplicaciones comerciales que suelen manejar grandes cantidades de datos y que requieren que dichos datos sean persistentes.

En Java la entrada/salida (I/O) de datos se maneja a través de streams.

La entrada y salida puede ser de archivos, y de teclado y pantalla. Se usan streams en ambos casos.

Un stream es un objeto que puede:

- ◆ Tomar datos de una fuente (archivo o teclado) y llevarlos a un programa.
- ◆ Tomar datos del programa y entregarlos a un destino (archivo o pantalla)

Ya se mencionó que para leer/escribir de consola y de archivos se usan streams. Para declarar la variable, crear el objeto y abrir el archivo utilizamos:

- ◆ `BufferedReader fileIn = new BufferedReader(new FileReader(nomArch));`
- ◆ `PrintWriter fileOut = new PrintWriter(new FileWriter(nomArch));`
- ◆ `PrintWriter fileApnd = new PrintWriter(new FileWriter(nomArch, true));`

Nota que en un solo paso estás haciendo las 3 acciones mencionadas.

Observa también que los objetos que estás creando son exactamente de las mismas clases que los que usamos para leer/escribir de consola.

Dado que para escribir en el archivo se utiliza la clase `PrintWriter`, se escribe en el archivo de la misma forma que se hace a pantalla:

```
fileOut.println(unString);  
fileOut.print(unString);  
fileOut.flush();
```

Estas funciones trabajan de la forma que ya conocemos.

Dado que para leer del archivo se utiliza la clase `BufferedReader`, se lee del archivo de la misma forma que se hace del teclado:

```
fileIn.readLine();
```

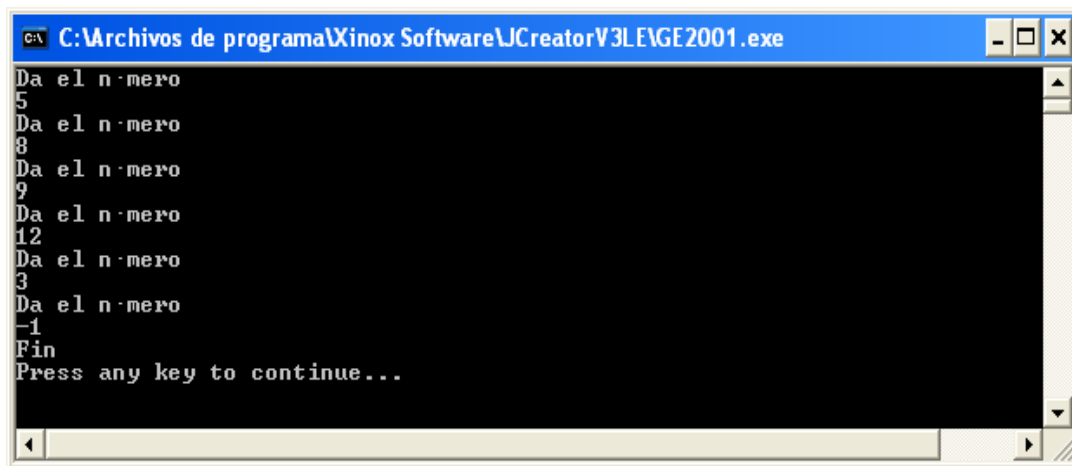
`readLine()` lee una línea del stream de entrada y regresa esa línea.

A continuación encontrarás un ejemplo que te muestra como crear un archivo de texto a partir de la consola:

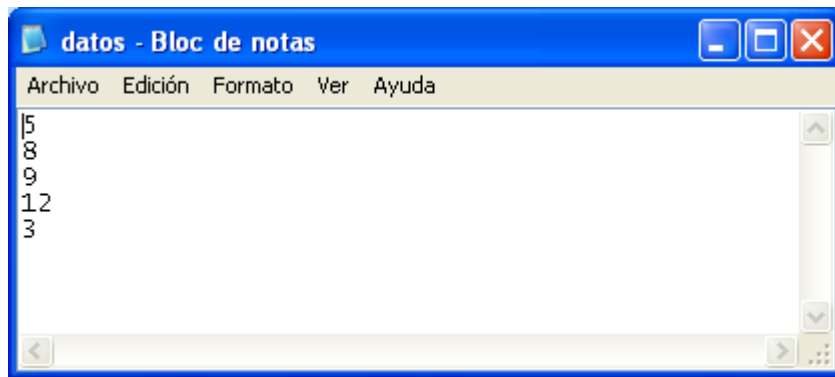
```
import java.io.*;

public class AplicacionFiles1 {
    public static void main(String[] args) throws IOException {
        BufferedReader ent = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter sal = new PrintWriter(new FileWriter("datos.txt"));
        PrintWriter salErr = new PrintWriter(System.err, true);
        salErr.println("Da el número");
        int numero = Integer.parseInt(ent.readLine());
        while (numero > 0) {
            sal.println("" + numero);
            salErr.println("Da el número");
            numero = Integer.parseInt(ent.readLine());
        }
        salErr.println("Fin");
        sal.close();
    }
}
```

La ejecución de esta aplicación pudiera ser:



El archivo generado se vería así:



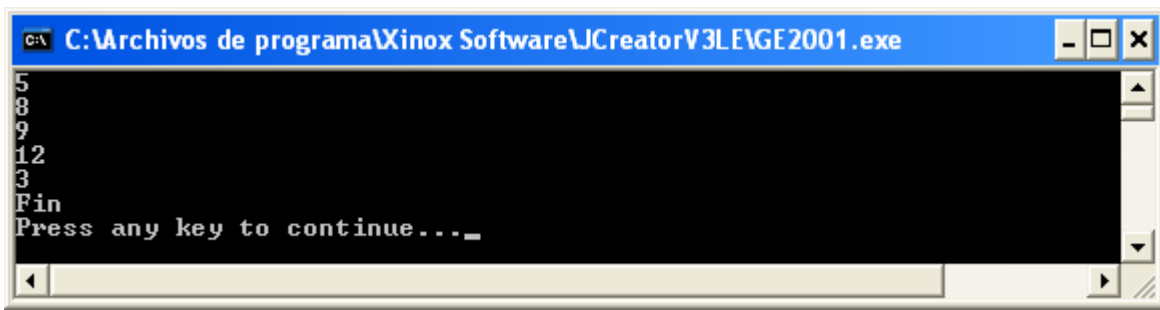
Ya que fue generado con extensión txt se abre desde el Block de Notas.

Una manera de leer el archivo es tomando el `BufferedReader` con un objeto de `FileReader` y entonces pasarle como parámetro el nombre del archivo, y en un ciclo leer hasta que no encontremos ningún valor más a leer, como el ejemplo:

```
import java.io.*;
```

```
public class AplicacionFiles2 {  
    public static void main(String[] args) throws IOException {  
        BufferedReader ent = new BufferedReader(new FileReader("datos.txt"));  
        PrintWriter sal = new    PrintWriter(System.out,true);  
        PrintWriter salErr = new    PrintWriter(System.err, true);  
        String linea = ent.readLine();  
        int numero;  
        while (linea != null) {  
            numero = Integer.parseInt(linea);  
            sal.println("" + numero);  
            linea = ent.readLine();  
        }  
        salErr.println("Fin");  
        ent.close();  
    }  
}
```

El cual al ejecutar se vería así:



◆ StringTokenizer y StreamTokenizer

Pertenece a `java.util.StringTokenizer`

Tokenizing es el proceso de romper un `String` en pequeñas piezas llamadas tokens.

Estos tokens están delimitados por uno o varios caracteres.

Los delimitadores más comunes son: el espacio, la coma y el guión bajo. Pero cualquier carácter puede ser delimitador.

Comúnmente lo utilizamos cuando tenemos un archivo y sacamos líneas de datos de él.

