
# Lenguaje de programación I

| | |
|---|---|
| **Curso** | Lenguaje de Programación I (4688) |
| **Formato** | Manual de curso |
| **Autor** | Cibertec |
| **Título** | Lenguaje de Programación I |
| **Versión** | 2022 |
| **Páginas** | 188 p. |
| **Elaborador** | Atuncar Guzman, Jose Augusto |
| **Revisor** | Belleza Porras, Alex Tito |

# Índice

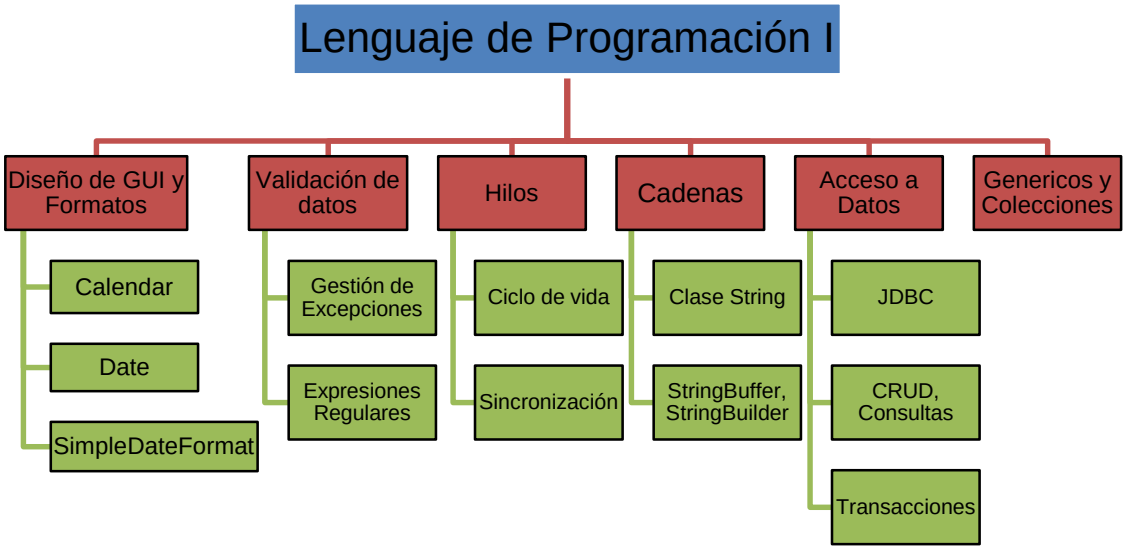
# Presentación

***Lenguaje de Programación I*** pertenece a la línea de Programación y Desarrollo de Aplicaciones y se dicta en la carrera de Computación e Informática de la institución. El curso brinda un conjunto de conceptos, técnicas y herramientas que permiten al alumno alcanzar un nivel avanzado de conocimiento en el lenguaje Java Standard Edition (JSE), implementando una aplicación Stand Alone con acceso a datos, utilizando Java como lenguaje de programación.

El manual para el curso ha sido diseñado bajo la modalidad de unidades de aprendizaje, las que se desarrollan durante semanas determinadas. En cada una de ellas, hallará los logros, que debe alcanzar al final de la unidad; el tema tratado, el cual será ampliamente desarrollado; y los contenidos, que debe desarrollar, es decir, los subtemas. Por último, encontrará las actividades que deberá desarrollar en cada sesión, que le permitirán reforzar lo aprendido en la clase.

El curso es eminentemente práctico: consiste de una sesión semanal de laboratorio. En primer lugar, se inicia con la definición de conceptos básicos para el manejo de Excepciones en Java, enfatizándose en las buenas prácticas existentes para manejar ambos aspectos. Continúa con la implementación de aplicaciones que apliquen conceptos avanzados de manejo de Strings, flujos de entrada y salida, serialización de objetos, formateo y parseo de datos, así como el uso del Framework Collections de Java. Luego, se desarrollan aplicaciones con acceso a datos.

# Red de contenidos



1

# INTERFAZ GRÁFICA DE USUARIO

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones utilizando de manera individual y combinada las clases JMenu, JMenuBar, JItemMenu, JDesktopPane, JInternalFrame, JTable JDialog.

**TEMARIO**



**ACTIVIDADES PROPUESTAS**

- Crear aplicaciones con formularios internos

## 1.1 CREACIÓN DE FORMULARIOS COM JINTERNAFRAME

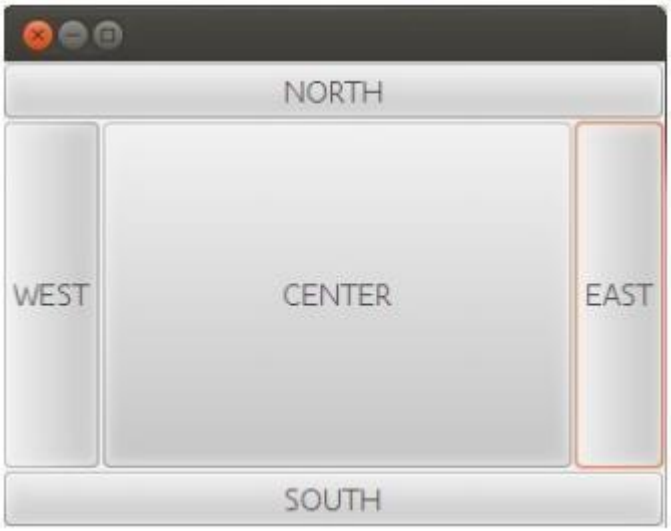
### 1.1.1. Manejo de Layouts

Los Layouts son clases que indican como se van a acomodar los controles (botones, labels, textfields, etc) en una interfaz gráfica en Java. Algunos de ellos son:

- BorderLayout
- GridLayout

BorderLayout

BorderLayout se usa para acomodar los componentes al norte, sur, este, oeste y centro de una ventana. Solo hay que agregar un componente a un BorderLayout y decir donde queremos que se vea, ya sea al centro, al norte, al sur, al este o al oeste.




GridLayout

El GridLayout se utiliza para crear matrices en una ventana. Acomoda los componentes en un mismo tamaño. Es útil para realizar calculadoras.

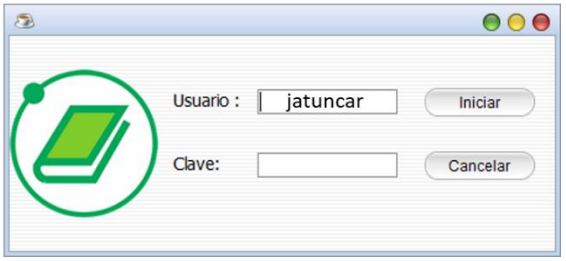

### 1.1.2. Temas

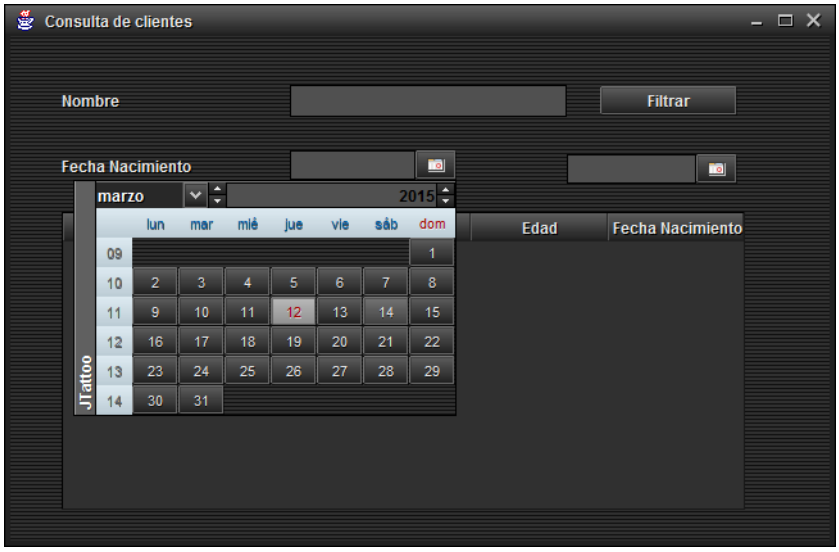
La apariencia de una aplicación puede hacer que el usuario se decida por ella, tanto es así, que muchos programadores profesionales la utilizan como un gancho para atraer a sus clientes, a pesar de requerir más tiempo y dedicación. En java, existe una forma muy sencilla con la que podemos darle

un toque más agradable a nuestras aplicaciones, en tan solo un par de líneas.

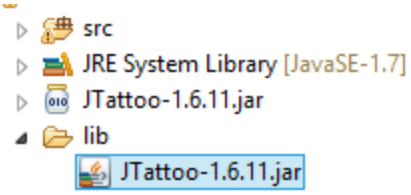
Se trata de los Looks and Feels, un mecanismo que funciona parecido a instalar un theme para un SO y que permite cambiar el aspecto de las ventanas, botones, cuadros de diálogos y demás componentes que integran una Interfaz de usuario.

JTattoo (http://www.jtattoo.net/Download.html )
Consta de varios Look para aplicaciones Swing.
Permite a los desarrolladores mejorar su aplicación con una interfaz de usuario sencilla


Para ello:
* descargamos la librería e instalamos:




* En el método main, escribimos lo siguiente:

```
UIManager.setLookAndFeel("com.jtattoo.plaf.mcwin.McWinLookAndFeel");
```

### 1.1.3. JMenuBar, JMenu, JMenuItem

JmenuBar

Este componente es muy útil ya que representa la barra de menú que vemos siempre en todo programa y es muy útil para acceder a la información de forma más fácil y organizada.

Jmenu

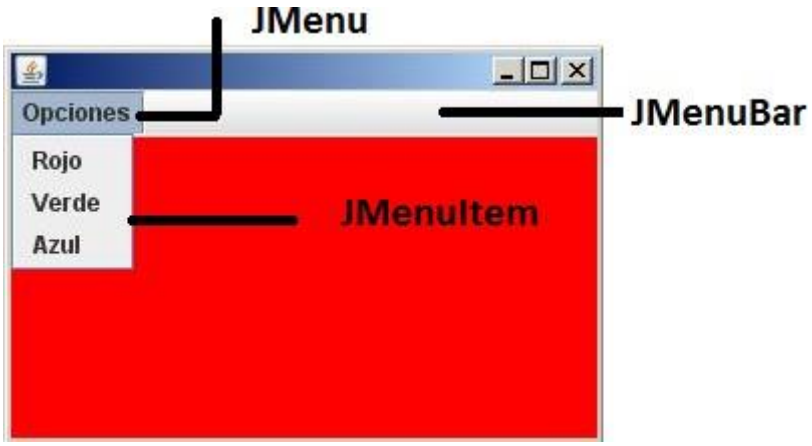
Un 12onv es un objeto que se le añade al JmenuBar, y sirve para almacenar 12onve comunes.

JmenuItem

Un JmenuItem es un elemento del 12onv y al ser pulsado genera un evento, es decir, abre una ventana, solicita datos o cualquier evento

Explicaremos las clases mediante ejemplos

Ejemplo01




```
package ejemplo01;

import java.awt.Color;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JFrame;
import javax.swing.JMenu;
import
javax.swing.JMenuBar;
import javax.swing.JMe
@SuppressWarnings("serial")
public class Ejemplo extends JFrame implements ActionListener {
```

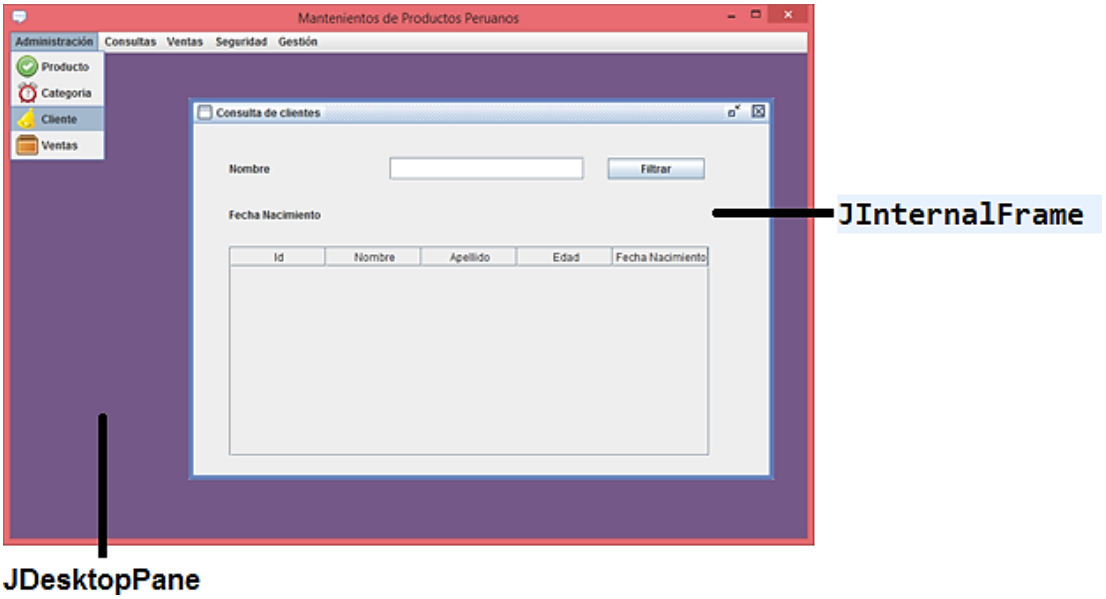
```
        private JMenuBar mb;
        private JMenu menu1;
        private JMenuItem mi1, mi2, mi3;

        public Ejemplo() {
              setLayout(null);
              mb = new
              JMenuBar();
              setJMenuBar(mb);
              menu1 = new
              JMenu("Opciones");
              mb.add(menu1);
              mi1     =     new     JMenuItem("Rojo");
              mi1.addActionListener(this);
              menu1.add(mi1);
mi2 = new JMenuItem("Verde");

        }

        public void actionPerformed(ActionEvent e)
              { Container f =
              this.getContentPane(); if
              (e.getSource() == mi1) {
                    f.setBackground(new Color(255, 0, 0));
              }
              if (e.getSource() == mi2) {
                    f.setBackground(new Color(0, 255,
                    0));
              }
              if (e.getSource() == mi3) {
                    f.setBackground(new Color(0, 0,
                    255));
              }
        }

        public static void main(String[] ar) {
              Ejemplo formulario1 = new
              Ejemplo();
              formulario1.setBounds(10, 20, 300,
              200); formulario1.setVisible(true);
        }
 }
```

### 1.1.4. JDesktopPane, JInternalFrame

Con estas dos clases se pueden crear ventanas internas, es muy 14onve ver este tipo de ventanas en programas de diseño en los cuales puedes abrir varias esto, 14onver puedes minimizar, maximizar y cerrar estas ventanas internas sin cerrar la ventana Padre.

Explicaremos las clases mediante ejemplos:

Ejemplo02




```
package vista;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.WindowEvent;
```

```
import java.awt.event.WindowListener;


import javax.swing.ImageIcon;
import javax.swing.JDesktopPane;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

@SuppressWarnings("serial")
public class FormularioPrincipal extends JFrame implements WindowListener {

	private JDesktopPane desktop = new JDesktopPane();

	public MenuPrincipal menu = new MenuPrincipal(desktop);

	public FormularioPrincipal(String cad, int x, int y) {
		super(cad);
		setLocation(0, 0);
		this.setIconImage(new
ImageIcon(getClass().getClassLoader().getResource("iconos/Bubble.gif")).ge
tIm age());
		setSize(x, y);


		this.setExtendedState(JFrame.MAXIMIZED_BOTH);


		desktop.setSize(600, 400);
		desktop.setBackground(new Color(116, 88, 135));

		add(menu, BorderLayout.NORTH);
		add(desktop, BorderLayout.CENTER);

		addWindowListener(this);

	}

	public static void main(String[] args) {
		try {
			FormularioPrincipal jf = new FormularioPrincipal(
					"Mantenientos de Productos Peruanos", 900,
600);
			jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
			jf.setVisible(true);
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public void windowOpened(WindowEvent e) {}

	public void windowClosing(WindowEvent e) {
		int n = JOptionPane.showConfirmDialog(e.getWindow(),
				"Desea Cerrar la Aplicación ?", "Confirmación",
				JOptionPane.YES_NO_OPTION);
		if (n == JOptionPane.YES_OPTION)
			System.exit(0);
	}
```

```
	public void windowClosed(WindowEvent e) {}
	public void windowIconified(WindowEvent e) {}
	public void windowDeiconified(WindowEvent e) {}
	public void windowActivated(WindowEvent e) {}
	public void windowDeactivated(WindowEvent e) {}
}
```

```
package vista;

import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JInternalFrame;
import javax.swing.JLabel;
import
javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.table.DefaultTableModel;

@SuppressWarnings("serial")

public class FrmConsultaCliente extends JInternalFrame {
	private JLabel jLabel1;
	private JTextField
	txtNombre; private JButton
	btnFiltrar; private JTable
	jTable1;
	private JScrollPane jScrollPane1;
	private JLabel jLabel2;
	private DefaultTableModel jTable1Model;

	public FrmConsultaCliente() {
		this.setVisible(false);
		this.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE)
		; this.setTitle("Consulta de clientes");
		this.setBounds(0, 0, 649, 423);
		this.setClosable(true);
		this.setIconifiable(true);

		// este centrado del formulario
		interno Dimension pantalla =
Toolkit.getDefaultToolkit().getScreenSize();
		Dimension ventana = getSize();
		setLocation((pantalla.width - ventana.width) / 2,
				(pantalla.height - ventana.height) /
				2);
		this.setPreferredSize(new java.awt.Dimension(649,
		423)); getContentPane().setLayout(null);
```

```
            jLabel1 = new JLabel();
            getContentPane().add(jLabel1);

            jLabel1.setText("Nombre");
            jLabel1.setBounds(40, 37, 125,
            26);

            txtNombre = new JTextField();
            getContentPane().add(txtNombre);
            txtNombre.addActionListener(new ActionListener()
            {
                  public void actionPerformed(ActionEvent evt)
                        { btnFiltrarActionPerformed(evt);
                  }
            });
            txtNombre.setBounds(218, 38, 216, 25);

            jLabel2 = new JLabel();
            getContentPane().add(jLabel2);
            jLabel2.setText("Fecha
            Nacimiento");
            jLabel2.setBounds(40, 89, 140,
            24);

            jScrollPane1 = new JScrollPane();
            getContentPane().add(jScrollPane1);
            jScrollPane1.setBounds(40, 137, 533,
            232);


            btnFiltrar = new JButton();
            getContentPane().add(btnFiltrar);
            btnFiltrar.setText("Filtrar");
            btnFiltrar.setBounds(460, 39, 107, 23);
            btnFiltrar.addActionListener(new ActionListener()
            {
                  public void actionPerformed(ActionEvent evt)
                        { btnFiltrarActionPerformed(evt);
                  }
            });

            jTable1Model = new DefaultTableModel(new String[][] {}, new
String[] {
                        "Id", "Nombre", "Apellido", "Edad", "Fecha
Nacimiento" });

            jTable1 = new JTable();
            jScrollPane1.setViewportView(jTable1
            ); jTable1.setModel(jTable1Model);
      }

      private void btnFiltrarActionPerformed(ActionEvent evt) {

      }

}
```

```
package vista;

import java.awt.event.ActionEvent; import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;

import javax.swing.ImageIcon;
```

```
import javax.swing.JDesktopPane;
import  javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;


public class MenuPrincipal extends JMenuBar implements ActionListener{

private static final long serialVersionUID = 1L;

 private JMenu  mMenu01          = new JMenu("Administración");
 private JMenu  mMenu02          = new JMenu("Consultas");
 private JMenu  mMenu03          = new JMenu("Ventas");
 private JMenu  mMenu04          = new JMenu("Seguridad");
 private JMenu  mMenu05          = new JMenu("Gestión");

 private JMenuItem mItem01    = new JMenuItem("Producto");
 private JMenuItem mItem02    = new JMenuItem("Categoria");
 private JMenuItem mItem03    = new JMenuItem("Cliente");
 private JMenuItem mItem04    = new JMenuItem("Ventas");
 private JMenuItem mItem08    = new JMenuItem("Inventario");

 private JMenuItem mItem05    = new JMenuItem("Ventas por cliente");
 private JMenuItem mItem06    = new JMenuItem("Ventas por producto");

 private JMenuItem mItem07    = new JMenuItem("Boleta");



private List<JMenuItem> listaItemMenus = new ArrayList<>();
private List<JMenu> listaMenus = new ArrayList<>();

@SuppressWarnings("unused")
private JDesktopPane desktop= new           JDesktopPane();


private FrmConsultaCliente modulo05 = new FrmConsultaCliente();

public MenuPrincipal(JDesktopPane desktop){
      this.desktop = desktop;

      listaItemMenus.add(mItem01);
      listaItemMenus.add(mItem02);
      listaItemMenus.add(mItem03);
      listaItemMenus.add(mItem04);
      listaItemMenus.add(mItem05);
      listaItemMenus.add(mItem06);
      listaItemMenus.add(mItem07);
      listaItemMenus.add(mItem08);
```

```
            listaMenus.add(mMenu01);
            listaMenus.add(mMenu02);
            listaMenus.add(mMenu03);
            listaMenus.add(mMenu04);
            listaMenus.add(mMenu05);
            mItem01.setIcon(new
ImageIcon(MenuPrincipal.class.getResource("/iconos/Accept.gif")));

            mItem01.setVisible(true);
            mItem02.setIcon(new
```

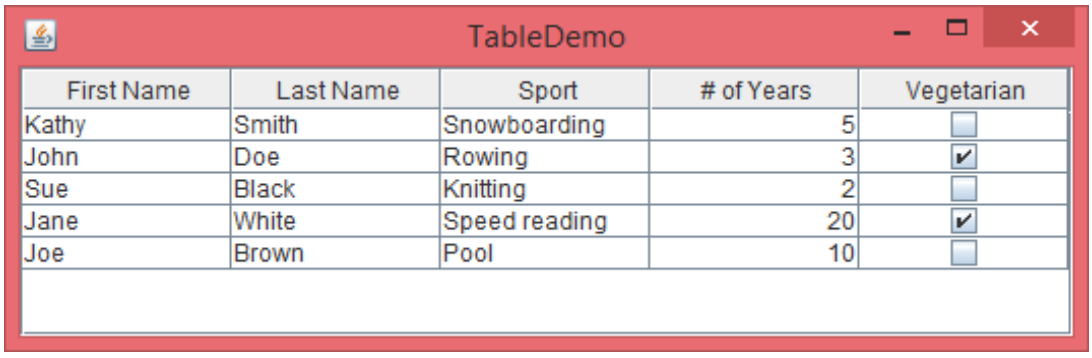
```
ImageIcon(MenuPrincipal.class.getResource("/iconos/Alarm.gif")));
            mItem02.setVisible(true);
            mItem03.setIcon(new

ImageIcon(MenuPrincipal.class.getResource("/iconos/Bell.gif")));
             mItem03.setVisible(true);
             mItem04.setIcon(new
ImageIcon(MenuPrincipal.class.getResource("/iconos/Box.gif")));
            mItem04.setVisible(true);
            mItem05.setVisible(true);
            mItem06.setVisible(true);
            mItem07.setVisible(true);
            mItem08.setVisible(true);

            mMenu01.setVisible(true);
            mMenu02.setVisible(true);
            mMenu03.setVisible(true);
            mMenu04.setVisible(true);
            mMenu05.setVisible(true);

            mItem01.addActionListener(this);
            mItem02.addActionListener(this);
            mItem03.addActionListener(this);
            mItem05.addActionListener(this);
            mItem07.addActionListener(this);
            mItem08.addActionListener(this);


             //Se agregan los items
             mMenu01.add(mItem01);
             mMenu01.add(mItem02);
             mMenu01.add(mItem03);
             mMenu01.add(mItem04);

             mMenu02.add(mItem05);
             mMenu02.add(mItem06);

             mMenu03.add(mItem07);


             mMenu05.add(mItem08);


             //Se agregan los menus
             add(mMenu01);
```

```
            add(mMenu02);
            add(mMenu03);
            add(mMenu04);
            add(mMenu05);

            desktop.add(modulo05);

      }

      public void actionPerformed(ActionEvent e) {

            if(e.getSource()==mItem05
                  ){
                  modulo05.setVisible
                  (true);
            }

       }

}
```

### 1.1.5. JTable

#### Ejercicio 01

Ejemplo básico de mostrar datos en una tabla:



```
package components;

import javax.swing.JFrame;
import javax.swing.JPanel;
import
javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import java.awt.Dimension;
import java.awt.GridLayout;

public class TableDemo extends JPanel {
    private boolean DEBUG = false;

    public TableDemo() {
        super(new GridLayout(1,0));
```

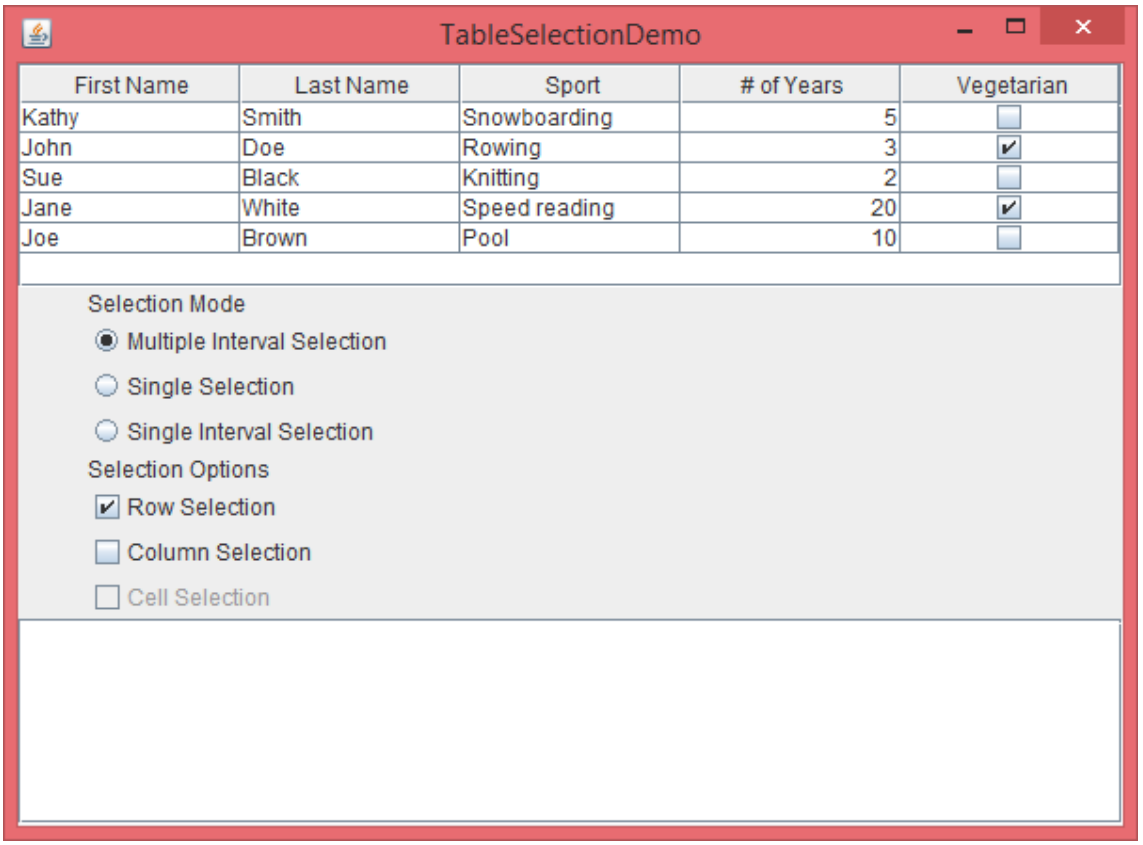
```
        JTable table = new JTable(new MyTableModel());
        table.setPreferredScrollableViewportSize(new Dimension(500, 70));
        table.setFillsViewportHeight(true);

        //Create the scroll pane and add the table to
        it. JScrollPane scrollPane = new
        JScrollPane(table);

        //Add the scroll pane to this
        panel. add(scrollPane);
    }

    class MyTableModel extends AbstractTableModel {
        private String[] columnNames = {"First Name",
                                        "Last Name",
                                        "Sport",
                                        "# of Years",
                                        "Vegetarian"};
        }private Object[][] data = {
         {"Kathy", "Smith",
           "Snowboarding", new Integer(5), new Boolean(false)},

                    "John", "Doe",
                    "Rowing", new Integer(3), new Boolean(true)},
                   {"Sue", "Black",
                    "Knitting", new Integer(2), new Boolean(false)},
                   {"Jane", "White",
                    "Speed reading", new Integer(20), new Boolean(true)},
                   {"Joe", "Brown",
                    "Pool", new Integer(10), new Boolean(false)}
                };

                public int getColumnCount() {
                    return columnNames.length;
                }

                public int getRowCount() {
                    return data.length;
                }

                public String getColumnName(int col) {
                    return columnNames[col];
                }

                public Object getValueAt(int row, int col) {
                    return data[row][col];
                }
                /*
      * JTable uses this method to determine the default renderer/
      * editor for each cell.              If we didn't implement this
        method,
      * rather than a check box.
```

```
     * then the last column would contain text ("true"/"false"),
                */
               public Class getColumnClass(int c) {
                   return getValueAt(0, c).getClass();
               }

               /*
     * Don't need to implement this method unless your table's
     * editable.
                */
               public boolean isCellEditable(int row, int col) {
                   //Note that the data/cell address is constant,
                   //no matter where the cell appears onscreen.
                   if (col < 2) {
                       return false;
                   } else {
                       return true;
                   }
               }

               /*
     * Don't need to implement this method unless your table's
     * data can change.
                */
               public void setValueAt(Object value, int row, int col) {
             if (DEBUG) {
                   System.out.println("Setting value at " + row + "," + col
                                      + " to " + value
                                              + " (an instance of "
                                              + value.getClass() + ")");
                   }

                   data[row][col] = value;
                       fireTableCellUpdated(row, col);

                   if (DEBUG) {
                       System.out.println("New value
                           of data:");
                           printDebugData();
                   }
               }

              private void printDebugData() {
                   int numRows = getRowCount();
                   int numCols = getColumnCount();

                   for (int i=0; i < numRows; i++) {
                       System.out.print("   row " +
                       i + ":"); for (int j=0; j <
                       numCols; j++) {
                           System.out.print("     " + data[i][j]);
                       }
                       System.out.println();
                   }
                   System.out.println("--------------------------");
               }
           }
```

```
    private static void createAndShowGUI() {
        //Create and set up the window.
        JFrame frame = new JFrame("TableDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
        ;

        //Create and set up the content pane.
        TableDemo newContentPane = new TableDemo();
        newContentPane.setOpaque(true); //content panes must be
        opaque frame.setContentPane(newContentPane);

        //Display the window.
        frame.pack();
        frame.setVisible(true)
        ;
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable()
        {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

## Ejercicio 02

Modo de selección:




```
package components;


import javax.swing.*;
import javax.swing.table.AbstractTableModel;
import javax.swing.event.ListSelectionEvent;
import
javax.swing.event.ListSelectionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.GridLayout;
import java.awt.Dimension;

public class TableSelectionDemo extends JPanel
                                implements ActionListener {
    private JTable table;
    private JCheckBox rowCheck;
    private JCheckBox columnCheck;
    private JCheckBox cellCheck;
    private ButtonGroup
    buttonGroup; private JTextArea
    output;

    public TableSelectionDemo() {
        super();
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
```

```
      table.setFillsViewportHeight(true);
      table.getSelectionModel().addListSelectionListener(new
RowListener());
      table.getColumnModel().getSelectionModel().
          addListSelectionListener(new ColumnListener());
      add(new JScrollPane(table));

      add(new JLabel("Selection Mode")); buttonGroup
      = new ButtonGroup();
      addRadio("Multiple Interval Selection").setSelected(true);
      addRadio("Single Selection");
      addRadio("Single Interval Selection");

      add(new JLabel("Selection Options"));
      rowCheck = addCheckBox("Row Selection");
      rowCheck.setSelected(true);
      columnCheck = addCheckBox("Column Selection");
      cellCheck = addCheckBox("Cell Selection");
      cellCheck.setEnabled(false);

      output = new JTextArea(5, 40);
      output.setEditable(false); add(new
      JScrollPane(output));
  }

  private JCheckBox addCheckBox(String text) { JCheckBox
      checkBox = new JCheckBox(text);
      checkBox.addActionListener(this); add(checkBox);
      return checkBox;
  }

  private JRadioButton addRadio(String text) {
      JRadioButton b = new JRadioButton(text);
      b.addActionListener(this); buttonGroup.add(b);
                  add(b);
                  return b;
              }
    public void actionPerformed(ActionEvent
        event) { String command =
        event.getActionCommand();
        //Cell selection is disabled in Multiple Interval Selection
        //mode. The enabled state of cellCheck is a convenient flag
        //for this status.
        if ("Row Selection" == command) {
            table.setRowSelectionAllowed(rowCheck.isSelect
            ed());
            //In MIS mode, column selection allowed must be the
            //opposite of row selection allowed.
            if (!cellCheck.isEnabled()) {
                table.setColumnSelectionAllowed(!rowCheck.isSelected());
            }
        } else if ("Column Selection" == command) {
            table.setColumnSelectionAllowed(columnCheck.isSelected());
            //In MIS mode, row selection allowed must be the
            //opposite of column selection allowed.
            if (!cellCheck.isEnabled()) {
                table.setRowSelectionAllowed(!columnCheck.isSelected());
            }
```

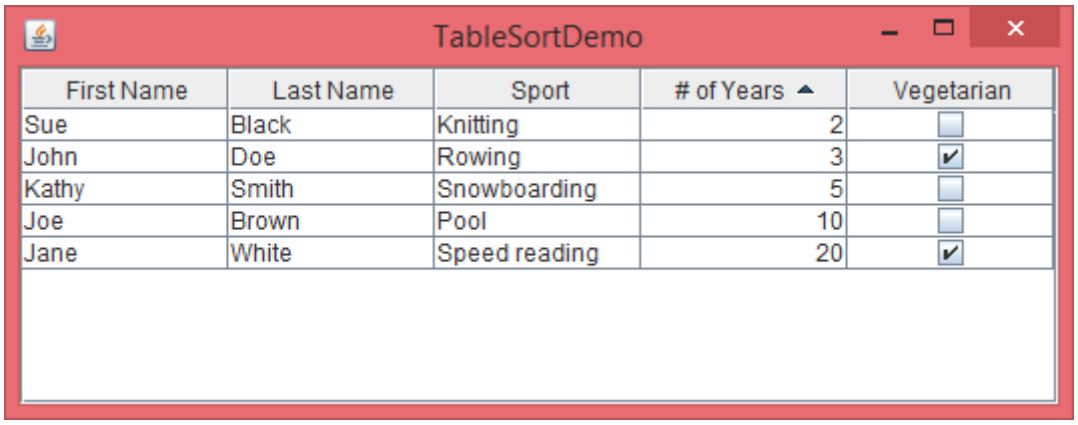
```
        } else if ("Cell Selection" == command) {
            table.setCellSelectionEnabled(cellCheck.isSelected());
            } else if ("Multiple Interval Selection" ==
                command) { table.setSelectionMode(
                        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
                //If cell selection is on, turn it off.
                if (cellCheck.isSelected()) {
                    cellCheck.setSelected(false);
                    table.setCellSelectionEnabled(false);
                }
                //And don't let it be turned back on.
                cellCheck.setEnabled(false);
            } else if ("Single Interval Selection" ==
                command) { table.setSelectionMode(
                        ListSelectionModel.SINGLE_INTERVAL_SELECTION);
                      //Cell selection is ok in this mode.
                      cellCheck.setEnabled(true);
              } else if ("Single Selection" == command) {
                  table.setSelectionMode(
                          ListSelectionModel.SINGLE_SELECTION);
                  //Cell selection is ok in this mode.
                  cellCheck.setEnabled(true);
              }

              //Update checkboxes to reflect selection mode
              side effects.
              rowCheck.setSelected(table.getRowSelectionAllow
              ed());
              columnCheck.setSelected(table.getColumnSelectionA
              llowed());  if (cellCheck.isEnabled()) {
                  cellCheck.setSelected(table.getCellSelectionEnabled());
              }
          }
          private void outputSelection() {
              output.append(String.format("Lead: %d, %d. ",
                          table.getSelectionModel().getLeadSelectionIndex(),
                          table.getColumnModel().getSelectionModel().
                              getLeadSelectionIndex()));
              output.append("Rows:");
              for (int c : table.getSelectedRows()) {
                  output.append(String.format(" %d", c));
              }
              output.append(". Columns:");
              for (int c : table.getSelectedColumns()) {
                  output.append(String.format(" %d", c));
              }
              output.append(".\n");
          }

            private class RowListener implements
                  ListSelectionListener {
               public void valueChanged(ListSelectionEvent
                                event) {
                      if (event.getValueIsAdjusting()) {
                          return;
                      }
                      output.append("ROW SELECTION EVENT. ");
```

```
        outputSelection();
    }
}

private class ColumnListener implements ListSelectionListener {

      public void valueChanged(ListSelectionEvent event) {
            if (event.getValueIsAdjusting()) {
                        return;
                    }
                    output.append("COLUMN SELECTION EVENT. ");
                    outputSelection();
                }
            }

      class MyTableModel extends AbstractTableModel {
            private String[] columnNames = {"First Name",
                                            "Last Name",
                                            "Sport",
                                            "# of Years",
                                            "Vegetarian"};
        private Object[][] data = {
          {"Kathy", "Smith",
           "Snowboarding", new Integer(5), new Boolean(false)},
          {"John", "Doe",
           "Rowing", new Integer(3), new Boolean(true)},
          {"Sue", "Black",
           "Knitting", new Integer(2), new Boolean(false)},
          {"Jane", "White",
           "Speed reading", new Integer(20), new Boolean(true)},
          {"Joe", "Brown",
           "Pool", new Integer(10), new Boolean(false)}
        };

        public int getColumnCount() {
            return columnNames.length;
        }

        public int getRowCount() {
            return data.length;
        }

        public String getColumnName(int col) {
            return columnNames[col];
        }

        public Object getValueAt(int row, int col) {
            return data[row][col];
        }

        public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
        }

         /*
          * Don't need to implement this method unless your table's
          * editable.
```

```
             */
            public boolean isCellEditable(int row, int col) {
                //Note that the data/cell address is constant,
                //no matter where the cell appears onscreen.
                if (col < 2) {
                    return false;
                } else {
                    return true;
                }
            }

            /*
             * Don't need to implement this method unless your table's
             * data can change.
             */
            public void setValueAt(Object value, int row, int col)
                { data[row][col] = value;
                fireTableCellUpdated(row, col);
            }

        }

        private static void createAndShowGUI() {
            //Disable boldface controls.
            UIManager.put("swing.boldMetal", Boolean.FALSE);

            //Create and set up the window.
            JFrame frame = new JFrame("TableSelectionDemo");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            //Create and set up the content pane.
            TableSelectionDemo newContentPane = new TableSelectionDemo();
            newContentPane.setOpaque(true); //content panes must be opaque
            frame.setContentPane(newContentPane);

            //Display the window. frame.pack();
            frame.setVisible(true);
        }

        public static void main(String[] args) {
            //Schedule a job for the event-dispatching thread:
            //creating and showing this application's GUI.
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    createAndShowGUI();
                }
            });
        }
    }
```

## Ejercicio 03

Ordenamiento:

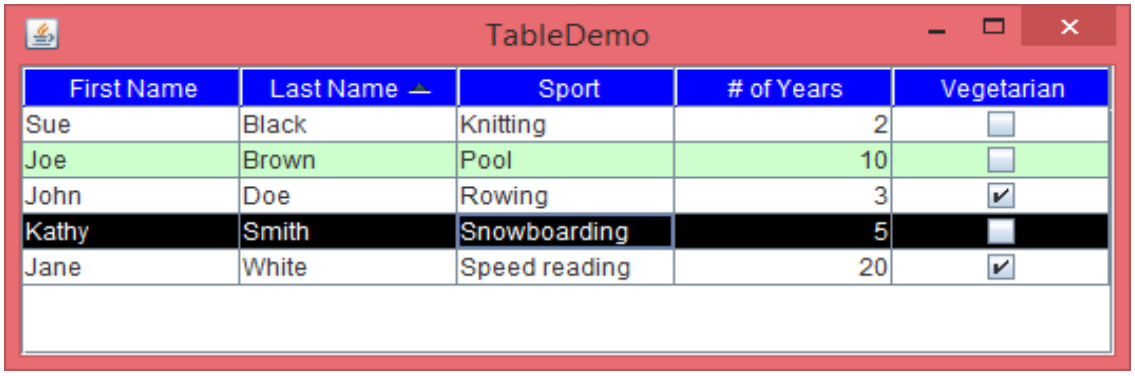



```
package components;



import javax.swing.JFrame;
import javax.swing.JPanel;
import
javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import java.awt.Dimension;
import java.awt.GridLayout;

public class TableSortDemo extends JPanel {
    private boolean DEBUG = false;

    public TableSortDemo() {
        super(new GridLayout(1,0));

        JTable table = new JTable(new MyTableModel());
        table.setPreferredScrollableViewportSize(new Dimension(500,
        70)); table.setFillsViewportHeight(true);
        table.setAutoCreateRowSorter(true);

        //Create the scroll pane and add the table to
        it. JScrollPane scrollPane = new
        JScrollPane(table);

        //Add the scroll pane to this
        panel. add(scrollPane);
    }

    class MyTableModel extends AbstractTableModel {
        private String[] columnNames = {"First Name",
                                        "Last
                                        Name",
                                        "Sport",
                                        "# of Years",
                                        "Vegetarian"}
                                        ;
        private Object[][] data = {
          {"Kathy", "Smith",
           "Snowboarding", new Integer(5), new Boolean(false)},
          {"John", "Doe",
```

```
            "Rowing", new Integer(3), new Boolean(true)},
           {"Sue", "Black",
            "Knitting", new Integer(2), new Boolean(false)},


            {"Jane", "White",
             "Speed reading", new Integer(20), new Boolean(true)},
            {"Joe", "Brown",
             "Pool", new Integer(10), new Boolean(false)}
          };

          public int getColumnCount() {
              return columnNames.length;
          }

          public int getRowCount() {
           return data.length;
          }

                public String getColumnName(int col) {
                    return columnNames[col];
                }

                public Object getValueAt(int row, int col) {
                    return data[row][col];
                }


        public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
        }

        /*
         * Don't need to implement this method unless your table's
         * editable.
         */
        public boolean isCellEditable(int row, int col) {
            //Note that the data/cell address is constant,
            //no matter where the cell appears onscreen.
            if (col < 2) {
                return false;
            } else {
                return true;
            }
        }

        /*
         * Don't need to implement this method unless your table's
         * data can change.
         */
        public void setValueAt(Object value, int row, int col) {
       if (DEBUG) {
                System.out.println("Setting value at " + row + "," + col
                                   + " to " + value
                                        + " (an instance of "
                                        + value.getClass() + ")");
            }
                    data[row][col] = value;
```

```
            if (DEBUG) {
                System.out.println("New value of
                data:"); printDebugData();
            }
        }

        private void printDebugData() {
            int numRows = getRowCount();
            int numCols = getColumnCount();

            for (int i=0; i < numRows; i++) {
                System.out.print("    row " + i + ":");
                for (int j=0; j < numCols; j++) {
                    System.out.print("  " + data[i][j]);
                }
                System.out.println();
            }
            System.out.println("--------------------------");
        }
    }

    private static void createAndShowGUI() {
        //Create and set up the window.
        JFrame frame = new JFrame("TableSortDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
        ;

        //Create and set up the content pane.
        TableSortDemo newContentPane = new TableSortDemo();
        newContentPane.setOpaque(true); //content panes must be opaque
        frame.setContentPane(newContentPane);

        //Display the window.
        frame.pack();
        frame.setVisible(true)
        ;
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable()
        {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

### 1.1.6. JXTable




```
package jxtable;

/*
* TableDemo.java requires no other files.
*/

import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;
import javax.swing.table.AbstractTableModel;

import org.jdesktop.swingx.JXTable;
import org.jdesktop.swingx.decorator.HighlighterFactory;

/**
*     TableDemo is just like SimpleTableDemo, except that it
*     uses a custom TableModel.
*/
public class JXTableDemo extends JPanel {
private boolean DEBUG = false;

public JXTableDemo() {
super(new GridLayout(1,0));

JXTable table = new JXTable(new MyTableModel());
table.setPreferredScrollableViewportSize(new Dimension(500,
70)); table.setFillsViewportHeight(true);

table.getTableHeader().setReorderingAllowed(false) ;
table.getTableHeader().setBackground(Color.BLUE);
table.getTableHeader().setForeground(Color.WHITE);

table.setHighlighters(HighlighterFactory.createSimpleStriping(Highlight
erFactory.CLASSIC_LINE_PRINTER));
table.setRolloverEnabled(true);
table.setSelectionBackground(Color.BLACK);
```

```
        table.setSelectionForeground(Color.WHITE);
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        //Create the scroll pane and add the table to it.
        JScrollPane scrollPane = new JScrollPane(table);

        //Add the scroll pane to this
        panel. add(scrollPane);
        }

        class MyTableModel extends AbstractTableModel {
              private String[] columnNames = {"First Name",
                                              "Last Name",
                                              "Sport",
                                              "# of Years",
                                              "Vegetarian"};


        private Object[][] data = {
                    {"Kathy", "Smith",
           "Snowboarding", new Integer(5), new Boolean(false)},
          {"John", "Doe",
           "Rowing", new Integer(3), new Boolean(true)},
          {"Sue", "Black",
           "Knitting", new Integer(2), new Boolean(false)},
          {"Jane", "White",
           "Speed reading", new Integer(20), new Boolean(true)},
          {"Joe", "Brown",
           "Pool", new Integer(10), new Boolean(false)}
        };

        public int getColumnCount() {
            return columnNames.length;
        }

        public int getRowCount() {
              return data.length;
        }

        public String getColumnName(int col) {
            return columnNames[col];
        }

        public Object getValueAt(int row, int col) {
            return data[row][col];
        }

        /*
         * JTable uses this method to determine the default renderer/
         * editor for each cell.            If we didn't implement this
           method,
         * then the last column would contain text ("true"/"false"),
         * rather than a check box.
         */
        public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
        }

        /*
         * Don't need to implement this method unless your table's
```

```
     * editable.
    */
        public boolean isCellEditable(int row, int col) {
            //Note that the data/cell address is constant,
            //no matter where the cell appears onscreen.
            if (col < 2) {
                return false;
            } else {
                return true;
            }
        }

        /*
         * Don't need to implement this method unless your table's
         * data can change.
         */
        public void setValueAt(Object value, int row, int col) {
        if (DEBUG) {

         System.out.println("Setting value at " + row + "," + col
                                    + " to " + value
                                + " (an instance of "
                                + value.getClass() + ")");
                    }

          data[row][col] = value;
          fireTableCellUpdated(row, col);

                    if (DEBUG) {
                        System.out.println("New value of data:");
                        printDebugData();
                    }
                }

          private void printDebugData() {
              int numRows = getRowCount();
              int numCols = getColumnCount();

              for (int i=0; i < numRows; i++)
                  { System.out.print("

                  row " + i + ":"); for (int
                  j=0; j < numCols; j++) {
                      System.out.print("       " + data[i][j]);
                  }
                  System.out.println();
              }
              System.out.println("--------------------------");
          }
      }


          private static void createAndShowGUI() {
              //Create and set up the window.
              JFrame frame = new JFrame("TableDemo");
              frame.setDefaultCloseOperation(JFrame.EXI
              T_ON_CLOSE);

              //Create and set up the content
```

```
                pane. JXTableDemo
                newContentPane = new

                JXTableDemo();
                newContentPane.setOpaque(true); //content
                panes must be opaque
                frame.setContentPane(newContentPane);

                //Display the
                window.
                frame.pack();
                frame.setVisi
                ble(true);
            }

            public static void main(String[] args) {
                //Schedule a job for the event-dispatching thread:
                //creating and showing this
                application's GUI.
                javax.swing.SwingUtilities.invokeLater
                (new Runnable() {
                    public void run() {
                        createAndShowGUI();
                    }
                });
            }
        }
```

## JDialog

Una ventana de diálogo es una ventana secundaria independiente destinado a llevar aviso temporal aparte de la principal ventana de aplicación Swing. La mayoría de los cuadros de diálogo de presentar un mensaje de error o una advertencia a un usuario, pero Diálogos pueden presentar imágenes, árboles de directorios, o lo que sea compatible con el Swing Application principal que los gestiona.

Para mayor comodidad, varias clases de componentes de Swing pueden crear instancias directamente y cuadros de diálogo de visualización. Para crear diálogos simples, estándar, se utiliza el JOptionPane clase. La clase ProgressMonitor puede poner un cuadro de diálogo que muestra el progreso de una operación. Otras dos clases, JColorChooser y JFileChooser, también suministran diálogos estándar. Para que aparezca un cuadro de diálogo de impresión, puede usar la impresión de API. Para crear un cuadro de diálogo personalizado, utilice la JDialog clase directamente.

El código para diálogos simples puede ser mínimo. Por ejemplo, aquí es un diálogo informativo:

Por diálogos modales más simples, puede crear y mostrar el cuadro de diálogo usando uno de JOptionPane 's show Xxx Dialog métodos. Si el diálogo debe ser un marco interno, a continuación, añadir Internal después de show - por ejemplo, showMessageDialog cambia a showInternalMessageDialog. Si usted necesita para controlar el comportamiento de las ventanas de diálogo de cierre o si usted no desea que el diálogo sea modal, entonces usted debe crear instancias directamente JOptionPane y agregarlo a un JDialog instancia. Entonces invocar setVisible(true) en el JDialog para que parezca.

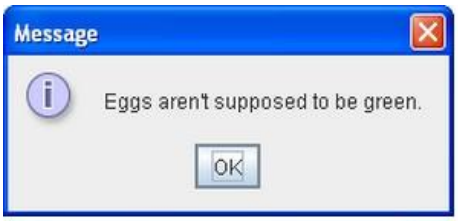
Los dos más útiles showXxxDialog métodos son showMessageDialog y showOptionDialog. El showMessageDialog método muestra un diálogo simple, con un solo botón. El showOptionDialog método muestra un cuadro de diálogo personalizado - puede mostrar una variedad de botones con el texto del botón personalizado, y puede contener un mensaje de texto estándar o una colección de componentes.

Los otros dos showXxxDialog métodos se utilizan con menos frecuencia. El showConfirmDialog método pide al usuario que confirme algo, pero presenta texto estándar botón (Sí / No o el equivalente localizada, por ejemplo) en lugar de texto del botón personalizado a la situación del usuario (Inicio / Cancelar, por ejemplo). Un cuarto método, showInputDialog , está diseñado para mostrar un cuadro de diálogo modal que recibe una cadena del usuario, utilizando un campo de texto, un cuadro combinado editable o una lista.

Estos son algunos ejemplos, tomados de DialogDemo.java, de utilizar showMessageDialog , showOptionDialog , y el JOptionPane constructor. Para obtener más código de ejemplo, ver DialogDemo.java y los demás programas listados en ejemplos que utilizan Diálogos.

### showMessageDialog

Muestra un cuadro de diálogo modal con un botón, que está etiquetado "OK" (o el equivalente localizado). Puede especificar fácilmente el mensaje, el icono y el título que el diálogo muestra. Aquí están algunos ejemplos del uso showMessageDialog:




```
// Título predeterminado y el icono
JOptionPane.showMessageDialog (marco,
    "Los huevos no se supone que son de color verde.");
```


```
// Título personalizado, icono de advertencia
JOptionPane.showMessageDialog (marco,
    "Los huevos no se supone que son de color verde.",
    "Advertencia Inane",
    JOptionPane.WARNING_MESSAGE);
```


```
// Título personalizado, icono de error
JOptionPane.showMessageDialog (marco,
    "Los huevos no se supone que son de color verde.",
    "Error Inane",
    JOptionPane.ERROR_MESSAGE);
```


```
// Título personalizado, ningún icono
JOptionPane.showMessageDialog (marco,
    "Los huevos no se supone que son de color verde.",
    "Un mensaje claro",
    JOptionPane.PLAIN_MESSAGE);
```


```
// Título personalizado, icono personalizado
JOptionPane.showMessageDialog (marco,
    "Los huevos no se supone que son de color verde.",
    "Diálogo personalizado Inane",
    JOptionPane.INFORMATION_MESSAGE,
    icono);
```

**Iconos utilizados por JOptionPane**

| Descripción de los iconos | Aspecto y Comportamiento | Ventanas ven y se sienten |
|---|---|---|
| pregunta |  |  |
| información |  |  |
| advertencia |  |  |
| error |  |  |

### 1.1.7. Calendar

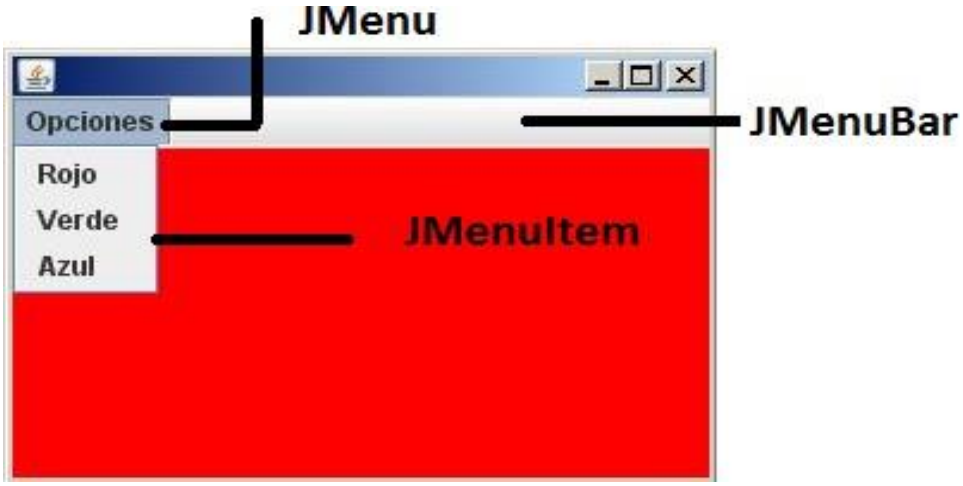

```
package calendar;

import java.awt.Dimension;
import java.awt.Toolkit;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import
javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.table.DefaultTableModel;

import com.toedter.calendar.JDateChooser;

@SuppressWarnings("serial")
public class FrmConsultaCliente extends JFrame {
	private JLabel  jLabel1;
	private JTextField txtNombre;
	private JButton btnFiltrar;
	private JDateChooser jdcFin;
	private JDateChooserjdcInicio;
	private JTable jTable1;
	private JScrollPane jScrollPane1;
	private JLabel jLabel2;
	private DefaultTableModel jTable1Model;

	public FrmConsultaCliente() {
		this.setVisible(false);
		this.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
		this.setTitle("Consulta de clientes");
		this.setBounds(0, 0, 649, 423);
		// este centrado del formulario interno
		Dimension pantalla =
```

```
Toolkit.getDefaultToolkit().getScreen
            Size(); Dimension
            ventana = getSize();
            setLocation((pantalla.width - ventana.width) / 2,
                        (pantalla.height - ventana.height) / 2);
            this.setPreferredSize(new java.awt.Dimension(649, 423));
            getContentPane().setLayout(null);

            jLabel1 = new JLabel();
            getContentPane().add(jLabel1);
            jLabel1.setText("Nombre");
            jLabel1.setBounds(40, 37, 125, 26);

            txtNombre = new JTextField();
            getContentPane().add(txtNombre);
            txtNombre.setBounds(218, 38, 216, 25);

            jLabel2 = new JLabel();
            getContentPane().add(jLabel2);
            jLabel2.setText("Fecha Nacimiento");
            jLabel2.setBounds(40, 89, 140, 24);

            jScrollPane1 = new JScrollPane();
            getContentPane().add(jScrollPane1);
            jScrollPane1.setBounds(40, 137, 533, 232);

            jdcInicio = new JDateChooser();
            getContentPane().add(jdcInicio);
            jdcInicio.setBounds(218, 89, 130, 23);

            jdcFin = new JDateChooser();
            getContentPane().add(jdcFin);
            jdcFin.setBounds(434, 92, 133, 23);

             btnFiltrar = new JButton();
             getContentPane().add(btnFiltrar);
             btnFiltrar.setText("Filtrar");
             btnFiltrar.setBounds(460, 39, 107, 23);

            new DefaultTableModel(new String[][] {}, new
          {
 String[] {{             "Id", "Nombre", "Apellido", "Edad", "Fecha

jTable1Model =
          Nacimiento" });

                    jTable1 = new JTable();
                    jScrollPane1.setViewportView(jTable1);
                    jTable1.setModel(jTable1Model);
              }
              public static void main(String[] args) {
                    try {
                          FrmConsultaCliente jf = new FrmConsultaCliente();
                          jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                          jf.setVisible(true);
                    } catch (Exception e) {
                          e.printStackTrace();
                    }
              }
```

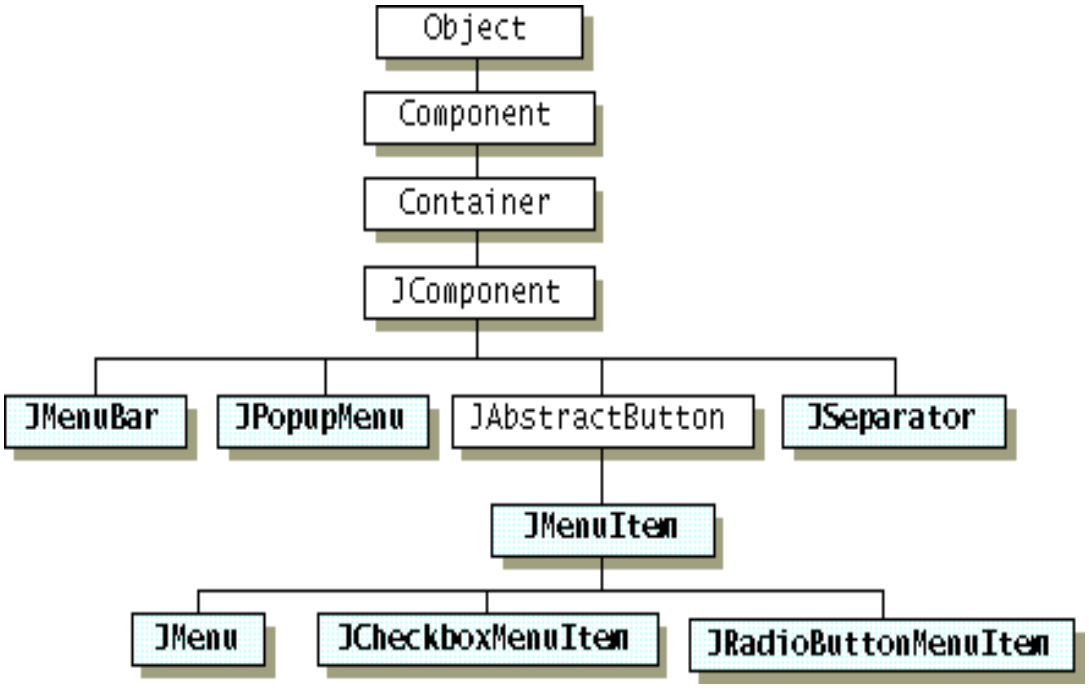
```
}
```

# Resumen

1 Componentes de la GUI




2 Jerarquía de clases




Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- http://docs.oracle.com/javase/tutorial/uiswing/components/menu.html

  Componentes

http://docs.oracle.com/javase/tutorial/uiswing/components/internalframe.html Formularios internos

3 Colores predetminados de las filas:

```
table.setHighlighters(HighlighterFactory.createSimpleStriping(Highl
ighter Factory.CLASSIC_LINE_PRINTER));
```

4 Cambio de color al seleccionar un registro:

```
table.setRolloverEnabled(true)
```

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

http://docs.oracle.com/javase/tutorial/uiswing/components/table.html#show Tablas

http://www.javalobby.org/java/forums/m91834162.html JXTable

5 Cambio de LookAndFeel

```
UIManager.setLookAndFeel("ruta del look and feel");
```

6 Creación del Calendar

```
private JDateChooser jdcFin;
```

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

http://www.jtattoo.net/ Framework jtatoo

http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html Dialogos en java

1

# FORMATOS EN JAVA

## LOGRO DE LA UNIDAD DE APRENDIZAJE

Al término de la unidad, el alumno crea aplicaciones utilizando de manera individual y combinada las clases de formateo y parseo de datos, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

## TEMARIO

**1.2 Tema 2 : Date**

1.2.1 : Date, Calendar
1.2.2 : DateFormat, NumberFormat
1.2.3 : Locale

## ACTIVIDADES PROPUESTAS

- Aplicar formatos a una determinada fecha
- Verificar formatos a cadenas

# 1.2. CONFIGURACIÓN NUMÉRICA Y FECHAS

## 1.2.1. Date, Calendar

La API java nos proporciona un amplio grupo de clases para trabajar con fechas, números y monedas.

A continuación, se muestran las principales:

**java.util.Date** se utiliza como una clase intermedia entre las clases Calendar y DateFormat. Una instancia de la clase Date representa una fecha y hora, expresada en milisegundos.

**java.util.Calendar** proporciona una amplia variedad de métodos que nos permitirán manipular fechas y horas. Por ejemplo, si queremos agregar un mes a una fecha en particular o determinar qué día de la semana será el primer de enero de 2009, la clase Calendar nos permitirá hacerlo.

**java.text.DateFormat** nos permite definer varios formatos y estilos de fecha tales como "01/01/70" o "Enero 1, 1970".

**Java.text.NumberFormat** permite dar formato a números y monedas.

**Java.util.Locale** se usa junto con DateFormat y NumberFormat para formatear fechas, números y monedas sobre la base de idiomas o idiomas-países específicos.

### Uso de las clases de tipo fecha y número

Es importante tener presentes algunos conceptos básicos para el uso de fechas y números en Java. Por ejemplo, si deseamos formatear una fecha para un locale específico, necesitamos crear, primero, el objeto Locale antes que el objeto DateFormat, debido a que el locale será un parámetro de los métodos de la clase DateFormat. A continuación, se muestra un resumen con los usos típicos relacionados con fechas y números:

| Caso de Uso | Pasos a seguir |
|---|---|
| Obtener la fecha y hora actual | 1. Creamos una fecha: Date d = new Date();<br>2. Obtenemos su valor: String s = d.toString(); |
| Obtener un objeto que nos permite realizar cálculos utilizando fechas y horas (locale por defecto) | 1. Creamos un objeto Calendar<br>2. Calendar c = Calendar.getInstance();<br>3. Utlizamos c.add(…) y c.roll(…) para manipular la fecha y la hora. |

| | |
|---|---|
| Obtener un objeto que nos permite realizar cálculos utilizando fechas y horas (locale específico) | 1. Creamos un objeto Locale:<br>Locale loc = new Locale (language); o<br>Locale loc = new Locale (language, country);<br>2. Creamos un objeto Calendar para ese Locale:<br>3. Calendar c = Calendar.getInstance(loc)<br>4. Utlizamos c.add(…) y c.roll(…) para manipular la fecha y la hora. |
| Obtenemos un objeto que nos permita realizar cálculos con fechas y horas, y crear formatos de salida para diferentes Locales con diferentes estilos de fecha. | 1. Creamos un objeto Calendar:<br>2. Calendar c = Calendar.getInstance();<br>3. Creamos un objeto Locale para cada localización:<br>4. Locale loc = new Locale(...);<br>5. Convertimos el objeto Calendar en un objeto Date:<br>6. Date d = c.getTime();<br>7. Creamos unobjeto DateFormat para cada Locale:<br>8. DateFormat df = DateFormat.getDateInstance (style, loc);<br>9. Usamos el método format() para crear fechas con formato:<br>10. String s = df.format(d); |
| Obtenemos un objeto que nos permita dar formato a números y monedas utilizando diferentes Locales. | 1. Creamos un objeto Locale para cada localización:<br>2. Locale loc = new Locale(...);<br>3. Creamos un objeto de tipo NumberFormat:<br>NumberFormat nf = NumberFormat.getInstance(loc); - **o** - NumberFormat nf = NumberFormat.getCurrencyInstanc e(loc);<br>3.Use el método format() para crear una salida con formato<br>String s = nf.format(someNumber); |

### 1.2.2.Date

Una instancia de la clase Date representa una fecha / hora específica. Internamente, la fecha y hora es almacenada como un tipo primitivo long. Específicamente, el valor representa el número de milisegundos entre la fecha que se quiere representar y el 1 de enero de 1970.

En el siguiente ejemplo, determinaremos cuánto tiempo tomaría pasar un trillón de milisegundos a partir del 1 de enero de 1970.

```
import java.util.*; class
TestDates {
  public static void main(String[] args) {
    Date d1 =
          new Date(1000000000000L); // un trillón de milisegundos
    System.out.println("1st date " + d1.tostring());
  }
}
```

Obtendremos la siguiente salida:

```
1st date Sat Sep OB 19:46:40 MDT 2001
```

Sabemos ahora que un trillón de milisegundos equivale a 31 años y 2/3 de año.

Aunque la mayoría de métodos de la clase Date se encuentran deprecados, aún es común encontrar código que utilice los métodos getTime y setTime:

```
import java.util.*; class
TestDates
   public static void main(String[] args) {
   Date d1 =
   new Date(1000000000000L); // a trillion!

   System.out.println("1st date " + d1.toString());
   d1.setTime(d1.getTime() + 3600000); // 3600000 millis / 1
   hora System.out.println("new time " + d1.toString());
  }
}
```

Obtendremos la siguiente salida:

```
1st date Sat Sep 08 19:46:40 MDT 2001
new time Sat Sep 08 20:46:40 MDT 2001
```

### Calendar

La clase Calendar ha sido creada para que podamos manipular fechas de manera sencilla. Es importante notar que la clase Calendar es una clase abstracta. No es posible decir:

Calendar c = new Calendar(); **// ilegal, Calendar es una clase abstracta**

Para crear una instancia de la clase Calendar, debemos usar uno de los métodos estáticos sobrecargados getInstance():

Calendar cal = Calendar.**getInstance();**

A continuación, mostramos un ejemplo de

```
aplicación: import java.util.*;
class Dates2 {
  public static void main(String[] args) {
  Date d1 = new Date(1000000000000L);
    System.out.println("1st date " + d1.toString());

    Calendar c = Calendar.
    getInstance(); c.setTime(d1);                  // #1

    if(c.SUNDAY == c. getFirstDayofWeek() // #2
    System.out.println("Sunday is the first day of the week");
    System.out.println("trillionth milli day of week is "
                    + c.get(c.DAY_OF_WEEK));                  // #3

    c. add(Calendar. MONTH, 1);                              // #4

    Date d2 = c.getTime();                                   // #5
    System.out.println("new date " + d2.toString() );
                }
              }
```

Al ejecutar la clase, obtendremos la siguiente salida:

```
1st date Sat Sep 08 19:46:40
MDT 2001 Sunday is the first
day of the week trillionth milli day
of week is 7
new date Mon Oct 08 20:46:40 MDT 2001
```

Asignamos el objeto d1 (de tipo Date) a la variable c (referencia a un objeto Calendar). Usamos luego el campo SUNDAY para determinar si para nuestra JVM, SUNDAY es considerado el primer día de la semana. (Para algunos locales, MONDAY es el primer día de la semana). La clase Calendar proporciona campos similares para los días de la semana, meses y días del mes, etc.

El método add. Este método permite agregar o sustraer unidades de tiempo en base a cualquier campo que especifiquemos para la clase Calendar. Por ejemplo:

```
c.add(Calendar.HOUR, -4);                    // resta 4 horas de c
c.add(Calendar.YEAR, 2);                     // agrega 2 años a c
c.add(Calendar.DAY_OF_WEEK, -2); // resta dos días a c
```

Otro método importante en la clase Calendar es roll(). Este método trabaja como add(), con la excepción de que cuando una parte de la fecha es incrementada o decrementada, la parte más grande de dicha fecha no será incrementada o decrementada. Por ejemplo:

```
// asumiendo que c representa el 8 de Octubre de 2001
c.roll(Calendar.MONTH, 9);                   // notemos el año en la
salida Date d4 = c.getTime();
System.out.println("new date " + d4.toString() );
```

La salida será la siguiente:

**new date Fri Jul 08 19:46:40 MDT 2001**

Debemos notar que el año no cambió, a pesar de haber agregado 9 meses a la fecha original (Octubre). De manera similar, si invocamos a roll() con el campo HOUR, no cambiará la fecha, el mes o el año.

### 1.2.3. DateFormat, NumberFormat

Dado que ya aprendimos a crear y manipular fechas, es tiempo de aprendar a darles un formato específico. A continuación, se muestra un ejemplo en el que una misma fecha es visualizada utilizando diferentes formatos:

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

class Dates3 {
    public static void main(String[] args) {
      Date d1 = new Date(1000000000000L);

      DateFormat[] dfa = new DateFormat[6];
      dfa[0] = DateFormat.getInstance();
      dfa[1] = DateFormat.getDateInstance();
      dfa[2] = DateFormat.getDateInstance(DateFormat.SHORT);
      dfa[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
```

```
  dfa[4] = DateFormat.getDateInstance(DateFormat.LONG);
  dfa[5] = DateFormat.getDateInstance(DateFormat.FULL);

  for(DateFormat df : dfa)
    System.out.println(df.format(d1));
    }
  }
```

Obtendremos la siguiente salida:

```
9/8/01 7:46 PM
Sep 8, 2001
9/8/01
Sep 8, 2001
September 8, 2001
Saturday, September 8, 2001
```

La clase DateFormat es **una clase abstracta**, no es posible utilizar el operador new para crear instancias de la clase DateFormat. En este caso, utilizamos dos métodos de tipo factory: **getInstance()** y **getDateInstance()**. Debemos notar que getDateInstance() está sobrecargado. Este método tiene otra variante para poder ser utilizado con objetos de tipo Locale.

Utilizamos campos estáticos de la clase DateFormat para customizar nuestras instancias de tipo DateFormat. Cada uno de estos campos estáticos representa un estilo de formato. Finalmente, utilizamos el método format() para crear representaciones de tipo String de las diferentes versiones de formato aplicadas a la fecha específica.

El último método con el que deberíamos familiarizarnos es el método parse(). Este método toma un String formateado por DateFormat y lo convierte en un objeto de tipo Date. En realidad, ésta puede ser una operación riesgosa, dado que podría recibir como input un String que se encuentre mal formateado. Debido a ésto, el método parse() podría lanzar la siguiente excepción: ParseException. El código mostrado a continuación, crea una instancia de la clase Date, utiliza el método DateFormat.format() para convertirla a String, y luego usa DateFormat.parse() para convertirla nuevamente a Date:

```
Date d1 = new Date(1000000000000L);
System.out.println("d1 = " +
d1.tostring());

DateFormat df = DateFormat.getDateInstance(
                    DateFormat.SHORT)
; String s = df.format(d1);
System.out.println(s);
```

```
try {
Date d2 = df.parse(s);
System.out.println("parsed = " + d2.toString());
} catch (ParseException pe) {
System.out.println("parse exc"); }
```

Obtendremos la siguiente salida:

```
d1 = Sat Sep 08 19:46:40 MDT 2001
9/8/01
parsed = Sat Sep 08 00:00:00 MDT 2001
```

**Importante:**

Debemos notar que al usar el estilo SHORT, hemos perdido precisión al convertir el objeto Date a un String. Se evidencia esta pérdida de precisión cuando retornamos la cadena a un objeto Date nuevamente, se visualiza como hora: media noche en lugar de 17:46:00.

### 1.2.4. Locale

Las clases DateFormat y NumberFormat pueden usar instancias de la clase Locale para customizer un formato a una localización específica. Para Java, un Locale es una región geográfica, política o cultural específica. Los constructores básicos de la clase Locale son:

- Locale(String language)
- Locale(String language, String country)

El argumento language representa un código ISO 639. Existen aproximadamente 500 códigos ISO para el lenguaje, incluído el lenguaje Klingon ("tlh").

Podemos ser más específicos y no sólo indicar un lenguaje en particular, sino también en qué país se habla ese lenguaje. En el ejemplo mostrado a continuación, se crea un Locale para italiano y otro para el italiano hablado en Suiza:

```
Locale locPT = new Locale("it");// Italian
Locale locBR = new Locale("it", "CH"); // Switzerland
```

El uso de estos locales en una fecha producirá la siguiente salida:

```
sabato 1 ottobre 2005
sabato, 1. ottobre 2005
```

A continuación, se muestra un ejemplo más detallado que involucra el uso de la clase Calendar:

```
Calendar c = Calendar.getInstance();
c.set(2010, 11, 14);                                    // 14 de Diciembre de 2010
                                // (month is 0-based

Date d2 = c.getTime();

Locale locIT = new Locale("it", "IT"); // Italy
Locale locPT = new Locale("pt"): // Portugal
Locale locBR = new Locale("pt", "BR"); // Brazil
Locale locIN = new Locale("hi", "IN"); // India
Locale locJA = new Locale("ja");  // Japan

DateFormat dfUS = DateFormat.getInstance();
System.out.println("US        " + dfUS.format(d2));

DateFormat dfUSfull = DateFormat.getDateInstance(
                                DateFormat.FULL);
System.out.println("US full " + dfUSfull.format(d2));

DateFormat dfIT = DateFormat.getDateInstance(
                                DateFormat.FULL, locIT);
System.out.println("Italy                           " + dfIT.format(d2));

DateFormat dfPT = DateFormat.getDateInstance(
                                DateFormat.FULL, locPT);
System.out.println("Portugal " + dfPT.format(d2));

DateFormat dfBR = DateFormat.getDateInstance(
                                DateFormat.FULL, locBR);
System.out.println("Brazil " + dfBR.format(d2));

DateFormat dfIN = DateFormat.getDateInstance(
                                DateFormat.FULL, locIN);
System.out.println("India                            " + dfIN.format(d2));

DateFormat dfJA = DateFormat.getDateInstance(
                                DateFormat.FULL, locJA);
System.out.println("Japan                             " + dfJA.format(d2));
```

Obtendremos la siguiente salida:

```
US   12/14/10 3:32 PM
US full Sunday, December 14, 2010
Italy   domenica 14 dicembre 2010
```

```
Portugal Domingo, 14 de Dezembro de 2010
Brazil. Domingo, 14 de Dezembro de 2010
India                    ??????, ??
??????, ????
Japan                    2010?12?14?
```

Notemos como el computador en el que se ejecutó el ejercicio previo no está configurado para soportar los locales de India o Japón.

**Importante:**

Recordemos que los objetos de tipo DateFormat y NumberFormat pueden tener Locales asociados sólo en tiempo de instanciación. No es posible cambiar el locale de una instancia ya creada: no existe ningún método que permita realizar dicha operación.

Mostramos, a continuación, un ejemplo de uso de los métodos getDisplayCountry() y getDisplayLanguage(). Ambos métodos nos permitirán crear cadenas que representen un lenguaje y país específicos correspondientes a un Locale:

```
Calendar c =
Calendar.getInstance(); c.set(2010,
11, 14);
Date d2 = c.getTime();

Locale locBR = new Locale("pt", "BR"); // Brazil
Locale locDK = new Locale("da", "DK"); // Denmark
Locale locIT = new Locale("it", "IT"); // Italy

System.out.println("def " + locBR.getDisplayCountry());
System.out.println("loc " +
locBR.getDisplayCountry(locBR));

System.out.println("def " + locDK.getDisplayLanguage());
System.out.println("loc " +
locDK.getDisplayLanguage(locDK)); System.out.println("D>I "
+ locDK.getDisplayLanguage(locIT));
```

Visualizaremos losiguiente:

```
def Brazil
loc Brasil
def Danish
loc dansk
D>I danese
```

El locale por defecto para el país Brasil es "Brazil", y el default para el lenguaje danés es "Danish". En Brasil, el país es llamado "Brasil", y en Dinamarca el lenguaje es llamado "dansk". Finalmente, podemos notar que, en Italia, el lenguaje danés es llamado "danese".

## Ejercicio 01

```
package fechas;

import
java.text.DateFormat;
import java.util.Date;
import java.util.Locale;

public class ManejoFechas {

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

      d = new Date();

      //Fecha actual(todos los parametros)
      System.out.println(d);

      //Fecha en milisegundos
      long f = System.currentTimeMillis();
      System.out.println(f);

      //Formatos
      DateFormat df1 = DateFormat.getDateInstance();
      System.out.println(df1.format(d));

      DateFormat df2 = DateFormat.getDateInstance(DateFormat.LONG);
      System.out.println(df2.format(d));

      DateFormat df3 = DateFormat.getDateInstance(DateFormat.SHORT);
      System.out.println(df3.format(d));

      DateFormat df4 = DateFormat.getDateInstance(DateFormat.MEDIUM);
      System.out.println(df4.format(d));

       DateFormat df5 =
 DateFormat.getDateInstance(DateFormat.LONG,Locale.FRENCH);
      System.out.println(df5.format(d));


      DateFormat df6 = DateFormat.getTimeInstance();
      System.out.println(df6.format(d));

      DateFormat df7 = DateFormat.getTimeInstance(DateFormat.LONG);
      System.out.println(df7.format(d));

      DateFormat df8 = DateFormat.getTimeInstance(DateFormat.SHORT);
      System.out.println(df8.format(d));

      DateFormat df9 = DateFormat.getTimeInstance(DateFormat.MEDIUM);
      System.out.println(df9.format(d));
```

```
            DateFormat df10 =
DateFormat.getTimeInstance(DateFormat.LONG,Locale.FRENCH);
            System.out.println(df10.format(d));

      }

}
```

```
      Thu Mar 12 15:24:23 COT 2015
      1426191864035
      12-mar-2015
      12 de marzo
      de 2015
      12/03/15
      12-mar-2015
      12 mars 2015
      15:24:23
      15:24:23 COT
      15:24
      15:24:23
      15:24:23 COT
```

## Ejercicio 02

```
package formatos;

public class ManejoFormatos {

      public static void main(String[] args) {

            String cad1 = "89";
            String cad2 = "1 ";
            String cad3 = "ab";

            System.out.println(cad1.matches("[0-5][0-9]"));
            System.out.println(cad1.matches("[0-9][0-9]"));
            System.out.println(cad2.matches("[0-9][0-9]"));
            System.out.println(cad3.matches("[0-9][0-9]"));

            System.out.println(cad1.matches("\\d\\d"));
            System.out.println(cad2.matches("\\d\\d"));
            System.out.println(cad3.matches("\\d\\d"));

            System.out.println(cad1.matches("\\d{2}"));
            System.out.println(cad2.matches("\\d{2}"));
            System.out.println(cad3.matches("\\d{2}"));

            String dni1="12345678";
            String dni2="12 45678";

            System.out.println(dni1.matches("\\d{8}"));
            System.out.println(dni2.matches("\\d{8}"));
```

```
            String          placa1
            ="AB1554";      String
            placa2       ="ab1454";
            String   placa3   ="AB-
            144";   String   placa4
            ="AB 144";

            System.out.println(placa1.matches("[A-Z]{2}[0-9]{4}"));
            System.out.println(placa2.matches("[A-Z]{2}[0-9]{4}"));
            System.out.println(placa3.matches("[A-Z]{2}[0-9]{4}"));
            System.out.println(placa4.matches("[A-Z]{2}[0-9]{4}"));

      }

}
```

```
false
true
false
false
true
false
false
true
false
false
true
false
true
false
false
false
```

## Ejercicio 03

```
package fechas;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class ManejoFormatoPersonalizado {


      public static void main(String[]

            args) { Date f = new Date();
```

```
            SimpleDateFormat sdf = new SimpleDateFormat("d - W - M
            "); System.out.println(sdf.format(f));

            SimpleDateFormat sdf1 = new SimpleDateFormat("HH:mm:ss
            a"); System.out.println(sdf1.format(f));

            SimpleDateFormat sdf2 = new SimpleDateFormat("'Lima,' d
'de' MMMM 'de' y");
            System.out.println(sdf2.format(f));

            SimpleDateFormat sdf3 = new SimpleDateFormat("M MM MMM
            MMMM"); System.out.println(sdf3.format(f));

            SimpleDateFormat sdf4 = new SimpleDateFormat("y yy yyy
            yyyy"); System.out.println(sdf4.format(f));

            SimpleDateFormat sdf5 = new
SimpleDateFormat("EEEE - MMMM",Locale.ENGLISH);
            System.out.println(sdf5.format(f));



            SimpleDateFormat sdf6 = new
                        SimpleDateFormat("'Ha trasncurrido del año' D
'días'");
            System.out.println(sdf6.format(f));

            SimpleDateFormat sdf7 = new
                        SimpleDateFormat("m 'minutos,' s 'segundos' 'con'
                        S
'milisegundos'");
            System.out.println(sdf7.format(f));

            SimpleDateFormat sdf8 = new
                        SimpleDateFormat("EEEE - MMMM -
            y"); System.out.println(sdf8.format(f));

            SimpleDateFormat sdf9 = new
                        SimpleDateFormat("EEEE - MMMM - y",
Locale.ENGLISH);
            System.out.println(sdf9.format(f));
      }
}
```

```
12 - 2 - 3
15:25:44 PM
Lima, 12 de marzo de 2015
3 03 mar marzo
2015 15 2015
2015 Thursday
- March
Ha trasncurrido del año 71 días
25 minutos, 44 segundos con 747
milisegundos jueves - marzo - 2015
```

# Resumen

1. Constantes para fechas SimpleDateformat

| Letter | Date or Time Component | Presentation | Examples |
|---|---|---|---|
| G | Era designator | Text | AD |
| y | Year | Year | 1996; 96 |
| M | Month in year | Month | July; Jul; 07 |
| w | Week in year | Number | 27 |
| W | Week in month | Number | 2 |
| D | Day in year | Number | 189 |
| d | Day in month | Number | 10 |
| F | Day of week in month | Number | 2 |
| E | Day in week | Text | Tuesday; Tue |
| a | Am/pm marker | Text | PM |
| H | Hour in day (0-23) | Number | 0 |
| k | Hour in day (1-24) | Number | 24 |
| K | Hour in am/pm (0-11) | Number | 0 |
| h | Hour in am/pm (1-12) | Number | 12 |
| m | Minute in hour | Number | 30 |
| s | Second in minute | Number | 55 |
| S | Millisecond | Number | 978 |
| z | Time zone | General time zone | Pacific Standard Time; PST; GMT-08:00 |
| Z | Time zone | RFC 822 time zone | -0800 |

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

UNIDAD

2

# GESTIÓN DE EXCEPCIONES EN JAVA

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones y gestiona los diferentes tipos de excepciones en Java haciendo uso de manejadores de excepciones, declarando y sobrescribiendo métodos que capturan y lanzan excepciones.

**TEMARIO**

**2.1 Tema 1 : Gestión de excepciones en Java**

2.1.1 : Manejo de excepciones usando las sentencias Try, Catch y Finally
2.1.2 : Propagación de excepciones
2.1.3 : Definición de excepciones y jerarquía de Excepciones
2.1.4 : Errores y excepciones comunes

**ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran la declaración y gestión de excepciones.
- Los alumnos implementan una aplicación java swing que gestiona excepciones usando bloques TRY – CATCH – FINALLY.

# 2.1. GESTIÓN DE EXCEPCIONES EN JAVA

En muchos lenguajes, escribir código que verifique y gestione errores suele ser tedioso; incluso, puede prestarse a confusión si la codificación se hace poco entendible, esto es lo que se conoce como un código “spaghetti”. Java proporciona un mecanismo eficiente para gestionar errores y excepciones: el gestor de excepciones de java.

Este mecanismo facilitará la detección de errores sin tener que escribir código especial para verificar los valores retornados. El código que gestiona las excepciones se encuentra claramente separado del código que las genera, esto permite utilizar el mismo código de gestión de excepciones con un amplio rango de posibles excepciones.

## 2.1.1. Manejo de excepciones usando las sentencias Try, Catch y Finally

El término "exception" significa "condición excepcional" y es una ocurrencia que altera el flujo normal de un programa. Son múltiples los factores que pueden generar una excepción, incluyendo fallas de hardware, agotamiento de recursos, entre otros.

Cuando un evento excepcional ocurre en Java, se dice que una excepción ha sido lanzada. El código responsable por hacer algo con respecto a la excepción generada es llamado el código gestor de excepciones y atrapa la excepción lanzada.

La palabra clave TRY es usada para definir un bloque de código dentro del cual puede ocurrir una excepción. Una o más cláusulas CATCH están asociadas ana excepción específica o grupo de excepciones. A continuación, se muestra un ejemplo:

```
try {
        // Aquí colocamos código que podría lanzar alguna excepción.