Constructor Summary	
<code>StringTokenizer(String str)</code>	Constructs a string tokenizer for the specified string.
<code>StringTokenizer(String str, String delim)</code>	Constructs a string tokenizer for the specified string.
<code>StringTokenizer(String str, String delim, boolean returnDelims)</code>	Constructs a string tokenizer for the specified string.

Sus métodos son:

Method Summary	
int	<code>countTokens()</code> Calculates the number of times that this tokenizer's <code>nextToken</code> method can be called before it generates an exception.
boolean	<code>hasMoreElements()</code> Returns the same value as the <code>hasMoreTokens</code> method.
boolean	<code>hasMoreTokens()</code> Tests if there are more tokens available from this tokenizer's string.
Object	<code>nextElement()</code> Returns the same value as the <code>nextToken</code> method, except that its declared return value is <code>Object</code> rather than <code>String</code> .
String	<code>nextToken()</code> Returns the next token from this string tokenizer.
String	<code>nextToken(String delim)</code> Returns the next token in this string tokenizer's string.

Al crear un objeto de la clase `StringTokenizer`, si no se define el delimitador, este es como un espacio, o a menos que con otro constructor se defina el delimitador, esta clase nos va a ayudar a obtener datos dentro de una línea de datos o un `String`, como se ve a continuación.

Considera la siguiente aplicación con un Área de Texto donde se muestran los resultados.

```
String s = "This string has five tokens";
StringTokenizer st = new StringTokenizer(s);

int i = 1;
while (st.hasMoreTokens()) {
    textAreaSalida.append("Token #" + i + ": " +
st.nextToken());
    ++i;
}
```

El código anterior considera el espacio como delimitador default. Y desplegará lo siguiente:

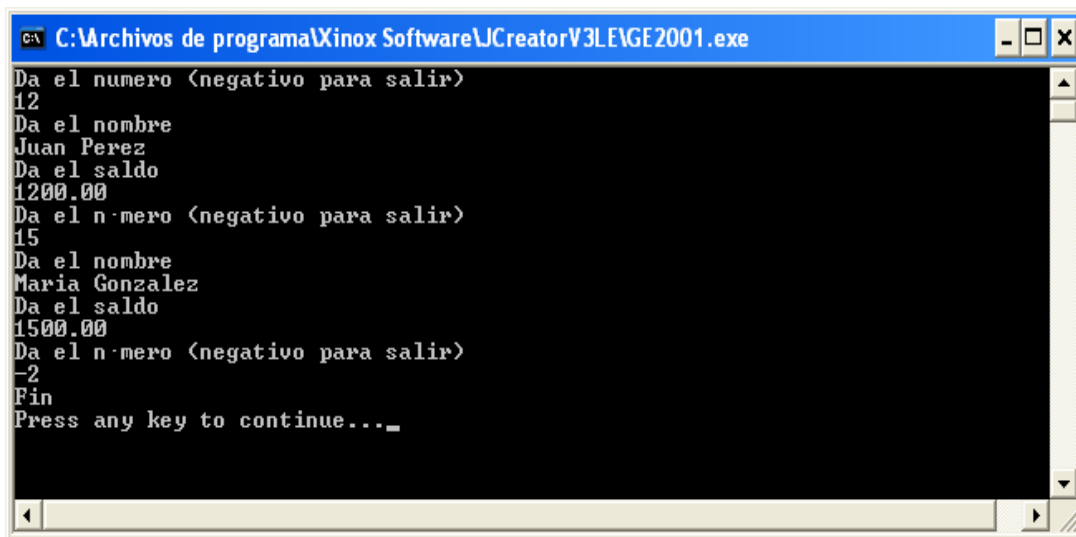
Token #1: This
Token #2: string
Token #3: has
Token #4: five
Token #5: tokens

A continuación se da un ejemplo de grabar un archivo con varios datos, pero usando un delimitador al grabar, esta aplicación podría ser como la siguiente:

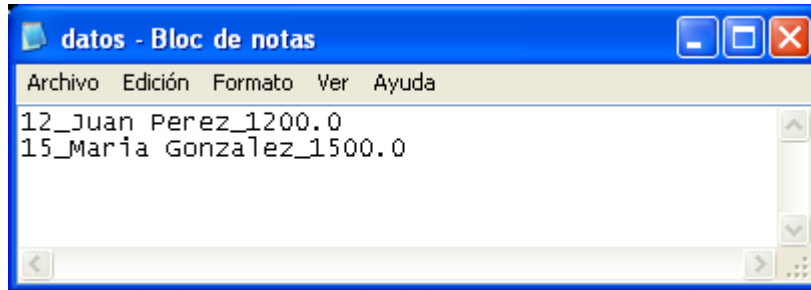
```
import java.io.*;
```

```
public class AplicacionFiles3 {  
    public static void main(String[] args) throws IOException {  
        BufferedReader ent = new BufferedReader(new InputStreamReader(System.in));  
        PrintWriter sal = new PrintWriter(new FileWriter("datos.txt"));  
        PrintWriter salErr = new PrintWriter(System.err, true);  
        salErr.println("Da el numero (negativo para salir");  
        int numero = Integer.parseInt(ent.readLine());  
        String nombre;  
        double saldo;  
        while (numero > 0) {  
            salErr.println("Da el nombre");  
            nombre = ent.readLine();  
            salErr.println("Da el saldo");  
            saldo = Double.parseDouble(ent.readLine());  
            // se graba la linea pero usando el delimitador guion bajo  
            sal.println("'" + numero + "_" + nombre + "_" + saldo);  
            salErr.println("Da el número (negativo para salir");  
            numero = Integer.parseInt(ent.readLine());  
        }  
        salErr.println("Fin");  
        sal.close();  
    }  
}
```

La ejecución de la aplicación podría ser:



Donde el archivo que se grabó sería:



Ahora la manera de leer estos datos es utilizando el StringTokenizer después de leer el objeto de la clase String, tomando el “_” como delimitador. La siguiente aplicación toma línea por línea del archivo y extrae los tokens que son número, nombre y saldo:

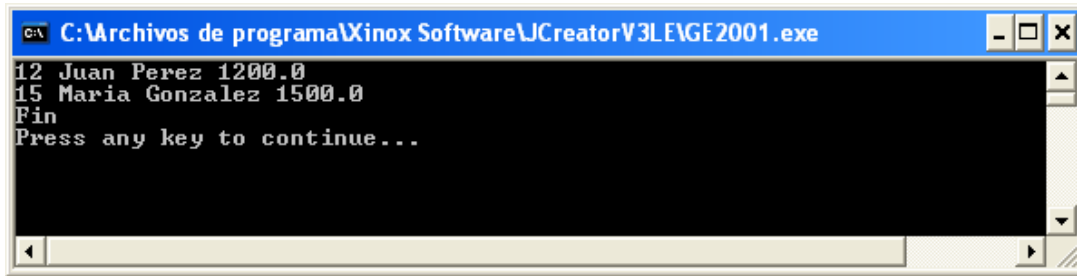
```
import java.io.*;
import java.util.*;

public class AplicacionFiles4 {
    public static void main(String[] args) throws IOException {
        BufferedReader ent = new BufferedReader(new FileReader("datos.txt"));
        PrintWriter sal = new PrintWriter(System.out,true);
        PrintWriter salErr = new PrintWriter(System.err, true);
        int numero;
        String nombre;
        double saldo;
        StringTokenizer stt;

        String linea = ent.readLine();
        while (linea != null) {
            // se tokeniza la linea leida del archivo
            stt = new StringTokenizer(linea, "_");
            // se extrae cada uno de los tokens de la línea
            numero = Integer.parseInt(stt.nextToken());
            nombre = stt.nextToken();
            saldo = Double.parseDouble(stt.nextToken());
            sal.println("" + numero + " " + nombre + " " + saldo);
            linea = ent.readLine();
        }
        salErr.println("Fin");
    }
}
```