} catch(MiPrimeraExcepcion) {
         // Aquí colocamos código que gestiona la excepción

 } catch(MiSegundaExcepcion) {
        // Aquí colocamos código que gestiona la excepción

 }

 // Aquí colocamos código que no es riesgoso.
```

En el ejemplo, las líneas que definamos dentro de la cláusula TRY constituyen la zona protegida. Las líneas dentro de la primera cláusula CATCH constituyen el gestor de excepciones para excepciones de tipo MiPrimeraExcepcion.

Imaginemos ahora que tenemos el siguiente código:

```
try {

        obtenerArchivoDeLaRed
        leerArchivoYCargarTabla

}catch(NoPuedeObtenerArchivoDeLaRed) {
        mostrarMensajeDeErrorDeRed

}
```

En el ejemplo anterior, podemos observar cómo el código susceptible de una operación riesgosa, en este caso, cargar una tabla con los datos de un archivo, es agrupado dentro de un bloque TRY. Si la operación obtenerArchivoDeLaRed falla, no leeremos el archivo ni cargaremos la tabla. Esto se debe a que la operación previa ya lanzó una excepción.

Las cláusulas TRY y CATCH proporcionan un mecanismo eficiente para atrapar y gestionar excepciones, sin embargo, aún no estamos controlando cómo inicializar variables o asignar valores a objetos que deben ser inicializados ocurra o no ocurra una excepción.

Un bloque FINALLY encierra código que es siempre ejecutado después del bloque TRY, sea que se produzca o no una excepción. Finally se ejecuta, incluso, si existe una sentencia return dentro del bloque TRY:

“El bloque finally se ejecuta después de que la sentencia return es encontrada y antes de que la sentencia return sea ejecutada”.

El bloque FINALLY es el lugar correcto para, por ejemplo, cerrar archivos, liberar sockets de conexión y ejecutar cualquier otra operación de limpieza que el código necesite. Si se generó una excepción, el bloque FINALLY se ejecutará después de que se haya completado el bloque CATCH respectivo.

La cláusula FINALLY no es obligatoria. Por otro lado, dado que el compilador no requiere una cláusula CATCH, podemos tener código con un bloque TRY seguido inmediatamente por un bloque FINALLY. Este tipo de código es útil cuando la excepción será retornada al método que hizo la llamada. Usar un bloque FINALLY permite ejecutar código de inicialización así no exista la cláusula CATCH.

A continuación, se muestran algunos ejemplos de aplicación de las cláusulas TRY, CATCH y FINALLY.

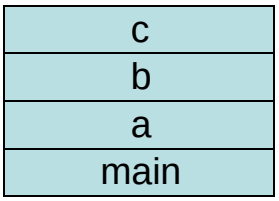
```
//El siguiente código muestra el uso de TRY y FINALLY:

 try {
  // hacer algo
} finally {
  //clean up
}


//El siguiente código muestra el uso de TRY, CATCH y FINALLY:

try {
  // hacer algo
} catch (SomeException ex) {
  // do exception handling
} finally {
  // clean up
}


//El siguiente código es inválido:

try {
  // hacer algo
}
// Se necesita un CATCH o un FINALLY aquí
System.out.println("fuera del bloque try");


//El siguiente código es inválido:

try {
  // hacer algo
}
// No se puede tener código entre una TRY y un CATCH
System.out.println("fuera del bloque try"); catch(Exception
ex) { }
```

## 2.1.2. Propagación de excepciones

### Pila de llamadas

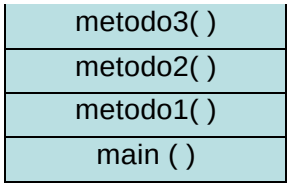
La mayoría de lenguajes manejan el concepto de pila de llamadas o pila de métodos. La pila de llamadas es la cadena de métodos que un programa ejecuta para obtener el método actual. Si nuestro programa inicia con el método main(), y

éste invoca al método a(), el cual invoca al método b() y éste al método c(), la pila de llamadas tendría la siguiente apariencia:




Como se puede observar, el último método invocado se encuentra en la parte superior de la pila, mientras que el primer método llamado se encuentra en la parte inferior de la misma. El método en la parte más alta de la pila será el método que actualmente estamos ejecutando. Si retrocedemos en la pila, nos estaremos moviendo desde el método actual hacia el método previamente invocado.

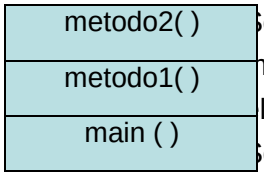
La pila de llamadas mientras el método 3 está ejecutándose:




El método 2 invoca al método 3
El método 1 invoca al método 2
main invoca al método 1
main inicia la ejecución

**Orden en el cual los métodos son colocados en la pila de llamadas**

La pila de llamadas después de terminar la ejecución del método 3.




Se completará el
método 2 Se completará
l método 1
Se completará el método main y la JVM hará un exit

**Orden en el cual los métodos completan su ejecución**

**Esquema de propagación de excepciones**

Una excepción es primero lanzada por el método en la parte más alta de la pila. Si no es atrapada por dicho método, la excepción caerá al siguiente nivel de la pila, al siguiente método. Se seguirá este proceso hasta que la excepción sea atrapada o hasta que llegar a la parte más baja de la pila. Esto se conoce como propagación de excepciones.

Una excepción que nunca fue atrapada ocasionará que se detenga la ejecución de nuestro programa. Una descripción de la excepción será visualizada y el detalle de la pila de llamadas será visualizado. Esto nos ayudará a hacer un “debug” sobre nuestra

aplicación indicándonos qué excepción fue lanzada y qué método la lanzó.

### 2.1.3. Definición de excepciones y jerarquía de Excepciones

Cada excepción es una instancia de una clase que tiene a la clase Exception dentro de su jerarquía de herencia. Es decir, las excepciones son siempre subclases de la clase java.lang.Exception.

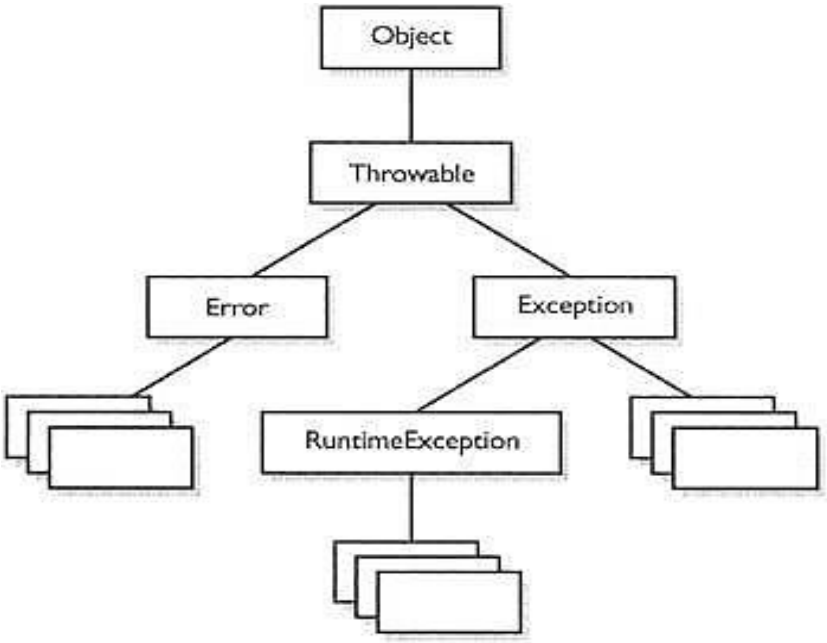
Cuando una excepción es lanzada, un objeto de un subtipo particular de excepción es instanciado y pasado al gestor de excepciones como un argumento de la cláusula CATCH. A continuación, se muestra un ejemplo:

```
try {
  // colocamos algún código aquí
}
catch (ArrayIndexOutOfBoundsException e) {
  e.printStackTrace();
}
```

En el ejemplo, e es una instancia de la clase ArrayIndexOutOfBoundsException. Como cualquier otro objeto, podemos invocar a cualquiera de sus métodos.

**Jerarquía de excepciones**

Todas las clases de excepciones son subtipos de la clase Exception. Esta clase se deriva de la clase Throwable (la cual deriva de la clase Object).




Podemos observar que existen dos subclases que se derivan de la clase Throwable:

Exception y Error. Las clases que se derivan de Error, representan situaciones inusuales que no son causadas por errores del programa e indican situaciones que normalmente no se darían durante la ejecución del programa, como por ejemplo, “JVM running out of memory”.

Generalmente, nuestra aplicación no estará capacitada para recuperarse de un error, por lo que no será necesario que los gestionemos. Los errores, técnicamente, no son excepciones, dado que no se derivan de la clase Exception.

En general, una excepción representa algo que sucedió no como resultado de un error de programación, sino, más bien, porque algún recurso no se encuentra disponible o alguna otra condición requerida para la correcta ejecución no está presente. Por ejemplo, si se supone que la aplicación debe comunicarse con otra aplicación o computadora que no está contestando, esta sería una excepción no causada por un bug. En la figura previa también se aprecia un subtipo de excepción llamado RuntimeException. Estas excepciones son un caso especial porque ellas, a veces, indican errores del programa. Ellas también representan condiciones excepcionales raras y/o difíciles de gestionar.

Existen dos maneras de obtener información relacionada con una excepción:

- La primera es del tipo de excepción en sí misma (el parámetro en la cláusula CATCH).
- La segunda, es obteniéndola del mismo objeto Exception.

La clase Throwable, ubicada en la parte superior de la jerarquía del árbol de excepciones, proporciona a sus descendientes algunos métodos que son útiles para los gestores de transacciones. Uno de estos métodos es printStackTrace(). Este método imprime primero una descripción de la excepción y el método invocado más recientemente, continua así imprimiendo el nombre de cada uno de los métodos en la pila de llamadas.

## 2.1.4. Errores y excepciones comunes

### Origen de las excepciones

Es importante entender qué causan las excepciones y de dónde vienen éstas. Existen dos categorías básicas:

- **Excepciones de la JVM**. Son las excepciones y errores lanzadas por la Java Virtual Machine.

- **Excepciones programáticas**. Aquellas que son lanzadas excplícitamente por la aplicación y/o APIs propias del desarrollador.

### Excepciones lanzadas por la JVM

Son excepciones en las que no es posible que el compilador pueda detectarlas antes del tiempo de ejecución. A continuación, se muestra un ejemplo:

```
class NPE {
static String s;
  public static void main(String [] args)
    { System.out.println(s.length());
  }
}
```

En el ejemplo anterior, el código compilará satisfactoriamente y la JVM lanzará un **NullPointerException** cuando trate de invocar al método **length().**

Otro tipo de excepción típica dentro de esta categoría es StackOverflowError. La manera más común de que esto ocurra es creando un método recursivo. Ejemplo:

```
void go() {                  //Aplicación errada de recursividad
   go();
}
```

Si cometemos el error de invocar al método go(), el programa caerá en un loop infinito, invocando al método go() una y otra vez hasta obtener la excepción stackOverflowError. Nuevamente, **sólo la JVM** sabe cuando ocurre este momento y la JVM será la fuente de este error.

**Excepciones lanzadas programáticamente**

Muchas clases pertenecientes a la API java tienen métodos que toman objetos String como argumentos y convierten estos Strings en valores numéricos primitivos.

Se muestra, a continuación, un ejemplo de excepción perteneciente a este contexto:

```
 int parseInt (String s) throws NumberFormatException {
 boolean parseSuccess
 = false; int result = 0;
 // se implementa la lógica de la
 conversion if (!parseSuccess) // si
 la conversion falló throw new
 NumberFormatException();
 return result;
}
```

Otro ejemplo de excepción programática es AssertionError (que en realidad no es una excepción, pero es lanzada programáticamente). Se genera un IllegalArgumentException.

Es posible, también, crear nuestras propias excepciones. Estas excepciones caen dentro de la categoría de excepciones generadas programáticamente.

### Resumen de excepciones y errores comunes

Se muestran, a continuación, los errores y excepciones más comunes generados por una aplicación java

| *Descripción y Fuentes de Excepciones comunes* | | |
|---|---|---|
| **Excepción** | **Descripción** | **Lanzada generalmente por:** |
| **ArrayIndexOutOfBoundsException** | Lanzada cuando se intenta acceder a un arreglo con un valor de índice inválido (sea éste negativo o superior a la longitud del arreglo). | JVM |
| **ClassCastException** | Lanzada cuando intentamos convertir una referencia a variable a un tipo que falla la prueba de casteo IS-A. | JVM |
| **IllegalArgumentException** | Lanzada cuando un método recibe un argumento formateado de manera diferente a lo que el método esperaba. | Programáticamente |
| **IllegalStateException** | Lanzada cuando el estado del entorno no coincide con la operación que se intenta ejecutar. Por ejemplo, usar un objeto de la clase Scanner que ha sido cerrado previamente. | Programáticamente |
| **NullPointerException** | Lanzada cuando intentamos acceder a un objeto con una variable de referencia cuyo valor actual es null. | JVM |
| **NumberFormatException** | Lanzada cuando un método que convierte un String a un número recibe un String que no puede ser convertido. | Programáticamente |

| | | |
|---|---|---|
| **AssertionError** | Lanzada cuando una sentencia Boolean retorna el valor falso después de ser evaluada. | Programáticamente |
| **ExceptionInInitializerError** | Lanzada cuando intentamos inicializar una variable estática o un bloque de inicialización. | JVM |
| **StackOverflowError** | Típicamente lanzada cuando un método es invocado demasiadas veces, por ejemplo, recursivamente. | JVM |
| **NoClassDefFoundError** | Lanzada cuando la JVM no puede ubicar una clase que necesita, por un error de línea de comando, problemas de classpath, o un archivo class perdido. | JVM |

## Ejercicio 01

```
package demo01;

public class Estructura {

	public static void main(String[] args) {
		// 1  Ingresa al catch si es que sucede una Exception
		// 2  Luego de la caida ya no se ejecuta
		// 3  El bloque finally se ejecuta suceda o no una caida

		System.out.println(1);
		try {
			System.out.println(2
			); int x = 5 / 0;
			System.out.println(3
			);
		} catch (Exception e) {
			//Finalizar el sistema
			//System.exit(0); No ingresa al
			finally System.out.println(4);
		} finally {
			System.out.println(5);
		}
		System.out.println(6);

	}
```

```
1
2
4
5
6
```

## Ejercicio 02

```
package demo02;

public class PosibilidadesDelTryCatch {

	public static void main(String[] args) {

		// 1 Puede existir un bloque try sin el bloque finally
		try {
			int x = 6 / 0;
		} catch (Exception e) {
			e.printStackTrace();
		}

		// 2 Puede existir un bloque try y varios catch, pero
		// las Exceptions estan de menor a mayor jerarquia

		try {
			int x = 6 / 0;
		} catch (ArithmeticException e)
			{ e.printStackTrace();
		} catch (Exception e) {
			e.printStackTrace();
		}

		// 3 Puede existir un bloque try con su finally pero
		sin
		// el catch. NO tien mucho sentido
		try {
			int x = 6 / 0;
		} finally {
			System.out.println("Division entre cero");
		}
		//	try{
		//			int x = 6 /
		//}

	}

}
```

```
java.lang.ArithmeticException: / by
	zero at
demo02.PosibilidadesDelTryCatch.main(PosibilidadesDelTryCatch.java:9)
java.lang.ArithmeticException: / by zero
	at
demo02.PosibilidadesDelTryCatch.main(PosibilidadesDelTryCatch.java:18)
Exception in thread "main" java.lang.ArithmeticException: / by zero
Division entre cero
	at
demo02.PosibilidadesDelTryCatch.main(PosibilidadesDelTryCatch.java:28)
```

## Ejercicio 03

```
package demo03;

import javax.swing.JButton;

public class TipoExceptions {

	public static void main(String[] args) {

				//Division entre cero
				try {
					int x = 5 / 0;
				} catch (ArithmeticException e)
					{ System.out.println(e);
				}

				//Fuera del tamaño del arreglo
				try {
					int [] y = new
					int[10]; y[100]= 15;
				} catch (ArrayIndexOutOfBoundsException e)
					{ System.out.println(e);
				}

				//Fuera del tamaño del String
				try {
					String cad
					="java";
					cad.charAt(50);
				} catch (StringIndexOutOfBoundsException e)
					{ System.out.println(e);
				}

				//Error de puntero null
				try {
					JButton x = null;
					x.setBounds(0,0,100,15);
				} catch (NullPointerException e)
					{ System.out.println(e);
				}
```

```
//Error de formato
try {
	Integer o = Integer.parseInt("jaja");
} catch (NumberFormatException
	e) {
	System.out.println(e);
}

//Error de casteo
try {
	//Polimorfi
	    smo
	Object obj = new
	JButton(); Integer a =
	(Integer)obj;
} catch (ClassCastException
	e) {
	System.out.println(e);
}
```

```
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 100
java.lang.StringIndexOutOfBoundsException: String index out of range: 50
java.lang.NullPointerException
java.lang.NumberFormatException: For input string: "jaja"
java.lang.ClassCastException: javax.swing.JButton cannot be cast
to java.lang.Integer
```

**Ejercicios de aplicación:**

1. Dado el siguiente código:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1() ;
    System.out.println(s);
  }
  void sl() {
    try { s2();
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception {
    s3(); s += "2";
    s3(); s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
  }
}
```

**¿Cuál será el resultado de su ejecución?**

a) -

b) -c

c) -c2

d) -2c

e) -c22b

f) -2c2b

g) -2c2bc

h) Compilation fails

2. Dada la siguiente sentencia:

```
try { int x = Integer.parselnt("two");}
```

**Seleccione todas las opciones que permitirían implementar un bloque CATCH adecuadame**

a. ClassCastException

b. IllegalStateException

c. NumberFormatException

d. IllegalArgumentException

e. ExceptionInInitializerError

f. ArrayIndexOutOfBoundsException

3. Dado el siguiente código:

```
class Swill {
  public static void main(String[] args) {
    String s = "-";
    switch(TimeZone.CST) {
      case EST: s += "e";
      case CST: s += "c";
      case MST: s += "m";
      default: s += "X";
      case PST: s += "p";
    }
    System.out.println(s);
  }
}
enum TimeZone {EST, CST, MST, PST }
```

¿Cuál será el resultado de su ejecución?

a) -c

b) -X

c) -cm

d) -cmp

e) -cmXp

f) Compilation fails.

g) An exception is thrown at runtime.

4. Dado la siguiente clase:

```
class Circus {
  public static void main(String[] args) {
    int x = 9;
    int y = 6;
    for(int z = 0; z < 6; z++, y--) {
      if(x > 2)  x--;
      label:
        if(x > 5) {
          System.out.print(x + " "};
          --X;
          continue label;

          }
        X--;
      }
   }
}
```

**¿Cuál será el resultado de su ejecución?**

a. 8

b. 8 7

c. 876

d. Compilation fails

e. An exception is thrown at runtime

5. Dada la siguiente clase:

```
class Mineral { }
class Gem extends Mineral { }
class Miner {
  static int x = 7; static
  String s = null;
  public static void getWeight(Mineral m) { int
    y = 0 / x;
    System.out.print(s + " ");
  }
  public static void main(String[] args) { Mineral[]
    ma = {new Mineral(), new Gem()};

   for(Object o : ma)
     getWeight((Mineral) o);
  }
}
```

Y la siguiente línea de comando: **java Miner.java**

**¿Cuál será el resultado de su ejecución?**

a) null

b) null null

c) A ClassCastException is thrown.

d) A NullPointerException is thrown.

e) A NoClassDefFoundError is thrown.

f) An ArithmeticException is thrown.

g) An IllegalArgumentException is thrown.

h) An ArrayIndexOutOfBoundsException is thrown

UNIDAD

2

# EXPRESIONES REGULARES

## LOGRO DE LA UNIDAD DE APRENDIZAJE

Al término de la unidad, el alumno crea aplicaciones utilizando de manera individual y combinada las Clases String, StringBuffer, StringBuilder, clases de manejo de flujos del paquete java.io, clases de formateo y parseo de datos, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

## TEMARIO

**2.2 Tema 4 : Expresiones Regulares**

2.2.1 : Regular Expression en Java.

## ACTIVIDADES PROPUESTAS

- Verificar formatos a cadenas

# 2.2. EXPRESIONES REGULARES

## 2.2.1. Regular Expression en Java

Las expresiones regulares son algo que se usa desde hace años en otros lenguajes de programación como Perl, Sed o Awk. En la versión 1.4 del JDK de Sun se incluye el paquete java.util.regex, que proporciona una serie de clases para poder hacer uso de la potencia de este tipo de expresiones en Java. Antes de nada, necesitamos saber qué es una expresión regular y para que nos puede servir:
Pues bien, una expresión regular es un patrón que describe a una cadena de caracteres. Todos hemos utilizado alguna vez la expresión *.doc para buscar todos los documentos en algún lugar de nuestro disco duro, pues bien, *.doc es un ejemplo de una expresión regular que representa a todos los archivos con extensión doc, el asterisco significa cualquier secuencia de caracteres (vale, los que ya conozcan esto dirán que no es correcto, y dirán bien, es mas preciso hablar de *.doc pero el ejemplo es muy gráfico).

Las expresiones regulares se rigen por una serie de normas y hay una construcción para cualquier patrón de caracteres. Una expresión regular sólo puede contener (aparte de letras y números) los siguientes caracteres:

< $, ^, ., *, +, ?, [, ], . >

Una expresión regular, nos servirá para buscar patrones en una cadena de texto, por ejemplo, encontrar cuantas veces se repite una palabra en un texto, para comprobar que una cadena tiene una detereminada estructura, por ejemplo que el nombre de archivo que nos proponen tiene una determinada extensión, o comprobar que un email esta bien escrito... Para cada uno de estos casos existe una expresión regular que los representa:
Por medio de la expresión regular "camion" podemos encontrar cuantas veces se repite camión en un texto. Es la construcción más sencilla.
Esta expresión "^www.*.es" comprueba que una cadena sea una dirección web que comience por www y sea de un servidor español.
Y esta, para ver la potencia de las expresiones regulares, comprueba la buena formación de los correos electrónicos: "[^A-Za-z0-9.@_-~#]+".

### Aplicaciones

El paquete java.util.regex esta formado por dos clases, la clase Matcher y la clase Pattern y por una excepción, PatternSyntaxException.
La clase Pattern (segun la documentacion del jdk1.4) es la representacion compilada de una expresion regular, o lo que es lo mismo, representa a la expresion regular, que en el paquete java.util.regex necesita estar compilada. En castellano significa patrón.
La clase Matcher es un tipo de objeto que se crea a partir de un patrón mediante la invocación del método Pattern.matcher. Este objeto es el que nos permite realizar operaciones sobre la secuencia de caracteres que queremos validar o la en la secuencia

de caracteres en la que queremos buscar. En castellano lo mas parecido a esto es la palabra encajador.
Por lo tanto, tenemos patrones que deben ser compilados, a partir de estos creamos objetos Matcher (encajadores) para poder realizar las operaciones sobre la cadena en cuestión.

Vamos con la clase Pattern, para crear un patrón necesitamos compilar una expresión regular, esto lo conseguimos con el método compile:
Pattern patron = Pattern.compile("camion");
El método pattern devuelve la expresión regular que hemos compilado, el método matcher crea un objeto Matcher a partir del patrón, el método split divide una cadena dada en partes que cumplan el patrón compilado y por último el método matches compila una expresión regular y comprueba una cadena de caracteres contra ella.
Ahora la clase Matcher. Esta clase se utiliza para comprobar cadenas contra el patrón indicado. Un objeto Matcher se genera a partir de un objeto Pattern por medio del método matcher:

```
Pattern patron = Pattern.compile("camion");
Matcher encaja = patron.matcher();
```

Una vez que tenemos el objeto creado, podemos realizar tres tipos de operaciones sobre una cadena de caracteres. Una es a través del método matches que intenta encajar toda la secuencia en el patrón (para el patrón "camion" la cadena "camion" encajaría, la cadena "mi camion es verde" no encajaría). Otra es a través del método lookingAt, intenta encajar el patrón en la cadena (para el patrón "camion" tanto la cadena "camion" como la cadena "mi camion es verde" encajaria). Otra es la proporcionada por el método find que va buscando subcadenas dentro de la cadena de caracteres que cumplan el patrón compilado (una vez encontrada una ocurrencia, se puede inspeccionar por medio de los métodos start que marca el primer carácter de la ocurrencia en la secuencia y el método end que marca el ultimo carácter de la ocurrencia). Todos estos métodos devuelven un booleano que indica si la operación ha tenido éxito.
Todo lo anterior esta orientado a la b6uacutesqueda de patrones en cadenas de caracteres, pero puede que queramos llegar mas allá, que lo que queramos sea reemplazar una cadena de caracteres que se corresponda con un patrón por otra cadena. Por ejemplo, un método que consigue esto es replaceAll que reemplaza toda ocurrencia del patrón en la cadena por la cadena que se le suministra.

### Ejemplos

El siguiente es un ejemplo del uso del método replaceAll sobre una cadena. El ejemplo sustituye todas las apariciones que concuerden con el patron "a*b" por la cadena "-".

```
// se importa el paquete
java.util.regex import
java.util.regex.*;

public class EjemploReplaceAll{
        public static void main(String args[]){
// compilamos el patron
Pattern patron = Pattern.compile("a*b");
// creamos el Matcher a partir del patron, la cadena como parametro
Matcher encaja =
patron.matcher("aabmanoloaabmanoloabmanolob");
// invocamos el metodo replaceAll String
resultado = encaja.replaceAll("-");
System.out.println(resultado);
        }
}
```

El siguiente ejemplo trata de validar una cadena que supuestamente contiene un email, lo hace con cuatro comprobaciones, con un patrón cada una, la primera que no contenga como primer caracter una @ o un punto, la segunda que no comience por www. , que contenga una y solo una @ y la cuarta que no contenga caracteres ilegales:

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

public class ValidacionEmail {
  public static void main(String[] args) throws Exception {
    String input = "www.?regular.com";
    // comprueba que no empieze por punto o @
    Pattern p = Pattern.compile("^.|^@");
    Matcher m = p.matcher(input);
    if (m.find())
      System.err.println("Las direcciones email no empiezan por punto o @");

    // comprueba que no empieze por www.
    p = Pattern.compile("^www.");
    m = p.matcher(input);
    if (m.find())
       System.out.println("Los emails no empiezan por www");

    // comprueba que contenga @
    p = Pattern.compile("@");
    m = p.matcher(input);
    if (!m.find())
        System.out.println("La cadena no tiene arroba");

    // comprueba que no contenga caracteres prohibidos
    p = Pattern.compile("[^A-Za-z0-9.@_-~#]+");
    m = p.matcher(input);
    StringBuffer sb = new StringBuffer();
    boolean resultado = m.find();
    boolean caracteresIlegales = false;
```

```
    while(resultado) {
      caracteresIlegales = true;
      m.appendReplacement(sb, "");
      resultado = m.find();
    }

    // Añade el ultimo segmento de la entrada a la cadena
    m.appendTail(sb);

    input = sb.toString();

    if (caracteresIlegales) {
      System.out.println("La cadena contiene caracteres ilegales");
    }
  }
}
```

**Conclusión**

Las expresiones regulares vienen a tapar un hueco en el JDK de Sun que venia siendo solicitado desde hace mucho tiempo. Con la inclusión de las expresiones regulares Java se convierte, en este tema, en un lenguaje de programación tan flexible como otros mas tradicionales en el tema de las expresiones regulares, Perl, Awk, etc... Hasta ahora la unica opción para conseguir un efecto parecido era el uso de StringTokenizer en conjunción con llamadas repetidas al método charAt que producía un código demasiado enrevesado. Las expresiones regulares tienen un amplio abanico de posibilidades, principalmente para hacer búsquedas, para sustituir ocurrencias y para comprobar la buena formación de cadenas, como se ha visto en el ejemplo del email.

# Resumen

1. Las excepciones pueden ser de dos tipos, controladas y no controladas (checked / unchecked).

2. Las excepciones controladas incluyen todos los subtipos de la clase Excepction, excluyendo las clases que heredan de RuntimeException.

3. Las excepciones controladas son sujetas de gestión o declaración de reglas. Cualquier método que pudiese lanzar una excepción controlada (incluyendo métodos que invocan otros que pueden lanzar una excepción controlada), deben declarar la excepción usando la cláusula throws o gestionarla con la sentencia TRY – CATCH más adecuada.

4. Los subtipos de las clases Error o RuntimeException no son controlables, por lo tanto el compilador no forzará la gestión o declaración de regla alguna.

5. Podemos crear nuestras propias excepciones, normalmente heredando de la clase Exception o de cualquiera de sus subtipos. Este tipo de excepción será considerada una excepción controlada por lo que el compilador forzará su gestión o la declaración de alguna regla para la excepción.

6. Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

   Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

   - http://today.java.net/pub/a/today/2003/12/04/exceptions.html

     Aquí hallará importantes reglas básicas para la gestión de excepciones en java.

   - http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Exception.html

     En esta página, hallará documentación detallada de la clase base de todas las excepciones controladas: Exception

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

Expresiones regulares

http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

# 3

# THREAD

## LOGRO DE LA UNIDAD DE APRENDIZAJE

Al término de la unidad, el alumno crea aplicaciones que manejen Hilos para controlar ejecuciones en paralelos, además utilizará el concepto de sincronización.

## TEMARIO

**3.1 Tema 5 : Thread**

- 3.1.1 : Clase Thread. Ciclo de Vida de un Thread
- 3.1.2 : Interface Runnable
- 3.1.3 : Sincronización
- 3.1.4 : Aplicaciones con Hilo seguro

## ACTIVIDADES PROPUESTAS

- Crear hilos que implementen hilos que herenden de Thread
- Crear hilos que implementen hilos que implementen de Runnable
- Aplicación de la palabra reservada synchronized

# THREAD

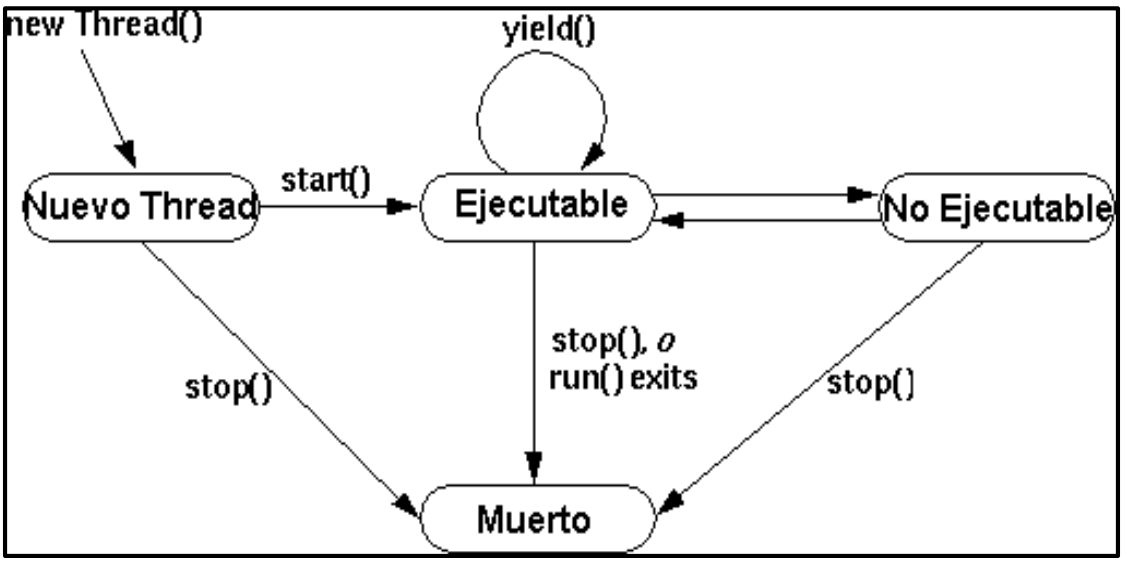
## 3.1.1. Clase Thread. Ciclo de Vida de un Thread

La Máquina Virtual Java (JVM) es un sistema multi-thread. Es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente. La JVM gestiona todos los detalles, asignación de tiempos de ejecución, prioridades, etc, de forma similar a como gestiona un Sistema Operativo múltiples procesos. La diferencia básica entre un proceso de Sistema Operativo y un Thread Java es que los Threads corren dentro de la JVM, que es un proceso del Sistema Operativo y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos. A este tipo de procesos donde se comparte los recursos se les llama a veces 'procesos ligeros' (lightweight process).

Java da soporte al concepto de Thread desde el mismo lenguaje, con algunas clases e interfaces definidas en el package java.lang y con métodos específicos para la manipulación de Threads en la clase Object.

Desde el punto de vista de las aplicaciones los threads son útiles porque permiten que el flujo del programa sea divido en dos o más partes, cada una ocupándose de alguna tarea. Por ejemplo, un Thread puede encargarse de la comunicación con el usuario, mientras otros actuan en segundo plano, realizando la transmisión de un fichero, accediendo a recursos del sistema (cargar sonidos, leer ficheros), etc. De hecho, todos los programas con interface gráfico (AWT o Swing) son multithread porque los eventos y las rutinas de dibujado de las ventanas corren en un thread distinto al principal.

El siguiente diagrama ilustra los distintos estados que puede tener un Thread Java en cualquier momento de su vida. También ilustra las llamadas a métodos que provocan las transiciones de un estado a otro. Este no es un diagrama de estado finito pero da un idea general del las facetas más interesantes y comunes en la vida de un thread. El resto de está página explica el ciclo de vida de un thread, basándose en sus estados.



## Ciclo de vida de un Thread

### Un "Nuevo Thread"

La siguiente sentencia crea un nuevo thread pero no lo arranca, por lo tanto deja el thread en el estado: "New Thread" = "Nuevo Thread".