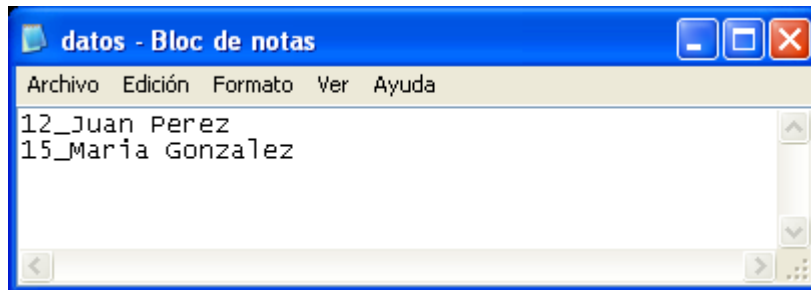
```
        ent.close();  
    }  
}
```

La ejecución de esta aplicación sería:

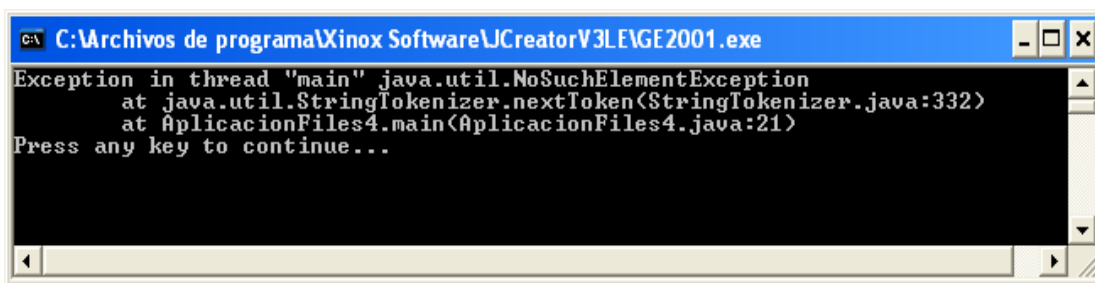


```
C:\Archivos de programa\Xinox Software\JCreatorV3\JEGE2001.exe  
12 Juan Perez 1200.0  
15 Maria Gonzalez 1500.0  
Fin  
Press any key to continue...
```

En la aplicación anterior debemos proteger la aplicación por si hay algún error al momento de tomar el archivo y que no contenga los datos como se espera, es decir supongamos que en lugar de tener tres valores por línea solo tuviera dos, veamos que pasaría:



Al ejecutar la aplicación nos daría:



```
C:\Archivos de programa\Xinox Software\JCreatorV3\JEGE2001.exe  
Exception in thread "main" java.util.NoSuchElementException  
    at java.util.StringTokenizer.nextToken(StringTokenizer.java:332)  
    at AplicacionFiles4.main(AplicacionFiles4.java:21)  
Press any key to continue...
```

Esto es porque al crear el valor double que no se encuentra en la primera línea del archivo, el nextToken no encuentra nada en esa línea, por lo tanto el error de ejecución sería NoSuchElementException.

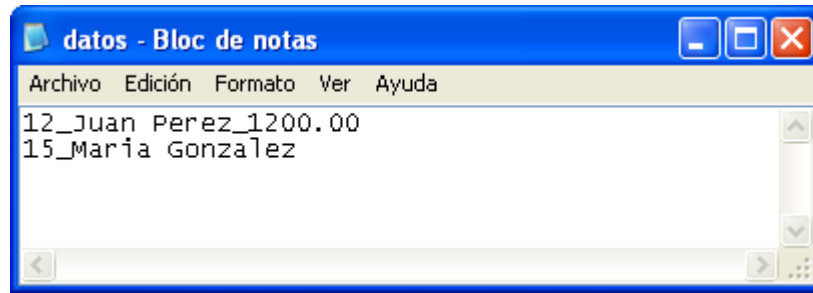
Utilizando el try/catch podemos corregir este error, la aplicación nos quedaría como:

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

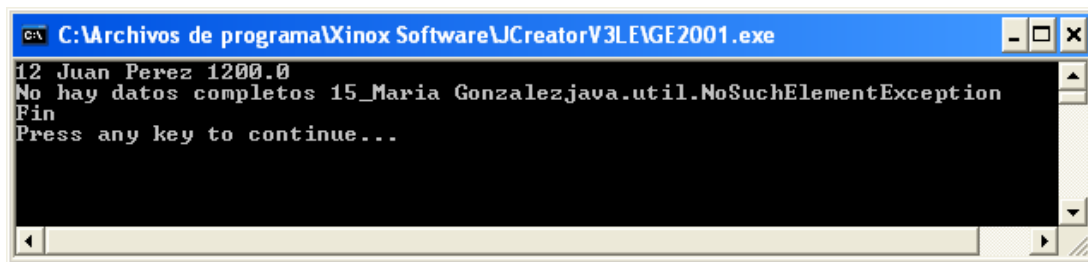
public class AplicacionFiles4 {
    public static void main(String[] args) throws IOException {
        BufferedReader ent = new BufferedReader(new FileReader("datos.txt"));
        PrintWriter sal = new PrintWriter(System.out,true);
        PrintWriter salErr = new PrintWriter(System.err, true);
        int numero;
        String nombre;
        double saldo;
        StringTokenizer stt;

        String linea = ent.readLine();
        while (linea != null) {
            try {
                // se tokeniza la linea leida del archivo
                stt = new StringTokenizer(linea, " ");
                // se extrae cada uno de los tokens de la línea
                numero = Integer.parseInt(stt.nextToken());
                nombre = stt.nextToken();
                saldo = Double.parseDouble(stt.nextToken());
                sal.println("'" + numero + " " + nombre + " " + saldo);
            }
            catch (NoSuchElementException nsee) {
                salErr.println("No hay datos completos " + linea + nsee.toString());
            }
            linea = ent.readLine();
        }
        salErr.println("Fin");
        ent.close();
    }
}
```

Suponiendo que tenemos el archivo con la primera línea correcta y la segunda no:



La aplicación ejecutada seria la siguiente:



Observamos que la primera línea esta correcta, pero en la segunda hay error y al mandar la línea a desplegar podemos observar que solo lleva hasta el nombre.

StreamTokenizer

El StreamTokenizer trabaja igual que el StringTokenizer, pero la diferencia radica en el tamaño del objeto a ser tokenizado, es decir el StringTokenizer utiliza un String el cual tiene un límite definido, mientras que el StreamTokenizer toma como parámetros un objeto de la clase InputStream o un objeto de la clase Reader.

Constructor Summary
StreamTokenizer (InputStream is) <i>Deprecated. As of JDK version 1.1, the preferred way to tokenize an input stream is to convert it into a character stream, for example:</i> <pre>Reader r = new BufferedReader(new InputStreamReader(is)); StreamTokenizer st = new StreamTokenizer(r);</pre>
StreamTokenizer (Reader r) Create a tokenizer that parses the given character stream.

Cuenta con una serie de métodos para tokenizer:

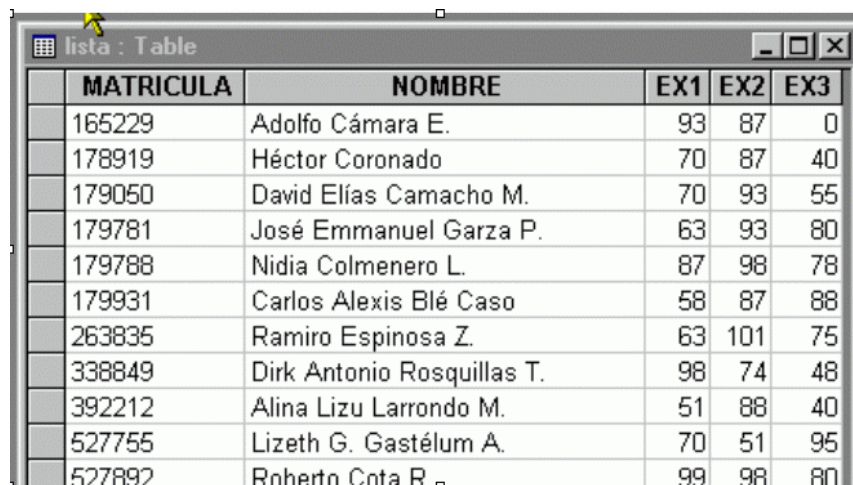
Method Summary	
void	<code>commentChar</code> (int ch) Specified that the character argument starts a single-line comment.
void	<code>eolIsSignificant</code> (boolean flag) Determines whether or not ends of line are treated as tokens.
int	<code>lineno</code> () Return the current line number.
void	<code>lowerCaseMode</code> (boolean fl) Determines whether or not word token are automatically lowercased.
int	<code>nextToken</code> () Parses the next token from the input stream of this tokenizer.
void	<code>ordinaryChar</code> (int ch) Specifies that the character argument is "ordinary" in this tokenizer.
void	<code>ordinaryChars</code> (int low, int hi) Specifies that all characters <i>c</i> in the range <code>low <= c <= high</code> are "ordinary" in this tokenizer.
void	<code>parseNumbers</code> () Specifies that numbers should be parsed by this tokenizer.
void	<code>pushBack</code> () Causes the next call to the <code>nextToken</code> method of this tokenizer to return the current value in the <code>tttype</code> field, and not to the value in the <code>nval</code> or <code>sval</code> field.
void	<code>quoteChar</code> (int ch) Specifies that matching pairs of this character delimit string constants in this tokenizer.
void	<code>resetSyntax</code> ()

4.2. Bases de Datos

4.2.1. Configuración de ODBC

Para el desarrollo de los ejemplos usaremos el puente JDBC-ODBC, es bastante bueno para el desarrollo, además de que es gratis. Para aplicaciones pequeñas el desempeño es bueno, si es que necesitas algo que procese con mayor rendimiento tendrás que conseguir algún driver, la ventaja es que tu programa no cambiará en lo absoluto, solamente la línea donde configuras el driver.

En los ejemplos que veremos manejaremos una base de datos en Access que se llama `curso1.mdb` (baja y graba el archivo en algún directorio en tu disco para poder realizar el siguiente paso).



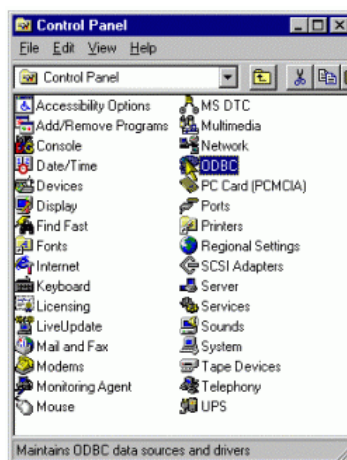
MATRICULA	NOMBRE	EX1	EX2	EX3
165229	Adolfo Cámara E.	93	87	0
178919	Héctor Coronado	70	87	40
179050	David Elías Camacho M.	70	93	55
179781	José Emmanuel Garza P.	63	93	80
179788	Nidia Colmenero L.	87	98	78
179931	Carlos Alexis Blé Caso	58	87	88
263835	Ramiro Espinosa Z.	63	101	75
338849	Dirk Antonio Rosquillas T.	98	74	48
392212	Alina Lizu Larrondo M.	51	88	40
527755	Lizeth G. Gastélum A.	70	51	95
527892	Roberto Cota R.	99	98	80

Como ves la base de datos tiene 4 campos, con los nombres que ves en la figura.

◆ Configuración del ODBC

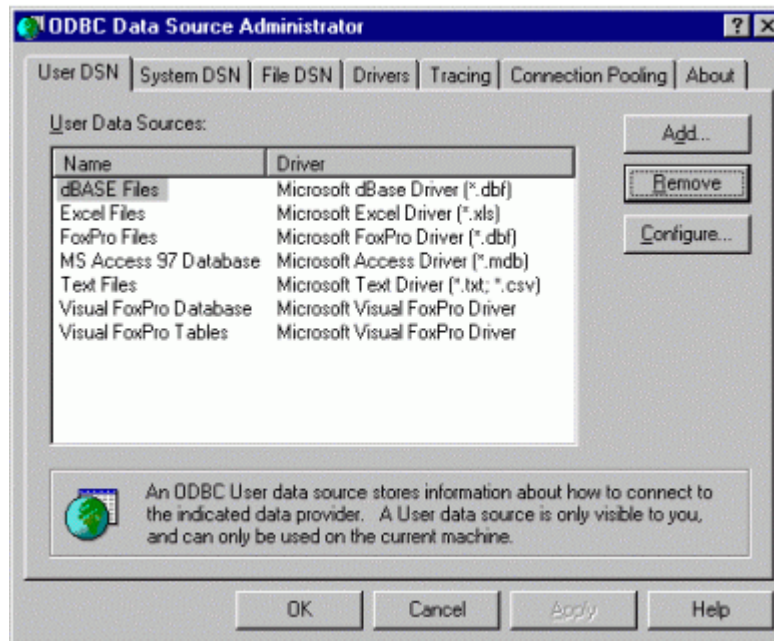
Para poder usar el puente JDBC-ODBC, primero tenemos que configurar el ODBC, lo que haremos es crear un nombre (data source name) con el cual podamos hacer referencia a la base de datos desde Java usando el ODBC. Los pasos están realizados en una computadora con Windows NT, es idéntico en Windows 95/98/2000 sólo cambian algunas ventanas (en forma mínima).

Primero ejecutar ODBC desde el Control Panel, en algunos sistemas se llama "32 bits ODBC" o algo por el estilo.

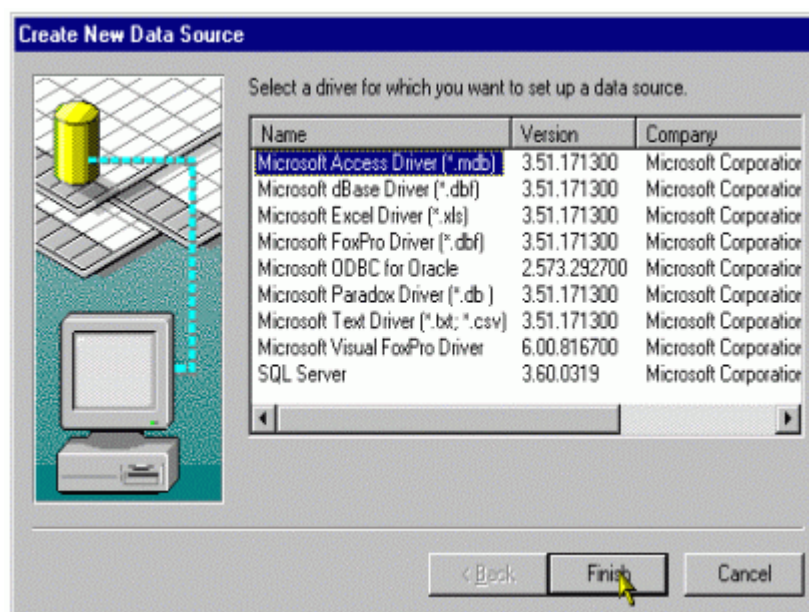


Al ejecutar ODBC se abre la siguiente ventana, es conveniente que selecciones la pestaña de System DSN, en caso que estes en un servidor, esto es necesario porque de otra manera el data source name no estará disponible cuando el usuario esté dado de baja, en Windows 95/98 esto es irrelevante. Lo anterior es importante para cuando dejes corriendo un servicio que haga accesos a

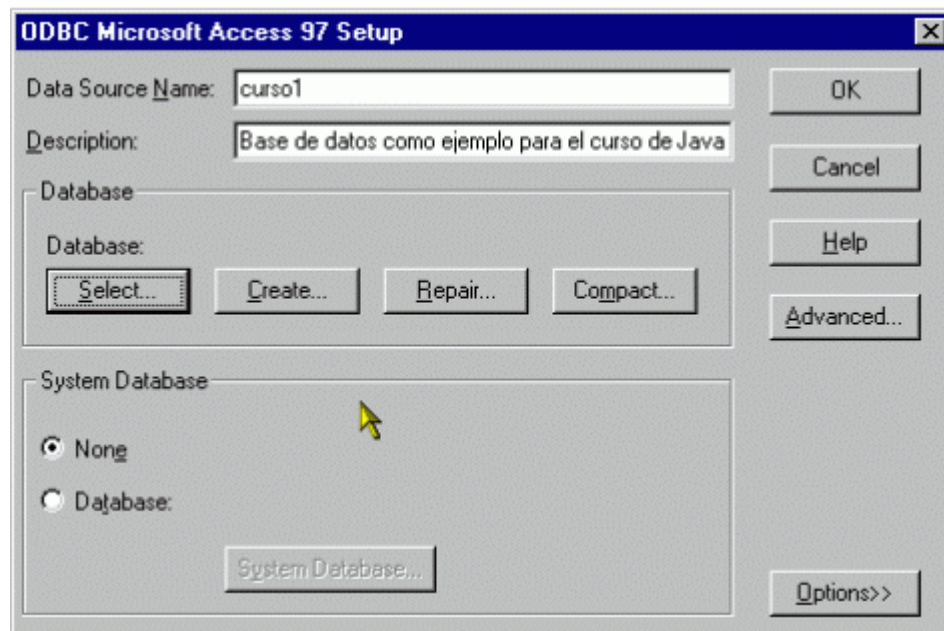
la base de datos, o cuando desarrolles cuestiones como servlets o JSPs, que veremos más adelante.



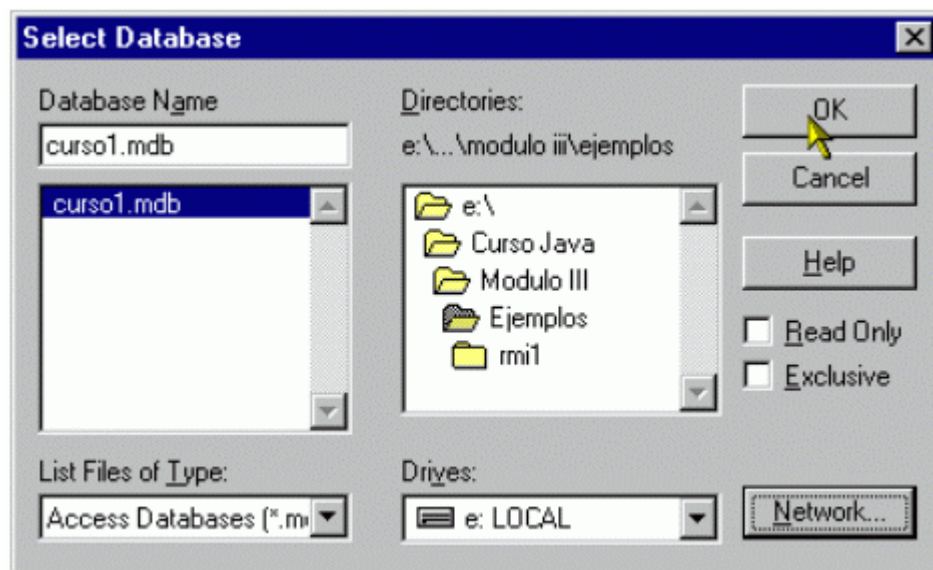
Después de dar agregar o add, se abre la siguiente ventana, en donde seleccionaras el tipo de driver que usará ODBC, en este caso será Microsoft Access Driver (*.mdb), después presiona el botón de terminar.



Acto seguido deberás asignar un nombre a la base de datos, *este es un nombre con el cual nos referiremos a la base de datos desde Java*, no necesita coincidir con el nombre real del archivo, en la siguiente figura es donde se hace el mapeo del nombre al archivo en disco, en este caso le pondremos curso1, la caja de descripción es opcional.



Como paso final tenemos que seleccionar cual es la base de datos a la cual apuntará el nombre que acabamos de escribir, presiona Seleccionar o Select.



y busca el archivo que bajaste y grabaste localmente, recuerda que los nombres no deben de coincidir, lo importante es el mapeo que haces y para Java tienes que recordar el data source name que seleccionaste, porque con ese nombre nos referiremos a la base de datos en los programas, en nuestro caso será curso1.

Eso fue todo con el ODBC, hay algunos otros detalles, como que puedes poner el password de acceso en caso de que tu base de datos tenga, en este caso no lo usaremos, y dejaremos que Java lo maneje. Así es que adelante, vamos a manejar Bases de datos desde Java.
Consulta a Tablas

◆ Pasos previos de un programa

Un programa siempre debe de realizar los siguientes tres pasos antes de tener acceso a una base de datos:

◆ Cargar el driver

La instrucción siguiente carga el driver, el parámetro String es el nombre del driver, será lo único que cambie en tu programa si decides utilizar otro driver, además de la manera de especificar la base de datos que vemos en la siguiente instrucción.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Lo que pasa es que se crea un ejemplar (instancia) del driver y lo registra en el DriverManager para su posterior uso. El método `forName` arroja la excepción `ClassNotFoundException`, así es que la tienes que atrapar para que un programa compile.

◆ Establecer una conexión a la Base de Datos