```
Thread miThread = new MiClaseThread();
```

Cuando un thread está en este estado, es sólo un objeto Thread vacío. No se han asignado recursos del sistema todavía para el thread. Así, cuando un thread está en este estado, lo único que se puede hacer es arrancarlo o pararlo. Llamar a otros métodos distintos de start() o stop() no tiene sentido y causa una excepción del tipo IllegalThreadStateException.

### Ejecutable

Ahora consideremos estas dos líneas de código.

```
Thread miThread = new MiClaseThread();
miThread.start();
```

Cuando el método start() crea los recursos del sistema necesarios para ejecutar el thread, programa el thread para ejecutarse, y llama al método run() del thread. En este punto el thread está en el estado "Ejecutable". Este estado se llama "Ejecutable" mejor que "Ejecutando" ya que el thread todavía no ha empezado a ejecutarse cuando está en este estado. Muchos procesadores tienen un sólo procesador, haciendo posible que todos los threads sean "Ejecutables" al mismo tiempo. Por eso, el sistema de ejecución de Java debe implementar un esquema de programación para compartir el procesador entre todos los threads "Ejecutables".
(Puedes ver la página Prioridad de un Thread para obtener más información sobre la programación.) Sin embargo, para la mayoría de los propositos puedes pensar en "Ejecutable" como un sencillo "Ejecutando". Cuando un thread se está ejecutanto -- está "Ejecutable" y es el thread actual -- las instrucciones de su método run()se ejecutan de forma secuencial.

### Ejecutable

Un thread entra en el estado "No Ejecutable" cuando ocurre uno de estos cuatro eventos.
Alguien llama a su método sleep().
Alguien llama a su método suspend().
El thread utiliza su método wait() para esperar una condición variable.
El thread está bloqueado durante la I/O.
Por ejemplo, la línea en negrita del siguiente fragmento de codigo pone a dormir miThread durante 10 segundos (10.000 milisegundos).

```
Thread miThread = new
MiClaseThread();
miThread.start();
```

```
try {
    miThread.sleep(10000);
} catch (InterruptedException e){
}
```

Durante los 10 segundos que miThread está dormido, incluso si el proceso se vuelve disponible miThread no se ejecuta. Después de 10 segundos, miThread se convierte en "Ejecutable" de nuevo y, si el procesar está disponible se ejecuta.
Para cada entrada en el estado "No Ejecutable" mostrado en figura, existe una ruta de escape distinta y específica que devuelve el thread al estado "Ejecutable". Una ruta de escape sólo trabaja para su entrada correspondiente. Por ejemplo, si un thread ha sido puesto a dormir dutante un cierto número de milisegundos deben pasar esos milisegundos antes de volverse "Ejecutable" de nuevo.

Llamar al método resume() en un thread dormido no tiene efecto.
Esta lista indica la ruta de escape para cada entrada en el estado "No Ejecutable".
Si se ha puesto a dormir un thread, deben pasar el número de milisegundos especificados.
Si se ha suspendido un thread, alguien debe llamar a su método resume().
Si un thread está esperando una condición variable, siempre que el objeto propietario de la variable renuncie mediante notify() o notifyAll().
Si un thread está bloqueado durante la I/O, cuando se complete la I/O.

**Muerto**

Un thread puede morir de dos formas: por causas naturares o siendo asesinado (parado). Una muerte natural se produce cuando su método run() sale normalmente. Por ejemplo, el bucle while en este método es un bucle finito -- itera 100 veces y luego sale.

```
public void run() { int i = 0;
    while (i < 100) { i++;
        System.out.println("i = " + i);
    }
}
```

Un thread con este método run() moriría natualmente después de que el bucle y el método run() se hubieran completado.
También puede matar un thread en cualquier momento llamando a su método stop(). El siguiente código crea y arranca miThread luego lo pone a dormir durante 10 segundos. Cuando el thread actual se despierta, la línea en negrita mata miThread.

```
Thread miThread =
new MiClaseThread();
miThread.start();
try {
    Thread.currentThread().sleep(10000);
} catch (InterruptedException e){
}
```

```
miThread.stop();
```

El método stop() lanza un objeto ThreadDeath hacia al thread a eliminar. Así, cuando se mata al thread de esta forma, muere de forma asíncrona. El thread moririá cuando reciba realmente la excepción ThreadDeath.
El método stop() provoca una terminación súbita del método run() del thread. Si el método run() estuviera realizando cálculos sensibles, stop() podría dejar el programa en un estado inconsistente. Normalmente, no se debería llamar al método stop() pero si se debería proporcionar una terminación educada como la selección de una bandera que indique que el método run() debería salir.

### 3.1.2. Interface Runnable

La interface Runnable proporciona un método alternativo a la utilización de la clase Thread, para los casos en los que no es posible hacer que nuestra clase extienda la clase Thread. Esto ocurre cuando nuestra clase, que deseamos correr en un thread independiente deba extender alguna otra clase. Dado que no existe herencia múltiple, nuestra clase no puede extender a la vez la clase Thread y otra más. En este caso nuestra clase debe implantar la interface Runnable, variando ligeramente la forma en que se crean e inician los nuevos thread

El siguiente ejemplo es equivalente al del apartado anterior, pero utilizando la interface Runnable:

```
public class ThreadEjemplo implements Runnable {
public void run() {
for (int i = 0; i < 5 ; i++)
System.out.println(i + " " +
Thread.currentThread().getName());
System.out.println("Termina thread " +
Thread.currentThread().getName());
}
public static void main (String [] args) {
new Thread ( new ThreadEjemplo() , "Pepe").start();
new Thread ( new ThreadEjemplo() , "Juan").start();
System.out.println("Termina thread main");
}
}
```

Observese en este caso:

- Se implanta la interface Runnable en lugar de extender la clase Thread.
- El constructor que había antes no es necesario.
- En el main observa la forma en que se crea el thread.

Esa expresión es equivalente a:

```
ThreadEjemplo ejemplo = new
ThreadEjemplo(); Thread thread = new Thread
( ejemplo , "Pepe") ; thread.start();
```

Primero se crea la instancia de nuestra clase.

- Después se crea una instancia de la clase Thread, pasando como parámetros la referencia de nuestro objeto y el nombre del nuevo thread.
- Por último, se llama al método start de la clase thread. Este método iniciará el nuevo thread y llamará al método run() de nuestra clase.
- Por útlimo, obsérvese la llamada al método getName() desde run(). getName es un método de la clase Thread, por lo que nuestra clase debe obtener una referencia al thread propio. Es lo que hace el método estático currentThread() de la clase Thread.

## Ejercicio01

Creación de un hilo que herede de Thread

```
package thread.estructura.porHerencia;


/*
 *  - Para que sea hilo debe heredar de Thread
 *  - El cuerpo es método RUN
 */
public class Hilo extends Thread{
```

```
	@Override
	public void run() {

		while(true){

			try {
				sleep(1000);
			} catch (InterruptedException
				e) { e.printStackTrace();
			}
			System.out.println("Hilo");
		}
	}

}
package thread.estructura.porHerencia;

public class Ejecucion {
	/*
	 *  - No se ejecuta con RUN, se jecuta con start
	 *  - Internamente el start llama al metodo RUN
	 */
	public static void main(String[]
		args) { Hilo h = new Hilo();
		h.start();
	}
}
```

Ejercicio02

Creación de un hilo que implemente de Runnable

```
package thread.estructura.porImplementacion;

	public class CuerpoHilo implements

		Runnable{ @Override
		public void run() {
			while(true){
				try {
					Thread.sleep(1000);
				} catch (InterruptedException e)
					{ e.printStackTrace();
				}
				System.out.println("Hilo");
			}


		}
```

```
package thread.estructura.porImplementacion;

public class Ejecucion {
```

```


	public static void main(String[] args) {

		CuerpoHilo c = new CuerpoHilo();
		Thread t = new Thread(c); t.start();
	}
}
```

# THREAD - SINCRONIZACIÓN

## 3.1.3. Sincronización

Cuando en un programa tenemos varios hilos corriendo simultáneamente es posible que varios hilos intenten acceder a la vez a un mismo sitio (un fichero, una conexión, un array de datos) y es posbible que la operación de uno de ellos entorpezca la del otro. Para evitar estos problemas, hay que sincronizar los hilos. Por ejemplo, si un hilo con vocación de Cervantes escribe en fichero "El Quijote" y el otro con vocación de Shakespeare escribe "Hamlet", al final quedarán todas las letras entremezcladas. Hay que conseguir que uno escriba primero su Quijote y el otro, luego, su Hamlet.

**Sincronizar usando un objeto**

Imagina que escribimos en un fichero usando la variable **fichero** de tipo PrintWriter. Para escribir uno de los hilos hará esto:

fichero.println("En un lugar de la Mancha...");

Mientras que el otro hará esto

fichero.println("... ser o no ser ...");

Si los dos hilos lo hacen a la vez, sin ningún tipo de sincronización, el

fichero al final puede tener esto

En un ... ser lugar de la o no Mancha ser ...

Para evitarlo debemos sincronizar los hilos. Cuando un hilo escribe en el fichero, debe marcar de alguna manera que el fichero está ocupado. El otro hilo, al intentar escribir, lo verá ocupado y deberá esperar a que esté libre. En java esto se hace fácilmente. El código sería así

```
synchronized (fichero)
{
  fichero.println("En un lugar de la Mancha...");
}

y el otro hilo

synchronized (fichero)
{
  fichero.println("... ser o no ser ...");
}
```

Al poner synchronized(fichero) marcamos fichero como ocupado desde que se abren las llaves de después hasta que se cierran. Cuando el segundo hilo intenta también su synchronized(fichero), se queda ahí bloqueado, en espera que de que el primero termine con fichero. Es decir, nuestro hilo Shakespeare se queda parado esperando en el

synchronized(fichero) hasta que nuestro hilo Cervantes termine.

synchronized comprueba si fichero está o no ocupado. Si está ocupado, se queda esperando hasta que esté libre. Si está libre o una vez que esté libre, lo marca como ocupado y sigue el código.

Este mecanismo requiere colaboración entre los hilos. El que hace el código debe acordarse de poner synchronized siempre que vaya a usar fichero. Si no lo hace, el mecanismo no sirve de nada.

**Métodos sincronizados**

Otro mecanismo que ofrece java para sincronizar hilos es usar métodos sincronizados. Este mecanismo evita además que el que hace el código tenga que acordarse de poner synchronized.

Imagina que encapsulamos fichero dentro de una clase y que ponemos un método synchronized para escribir, tal que así:

```
public class ControladorDelFichero
{
  private PrintWriter fichero;

  public ControladorFichero()
  {
    // Aqui abrimos el fichero y lo dejamos listo
    // para escribir.
  }

  public synchronized void println(String cadena)
  {
     fichero.println(cadena);
  }
}
```

Una vez hecho esto, nuestros hilos Cervantes y Shakespeare sólo tienen que hacer esto:

```
ControladorFichero control = new ControladorFichero();
...
// Hilo Cervantes
control.println("En un lugar de la Mancha ...");
...
// Hilo Shakespeare control.println("... ser o no ser ...");
```

Al ser el método println() synchronized, si algún hilo está dentro de él ejecutando el código, cualquier otro hilo que llame a ese método se quedará bloqueado en espera de

que el primero termine.

Este mecanismo es más mejor porque, siguiendo la filosfía de la orientación a objetos, encapsula más las cosas. El fichero requiere sincronización, pero ese conocimiento sólo lo tiene la clase ControladorFichero. Los hilos Cervantes y Shakespeare no saben nada del tema y simplemente se ocupan de escribir cuando les viene bien. Tampoco depende de la buena memoria del programador a la hora de poner el synchronized(fichero) de antes.

**Otros objetos que necesitan sincronización**

Hemos puesto de ejemplo un fichero, pero requieren sincronización en general cualquier entrada y salida de datos, como pueden ser ficheros, sockets o incluso conexiones con bases de datos.

También pueden necesitar sincronización almacenes de datos en memoria, como LinkedList, ArrayList, etc. Imagina, por ejemplo, en una LinkedList que un hilo está intentando sacar por pantalla todos los datos.

```
LinkedList lista = new LinkedList();
...
for (int i=0;i<lista.size(); i++)
System.out.println(lista.get(i));
```

Estupendo y maravilloso pero ... ¿qué pasa si mientras se escriben estos datos otro hilo borra uno de los elementos?. Imagina que lista.size() nos ha devuelto 3 y justo antes de intentar escribir el elemento 2 (el último) viene otro hilo y borra cualquiera de los elementos de la lista. Cuando intentemos el lista.get(2) nos saltará una excepción porque la lista ya no tiene tantos elementos.

La solución es sincronizar la lista mientras la estamos usando:

```
LinkedList lista = new LinkedList();
    ...
synchronized (lista)
{
for (int i=0;i<lista.size(); i++)
  System.out.println(lista.get(i));
}
```

Además, este tipo de sincronización es la que se requiere para mantener "ocupado" el objeto lista mientras hacemos varias llamadas a sus métodos (size() y get()), no queda más remedio que hacerla así. Por supuesto, el que borra también debe preocuparse del synchronized.

### 3.1.4. Aplicaciones con Hilo seguro

```
package sincronizacion02;

public class Data {
	private int x = 10000;


	public synchronized void avance(String nomHilo, int c)
		{ x += c;
		System.out.println("Avance " + nomHilo + " : " + x);
	}

	public synchronized void retroceso(String nomHilo, int c)
		{ x -= c;
		if(x<=0){
			System.out.println("Quebrado por : "+
			nomHilo); System.exit(0);
		}

		System.out.println("Retroceso " + nomHilo + " : " + x);
	}

}
```

```
package sincronizacion02;

public class Ejecucion {


	public static void main(String[] args)
		{ Data d = new Data();

		Padre h1 = new
		Padre(d); h1.start();

		Esposa h2 = new
		Esposa(d); h2.start();

		Suegra h3 = new
		Suegra(d); h3.start();

		Hijo h4 = new
		Hijo(d); h4.start();
	}


}
```

```
package sincronizacion02;


public class Esposa extends Thread {

	private Data data;

	public Esposa(Data data) {
		this.data = data;
	}
```

```
                    try {
                        sleep(100);
                    } catch (InterruptedException e)
                        { e.printStackTrace();
                    }

                }

        }

}
```

```
package sincronizacion02;


public class Hijo extends Thread {

        private Data data;

        public Hijo(Data data) {
                this.data = data;
        }
```

```
        @Override
        public void run() {
                while (true) {
                        data.retroceso("Hijo", 1000);

                        try {
                                sleep(500);
                        } catch (InterruptedException e)
                                { e.printStackTrace();
                        }
                }

        }

}
```

```
 package sincronizacion02;


 public class Padre extends Thread {

         private Data data;

         public Padre(Data data) {
                 this.data = data;
         }

         @Override
         public void run() {
                 while (true) {
                         data.avance("Padre", 500);
```

```
                        try {
                              sleep(500);
                        } catch (InterruptedException e)
                                { e.printStackTrace();
                        }
                }

        }

}
```

```
package sincronizacion02;


public class Suegra extends Thread {

        private Data data;

        public Suegra(Data data) {
                this.data = data;
        }

        @Override
        public void run() {
                while (true) {
                        data.retroceso("Suegra", 400);

                        try {
                                sleep(500);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }

        }
```

## Diferencia entre método sincronizado y bloque sincronizado en Java

Seguidamente, se van a repasar algunas de las diferencias entre los métodos y bloques sincronizados en Java basados en la experiencia y las reglas sintácticas de la palabra reservada synchronized en Java. Aunque ambos pueden ser usados para proveer un alto grado de sincronización en Java, el uso del bloque sincronizado sobre el método es considerado en Java una mejor práctica de programación.

- Una diferencia significativa entre método y bloque sincronizado es que el bloque sincronizado generalmente limita el alcance del bloqueo. El alcance del bloqueo es inversamente proporcional al rendimiento, con lo que siempre es mejor bloquear únicamente la sección crítica que haya en el código. Uno de los mejores ejemplos de usar bloque sincronizado es la doble comprobación de bloqueo en patrón Singleton en lugar del bloqueo completo en métodos sincronizados con getInstance(), sólo bloqueamos la sección crítica de código en la cual se crear la instancia Singleton. Esto mejora el rendimiento considerablemente debido a que el bloqueo se requiere únicamente una o dos veces.

- El bloque sincronizado provee control granular sobre el bloqueo, con lo que se puede utilizar arbitrariamente cualquier bloqueo para conseguir exclusión mutua en una sección crítica de código. Por el otro lado, el método sincronizado siempre bloquea el objeto actual representado por su palabra reservada o bloqueo a nivel de clase, si el método sincronizado es estático.

- El bloque sincronizado puede lanzar una java.lang.NullPointerException si la expresión resultante al bloquear es null, al contrario del caso de los métodos sincronizados.

- En el caso de los métodos sincronizados, el bloqueo conseguido por el hilo cuando entra en el método y cuando se libera al salir del método, o sale normalmente o lanzando una Exception. Por el otro lado, en el caso de los bloques sincronizados, el hilo bloquea cuando entra en el bloque propiamente dicho, y libera cuando deja el bloque.

# Resumen

1. Parea ejecutar un hilo se invoca al método start.

2. El cuerpo del hilo se confecciona en el método run.

3. Thread.sleep(tiempo en milisegundos), permite dormir al hilo en un determinado tiempo en milisegundos.

   Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

   http://docs.oracle.com/javase/tutorial/essential/concurrency/
   Java Concurrency

   http://labojava.blogspot.com/2012/10/estados-de-un-thread.html

4. Los bloques sincronizados y los métodos sincronizados son dos formar de utilizar la sincronización en Java e implementar la exclusión mutua en secciones críticas de código. Java se utiliza de forma habitual en programas multi-hilo, los cuales presentan varios tipos de peligros como la seguridad del hilo, deadlocks y condiciones de carrera, que pueden aparecer en el código por un mal entendimiento del mecanismo de sincronización provisto por Java.

   Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

   - http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html
     Sincronización

   - http://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html
     Bloques de sincronización

4

# GESTIÓN DE CADENAS

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones utilizando de manera individual y combinada las Clases String, StringBuffer, StringBuilder, clases de manejo de flujos del paquete java.io, clases de formateo y parseo de datos, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

**TEMARIO**

**4.1 Tema 6 : La clase String**

4.1.1 : La clase String – Definición y gestión de memoria
4.1.2 : Principales métodos de la clase String
4.1.3 : Las clases StringBuffer y StringBuilder

**ACTIVIDADES PROPUESTAS**

- Apliacion de métodos de la calse String.
- Mostra las diferencias entre las clases String, StringBuffer, StringBuilder.

# 4.1. LA CLASE STRING

## 4.1.1. La clase String – Definición y gestión de memoria

**Ejercicio 01**

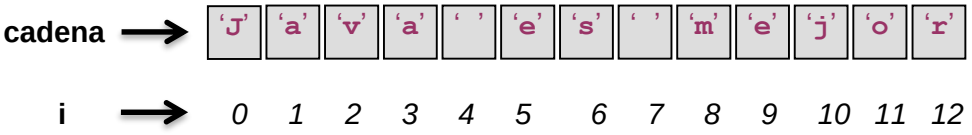
```
package cadenas;

public class ManejoString {

	public static void main(String[] args) {
		//1 Comparaciones
		String cadena ="LucIa_na";
		System.out.println(cadena.equals("LucIa_na"));
		System.out.println(cadena.equals("LUCIA_NA"));
		System.out.println(cadena.equalsIgnoreCase("LUCIA_NA"));

		System.out.println(cadena.startsWith("Lu"));
		System.out.println(cadena.endsWith("na"));

		//2 Conversión a mayúscula y minúscula
		System.out.println(cadena.toLowerCase());
		System.out.println(cadena.toUpperCase());

		//3 Remplazo de caracteres
		System.out.println(cadena.replace('a', 'x'));

		//4 Remplazo de palabras
		System.out.println(cadena.replaceAll("na", "nita"));
		System.out.println(cadena.replaceFirst("a", "-"));

		//5 Corte a la cadena :Número de caracteres
		System.out.println(cadena.substring(3));

		//5 Corte a la cadena : Doble corte
		System.out.println(cadena.substring(2,3));

		//Ejercicio 1

		//Como extraigo el último caracter para cualquier cadena
		String cad1 = "El mejor";

		System.out.println(cad1.substring(cad1.length()-1));
		System.out.println(cad1.charAt(cad1.length()-1));


		//6 Como separa cadenas de una trama
		String cad2 = "luis,pedro,fernamdez,luz,ana,maria";
		String[] s = cad2.split(",");
		for (String aux : s) {
			System.out.println(aux);
		}
	}

}
```

```
true false
true true
true
lucia_na
LUCIA_NA
LucIx_nx
LucIa_nita
LucI-_na
Ia_na
c r r
luis pedro
fernamdez
luz
ana
maria
```

### 4.1.2. Principales métodos de la clase String

Esta Clase dispone de diversos métodos para manipular cadenas.

```
String          cadena = "Java es mejor";
```




| METODO | DESCRIPCION |
|---|---|
| `length()` | Devuelve la longitud de la cadena.<br><br>`int longitud = cadena.length();`<br>`longitud ← 13` |
| `charAt(int)` | Devuelve una copia del carácter que encuentre en la posición indicada por el parámetro.<br><br>`char caracter = cadena.charAt(8);`<br>`caracter ← 'm'` |
| `equals(String)` | Comprueba si dos cadenas son iguales. En este caso comprueba que el objeto dado como argumento sea de tipo *String* y contenga la misma cadena de caracteres que el objeto actual.<br><br>`String s = "Java";`<br>`boolean x = cadena.equals(s);`<br>`x ← false` |

| | |
|---|---|
| equalsIgnoreCase(*String*) | Realiza la misma tarea que *equals* pero sin tener en cuenta las mayúsculas o minúsculas.<br><br>*String* **s** = "java Es MeJOR";<br>*boolean* **x** =<br>**cadena**.equalsIgnoreCase(**s**);<br>**x** ← *true* |

| | |
|---|---|
| compareTo(*String*) | Devuelve un entero menor que cero si la cadena es alfabéticamente menor que la dada como argumento, cero si las dos cadenas son léxicamente iguales y un entero mayor que cero si la cadena es mayor alfabéticamente.<br><br>*String* **s1** = "Java es lo máximo",<br>**s2** = "Java es mejor",<br>**s3** = "Java es ok";<br><br>*int* **x** =<br>**cadena**.compareTo(**s1**)<br>, **y** =<br>**cadena**.compareTo(**s2**)<br>, **z** =<br>**cadena**.compareTo(**s3**)<br>;<br><br>***x*** ← *1 //* ***cadena*** *mayor que* ***s1*** *alfabéticamente*<br>***y*** ← *0 //* ***cadena*** *contiene lo mismo que* ***s2***<br>***z*** ← *-2 //* ***cadena*** *menor que* ***s3*** *alfabéticamente* |
| startsWith(*String*) | Comprueba si el comienzo de la cadena actual coincide con la cadena pasada como parámetro.<br><br>*String* **s** = "JavaX";<br>*boolean* **x** =<br>**cadena**.startsWith(**s**)<br>; **x** ← *false* |
| endsWith(*String*) | Comprueba si el final de la cadena actual coincide con la cadena pasada como parámetro.<br><br>*String* **s** = "mejor";<br>*boolean* **x** =<br>**cadena**.endsWith(**s**)<br>; **x** ← *true* |
| indexOf(*int*) | Devuelve la posición que por primera vez aparece el carácter (expresado como entero) pasado como parámetro. En caso no exista devuelve -1.<br><br>*int* **i** =<br>**cadena**.indexOf('e')<br>; **i** ← *5* |
| indexOf(*int*,*int*) | Devuelve la posición que por primera vez aparece el carácter (expresado como entero) a partir de la posición especificada como segundo parámetro.<br><br>*int* **i** =<br>**cadena**.indexOf('e',6)<br>; **i** ← *9* |

| | |
|---|---|
| indexOf(*String*) | Devuelve la posición que por primera vez aparece la cadena pasada como parámetro.<br><br>*int* **i** =<br>**cadena**.indexOf("va")<br>*;* **i** ← *2* |
| indexOf(*String*,*int*) | Devuelve la posición que por primera vez aparece la cadena pasada como parámetro, pudiendo especificar en un segundo parámetro a partir de dónde buscar.<br><br>*int* **i** =<br>**cadena**.indexOf("ej",5)<br>*;* **i** ← *9* |
| lastIndexOf(*int*)<br>lastIndexOf(*int*,*int*)<br>lastIndexOf(*String*)<br>lastIndexOf(*String*,*int*) | Devuelve la última vez que aparece el carácter (expresado como entero) o cadena pasada como parámetro, pudiendo especificar en un segundo parámetro, a partir de dónde buscar (búsqueda hacia atrás).<br><br>*String* **s** = "e";<br>*int* **i** = **cadena**.lastIndexOf(**s**);<br>**i** ← *9* |

| | |
|---|---|
| toLowerCase() | Convierte la cadena a minúsculas.<br><br>*String* **s** = "CiberJava – Lima - Perú";<br>**s** = **s**.toLowerCase();<br>**s** ←"*ciberjava – lima – perú*" |
| toUpperCase() | Convierte la cadena a mayúsculas.<br><br>*String* **s** = "CiberJava – Lima - Perú";<br>**s** = **s**.toUpperCase();<br>**s** ←"*CIBERJAVA – LIMA – PERÚ*" |
| trim() | Elimina espacios al principio y al final de la cadena. *String* **s** = "<br>CiberJava<br>Lima"; **s** = **s**.trim();<br>**s** ←"*CiberJava Lima*" |
| substring(*int*) | Devuelve una subcadena de la cadena actual, empezando por el primer índice indicado hasta antes del segundo índice (si se especifica) o hasta el final de la cadena.<br><br>*String* **s1** = "Viva el Perú",<br>**s2** = s1.substring(5),<br>**s3** = s1.substring(3,9);<br>**s2** ← "*el Perú*"<br>**s3** ← "*a el P*" |

| | |
|---|---|
| substring(*int*,*int*) | |

| | |
|---|---|
| `replace(char,char)` | Reemplaza todos los caracteres iguales al primer parámetro y los sustituye por el carácter que pasamos en segundo lugar, teniendo en cuenta lo mismo una mayúscula que una minúscula.<br><br>`String s = "biba el Perú";`<br>`s = s.replace('b','v');`<br>`s ←"viva el Perú"` |
| `toCharArray()` | Convierte la cadena a un vector de caracteres.<br><br>`char[] arreglo = cadena.toCharArray();` |
| **Métodos estáticos de conversión** | La clase *String* dispone de varios métodos para transformar valores de otros tipos de datos a cadena. Todos se llaman valueOf y son estáticos. |
| `String.valueOf(boolean)`<br>`String.valueOf(int)`<br>`String.valueOf(long)`<br>`String.valueOf(float)`<br>`String.valueOf(double)`<br>`String.valueOf(Object)` | `double r = 3.1416;`<br>`String s = String.valueOf(r);`<br>`s ←"3.1416"` |

| | |
|---|---|
| `String.valueOf(char[])` | |
| `String.valueOf(char[],int,int)` | Transforma una subcadena de un arreglo de caracteres, especificando una posición y la longitud.<br><br>`char c[]={'C','i','b','e','r','J','a','v','a'};`<br>`String s = String.valueOf(c,3,5);`<br>`s ←"erJav"` |

## Ejercicio 02

Dada la cadena:

```
String cad2 = "luis,pedro,fernamdez,luz,ana,maria";
```

Encuentre:

- ✓ Número total de personas
- ✓ Número total de personas que terminan en vocal a
- ✓ Número total de personas que empiezan en vocal a
- ✓ Número total de personas que terminan en cualquier vocal
- ✓ Número total de personas que empiezan en cualquier vocal

```
package cadenas;

public class Ejercicio01 {

	public static void main(String[] args) {

		//Encuentre:
		//1 Número total de personas
		//2 Número total de personas que terminan en vocal a
		//3 Número total de personas que empiezan en vocal a
		//4 Número total de personas que terminan en cualquier vocal
		//5 Número total de personas que empiezan en cualquier vocal

		String cad2 =

		"luis,pedro,fernamdez,luz,ana,maria"; String[] s =

		cad2.split(",");
		int cont1=0, cont2=0, cont3=0, cont4=0;
		for (String aux : s) {
			if(aux.endsWith("a")) cont1++;
			if(aux.startsWith("a"))
			cont2++;
			if(aux.endsWith("a")||aux.endsWith("e")||
					aux.endsWith("i")||aux.endsWith("o")|
					| aux.endsWith("u"))
				cont3++;
			if(aux.startsWith("a")||aux.startsWith("e")||
					aux.startsWith("i")||aux.startsWith("o")|
					| aux.startsWith("u"))
```

```
		System.out.println("#Personas: " + s.length);
		System.out.println("#Personas terminan en a: " + cont1);
		System.out.println("#Personas empiezan en a: " + cont2);
		System.out.println("#Personas termina en vocal: " +
		cont3); System.out.println("#Personas empiezan en vocal: "
		+ cont4);

	}
}
```

```
#Personas: 6
#Personas terminan en a: 2
#Personas empiezan en a: 1
#Personas termina en vocal:
3 #Personas empiezan en
vocal: 1
```

## Eercicio 03

Dada la cadena:

```
String usuarios =
                "María-F,juan-M,Pedro-M,Sonia-F,"
                + "Alberto-M,Rosa-F,Elias-M,Ana-F,"
                + "Jorge-M,Luis-M,Juana-F,Elvira-F";
```

Elabore un programa donde encuentre

- número de personas
- número de sexo Masculino
- número de sexo Femenino
- Imprima todos los nombres(no sexo) de los usuarios

```
package cadenas;

public class Ejercicio02 {

	public static void main(String[] args) {

		// Elabore un programa donde encuentre
		// a)número de personas
		// b)número de sexo Masculino
		// c)número de sexo Femenino
		// d)Imprima todos los nombres(no sexo) de los usuarios

		String usuarios =
				"María-F,juan-M,Pedro-M,Sonia-F,"
				+ "Alberto-M,Rosa-F,Elias-M,Ana-F,"
				+ "Jorge-M,Luis-M,Juana-F,Elvira-F";

		String[] s = usuarios.split(",");
		int sexM =0, sexF=0;
		for (String aux : s) {
			if(aux.endsWith("F
			   ")) sexF ++;
			else
				sexM ++;
			System.out.print
			ln(
					aux.replaceAll("-F", "").replaceAll("-M",
"", ""));
		}

		System.out.println("#número de personas : " +
		s.length); System.out.println("#número de sexo
		Masculino : " + sexM); System.out.println("#número de
		sexo Femenino : " + sexF);
	}
}
```

```
María
juan
Pedro
Sonia
Alber
to
Rosa
Elias
Ana
Jorge
Luis
Juana
Elvir
a
#número de personas : 12
#número de sexo Masculino :
```

### 4.1.3. Las clases StringBuffer y StringBuilder

La clase StringBuilder fue agregada en la versión 5 de java. Tiene exactamente las mismas características que la clase StringBuffer, solo que no es un hilo de ejecución seguro (thread safe). En otras palabras, sus métodos no están sincronizados.

Sun recomienda que usemos StringBuilder en vez de StringBuffer siempre que sea posible, porque StringBuilder se ejecutará más rápido. Fuera del tema de sincronización, cualquier cosa que podamos decir de los métodos de la clase StringBuilder la podremos decir de StringBuffer y viceversa.

En el presente capítulo, hemos visto varios ejemplos que evidencian la inmutabilidad de los objetos de tipo String:

```
String x
= "abc";
x.concat
("def");
System.out.println("x = " + x);                    // la salida es "x = abc"
```

Dado que no hubo ninguna asignación usando el operador new, el Nuevo objeto String creado con el método concat() fue abandonada instantáneamente. Ahora veamos el siguiente ejemplo:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x);                    // la salida es "x = abcdef"
```

En el ejemplo previo, hemos creado un nuevo objeto String; sin embargo, el antiguo objeto x con el valor abc, ha sido perdido en el pool; por lo tanto, estamos desperdiciando memoria. Si estuviéramos usando un objeto de tipo StringBuffer en vez de un String, el código se vería de la siguiente manera:

```
StringBuffer sb = new
StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb);                    // la salida es "sb = abcdef"
```

Todos los métodos del objeto StringBuffer operan sobre el valor asignado al objeto del método que invocamos. Por ejemplo, una llamada a sb.append("def"); implica que estamos agregando "def" al mismo objeto de tipo StringBuffer sb. A continuación, se muestra un ejemplo:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb );                          // la
salida es "fed --- cba"
```

Debemos notar que en los dos ejemplos previos, solo hemos realizado una llamada al operador new y no hemos creado tampoco ningún objeto String adicional. Cada ejemplo solo necesitó ejecutar un único objeto StringXXX.

public synchronized StringBuffer append(String s). Este método actualiza el valor del objeto que lo invocó. El método puede tomar argumentos de diferentes tipos tales como boolean, char, double, float, int, long, entre otros, siendo el más usado el argumento de tipo String.

```
StringBuffer sb = new
StringBuffer("set ");
sb.append("point");
System.out.println(sb);                    // la
salida es "set point" StringBuffer sb2 = new
StringBuffer("pi = "); sb2.append(3.14159f);
System.out.println(sb2);                    // la salida es "pi = 3.14159"
```

public StringBuilder delete(int start, int end) retorna un objeto StringBuilder y actualiza el valor del objeto StringBuilder que invocó al método. En ambos casos una subcadena es removida del objeto original. El índice de inicio de la subcadena a eliminar es definido por el primer argumento (el cual está basado en cero), y el índice final de la subcadena a ser removida es definido por el segundo argumento (el cual está basado en 1). A continuación, se muestra un ejemplo:

```
StringBuilder sb = new StringBuilder("0123456789");
```

```
System.out.println(sb.delete(4,6));      // la salida es  "01236789"
```

Tip:

Dado que los objetos StringBuffer son modificables, el código mostrado a continuación se comportará de manera diferente que un código similar aplicado sobre objetos String:

```
StringBuffer sb = new
StringBuffer("abc");
sb.append("def");
System.out.println( sb );
```

En este caso, la salida sera: "abcdef"

public StringBuilder insert(int offset, String s) retorna un objeto StringBuilder y actualiza el valor del objeto StringBuilder que invocó al método. En ambos casos el string pasado en el segundo argumento es insertado en el objeto original sobre la base del primer argumento (el desplazamiento está basado en cero). Podemos pasar otros tipos de datos como segundo argumento (boolean, char, double, float, int, long, etc; sin embargo, el tipo String es el más usado.

```
StringBuilder sb = new
StringBuilder("01234567");
sb.insert(4, "---");
System.out.println( sb );         // la salida es  "0123---4567"
```

public synchronized StringBuffer reverse() This method returns a StringBuffer object and updates the value of the StringBuffer object that invoked the method call. En ambos casos, los caracteres en el objeto StringBuffer son invertidos.

```
StringBuffer s = new StringBuffer("A man a plan
a canal Panama"); sb.reverse();
System.out.println(sb); // salida : "amanaP lanac a nalp a nam A"
```

public String toString() Este método retorna el valor del objeto StringBuffer object que invocó al método como un String:

```
StringBuffer sb = new StringBuffer("test
string"); System.out.println( sb.toString() );
// la salida es "test string"
```

Esto se cumple para StringBuffers y StringBuilders: a diferencia de los objetos de tipo String, los objetos StringBuffer y StringBuilder pueden ser cambiados (no son inmutables).

Importante:

Es común utilizar en Java métodos encadenados ("chained methods"). Una sentencia con métodos encadenados tiene la siguiente apariencia:

*result = method1().method2().method3();*

En teoría, cualquier número de métodos pueden estar encadenados tal como se muestra en el esquema previo, aunque de manera típica encontraremos como máximo tres.

Sea x un objeto invocando un segundo método. Si existen solo dos métodos encadenados, el resultado de la llamada al segundo método será la expresión final resultante.

Si hay un tercer método, el resultado de la llamada al segundo método será usado para invocar al tercer método, cuyo resultado será la expresión final resultante de toda la operación:

```
String x = "abc";
String y = x.concat("def").toUpperCase().replace('C','x');
//metodos encadenados
System.out.println("y = " + y); // el resultado es "y = ABxDEF"
```

El literal def fue concatenado con abc, creando un String temporal e intermedio (el cual pronto sera descartado), con el valor abcdef. El método toUpperCase() creó un nuevo String temporal (que también pronto sera perdido) con el valor ABCDBF. El método replace() creó un String final con el valor ABxDEF el cual es referenciado por la variable y.

Comparación en concatenar cadenas de String, StringBuilder, StringBuffer

```
package cadenas;

     public class
    Comparacion {

      public static void main(String[] args) {

          {
             long ti = System.currentTimeMillis();
             String cad = "";
             for (int i = 0; i < 10000; i++) {
             cad = cad + "Elba";
             }
             long tf = System.currentTimeMillis();
             System.out.println("String con + : " + (tf - ti));
```

```
            }
            {

                  long ti =
                  System.currentTimeMillis(); String
                  cad = "";
                  for (int i = 0; i < 1000000; i++)
                        { cad.concat("Elba");
                  }
                  long tf = System.currentTimeMillis();
                  System.out.println("String con concat : " + (tf -
                  ti));
            }
            {
                  //StrinBuffer: Es de tipo hilo seguros
                  //Un solo hilo puede aceder a la cadena
                  //Es por eso que es lento
                  long ti = System.currentTimeMillis();
                  StringBuffer cad = new
                  StringBuffer(); for (int i = 0; i <
                  1000000; i++) {
                        cad.append("Elba");
                  }
                  long tf = System.currentTimeMillis();
                  System.out.println("StringBuffer  : " + (tf -
                  ti));
            }
            {
```

```
                  //StringBuilder no es de tipo hilo
                  protegido long ti =
                  System.currentTimeMillis(); StringBuilder
                  cad = new StringBuilder();  for (int i =
                  0; i < 1000000; i++) {
                        cad.append("Elba");
                  }
                  long tf = System.currentTimeMillis();
                  System.out.println("StringBuilder  : " + (tf - ti));
            }
      }
}
```

```
String con + : 263 String
con concat : 40
StringBuffer : 42
StringBuilder : 21
```

### Metodos de StringBuilder

```
package cadenas;

public class ManejoStringBuilder {

public static void main(String[] args) {

//No genera una nueva cadena, sino se aplica a la cadena
//original
StringBuilder cad = new StringBuilder("hola leo como estas");

//Elimina caracteres
cad.delete(5, 8);
System.out.println(cad);

//reversa
cad.reverse();
System.out.println(cad);

//agrega cadenas cad.insert(9,
"¡eres todo!");
System.out.println(cad);

//remplaza la caracteres
cad.setCharAt(2, '*');
cad.setCharAt(4, '*');
System.out.println(cad);
}


}
```

```
hola  como estas
satse omoc aloh
satse omo¡eres todo!c
aloh sa*s* omo¡eres
todo!c aloh
```

# Resumen

**1** Metodos de la clase String

- ✓ public char charAt(int index): Devuelve el carácter alojado en el índice pasado como parámetro.
- ✓ public String concat(String s): Agrega un String al final de otro (como el operador '+').
- ✓ public boolean equalsIgnoreCase(String s): Determina la igualdad de dos String, ignorando las diferencias de mayúsculas y minúsculas.
- ✓ public int length(): Devuelve la cantidad de caracteres que componen un String.
- ✓ public String replace(char old, char new): Reemplaza las ocurrencias de un carácter con otro nuevo.
- ✓ public String substring(int begin) – public String substring(int begin, int end): Devuelve una parte de un String.
- ✓ public String toLowerCase(): Devuelve el String con todos los caracteres convertidos a minúsculas.
- ✓ public String toUpperCase(): Devuelve el String con todos los caracteres convertidos a mayúsculas.
- ✓ public String trim(): Remueve los espacios en blanco del principio y final del String.

**2** La clase StringBuilder (agregada en Java 5), posee exactamente los mismos métodos que la clase StringBuffer. La única diferencia entre ellas, es que la clase StringBuilder es más rápida debido a que sus métodos no están sincronizados. Estas dos clases se utilizan para realizar muchas modificaciones sobre los caracteres de una cadena. Los objetos de estas clases no crean un nuevo objeto cada vez que se modifican (como los objetos String), es por esto que son más eficientes para realizar muchas modificaciones sobre cadenas.

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- ➢ https://javierrguez.wordpress.com/2010/01/12/resumenes-scjp-capitulo-6-strings- io-formateo-y-parseo/
  String, StringBuilder, StringBuffer

# 5

# APLICACIONES DE ESCRITORIO EN JAVA CON CONEXIÓN A BASE DE DATOS

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones que utilizan una fuente de datos Mysql y podrá realizar mantenimientos, consultas y transacciones utilizando las clases Connection, PreparedStatement, etc; del JDBC. Además, podrá acceder a procedimientos almacenados.

**TEMARIO**

**5.1 Tema 7 : JDBC- Básico**

5.1.1 : Connection, Class.forName

5.1.2 : Generar Conexiones Registros a base de datos

**ACTIVIDADES PROPUESTAS**

- Creación de conexiones a base de datos
- Creacion de inserción en tablas

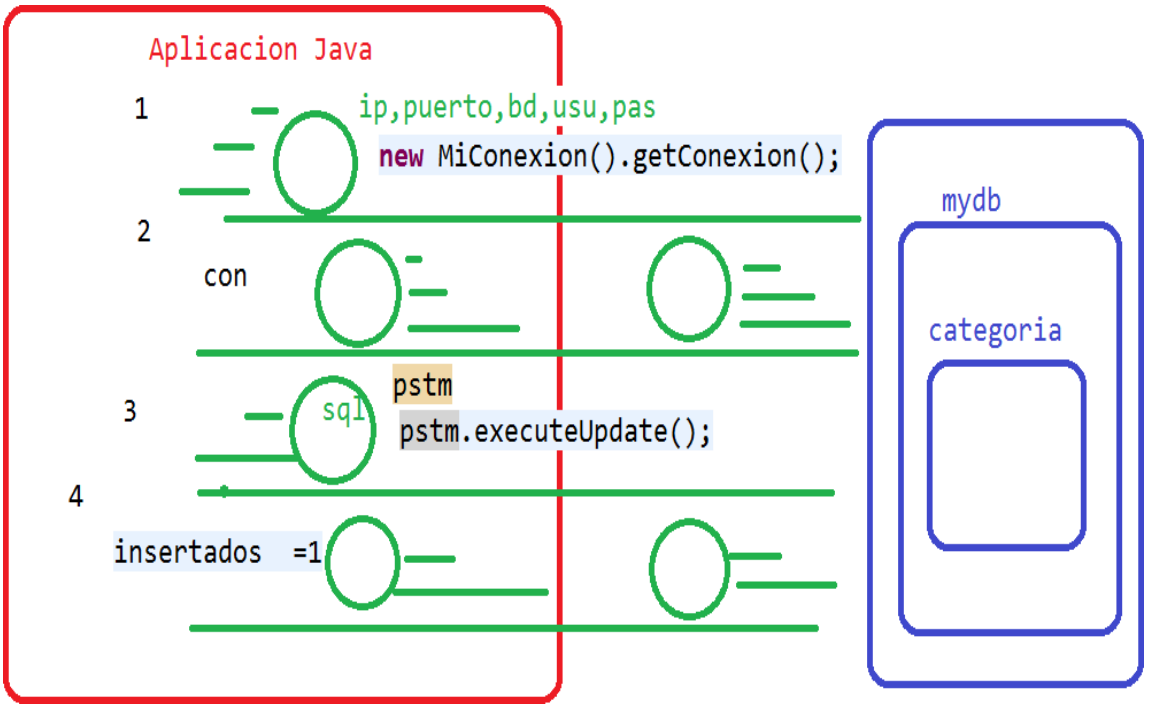
# 5.1. JDBC. BÁSICO

## 5.1.1. Connection, Class.forName

Java Database Connectivity, más conocida por sus siglas JDBC,1 2 es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.

El API JDBC se presenta como una colección de interfaces Java y métodos de gestión de manejadores de conexión hacia cada modelo específico de base de datos. Un manejador de conexiones hacia un modelo de base de datos en particular es un conjunto de clases que implementan las interfaces Java y que utilizan los métodos de registro para declarar los tipos de localizadores a base de datos (URL) que pueden manejar. Para utilizar una base de datos particular, el usuario ejecuta su programa junto con la biblioteca de conexión apropiada al modelo de su base de datos, y accede a ella estableciendo una conexión; para ello provee el localizador a la base de datos y los parámetros de conexión específicos. A partir de allí puede realizar cualquier tipo de tarea con la base de datos a la que tenga permiso: consulta, actualización, creación, modificación y borrado de tablas, ejecución de procedimientos almacenados en la base de datos, etc.

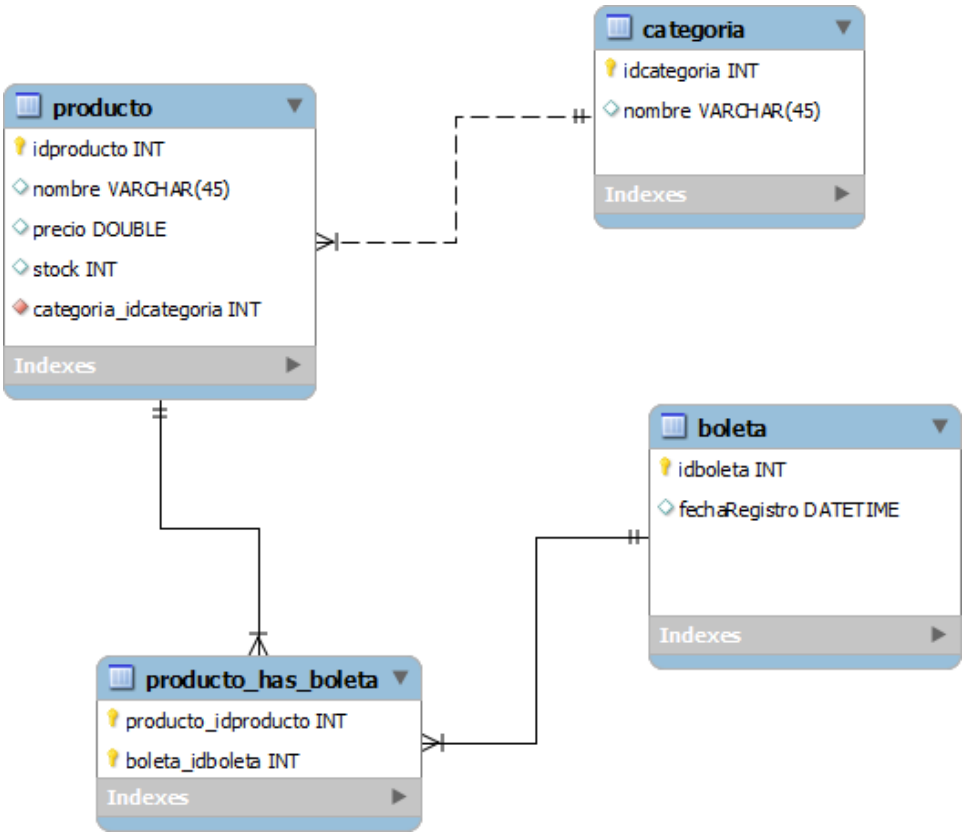
**Pasos de la Conexión**



## 5.1.2. Generar Conexiones Registros a base de datos

**Ejercicio 01**

Insertar Categoría




```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE,
SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

CREATE SCHEMA IF NOT EXISTS `ejemplo` DEFAULT CHARACTER SET utf8
COLLATE
    utf8_general_ci ; USE `ejemplo` ;

-- -----------------------------------------------------
-- Table `ejemplo`.`categoria`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `ejemplo`.`categoria` (
  `idcategoria` INT NOT NULL AUTO_INCREMENT,
  `nombre`
  VARCHAR(45)
  NULL,
  PRIMARY KEY
  (`idcategor
  ia`))
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Table `ejemplo`.`producto`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `ejemplo`.`producto` (
  `idproducto` INT NOT NULL AUTO_INCREMENT,
  `nombre` VARCHAR(45) NULL,
  `precio` DOUBLE NULL,
  `stock` INT NULL,
  `categoria_idcategor
  ia` INT NOT NULL,
  PRIMARY KEY
  (`idproducto`),
  INDEX `fk_producto_categoria_idx`
  (`categoria_idcategoria` ASC), CONSTRAINT
  `fk_producto_categoria`
FOREIGN KEY (`categoria_idcategoria`)
```

```
REFERENCES `ejemplo`.`categoria` (`idcategoria`)
      ON DELETE NO ACTION ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `ejemplo`.`boleta`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `ejemplo`.`boleta` (
  `idboleta` INT NOT NULL AUTO_INCREMENT,
  `fechaRegistr
  o` DATETIME
  NULL, PRIMARY
  KEY
  (`idboleta`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `ejemplo`.`producto_has_boleta`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `ejemplo`.`producto_has_boleta` (
  `producto_idproducto` INT NOT NULL,
  `boleta_idboleta` INT NOT NULL,
  PRIMARY KEY (`producto_idproducto`, `boleta_idboleta`),
  INDEX `fk_producto_has_boleta_boleta1_idx`
  (`boleta_idboleta`  ASC), INDEX
  `fk_producto_has_boleta_producto1_idx`
  (`producto_idproducto` ASC), CONSTRAINT
  `fk_producto_has_boleta_producto1`
    FOREIGN KEY (`producto_idproducto`)
    REFERENCES `ejemplo`.`producto` (`idproducto`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT
    `fk_producto_has_boleta_b
    oleta1` FOREIGN KEY
    (`boleta_idboleta`)
```

```
          REFERENCES

        `ejemplo`.`boleta` (`idboleta`)
        ON DELETE NO ACTION
        ONUPDATE NO ACTION) ENGINE = InnoDB;


      SET SQL_MODE=@OLD_SQL_MODE;
      SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
      SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

```
package utils;

import
java.sql.Connection;
import
java.sql.DriverManager;
import
java.sql.SQLException;


/*
 * Permite crer una conexion a la BD
 * Se debe tener:
 * 1) Driver JDBC
 * 2) Ip del Servidor
 * 3) puerto
 * 4) Nombre de la BD
 * 5) Usuario
 * 6) Password

 */
    public class
    MiConexion {

      static{
            try {
                  Class.forName("com.mysql.jdbc.Driver");
            } catch (ClassNotFoundException
                  e) { e.printStackTrace();
            }
      }

      /*
       *  IP :localhost
       *  PUERTO: 3306
       *  BD: mydb
       *  USUARIO: root
       *  PASSWORD: mysql
       *
       */
      public Connection
            getConexion(){
            Connection conn =
            null;
```

```
 *
            try {
                  conn = DriverManager.getConnection(
                                    "jdbc:mysql://localhost:3306/eje
                                    mplo", "root",
                                    "mysql");
            } catch
                  (SQLException
                  e) {
                  e.printStackTr
                  ace();
            }

            return conn;
      }


}
```

```
package ejecuciones;

import java.sql.Connection;
import java.sql.PreparedStatement;

import utils.MiConexion;

public class InsertaCategoria {


      public static void main(String[] args) {

            Connection conn = null;
            PreparedStatement pstm = null;

            try {
                  //1 se crea la conexion
                  conn = new MiConexion().getConexion();

                  //2 se prepara el SQL
                  String sql ="insert into categoria values(null,?)";
                  pstm = conn.prepareStatement(sql);
                  pstm.setString(1, "Comestibles");

                  //3 se envia el SQL a la BD
                  int insertados = pstm.executeUpdate();

                  System.out.println("Insertados : " + insertados);
            } catch (Exception e) {
                  e.printStackTrace();
            } finally{
                  try {
                        if(pstm!= null) pstm.close();
                        if(conn!= null) conn.close();
                  } catch (Exception e2) {}
            }
      }

}
```

# Resumen

1 Información para conectarse a base de datos
- ✓ **IP :localhost**
- ✓ **PUERTO: 3306**
- ✓ **BD: mydb**
- ✓ **USUARIO: root**
- ✓ **PASSWORD: mysql**

2 Para conectar a una base de datos se debe tener el conector JDBC, el error que sale al no colocarlo

```
java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
      at java.net.URLClassLoader.findClass(Unknown
      Source) at
      java.lang.ClassLoader.loadClass(Unknown Source)
      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown
      Source) at java.lang.ClassLoader.loadClass(Unknown Source)
      at java.lang.Class.forName0(Native
      Method) at
      java.lang.Class.forName(Unknown Source)
      at utils.MiConexion.<clinit>(MiConexion.java:23)
      at
ejecuciones.InsertaCategoria.main(InsertaCategoria.java:18)
java.sql.SQLException: No suitable driver found for
jdbc:mysql://localhost:3306/ejemplo
      at      java.sql.DriverManager.getConnection(Unknown
      Source)                                          at
      java.sql.DriverManager.getConnection(Unknown  Source)
      at utils.MiConexion.getConexion(MiConexion.java:41)
```

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- http://stackoverflow.com/questions/18058714/java-class-forname-jdbc-connection-loading-driver

  JDBC

- http://dev.mysql.com/downloads/connector/odbc/

# 5

# APLICACIONES DE ESCRITORIO EN JAVA CON CONEXIÓN A BASE DE DATOS

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones que utilizan una fuente de datos Mysql y podrá realizar mantenimientos, consultas y transacciones utilizando las clases Connection, PreparedStatement, etc; del JDBC. Además podrá acceder a procedimientos almacenados.

**TEMARIO**

**5.2 Tema 8 : JDBC - Mantenimientos**

5.2.1 : Clases Connection, PreparedStatement y ResultSet

5.2.2 : Insertar,eliminar, listar, actualizar

**ACTIVIDADES PROPUESTAS**

- Crear un mantenimiento en base de datos

# JDBC- MANTENIMIENTOS

## 5.2.1. Clases Connection, PreparedStatement y ResultSet

### Connection

Representa la conexión con la Base de Datos.
El encargado de abrir una conexión es el Driver Manager mediante el método estático:

```
public static Connection getConnection(url, usr, pwr)trows
java.sql.SQLException
```

**Donde:**

**url:** Identificador de la Base de Datos
**usr:** Usuario con el que se abre la conexión (opcional)
**pwr:** Contraseña del Usuario (opcional)

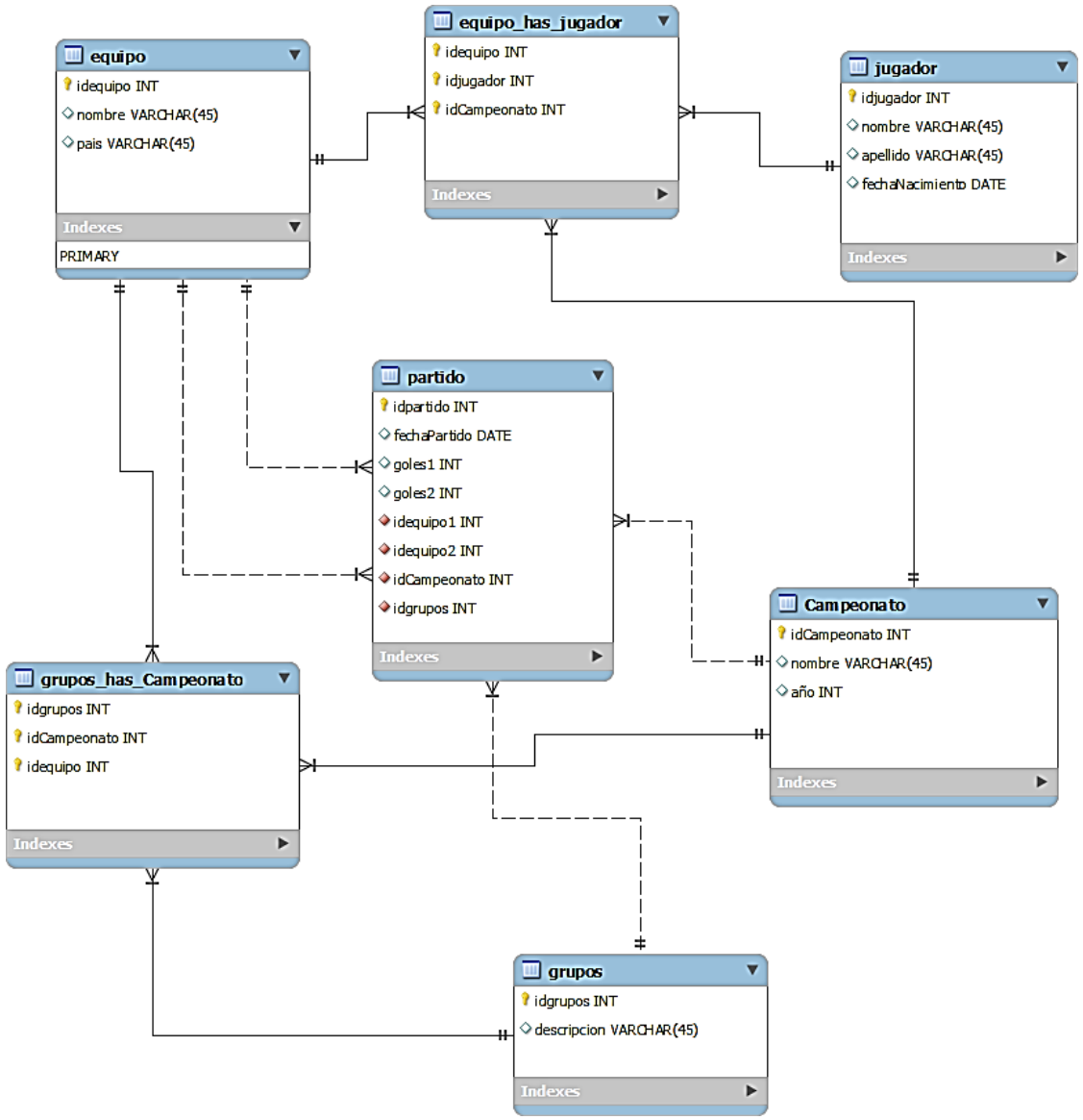
### PreparedStatement

Cuando trabajamos con una base de datos es posible que haya sentencias *SQL* que tengamos que ejecutar varias veces durante la sesión, aunque sea con distintos parámetros. Por ejemplo, durante una sesión con base de datos podemos querer insertar varios registros en una tabla. Cada vez los datos que insertamos serán distintos, pero la sentencia *SQL* será la misma: Un *INSERT* sobre determinada tabla que será simpre igual, salvo los valores concretos que queramos insertar.