```
Connection conexion=DriverManager.getConnection("jdbc:odbc:curso1");
```

Para poder tener acceso a la base de datos necesitamos abrir una conexión, normalmente abrimos una sola conexión, pero puedes abrir varias si es que lo necesitas. Como ves en el String le tienes que especificar el data source name que configuraste en un paso previo. Este método arroja `SQLException`.

◆ Crear un estatuto asociado a la conexión

```
Statement estatuto = conexion.createStatement();
```

El estatuto se necesita porque por medio de él haremos los accesos a la base de datos.

Los tres pasos anteriores se realizan siempre, lo que varía es el tipo de acceso que se hará a la base de datos, en los ejemplos posteriores iremos desde una simple consulta hasta poder modificar una tabla y crear tablas adicionales en la base de datos. Como ya viste en la introducción el lenguaje que se usa para el acceso a base de datos es SQL, el cual es bastante standard y lo iremos conociendo de a poco, si necesitas hacer cosas más elaboradas te recomiendo que comiences a aprender SQL para sacar el máximo provecho de JDBC. En los siguientes ejemplos veremos suficiente del lenguaje como para que comiences a hacer tus primeros programas sin problemas. Este método también arroja SQLException.

◆ Consulta a una tabla

Comencemos con lo más sencillo. La consulta a una tabla se hace por medio del estatuto

```
executeQuery("query en SQL")
```

Donde el query típicamente es un estatuto de SQL, aunque esto no es necesario, lo que necesitas poner ahí es un estatuto que entienda el manejador de base de datos que estás usando, pero como dijimos SQL es el standard, así es que es el que utilizaremos. El query que, como ves es un String, hay que construirlo con cuidado, ya que un error no será detectado en Java, lo detectará el manejador y te mandará un error (excepción). El resultado que arroja el estatuto anterior es un objeto de tipo ResultSet, el cual puedes manejar en Java para desplegar la información o hacer lo que necesites hacer, además el método executeQuery se tiene que aplicar a un estatuto que normalmente ya tienes creado asociado a una conexión. La instrucción completa quedaría de la siguiente forma:

```
ResultSet rs = estatuto.executeQuery("SELECT...");
```

El resultado arrojado en el objeto rs de tipo ResultSet tiene un apuntador que apunta, inicialmente, al primer registro del grupo de registros obtenidos de la base de datos, con base en el query hecho. la forma de avanzar en este objeto para obtener cada uno de los registros es con el método next() de la clase ResultSet, el cual regresa un valor booleano, indicando si todavía hay más registros en el objeto ResultSet. Los dos métodos anteriores también arrojan SQLException. Una vez que obtuvimos el objeto ResultSet y podemos obtener cada uno de los registros, tenemos que obtener los campos de cada registro, esto lo hacemos con una serie de métodos get...(), donde la versión particular de get dependerá del tipo de dato que quieras obtener, el parámetro que tienes que poner en el get es el nombre de la columna o campo del registro, aunque también puedes poner un entero, el cual indica el número de columna, este último método no lo

recomiendo porque tu programa se hace poco legible y mantenible, pero lo usaremos, a veces en nuestros ejemplos para que veas como se utiliza, a continuación puedes algunos ejemplos:

```
rs.getString("nombre")  
rs.getInt("EX2")  
rs.getBoolean(3)
```

El tipo de dato que regresan estos métodos depende de la versión que uses de get.
Veamos un ejemplo completo, el nombre del archivo es BaseDeDatos1.java

Antes de ver el primer ejemplo entendamos el estatuto SELECT de SQL. En un estatuto select especificamos los campos que queremos que regrese el query, además de la tabla en la cual se quiere hacer ese query, la tabla pertenece a una base de datos, la cual puede tener otras tablas, por ejemplo el siguiente query:

```
select campo1, campo2 from mitabla
```

Me regresará todos los campos con nombre campo1 y campo2 de la tabla con nombre mitabla, podemos usar el carácter * como comodín

```
select * from mitabla
```

en este caso regresará todos los campos de todos los registros en la tabla mitabla, también podemos especificar ciertas condiciones, que tiene que cumplir cada uno de los registros que me debe de regresar el manejador de la base de datos, por ejemplo:

```
select * from mitabla where campo1>4
```

me regresará todos los registros que tenga campo1 mayor que 4 en la tabla mitabla. En los siguientes ejemplos iremos cubriendo algunos atributos adicionales de los queries

```
import java.sql.*; //Importar las clases necesarias
```

```
public class BaseDeDatos1{
```

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

```
// Cargar y registrar el driver
```

```
    try{
```

```
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
    }catch(ClassNotFoundException e){
```

```
        System.out.println(e.getMessage());
```

```
    }
```

```
// Poner un sólo try para hacer la discusión más ágil, todos los métodos arrojan
```

```
// SQLException, normalmente en una aplicación real se será un poco más selectivo
```

```
try{

// Abrir la conexión a la base de datos con data source name curso1
    Connection conexion = DriverManager.getConnection("jdbc:odbc:curso1");

// Crear el estatuto para hacer los accesos
    Statement estatuto = conexion.createStatement();

// Hacer la consulta, en rs obtenemos todos los registros que nos regresa el query
    ResultSet rs = estatuto.executeQuery("select * FROM lista");

// En un ciclo barremos todos los registros uno por uno hasta que no haya más
    while(rs.next()){

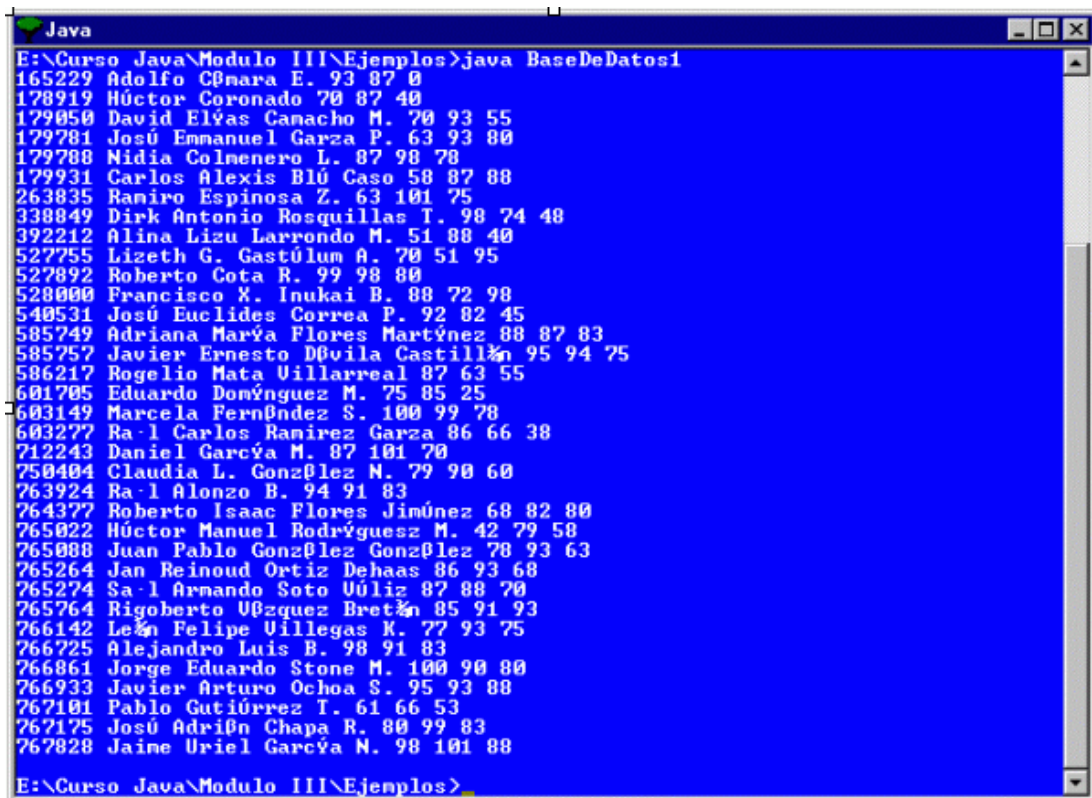
// Obtener todos los campos de cada uno de los registros, con el método get adecuado
        String matricula = rs.getString("MATRICULA");
        String nombre = rs.getString("NOMBRE");
        int ex1 = rs.getInt("EX1");
        int ex2 = rs.getInt("EX2");
        int ex3 = rs.getInt("EX3");

// Imprimir en pantalla todos los campos obtenidos para un registro
        System.out.println(matricula+" "+nombre+" "+ex1+" "+ex2+" "+ex3);
    }

// Cerrar resultset, estatuto y conexión. En teoría al cerrar la conexión
// se cierran automáticamente el estatuto y el resultset, pero anteriormente
// he tenido problemas con esto, así es que como medida de seguridad cerramos todo

        rs.close();
        estatuto.close();
        conexion.close();
    }catch(SQLException e){
        System.out.println(e.getMessage());
    }
}
```

El ejemplo anterior arroja la siguiente impresión a pantalla:



```
E:\Curso Java\Modulo III\Ejemplos>java BaseDeDatos1
165229 Adolfo Cmara E. 93 87 0
178919 Hctor Coronado 70 87 40
179050 David Elías Canacho M. 70 93 55
179781 Josu Emmanuel Garza P. 63 93 80
179788 Midia Colmenero L. 87 98 70
179931 Carlos Alexis Blú Caso 58 87 88
263835 Raniro Espinosa Z. 63 101 75
338849 Dirk Antonio Rosquillas T. 98 74 48
392212 Alina Lizu Larrondo M. 51 88 40
527755 Lizeth G. Gastólum A. 70 51 95
527892 Roberto Cota R. 99 98 80
528000 Francisco X. Inukai B. 88 72 98
540531 Josu Euclides Correa P. 92 82 45
585749 Adriana Maríya Flores Martínez 88 87 83
585757 Javier Ernesto Dóvila Castillón 95 94 75
586217 Rogelio Mata Villarreal 87 63 55
601705 Eduardo Domínguez M. 75 85 25
603149 Marcela Fernández S. 100 99 78
603277 Raúl Carlos Ramírez Garza 86 66 38
712243 Daniel García M. 87 101 70
750404 Claudia L. González N. 79 90 60
763924 Raúl Alonzo B. 94 91 83
764377 Roberto Isaac Flores Jiménez 68 82 80
765022 Hctor Manuel Rodríguez M. 42 79 58
765088 Juan Pablo González González 70 93 63
765264 Jan Reinoud Ortiz Dehaas 86 93 68
765274 Saúl Armando Soto Uúliz 87 88 70
765764 Rigoberto Ubezque Bretón 85 91 93
766142 León Felipe Villegas K. 77 93 75
766725 Alejandro Luis B. 78 91 83
766861 Jorge Eduardo Stone M. 100 90 80
766933 Javier Arturo Ochoa S. 95 93 88
767101 Pablo Gutiérrez T. 61 66 53
767175 Josu Adrián Chapa R. 80 99 83
767828 Jaime Uriel García N. 98 101 88
E:\Curso Java\Modulo III\Ejemplos>
```

Veamos otra consulta un poco más elaborada, el código completo está en BaseDeDatos2.java

```
// Abrir conexión y crear estatuto para la consulta
```

```
Connection conexion = DriverManager.getConnection("jdbc:odbc:curso1","jdbc","jdbc");
Statement estatuto = conexion.createStatement();
```

```
// Regresar los campos MATRICULA y NOMBRE de todos los registros cuyo campo
// nombre comience con b, c, d y e, la forma de especificar strings dentro
// del query, ya que este es un string, es con '...'
// ordenados por nombre en forma descendente
// Todo el query es un string, la suma de strings simplemente es para que puedas
// ver toda la instrucción sin necesidad de usar la barra de scroll
```

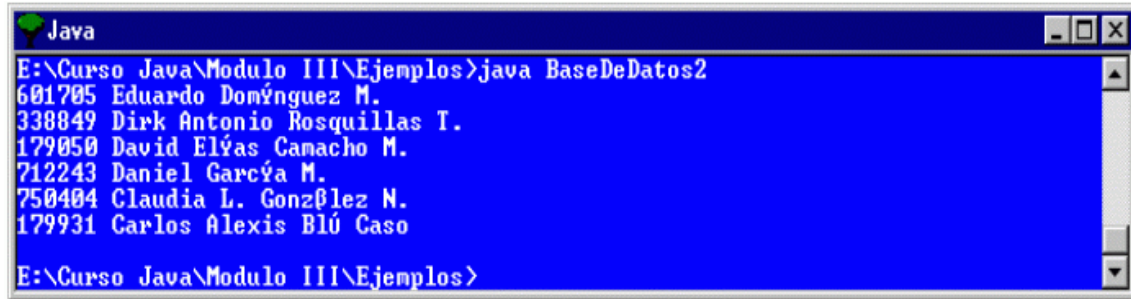
```
ResultSet rs = estatuto.executeQuery("select MATRICULA,NOMBRE "+
    "from lista where NOMBRE>'b' and NOMBRE<'f' "+
    "order by NOMBRE DESC");
```

```
// barrer todos los registros que arroja el query
while(rs.next()){
```

```
String matricula = rs.getString("MATRICULA");

// Podemos hacer mención a un campo con un entero, indicando (en este caso)
// el segundo campo regresado de cada registro
String nombre = rs.getString(2);
System.out.println(matricula+" "+nombre);
}
```

El programa anterior arroja la siguiente impresión a pantalla:



```
Java
E:\Curso Java\Modulo III\Ejemplos>java BaseDeDatos2
601705 Eduardo Domínguez M.
338849 Dirk Antonio Rosquillas T.
179050 David Elías Camacho M.
712243 Daniel García M.
750404 Claudia L. González N.
179931 Carlos Alexis Blú Caso
E:\Curso Java\Modulo III\Ejemplos>
```

Modificación de Tablas

Para modificar tablas necesitamos ejecutar otro método a un estatuto diferente de executeQuery

```
estatuto.executeUpdate("estatuto de SQL");
```

Los comandos típicos SQL para la actualización de tablas son

- ◆ UPDATE
- ◆ INSERT INTO
- ◆ DELETE FROM

El siguiente ejemplo contiene una serie de estatutos que modifican la tabla que hemos venido utilizando y está en BaseDeDatos3.java

Nos saltaremos los pasos previos de cargar el driver, establecer la conexión y crear el estatuto, iremos directo a la secuencia de estatutos que modifican la tabla y al final imprimiremos en pantalla todos los registros de la tabla, como quedó al final de todas las modificaciones.

executeUpdate regresa un entero, el cual indica el número de registros modificados, no es necesario que lo llames dentro de una expresión, esto lo haces si es que necesitas comprobar la cantidad de registros actualizados en la base de datos.

Veamos cada uno de los estatutos que se ejecutan:

```
// Sumaremos 10 puntos al campo EX1 y 5 al campo EX2 a todos los registros  
estatuto.executeUpdate("update lista set EX1 = EX1 + 10,EX2 = EX2 + 5");
```

```
// Le pondremos 100 a todos los exámenes de alumnos cuyo nombre inicia con 'a'  
estatuto.executeUpdate("update lista set EX1=100,EX2=100,EX3=100  
where NOMBRE>'a' AND NOMBRE<'b'");
```

```
// Borraremos todos los alumnos con matrícula mayor a 700,000,  
estatuto.executeUpdate("delete from lista where MATRICULA>700000");
```

```
// Insertamos al alumno Pablo Díaz matrícula 49052 y con calificaciones de 100  
// Este es el formato que se usa en caso de que quieras insertar valores a todos  
// los campos, el primer 0 es porque Access inserta un campo de ID por default  
// y el 0 causa que el manejador le inserte un valor en forma automática  
estatuto.executeUpdate("insert into lista values(0,'49052','Pablo  
Díaz',100,100,100)");
```

```
// En caso de que no quieras asignar valor a todos los campos necesitas  
// especificar a que campos les vas a asignar valor  
estatuto.executeUpdate("insert into lista (MATRICULA,NOMBRE,EX3)  
values ('123456','Tonto',50)");
```

Por último imprimimos todos los registros que quedan con la tabla modificada y arroja el siguiente resultado a pantalla:



```
Java
E:\Curso Java\Modulo III\Ejemplos>java BaseDeDatos3
E:\Curso Java\Modulo III\Ejemplos>java BaseDeDatos1
165229 Adolfo C mara E. 100 100 100
178919 H ctor Coronado 80 92 40
179050 David El as Camacho M. 80 98 55
179781 Jos  Emmanuel Garza P. 73 98 80
179788 Nidia Colmenero L. 97 103 78
179931 Carlos Alexis Bl  Caso 68 92 88
263835 Ramiro Espinosa Z. 73 106 75
338849 Dirk Antonio Rosquillas T. 108 79 48
392212 Alina Lizu Larrondo M. 100 100 100
527755 Lizeth G. Gast lum A. 80 56 95
527892 Roberto Cota R. 109 103 80
528000 Francisco X. Inukai B. 98 77 98
540531 Jos  Euclides Correa P. 102 87 45
585749 Adriana Mar a Flores Mart nez 100 100 100
585757 Javier Ernesto D vila Castell n 105 99 75
586217 Rogelio Mata Villarreal 97 68 55
601705 Eduardo Dom nguez M. 85 90 25
603149 Marcela Fern ndez S. 110 104 78
603277 Ra l Carlos Ramirez Garza 96 71 38
49052 Pablo D az 100 100 100
123456 Tonto 0 0 50
E:\Curso Java\Modulo III\Ejemplos>
```

En el siguiente ejemplo crearemos otra tabla dentro de la base de datos, lo que haremos ser  crear una tabla que se llamar  final, la cual guardar  matr cula y la calificaci n final calculada como el promedio de los tres parciales, veremos que necesitamos crear otro estatuto porque un estatuto lo usaremos para obtener los datos y otro para ir actualizando la tabla nueva, esto lo iremos haciendo en forma simult nea en las dos tablas, por eso necesitamos dos estatutos, porque estamos haciendo dos accesos simult neos.

El archivo se encuentra en BaseDeDatos4.java