### ResultSet

Puede utilizar un objeto ResultSet para acceder a una tabla de datos generada ejecutando una consulta. Las filas de la tabla se recuperan en secuencia. Dentro de una fila, es posible acceder a los valores de las columnas en cualquier orden.
Los datos almacenados en ResultSet se recuperan mediante los diversos métodos get, en función del tipo de datos que se vaya a recuperar.El método next() permite desplazarse a la fila siguiente.
ResultSet permite obtener y actualizar columnas por nombre, aunque el uso del índice de columna mejora el rendimiento.

Se crea la Base de datos champion:




```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL,ALLOW_INVALID_DATES';

CREATE SCHEMA IF NOT EXISTS `champion` DEFAULT CHARACTER SET utf8 COLLATE
utf8_general_ci ;
USE `champion` ;

-- -----------------------------------------------------
-- Table `champion`.`equipo`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `champion`.`equipo` (
  `idequipo` INT NOT NULL AUTO_INCREMENT,
```

```
`nombre` VARCHAR(45) NULL,
`pais` VARCHAR(45) NULL,
  PRIMARY KEY
(`idequipo`)) ENGINE
= InnoDB;


-- -----------------------------------------------------
-- Table `champion`.`jugador`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `champion`.`jugador` (
`idjugador` INT NOT NULL AUTO_INCREMENT,
`nombre` VARCHAR(45) NULL,
`apellido` VARCHAR(45) NULL,
`fechaNacimiento` DATE NULL,
PRIMARY KEY (`idjugador`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `champion`.`Campeonato`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `champion`.`Campeonato` (
`idCampeonato` INT NOT NULL AUTO_INCREMENT,
`nombre` VARCHAR(45) NULL,
`aÃ±o` INT NULL,
  PRIMARY KEY (`idCampeonato`))
   ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `champion`.`equipo_has_jugador`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `champion`.`equipo_has_jugador` (
`idequipo` INT NOT NULL,
`idjugador` INT NOT NULL,
`idCampeonato` INT NOT NULL,
PRIMARY KEY (`idequipo`, `idjugador`, `idCampeonato`),
INDEX `fk_equipo_has_jugador_jugador1_idx` (`idjugador` ASC),
INDEX `fk_equipo_has_jugador_equipo_idx` (`idequipo` ASC),
INDEX `fk_equipo_has_jugador_Campeonato1_idx` (`idCampeonato` ASC),
CONSTRAINT `fk_equipo_has_jugador_equipo`
  FOREIGN KEY (`idequipo`)
  REFERENCES `champion`.`equipo` (`idequipo`)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,
CONSTRAINT `fk_equipo_has_jugador_jugador1`
  FOREIGN KEY (`idjugador`)
  REFERENCES `champion`.`jugador` (`idjugador`)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION,
CONSTRAINT `fk_equipo_has_jugador_Campeonato1`
  FOREIGN KEY (`idCampeonato`)
  REFERENCES `champion`.`Campeonato` (`idCampeonato`)
  ON DELETE NO ACTION
  ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
```

```
-- Table `champion`.`grupos`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `champion`.`grupos` (
  `idgrupos` INT NOT NULL AUTO_INCREMENT,
  `descripcion` VARCHAR(45) NULL,
  PRIMARY KEY (`idgrupos`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `champion`.`grupos_has_Campeonato`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `champion`.`grupos_has_Campeonato` (
  `idgrupos` INT NOT NULL,
  `idCampeonato` INT NOT NULL,
  `idequipo` INT NOT NULL,
  PRIMARY KEY (`idgrupos`, `idCampeonato`, `idequipo`),
  INDEX `fk_grupos_has_Campeonato_Campeonato1_idx` (`idCampeonato` ASC),
  INDEX `fk_grupos_has_Campeonato_grupos1_idx` (`idgrupos` ASC),
  INDEX `fk_grupos_has_Campeonato_equipo1_idx` (`idequipo` ASC),
  CONSTRAINT `fk_grupos_has_Campeonato_grupos1`
    FOREIGN KEY (`idgrupos`)
    REFERENCES `champion`.`grupos` (`idgrupos`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_grupos_has_Campeonato_Campeonato1`
    FOREIGN KEY (`idCampeonato`)
    REFERENCES `champion`.`Campeonato` (`idCampeonato`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_grupos_has_Campeonato_equipo1`
    FOREIGN KEY (`idequipo`)
    REFERENCES `champion`.`equipo` (`idequipo`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `champion`.`partido`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `champion`.`partido` (
  `idpartido` INT NOT NULL AUTO_INCREMENT,
  `fechaPartido` DATE NULL,
  `goles1` INT NULL,
  `goles2` INT NULL,
  `idequipo1` INT NOT NULL,
  `idequipo2` INT NOT NULL,
  `idCampeonato` INT NOT NULL,
  `idgrupos` INT NOT NULL,
  PRIMARY KEY (`idpartido`),
  INDEX `fk_partido_Campeonato1_idx` (`idCampeonato` ASC),
  INDEX `fk_partido_equipo1_idx` (`idequipo1` ASC),
  INDEX `fk_partido_equipo2_idx` (`idequipo2` ASC),
  INDEX `fk_partido_grupos1_idx` (`idgrupos` ASC),
  CONSTRAINT `fk_partido_Campeonato1`
    FOREIGN KEY (`idCampeonato`)
    REFERENCES `champion`.`Campeonato` (`idCampeonato`)
    ON DELETE NO ACTION
```

```
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_partido_equipo1`
    FOREIGN KEY (`idequipo1`)
    REFERENCES `champion`.`equipo` (`idequipo`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_partido_equipo2`
    FOREIGN KEY (`idequipo2`)
    REFERENCES `champion`.`equipo` (`idequipo`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_partido_grupos1`
    FOREIGN KEY (`idgrupos`)
    REFERENCES `champion`.`grupos` (`idgrupos`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

## Ejercicio:

## Insertar Campeonato


```
package entidad;

public class Campeonato {

      private int idCampeonato, anno;
```

```
	private String descripcion;

	public int getIdCampeonato() {
		return idCampeonato;
	}
	public void setIdCampeonato(int idCampeonato) {
		this.idCampeonato = idCampeonato;
	}
	public int getAnno() {
		return anno;
	}
	public void setAnno(int anno) {
		this.anno = anno;
	}
	public String getDescripcion() {
		return descripcion;
	}
	public void setDescripcion(String descripcion) {
		this.descripcion = descripcion;
	}

}
```

```
package model;

import  java.sql.Connection;
import
java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import util.MiConexion;
import entidad.Campeonato;

public class ModelCampeonato {

	public int eliminaCampeonato(int idCampeonato){
		int eliminados = -1;

		Connection con = null;
		PreparedStatement pstm = null;

		try {
			con = new MiConexion().getConexion();

			String sql ="delete from campeonato where idCampeonato =
?";
			pstm = con.prepareStatement(sql);
			pstm.setInt(1, idCampeonato);

			eliminados = pstm.executeUpdate();
			System.out.println("Eliminados :  " + eliminados);
```

```
                    e.printStackTrace();
              } finally {
                    try {
                          if (pstm != null)pstm.close(); i
                          f (con != null)con.close();
                    } catch (SQLException e) {
                          e.printStackTrace();
                    }
              }

              return eliminados;
       }

       public int actualizaCampeonato(Campeonato obj){
              int actualizados = -1;

              Connection con = null;
              PreparedStatement pstm = null;

             try {
                   con = new MiConexion().getConexion();

                   String sql ="update campeonato set nombre=?, año=? where
idCampeonato=?" ;
                   pstm = con.prepareStatement(sql);
                   pstm.setString(1, obj.getDescripcion());
                   pstm.setInt(2, obj.getAnno());
                   pstm.setInt(3, obj.getIdCampeonato());

                   actualizados = pstm.executeUpdate();
                   System.out.println("Eliminados :   " + actualizados);

          } catch (Exception e) {
                e.printStackTrace();
          } finally {
                try {
                      if (pstm != null)pstm.close();
                      if (con != null)con.close();
                 } catch (SQLException e) { e.printStackTrace();
                 }
           }
           return actualizados;
    }

   public List<Campeonato> listaCampeonato(){
         List<Campeonato>     data = new ArrayList<Campeonato>();

         Connection con = null;
         PreparedStatement pstm = null;
         ResultSet     rs = null;
         try {
               con = new MiConexion().getConexion();

               String sql ="select * from campeonato";
               pstm = con.prepareStatement(sql);
               rs = pstm.executeQuery();
               Campeonato c = null;
               while(rs.next()){
```

```
                        c = new Campeonato();
                        c.setIdCampeonato(rs.getInt("idCampeonato"));
                        c.setDescripcion(rs.getString("nombre"));
                        c.setAnno(rs.getInt("año"));
                        data.add(c);
                  }
            } catch (Exception e) {
                  e.printStackTrace();
            } finally {
                  try {
                        if (pstm != null)pstm.close();
                        if (con != null)con.close();
                  } catch (SQLException e) {
                        e.printStackTrace();
                  }
            }

            return data;
      }

      public int insertaCampeonato(Campeonato obj) {
            int insertados = -1;

            Connection con = null;
            PreparedStatement pstm = null;

            try {
                  con = new MiConexion().getConexion();

                  String sql ="insert into campeonato values(null,?,?)";
                  pstm = con.prepareStatement(sql);
                  pstm.setString(1, obj.getDescripcion());
                  pstm.setInt(2, obj.getAnno());

                  insertados = pstm.executeUpdate();
                  System.out.println("Insertados :  " + insertados);

            } catch (Exception e) {
                  e.printStackTrace();
            } finally {
                  try {
                        if (pstm != null)pstm.close();
                        if (con != null)con.close();
                  } catch (SQLException e) {
                        e.printStackTrace();
                  }
            }

            return insertados;
      }

}
```

```
package gui;

import java.awt.Color;
import java.awt.EventQueue;
```

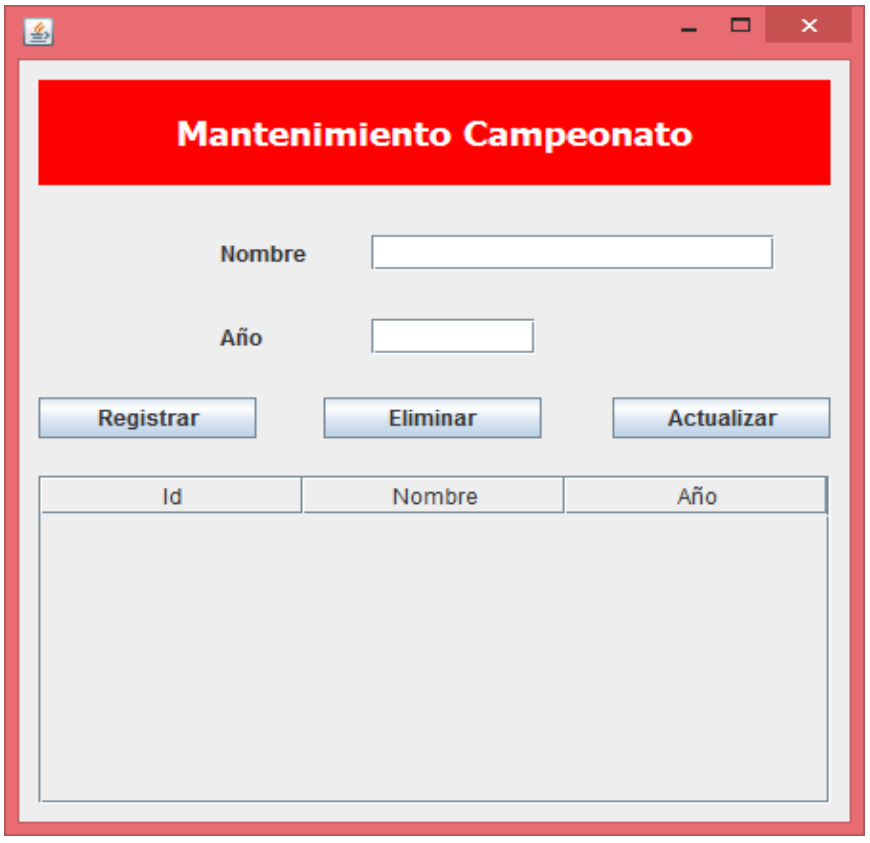
```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;
import javax.swing.border.EmptyBorder;

import model.ModelCampeonato;
import entidad.Campeonato;

public class FrmInsertaCampeonato extends JFrame implements ActionListener {
	private JPanel contentPane;
	private JTextField txtNombre;
	private JTextField txtAnno;
	private JButton btnRegistrar;

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

	/**
	 * Create the frame.
	 */
	public FrmInsertaCampeonato() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 450, 362);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);

		JLabel lblRegistraCampeonato = new JLabel("Registra
Campeonato");
		lblRegistraCampeonato.setFont(new Font("Tahoma", Font.BOLD,
23));

	lblRegistraCampeonato.setHorizontalAlignment(SwingConstants.CENTER);
		lblRegistraCampeonato.setOpaque(true);
		lblRegistraCampeonato.setForeground(Color.YELLOW);
```

```
                        lblRegistraCampeonato.setBackground(Color.RED);
                      lblRegistraCampeonato.setBounds(10, 11, 414, 46);
                                contentPane.add(lblRegistraCampeonato);
              JLabel lblNombr = new JLabel("Nombre");
              lblNombr.setBounds(41, 112, 69, 27);
              contentPane.add(lblNombr);

              JLabel lblAo = new JLabel("A\u00F1o");
              lblAo.setBounds(41, 170, 50, 20);
              contentPane.add(lblAo);

              txtNombre = new JTextField();
              txtNombre.setBounds(124, 115,
              261, 20);
              contentPane.add(txtNombre);
              txtNombre.setColumns(10);

              txtAnno = new JTextField();
              txtAnno.setBounds(124, 170,
              86, 20);
              contentPane.add(txtAnno);
              txtAnno.setColumns(10);

              btnRegistrar = new JButton("Registrar");
              btnRegistrar.addActionListener(this);
              btnRegistrar.setBounds(164, 259, 89, 23);
              contentPane.add(btnRegistrar);
       }
       public void actionPerformed(ActionEvent arg0) {
              if (arg0.getSource() == btnRegistrar) {
                     do_btnRegistrar_actionPerformed(arg0);
              }
       }
       protected void do_btnRegistrar_actionPerformed(ActionEvent arg0) {

              String des = txtNombre.getText().trim();
              String anno =           txtAnno.getText().trim();

              if(!des.matches("[a-z_A-Z0-9\\s]{4,50}")){
                     JOptionPane.showMessageDialog(this,
                                   "Debe ser alfanuméricos entre 4 y 50
caracteres");
               return;
              }

                     if(!anno.matches("[0-9]{4}")){
                            JOptionPane.showMessageDialog(th
                            is,
                                          "Debe ser numero de 4 dígitos");
                            return;
                     }


       Campeonato obj = new Campeonato();
       obj.setDescripcion(des);
       obj.setAnno(Integer.parseInt(anno));
```

```
        ModelCampeonato m = new ModelCampeonato();
        int     insertados      =      m.insertaCampeonato(obj);
        if(insertados>0){
              JOptionPane.showMessageDialog(this,
                          "Se insertó correctamente");

              }

              txtNombre.setText("");
              txtAnno.setText("");
              txtNombre.requestFocus();
        }
}
```

### 5.2.2. Generar Insertar, eliminar, listar, actualizar

**Mantenimiento de Campeonato**




```
package gui;

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

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JLabel;
```

```
import java.awt.Font;
import java.awt.Color;

import javax.swing.SwingConstants;
import javax.swing.JTextField;
import  javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

import entidad.Campeonato;
import model.ModelCampeonato;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.util.List;

public class FrmCrudCampeonato extends JFrame implements
ActionListener, MouseListener {

	private JPanel contentPane;
	private JTextField txtNombre;
	private JTextField txtAnno;
	private JTable  table;
	private JButton btnRegistrar;
	private JButton btnEliminar;
	private JButton btnActualizar;

	//ModelCampeonato-->Es la clase donde estan los
	//métodos insert, update, delete, listar en la BD

	ModelCampeonato model = new ModelCampeonato();
	//El id del campeonato que selecciono en el jtable
	int idSeleccionado = -1;

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

	 /**
	 * Create the frame.
	*/
```

```
public FrmCrudCampeonato() {
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setBounds(100, 100, 450, 466);
      contentPane = new JPanel();
      contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
      setContentPane(contentPane);
      contentPane.setLayout(null);
labellblMantenimientoCampeonato.setHorizontalAlignment(SwingConsta
nts.CENTER);

lblMantenimientoCampeonato.setForeground(Color.WHITE);
lblMantenimientoCampeonato.setFont(new Font("Tahoma",
Font.BOLD,19));

lblMantenimientoCampeonato.setBounds(10, 11, 414, 59);
contentPane.add(lblMantenimientoCampeonato);

JLabel lblNombre = new JLabel("Nombre");
lblNombre.setBounds(105, 95, 84, 26);
contentPane.add(lblNombre);

JLabel lblAnno = new JLabel("A\u00F1o");
lblAnno.setBounds(105, 142, 46, 26);
contentPane.add(lblAnno);

txtNombre = new JTextField();
txtNombre.setBounds(184, 98,
211, 20);
contentPane.add(txtNombre);
txtNombre.setColumns(10);

txtAnno = new JTextField();
txtAnno.setBounds(184, 145,
86, 20);
contentPane.add(txtAnno);
txtAnno.setColumns(10);

btnRegistrar = new JButton("Registrar");
btnRegistrar.addActionListener(this);
btnRegistrar.setBounds(10, 189, 114, 23);
contentPane.add(btnRegistrar);

btnActualizar = new JButton("Actualizar");
btnActualizar.addActionListener(this);
btnActualizar.setBounds(310, 189, 114, 23);
contentPane.add(btnActualizar);

btnEliminar = new JButton("Eliminar");
btnEliminar.addActionListener(this);
btnEliminar.setBounds(159, 189, 114, 23);
contentPane.add(btnEliminar);

JScrollPane scrollPane = new JScrollPane();
scrollPane.setBounds(10, 233, 414, 184);
contentPane.add(scrollPane);

table = new JTable();
```

```
            table.addMouseListener(this);
            table.setModel(new
            DefaultTableModel(
                  new Object[][] {
                  },
                  new String[] {
                     "Id", "Nombre", "A\u00F1o"
                  }
            ));
            scrollPane.setViewportView(table);

            //Trae los campeonatos
            listaCampeonato();
      }
      public void actionPerformed(ActionEvent arg0) {
            if (arg0.getSource() == btnActualizar) {
                  do_btnActualizar_actionPerformed(arg0);
            }
            if (arg0.getSource() == btnEliminar) {
                  do_btnEliminar_actionPerformed(arg0);
            }
            if (arg0.getSource() == btnRegistrar) {
                  do_btnRegistrar_actionPerformed(arg0);
            }
      }
      protected void do_btnRegistrar_actionPerformed(ActionEvent arg0)
            { String nom = txtNombre.getText().trim();
            String anno = txtAnno.getText().trim();

            Campeonato c = new
            Campeonato();
            c.setDescripcion(nom);
            c.setAnno(Integer.parseInt(
            anno));

            model.insertaCampeonato(c);

            listaCampeonato();
      }
      protected void do_btnEliminar_actionPerformed(ActionEvent arg0) {

            model.eliminaCampeonato(idSelecci
            onado); listaCampeonato();
      }
      protected void do_btnActualizar_actionPerformed(ActionEvent
            arg0) { String nom = txtNombre.getText().trim();
            String anno = txtAnno.getText().trim();

            Campeonato c = new Campeonato();
            c.setIdCampeonato(idSeleccionado);
            c.setDescripcion(nom);
            c.setAnno(Integer.parseInt(anno));

            model.actualizaCampeonato(c);

            listaCampeonato();
      }
```

```
	public void mouseClicked(MouseEvent arg0) {
		if (arg0.getSource() == table) {
			do_table_mouseClicked(arg0);
		}
	}
	public void mouseEntered(MouseEvent arg0) {
	}
	public void mouseExited(MouseEvent arg0) {
	}
	public void mousePressed(MouseEvent arg0) {
	}
	public void mouseReleased(MouseEvent arg0) {
	}
	protected void do_table_mouseClicked(MouseEvent arg0) {

		int fila = table.getSelectedRow();

		DefaultTableModel dtm =

		(DefaultTableModel)table.getModel();

		idSeleccionado = (Integer) dtm.getValueAt(fila, 0);
		String nom = (String) dtm.getValueAt(fila, 1);
		int anno  = (Integer) dtm.getValueAt(fila, 2);

		txtNombre.setText(nom);
		txtAnno.setText(String.valueOf(anno));
	}

	//--Muestra todo los registros de campeonato de la BD
	//al jtable
	private void listaCampeonato(){

		//1 traemos la data de la BD
		List<Campeonato> data = model.listaCampeonato();

		//2 limpiar el jtable
		//DefaultTableModel-->Maneja los datos del jtable
		DefaultTableModel dtm = (DefaultTableModel)table.getModel();
		dtm.setRowCount(0);

		//3 Agregamo los objetos de data al dtm
		for (Campeonato c : data) {
			Object[] fila = {c.getIdCampeonato(),
					c.getDescripcion(),
					c.getAnno()};
			 dtm.addRow(fila);
		}

		//4 Actualizamos el dtm al
		jtable
		dtm.fireTableDataChanged();

	}

}
```

# Resumen

1 Para realizar una inserción, eliminación o actuialización se debe usar la sentencia

**s = pstm.executeUpdate();**

2 Para realizar una consulta usar

**rs = pstm.executeQuery();**

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- http://tutorials.jenkov.com/jdbc/resultset.html
  ResultSet

- http://www.tutorialspoint.com/jdbc/jdbc-statements.htm
  Statements

# 5

# APLICACIONES DE ESCRITORIO EN JAVA CON CONEXIÓN A BASE DE DATOS

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones que utilizan una fuente de datos Mysql y podrá realizar mantenimientos, consultas y transacciones utilizando las clases Connection, PreparedStatement, etc; del JDBC. Además podrá acceder a procedimientos almacenados.

**TEMARIO**

**5.3 Tema 9 : JDBC - Consultas**

5.3.1 : Clases Connection, PreparedStatement y ResultSet

5.3.2 : Listar

**ACTIVIDADES PROPUESTAS**

- Listado de una tabla

## 5.3. JDBC- Consultas

### 5.3.1 Clases Connection, PreparedStatement y ResultSet

**Connection**

Representa la conexión con la Base de Datos.
El encargado de abrir una conexión es el Driver Manager mediante el método estático:

```
public static Connection getConnection(url, usr, pwr)trows
java.sql.SQLException
```

**Donde:**
**url:** Identificador de la Base de Datos
**usr:** Usuario con el que se abre la conexión (opcional)
**pwr:** Contraseña del Usuario (opcional)

**PreparedStatement**

Cuando trabajamos con una base de datos es posible que haya sentencias *SQL* que tengamos que ejecutar varias veces durante la sesión, aunque sea con distintos parámetros. Por ejemplo, durante una sesión con base de datos podemos querer insertar varios registros en una tabla. Cada vez los datos que insertamos serán distintos, pero la sentencia *SQL* será la misma: Un *INSERT* sobre determinada tabla que será simpre igual, salvo los valores concretos que queramos insertar.

**ResultSet**

Puede utilizar un objeto ResultSet para acceder a una tabla de datos generada ejecutando una consulta. Las filas de la tabla se recuperan en secuencia. Dentro de una fila, es posible acceder a los valores de las columnas en cualquier orden.
Los datos almacenados en ResultSet se recuperan mediante los diversos métodos get, en función del tipo de datos que se vaya a recuperar.El método next() permite desplazarse a la fila siguiente.
ResultSet permite obtener y actualizar columnas por nombre, aunque el uso del índice de columna mejora el rendimiento.

### 5.3.2 Listar

El administrador de la empresa CiberFarma, necesita un reporte (**listado**) de los usuarios del sistema. Diseñe el formulario y los procesos de Gestión.

### Método listado (MySqlUsuarioDAO.java)

```
public ArrayList<Usuario> listado() {
    ArrayList<Usuario> lista = new ArrayList<Usuario>();
    ResultSet rs = null;
    Connection con = null;
    PreparedStatement pst = null;
    try {
       con = MySQLConexion.getConexion();
       String sql = "select * from tb_usuarios"; // sentencia sql

       pst = con.prepareStatement(sql);
       // parámetros según la sentencia

       rs = pst.executeQuery(); // tipo de ejecución
       // Acciones adicionales en caso de consultas
       while (rs.next()){                    // mientras hay datos en la consulta
           Usuario u = new Usuario();
           u.setCodigo(rs.getInt(1));
           u.setNombre(rs.getString(2));
           // otros campos
           lista.add(u);
       }
    } catch (Exception e) {
       System.out.println("Error en la sentencia " + e.getMessage());
    } finally {
        try {
            if (pst != null) pst.close();
            if (con != null) con.close();
        } catch (SQLException e) {
            System.out.println("Error al cerrar ");
        }
     }
     return lista;
}
```

El método next() del ResultSet, hace que el puntero avance al siguiente registro. Si lo consigue devuelve true.

| codigo | nombre | apellido | usuario | clave | facceso | tipo | estado |
|---|---|---|---|---|---|---|---|
| 1 | Tito | Siber | U001 | 10001 | 2017-05-13 | 2 | 1 |
| 2 | Zoila | Baca | U002 | 10002 | 2017-05-13 | 2 | 1 |

Invocar en el botón Reporte el meotod listado que en su interior llama al metodo lsitado que se encuentrta en la case **MySqlUsuarioDAO.java**


```
void listado(){
    // Instancia la clase de gestion
    GestionUsuario gu = new GestionUsuario();

    // Obtiene el listado de usuarios -> ArrayList
    ArrayList<Usuario> lista = gu.listado();

    // imprime la lista en el área de texto
    if (lista == null){
        txtS.setText("Lista vacia");
    } else {
        txtS.setText("Código\tNombre\n");
        for (Usuario u : lista) {
            txtS.append(u.getCodigo() + "\t" + u.getNombre() + "\n");
        }
    }
}
```

# Resumen

1 Para realizar la ejecución de un select

**rs = pstm.executQuery();**

2 Para realizar un recorrido en un objeto de la clase ResultSet consulta usar

**while(rs.next()){**

**}**

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- http://tutorials.jenkov.com/jdbc/resultset.html
  ResultSet

- http://www.tutorialspoint.com/jdbc/jdbc-statements.htm
  Statements

# 5

# APLICACIONES DE ESCRITORIO EN JAVA CON CONEXIÓN A BASE DE DATOS

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones que utilizan una fuente de datos Mysql y podrá realizar mantenimientos, consultas y transacciones utilizando las clases Connection, PreparedStatement, etc; del JDBC. Además, podrá acceder a procedimientos almacenados.

**TEMARIO**

**5.4 Tema 10 : JDBC – Procedimientos almacenados**

5.4.1 : Clases Connection, CallStatetement y ResultSet

5.4.2 : Ejercicios

**ACTIVIDADES PROPUESTAS**

- Crear aplicaciones con procedimientos almacenados

# 5.4. JDBC- PROCEDIMIENTOS ALMACENADOS

## 5.4.1. Clases Connection, CallStatetement y ResultSet

Se crea el procedimiento almacenado

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `consulta_jugador`(IN  param1
varchar(30
0)) BEGIN

select * from jugador where nombre like param1;

END
```