```
// Creamos los dos estatutos que utilizaremos
Statement estatuto = conexion.createStatement();
Statement estatuto1 = conexion.createStatement();

// Creamos la tabla final con los dos campos, uno de texto de 6 caracteres
// y otro de punto flotante que guardar  la calificaci n final
estatuto.executeUpdate("create table final (MATRICULA varchar(6),
NOTA double)");

// Hacemos el query para obtener todos los registros de la tabla
ResultSet rs = estatuto.executeQuery("select * FROM lista");
```

```
// Barremos toda la tabla registro por registro
while(rs.next()){

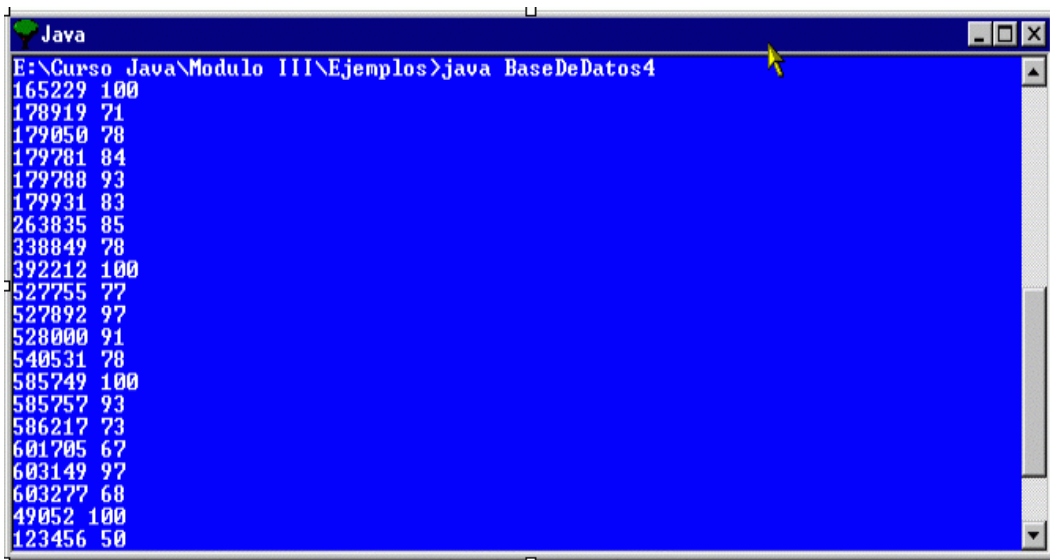
// Obtenemos los datos que necesitamos
    String matricula = rs.getString("MATRICULA");
    int ex1 = rs.getInt("EX1");
    int ex2 = rs.getInt("EX2");
    int ex3 = rs.getInt("EX3");

// calculamos la calificación final, nota el casting que se tiene que hacer
// para que el resultado sea double, si no lo ponemos el resultado será double
// pero la división la hará entera, entonces quedará como double pero truncado
// porque la división se hace que la signación, con el casting forzamos a que el tipo
// de dato cambie antes de la división
    double nota = (double)(ex1+ex2+ex3)/3;

// insertamos un nuevo registro en la nueva tabla
// Es importante notar que lo que se manda es un string, por eso necesitamos
// sumar la variable matricula al estatuto SQL porque si ponemos solamente
// matricula lo que se mandará a la base de datos será la palabra matricula y no el
// valor de la variable matricula, lo mismo pasa con nota, necesitamos transformar
// esa variable a string y después agregarlo a al estatuto SQL para que se mande lo
// que queremos
    estatuto1.executeUpdate("insert into final
    values('"+matricula+String.valueOf(nota)+"')");
}

// Por último veamos como quedo la nueva tabla imprimiendo todos los registros
// en pantalla
rs = estatuto.executeQuery("select * FROM final");
while(rs.next()){
    String mat = rs.getString("MATRICULA");
    double not = rs.getDouble("NOTA");
    System.out.println(mat+" "+Math.round(not));
}
```

En pantalla queda lo siguiente:



```
Java
E:\Curso Java\Modulo III\Ejemplos>java BaseDeDatos4
165229 100
178919 71
179050 78
179781 84
179788 93
179931 83
263835 85
338849 78
392212 100
527755 77
527892 97
528000 91
540531 78
585749 100
585757 93
586217 73
601705 67
603149 97
603277 68
49052 100
123456 50
```

EJERCICIOS DE AUTOEVALUACIÓN

1. Crear un aplicativo le permita escribir información básica de una agenda de citas
2. Crear un aplicativo que permite ingresar información en una tabla y mostrarla
3. Crear un aplicativo que permita en forma básica, agregar, consultar y listar información de un archivo

Prueba Final

Crear una aplicativo que permite modificar información de un archivo (tema para complemento investigativo)

Actividad

Crear un aplicativo que permita insertar, consultar, eliminar y actualizar información en una tabla.

5. RELACIÓN CON OTROS TEMAS

Lenguaje de programación II, es un tema relacionado con temas como Lenguaje de Programación I, Algoritmos I, Algoritmos II, además complementara con Estructuras de Datos, Bases de Datos, Ingeniería de Software y con Lenguaje de Programación III, hace parte del ciclo de procesos de desarrollo dentro de la línea principal de la carrera.

5.1. Fuentes

5.1.1. Libros

HOLZNER, Steven. La biblia de java 2. Anaya multimedia 2000

CEBALLOS, Francisco Javier. Java 2 Interfaces gráficas y aplicaciones para internet. Alfaomega Ra-Ma. 2006

JOYANES AGUILAR, Luis; FERNANDEZ AZUELA, Matilde. Java 2 Manual del programador. Ra-Ma. 2001

CEBALLOS, Francisco Javier. Java 2 Curso de Programación. Ra-Ma. 2005

ECKEL, Bruce. Pensando en Java. Prentice-Hall. 2000

VILLALOBOS, Jorge; CASALLAS, Ruby. Fundamentos de Programación. Prentice Hall. 2006

DEAN, John; DEAN, Raymond. Introducción a la programación en java. McGraw Hill

FERNANDEZ, Carmen. Java Básico. Starbook.

6. PÁGINAS WEB

www.sun.com.co

<http://profesores.fi-b.unam.mx>

www.java.com

[**cupi2**.uniandes.edu.co](http://cupi2.uniandes.edu.co)

www.programacion.com/java

www.monografias.com

<http://www.javahispano.org/>

1 METODOLOGÍA

1.1 PRESENCIAL

El proceso de participación dentro de la materia lenguaje de programación I (java), es netamente práctica y participativa, acompañado de investigación complementaria a los temas tratados.

1.2 DISTANCIA

El manejo de distancia no dista mucho del método presencial, dadas las características del área, es un ambiente teórico práctica, permitiendo la interacción permanente del aprendiz y de la complementación de los temas en mediante investigación y practica de los temas.

2 EVALUACIÓN

El tema evaluativo se conforma de un 80% del tema en forma práctica con los temas tratados e investigados y 20% teórico con los conceptos de la materia demanda y que son fundamentales para una comprensión optima de los temas.