Para la ejecución se ejecuta el procedimiento almacenado mediante prepareCall

```
public List<Jugador> buscaXNombre(String nombre){
            ArrayList<Jugador> data = new
            ArrayList<Jugador>();

            Connection con = null;
            PreparedStatement pstm =
            null; ResultSet     rs =
            null;
            try {
                  con = new MiConexion().getConexion();
                  String sql ="{call consulta_jugador(?)}";
                  pstm = con.prepareCall(sql);
                  pstm.setString(1, nombre + "%");
                  rs = pstm.executeQuery();
                  Jugador c = null; while(rs.next()){
                        c = new Jugador();
                        c.setIdJugador(rs.getInt("idJugador"));
                        c.setNombre(rs.getString("nombre"));
                        c.setApellido(rs.getString("apellido"));

      c.setFechaNacimiento(rs.getDate("fechaNacimiento"));
                        data.add(c);
                  }

            } catch (Exception e) {
                  e.printStackTrace();
            } finally {
                  try {
                        if (pstm != null)pstm.close();
                        if (con != null)con.close();
                  } catch (SQLException e) {
                        e.printStackTrace();
                  }
            }

            return data;
      }
```

### 5.4.2. Ejercicios

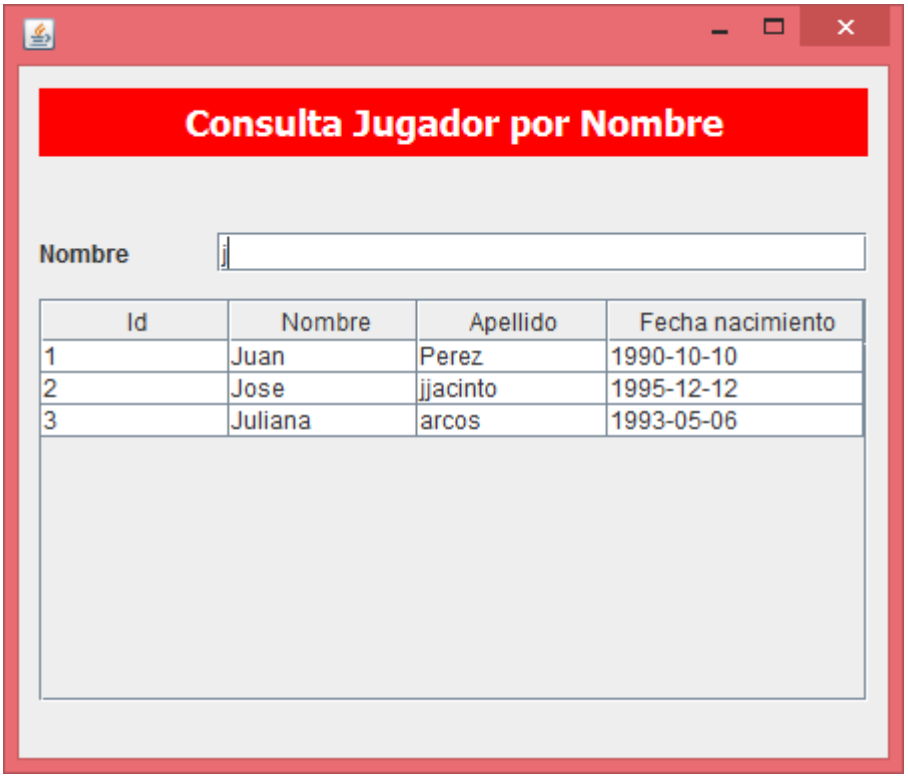
Se crea el procedimiento almacenado

```
DELIMITER $$

CREATE DEFINER=`root`@`localhost` PROCEDURE `consulta_jugador`(IN  param1
varchar(30
0)) BEGIN

select * from jugador where nombre like param1;

END
```

Para la ejecución se ejecuta el procedimiento almacenado mediante prepareCall

```
public List<Jugador> buscaXNombre(String nombre){
            ArrayList<Jugador> data = new
            ArrayList<Jugador>();

            Connection con = null;
            PreparedStatement pstm =
            null; ResultSet     rs =
            null;
            try {
                  con = new MiConexion().getConexion();
                  String sql ="{call consulta_jugador(?)}";
                  pstm = con.prepareCall(sql);
                  pstm.setString(1, nombre + "%");
                  rs = pstm.executeQuery();
                  Jugador c = null; while(rs.next()){
                        c = new Jugador();
                        c.setIdJugador(rs.getInt("idJugador"));
                        c.setNombre(rs.getString("nombre"));
                        c.setApellido(rs.getString("apellido"));

      c.setFechaNacimiento(rs.getDate("fechaNacimiento"));
                        data.add(c);
                  }

            } catch (Exception e) {
                  e.printStackTrace();
            } finally {
                  try {
                        if (pstm != null)pstm.close();
                        if (con != null)con.close();
                  } catch (SQLException e) {
                        e.printStackTrace();
                  }
            }

            return data;
      }
```

## Generar consultas




```
package entidad;

import

java.util.Date;

public class Jugador

{

	private int idJugador;
	private String nombre, apellido;
	private Date fechaNacimiento;

	public int getIdJugador() {
		return idJugador;
	}
	public void setIdJugador(int idJugador) {
		this.idJugador = idJugador;
	}
	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}
	public String getApellido() {
		return apellido;
```

```
	public Date getFechaNacimiento() {
		return fechaNacimiento;
	}
	public void setFechaNacimiento(Date fechaNacimiento) {
		this.fechaNacimiento = fechaNacimiento;
	}


}
```

```
package model;

import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import util.MiConexion;
import entidad.Jugador;

public class ModelJugador {

	public List<Jugador> buscaXNombre(String nombre){
		ArrayList<Jugador> data = new ArrayList<Jugador>();

		Connection con = null;
		PreparedStatement pstm = null;
		ResultSet  rs = null;
		try {
			con = new MiConexion().getConexion();

			String sql ="{call consulta_jugador(?)}";
			pstm = con.prepareCall(sql);
			pstm.setString(1, nombre + "%");
			rs = pstm.executeQuery();
			Jugador c = null;
			while(rs.next()){
				c = new Jugador();
				c.setIdJugador(rs.getInt("idJugador"));
				c.setNombre(rs.getString("nombre"));
				c.setApellido(rs.getString("apellido"));

	c.setFechaNacimiento(rs.getDate("fechaNacimiento"));
				data.add(c);
			}

		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			try {
				if (pstm != null)pstm.close();
				if (con != null)con.close();
			} catch (SQLException e) {
				e.printStackTrace();
```

```
                    }
                }

                return data;
        }



}
```

```
package gui;

import java.awt.Color;
import java.awt.EventQueue;
import java.awt.Font;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.List;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.SwingConstants;
import javax.swing.border.EmptyBorder;
import javax.swing.table.DefaultTableModel;

import model.ModelJugador;
import entidad.Jugador;

@SuppressWarnings("serial")
public class FrmConsultaJugador extends JFrame implements KeyListener {

	private JPanel contentPane;
	private JTextField txtNombre;
	private JTable table;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					FrmConsultaJugador frame = new
FrmConsultaJugador();
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}
```

```
	/**
	 * Create the frame.
	 */
	public FrmConsultaJugador() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
		; setBounds(100, 100, 450, 384);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);

		JLabel lblConsultaJugadorPor = new JLabel("Consulta Jugador por
Nombre");
		lblConsultaJugadorPor.setBackground(Color.RED);
		lblConsultaJugadorPor.setOpaque(true);

	lblConsultaJugadorPor.setHorizontalAlignment(SwingConstants.CENTER);
		lblConsultaJugadorPor.setForeground(Color.WHITE);
		lblConsultaJugadorPor.setFont(new Font("Tahoma", Font.BOLD,
18));
		lblConsultaJugadorPor.setBounds(10, 11, 414, 34);
		contentPane.add(lblConsultaJugadorPor);

		JLabel lblNombre = new JLabel("Nombre");
		lblNombre.setBounds(10, 82, 46, 23);
		contentPane.add(lblNombre);

		txtNombre = new JTextField();
		txtNombre.addKeyListener(this);
		txtNombre.setBounds(99, 83, 325, 20);
		contentPane.add(txtNombre);
		txtNombre.setColumns(10);

		JScrollPane scrollPane = new JScrollPane();
		scrollPane.setBounds(10, 116, 414, 201);
		contentPane.add(scrollPane);

		table = new JTable();
		table.setModel(new DefaultTableModel(
			new Object[][] {
			},
			new String[] {
				"Id", "Nombre", "Apellido", "Fecha nacimiento"
			}
		));
		table.getColumnModel().getColumn(3).setPreferredWidth(109);
		scrollPane.setViewportView(table);
	}
	public void keyPressed(KeyEvent arg0) {
	}
	public void keyReleased(KeyEvent arg0) {
		if (arg0.getSource() == txtNombre) {
			do_txtNombre_keyReleased(arg0);
		}
	}
	public void keyTyped(KeyEvent arg0) {
	}
	protected void do_txtNombre_keyReleased(KeyEvent arg0) {
		//1 obtener el texto
```

```
            String nombre = txtNombre.getText().trim();

            //2 Invocamos al ModelJugador
            ModelJugador m = new
            ModelJugador();
            List<Jugador> data = m.buscaXNombre(nombre);


            //3 Limpiamos el JTable
            DefaultTableModel dataTable =
(DefaultTableModel) table.getModel();
            dataTable.setRowCount(0);


            //4 Agregammos al JTable
            for (Jugador j : data) {
                  Object[] fila = {j.getIdJugador(),
                                          j.getNombre(),
                                          j.getApellido(),
                                          j.getFechaNacimiento(
                                          )} ;

                  dataTable.addRow(fila);
            }


      }
}
```

# Resumen

1 Para realizar una ejecución en procedimientos almacenados

```
String sql ="{call consulta_jugador(?)}";
pstm = con.prepareCall(sql);
```

2 Para pasar parámetros al procedimiento almacenado

```
pstm.setString(1, nombre + "%");
```

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- http://www.mkyong.com/jdbc/jdbc-callablestatement-stored-procedure-in-parameter-example/
  Ejemplos de ejecucion de procedimientos almacenados

- http://www.tutorialspoint.com/jdbc/jdbc-stored-procedure.htm
  Jdbc y procedimientos almacenados

# 5

# APLICACIONES DE ESCRITORIO EN JAVA CON CONEXIÓN A BASE DE DATOS

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones que utilizan una fuente de datos Mysql y podrá realizar mantenimientos, consultas y transacciones utilizando las clases Connection, PreparedStatement, etc; del JDBC. Además, podrá acceder a procedimientos almacenados.

**TEMARIO**

**5.5 Tema 11 : JDBC - Transacciones**

5.5.1 : Definición de Transacción

5.5.2 : Transacciones en JDBC

**ACTIVIDADES PROPUESTAS**

- Crear una transacción en Java

# 5.5. JDBC- TRANSACCIONES

## 5.5.1. Definición de Transacción

Un sistema de procesamiento de transacciones (TPS por sus siglas en inglés) es un tipo de sistema de información que recolecta, almacena, modifica y recupera toda la información generada por las transacciones producidas en una organización. Una transacción es un evento que genera o modifica los datos que se encuentran eventualmente almacenados en un sistema de información. Para que un sistema informático pueda ser considerado como un TPS, este debe superar el test ACID.

Desde un punto de vista técnico, un TPS monitoriza los programas transaccionales (un tipo especial de programas). La base de un programa transaccional está en que gestiona los datos de forma que estos deben ser siempre consistentes (por ejemplo, si se realiza un pago con una tarjeta electrónica, la cantidad de dinero de la cuenta sobre la que realiza el cargo debe disminuir en la misma cantidad que la cuenta que recibe el pago, de no ser así, ninguna de las dos cuentas se modificará), si durante el transcurso de una transacción ocurriese algún error, el TPS debe poder deshacer las operaciones realizadas hasta ese instante. Si bien este tipo de integridad es que debe presentar cualquier operación de procesamiento de transacciones por lotes, es particularmente importante para el procesamiento de transacciones on-line: si, por ejemplo, un sistema de reserva de billetes de una línea aérea es utilizado simultáneamente por varios operadores, tras encontrar un asiento vacío, los datos sobre la reserva de dicho asiento deben ser bloqueados hasta que la reserva se realice, de no ser así, otro operador podría tener la impresión de que dicho asiento está libre cuando en realidad está siendo reservado en ese mismo instante. Sin las debidas precauciones, en una transacción podría ocurrir una reserva doble. Otra función de los monitores de transacciones es la detección y resolución de interbloqueos (deadlock), y cortar transacciones para recuperar el sistema en caso de fallos masivos.

**Características**

**Respuesta rápida**
En este tipo de sistemas resulta crítico que exista un rendimiento elevado con tiempos de respuesta cortos. Una empresa no puede permitirse tener clientes esperando por una respuesta del SPT; el tiempo total transcurrido desde que se inicia la transacción hasta que se produce la salida correspondiente debe ser del orden de unos pocos segundos o menos.

**Fiabilidad**
Muchas organizaciones basan su fiabilidad en los SPT; un fallo en un SPT afectará negativamente a las operaciones o incluso parará totalmente el negocio. Para que un SPT sea efectivo, su tasa de fallos debe ser muy baja. En caso de fallo de un SPT, debe existir algún mecanismo que permita una recuperación rápida y precisa del sistema. Esto convierte en esencial la existencia procedimientos de copia de seguridad y de recuperación ante fallos correctamente diseñados.
**Inflexibilidad**

Un SPT requiere que todas las transacciones sean procesadas exactamente de la misma forma, independientemente del usuario, el cliente o la hora del día. Si los SPT fuesen flexibles, habría entonces demasiadas posibilidades de ejecutar operaciones no estándar. Por ejemplo, una aerolínea comercial necesita aceptar de forma consistente reservas de vuelos realizadas por un gran número de agencias de viaje distintas; aceptar distintos datos de transacción de cada agencia de viajes supondría un problema.

**Procesamiento controlado**
El procesamiento en un SPT debe apoyar las operaciones de la organización. Por ejemplo, si una organización establece roles y responsabilidades para determinados empleados, el SPT debe entonces mantener y reforzar este requisito.

**Propiedades**

**Atomicidad**
Los cambios de estado provocados por una transacción son atómicos: o bien ocurren todos o bien no ocurre ninguno. Estos cambios incluyen tanto modificaciones de la base de datos, como envío de mensajes o acciones sobre los transductores.

**Consistencia**
Una transacción es una transformación de estado correcta. Las acciones consideradas en su conjunto no violan ninguna de las restricciones de integridad asociadas al estado. Esto implica que la transacción debe ser un programa correcto.

**Aislamiento**
Incluso cuando varias transacciones se ejecuten de forma concurrente, para cada transacción T debe parecer que el resto de transacciones se han ejecutado antes o después de T, pero no antes y después.

**Durabilidad**
Una vez que una transacción ha finalizado con éxito (compromiso), cambia hacia un estado estable a prueba de fallos.

## 5.5.2. Transacciones en JDBC

Hay momentos en los que no desea una declaración entre en vigor a menos que otra completa. Por ejemplo, cuando el propietario de The Coffee Break actualiza la cantidad de café que se vende cada semana, el propietario también desee actualizar la cantidad total vendida hasta la fecha. Sin embargo, la cantidad vendida por semana y la cantidad total vendida debe actualizarse al mismo tiempo; de lo contrario, los datos serán inconsistentes. La manera de estar seguro de que se produzcan ya sea ambas acciones o ninguna acción ocurre es utilizar una transacción. Una transacción es un conjunto de una o más sentencias que se ejecuta como una unidad, por lo que o bien todas las sentencias se ejecutan, o ninguna de las declaraciones se ejecuta.
**Desactivación del modo de confirmación automática**
Cuando se crea una conexión, que está en modo auto-commit. Esto significa que cada sentencia SQL individuo es tratado como una transacción y se compromete de forma

automática después de que se ejecuta. (Para ser más precisos, el valor predeterminado es para una sentencia SQL que se comete cuando se haya completado, no cuando se ejecuta. Se completa una declaración cuando todos sus conjuntos de resultados y cuentas de actualización se han recuperado. En casi todos los casos, sin embargo, se completa una declaración, y por lo tanto comprometido, justo después de que se ejecute.)

La manera de permitir que dos o más declaraciones a agruparse en una transacción es desactivar el modo auto-commit. Esto se demuestra en el siguiente código, donde con una conexión activa:

con.setAutoCommit (false);

**Cometer Transacciones**

Después de que el modo de confirmación automática está desactivado, no hay sentencias SQL se han comprometido hasta que llame al método commit explícitamente. Todas las declaraciones realizadas después de la llamada anterior al método commit se incluyen en la transacción actual y se comprometieron juntos como una unidad. El siguiente método, CoffeesTable.updateCoffeeSales , en el que con una conexión activa, ilustra una transacción:

```
public void updateCoffeeSales(HashMap<String, Integer>
salesForWeek) throws SQLException {

    PreparedStatement updateSales =
    null; PreparedStatement
    updateTotal = null;

    String updateString =
        "update " + dbName +
        ".COFFEES " + "set SALES = ?
        where COF_NAME = ?";

    String updateStatement =
        "update " + dbName + ".COFFEES
        " + "set TOTAL = TOTAL + ? " +
        "where COF_NAME = ?";
    try {
        con.setAutoCommit(f
        alse);
        updateSales = con.prepareStatement(updateString);
        updateTotal = con.prepareStatement(updateStatement);
```

```
        for (Map.Entry<String, Integer> e : salesForWeek.entrySet())
        { updateSales.setInt(1, e.getValue().intValue());
        updateSales.setString(2, e.getKey());
        updateSales.executeUpdate();
            updateTotal.setInt(1,
            e.getValue().intValue());
            updateTotal.setString(2, e.getKey());
            updateTotal.executeUpdate();
            con.commit();
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e
        ); if (con != null) {
            try {
                System.err.print("Transaction is being rolled back");
                con.rollback();
            } catch(SQLException excep) {
                JDBCTutorialUtilities.printSQLException(excep);
            }
        }
    } finally {
        if (updateSales != null)
            {
            updateSales.close();
        }
        if (updateTotal != null)
            {
            updateTotal.close();
        }
        con.setAutoCommit(true);
    }
}
```

En este método, el modo de confirmación automática está deshabilitada para la conexión con, lo que significa que las dos sentencias preparadas updateSales y updateTotal están comprometidos juntos cuando el método commit se llama. Cada vez que el commit se llama al método (ya sea de forma automática cuando el modo de confirmación automática está habilitada o explícitamente cuando está desactivada), todos los cambios resultantes de las declaraciones en la transacción se hizo permanente. En este caso, eso significa que las SALES y TOTAL columnas para el café colombiano se han cambiado a 50 (si TOTAL había sido 0 anteriormente) y mantendrán este valor hasta que se cambien con otra sentencia de actualización.

La declaración con.setAutoCommit(true); activa el modo auto-commit, lo que significa que cada declaración es una vez más el compromiso de forma automática cuando se haya completado. Entonces, usted está de vuelta al estado predeterminado en el que no tienes que llamar al método commit usted mismo. Es recomendable desactivar el

modo auto-commit sólo durante el modo de transacción. De esta manera, se evita que contienen bloqueos de base de datos para múltiples declaraciones, lo que aumenta la probabilidad de conflictos con otros usuarios.

**Uso de transacciones para Preservar la integridad de los datos**

Además de la agrupación de estados juntos para la ejecución como una unidad, las transacciones pueden ayudar a preservar la integridad de los datos en una tabla. Por ejemplo, imagine que un empleado tenía que entrar en nuevos precios del café en la mesa COFFEES pero retrasó haciéndolo durante unos días. Mientras tanto, los precios subieron, y en la actualidad el propietario es en el proceso de introducción de los precios más altos. El empleado finalmente consigue alrededor de entrar en los precios ahora obsoletos, al mismo tiempo que el dueño está tratando de actualizar la tabla. Después de insertar los precios desfasados, el empleado se da cuenta de que ya no son válidos y llama a la Connection método rollback para deshacer sus efectos. (El método rollback aborta la transación y restaura los valores a lo que eran antes del intento de actualización.) Al mismo tiempo, el propietario está ejecutando un SELECT declaración y la impresión de los nuevos precios. En esta situación, es posible que el propietario se imprimirá un precio que se había deshecho de su valor anterior, por lo que el precio impreso incorrecta.

Este tipo de situación se puede evitar mediante el uso de transacciones, proporcionar algún nivel de protección contra los conflictos que surgen cuando dos usuarios acceder a los datos al mismo tiempo.

Para evitar conflictos durante una transacción, un DBMS utiliza bloqueos, mecanismos para bloquear el acceso de otros a los datos que se esté accediendo a la transacción. (Tenga en cuenta que el modo de auto-commit, donde cada sentencia es una transacción, bloqueos se mantienen durante una sola declaración.) Después se establece un bloqueo, que permanece en vigor hasta que la transacción se confirma o se deshace. Por ejemplo, un DBMS podría bloquear una fila de una tabla hasta cambios a que se han comprometido. El efecto de este bloqueo sería la de evitar que un usuario conseguir una lectura sucia, es decir, la lectura de un valor antes de que se haga permanente. (Acceso a un valor actualizado que no se ha cometido se considera una lectura sucia porque es posible que el valor que se revierte a su valor anterior. Si se lee un valor que luego se deshace, se ha leído un valor no válido. )

¿Cómo se establecen bloqueos está determinado por lo que se llama un nivel de aislamiento, que puede ir desde no apoyar transacciones en absoluto para apoyar transacciones que hacen cumplir las reglas de acceso muy estrictas.

Un ejemplo de un nivel de aislamiento se TRANSACTION_READ_COMMITTED , que no permitirá que un valor que se accede hasta después de que se ha cometido. En otras palabras, si el nivel de aislamiento de transacción se establece en TRANSACTION_READ_COMMITTED , el DBMS no permite lecturas sucias que se produzca. La interfaz Connection incluye cinco valores que representan los niveles de aislamiento de transacción que puede utilizar en JDBC:

| Isolation Level | Transactions | Dirty Reads | Non-Repeatable Reads | Phantom Reads |
|---|---|---|---|---|
| TRANSACTION_NONE | Not supported | *Not applicable* | *Not applicable* | *Not applicable* |
| TRANSACTION_READ_COMMITTED | Supported | Prevented | Allowed | Allowed |
| TRANSACTION_READ_UNCOMMITTED | Supported | Allowed | Allowed | Allowed |
| TRANSACTION_REPEATABLE_READ | Supported | Prevented | Prevented | Allowed |
| TRANSACTION_SERIALIZABLE | Supported | Prevented | Prevented | Prevented |

Una lectura no repetible se produce cuando la transacción A recupera una fila, la transacción B posteriormente actualiza la fila, y la transacción Una tarde recupera la misma fila de nuevo. Transacción A recupera la misma fila dos veces, pero ve datos diferentes.

Una lectura fantasma se produce cuando la transacción A recupera un conjunto de filas que satisfacen una condición dada, la transacción B posteriormente inserta o actualiza una fila de tal manera que la fila ahora cumple la condición en la transacción A, y la transacción Una tarde repite la recuperación condicional. Transacción A ahora ve una fila adicional. Esta fila se conoce como un fantasma.

Por lo general, usted no tiene que hacer nada sobre el nivel de aislamiento de transacción; sólo puede utilizar el valor predeterminado para su DBMS. El nivel de aislamiento de transacción por defecto depende de su DBMS. Por ejemplo, para Java DB, se TRANSACTION_READ_COMMITTED . JDBC permite a averiguar cuál es el nivel de aislamiento de transacción de su DBMS está ajustado en (utilizando la Connection método getTransactionIsolation ) y también le permite establecer a otro nivel (usando la Connection método setTransactionIsolation ).

Nota: Un controlador JDBC no admita todos los niveles de aislamiento de transacción. Si un conductor no soporta el nivel de aislamiento especificado en una invocación de setTransactionIsolation , el conductor puede sustituir un nivel de aislamiento más restrictivo superior. Si un conductor no puede sustituir a un nivel de transacción superior, que lanza una SQLException . Utilice el método DatabaseMetaData.supportsTransactionIsolationLevel para determinar si el controlador es compatible con un determinado nivel.

Ajuste y Recuperando la Puntos de salvaguarda

El método Connection.setSavepoint , establece un Savepoint objeto dentro de la transacción actual. El Connection.rollback método está sobrecargado para tomar un Savepoint argumento.

El siguiente método, CoffeesTable.modifyPricesByPercentage , eleva el precio de un café particular, en un porcentaje, priceModifier . Sin embargo, si el nuevo precio es superior a un precio determinado, maximumPrice , entonces el precio se revierte al precio original:

```
public void
    modifyPricesByPercentage(
    String coffeeName,
    float
    priceModifier,
    float maximumPrice)
    throws SQLException
    {

    con.setAutoCommit(false);

    Statement getPrice = null;
    Statement updatePrice =
    null; ResultSet rs = null;
    String query =
        "SELECT COF_NAME, PRICE FROM COFFEES " +
        "WHERE COF_NAME = '" + coffeeName + "'";

    try {
        Savepoint save1 =
        con.setSavepoint(); getPrice =
        con.createStatement(
                       ResultSet.TYPE_SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_READ_ONLY
        ); updatePrice = con.createStatement();

        if (!getPrice.execute(query))
            { System.out.println(
                "Could not find entry "
                + "for coffee named " +
                coffeeName);
        } else {
            rs =
            getPrice.getResultSet();
            rs.first();
            float oldPrice = rs.getFloat("PRICE");
            float newPrice = oldPrice + (oldPrice * priceModifier);
            System.out.println(
                "Old price of " + coffeeName
                + " is " + oldPrice);

            System.out.println(
                "New price of " + coffeeName
                + " is " + newPrice);

            System.out.println(
                "Performing
                update...");

            updatePrice.executeUpdate(
                "UPDATE COFFEES SET PRICE = " +
                newPrice +
                " WHERE COF_NAME = '" +
                coffeeName + "'");
```

```
            System.out.println(
                "\nCOFFEES table after
                " + "update:");

            CoffeesTable.viewTable(con)

            ; if (newPrice >

            maximumPrice) {

                System.out.println( "\nThe new
                    price, " + newPrice +
                    ", is greater than the
                    " + "maximum price, " +
                    maximumPrice +
                    ". Rolling back the " +
                    "transaction...");

                con.rollback(save1);

                System.out.println(
                    "\nCOFFEES table " +
                    "after rollback:");

                CoffeesTable.viewTable(con);
            }
            con.commit();
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (getPrice != null) { getPrice.close(); } if
        (updatePrice != null) {
            updatePrice.close();
        }
        con.setAutoCommit(true);
    }
}
```

La siguiente declaración especifica que el cursor del ResultSet objeto generado desde el getPrice consulta se cierra cuando el commit se llama al método. Tenga en cuenta que si su DBMS no soporta ResultSet.CLOSE_CURSORS_AT_COMMIT , entonces se ignora esta constante

getPrice=con.prepareStatement (consulta,
ResultSet.CLOSE_CURSORS_AT_COMMIT);

El método comienza por la creación de un Savepoint con la siguiente declaración:

Savepoint GUARDAR1 = con.setSavepoint ();
El método comprueba si el nuevo precio es mayor que el maximumPrice valor. Si es así,

el método revierte la transacción con la siguiente declaración:

con.rollback (GUARDAR1);

En consecuencia, cuando el método confirma la transacción mediante una llamada al Connection.commit método, no va a cometer ningún filas cuyos asociados Savepoint se ha deshecho; comprometerá todas las otras filas actualizadas.

**Liberar Puntos de salvaguarda**

El método Connection.releaseSavepoint toma un Savepoint objeto como parámetro y lo elimina de la transacción actual.

Después de un punto de rescate ha sido puesto en libertad, tratando de hacer referencia a ella en una operación de reversión causa una SQLException que se lance. Cualquier punto de salvaguarda que se han creado en una transacción se liberan automáticamente y dejan de ser válidas cuando se confirma la transacción, o cuando la transacción se retrotrae. Rodar una transacción a un punto de salvaguarda se desconecta automáticamente y hace inválida cualquier otro puntos de rescate que se crearon tras el punto en cuestión.

**Al llamar al método rollback**

Como se mencionó anteriormente, la llamada al método rollback termina una transacción y devuelve los valores que se han modificado a sus valores anteriores. Si usted está tratando de ejecutar una o más declaraciones en una transacción y obtener una SQLException , llame al método rollback para poner fin a la operación y comenzar la operación de nuevo. Esa es la única manera de saber lo que se ha cometido y lo que no se ha cometido. La captura de un SQLException te dice que algo está mal, pero no le dice lo que fue o no fue cometido. Porque no se puede contar con el hecho de que nada se haya cometido, una llamada al método rollback es la única manera de estar seguro.

El método CoffeesTable.updateCoffeeSales demuestra una transacción e incluye una catch bloque que invoca el método rollback . Si la aplicación continúa y utiliza los resultados de la transacción, esta llamada a la rollback método en la catch bloque impide el uso de datos posiblemente incorrectos.

**Ejemplo de inserción de Boleta**

```
package model;

import  java.sql.Connection;
importjava.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
```

```
import util.ConexionDB;
import entidad.BoletaBean;
import entidad.DetalleBoletaBean;

public class ModelBoleta {

      public int inserta(BoletaBean boletaBean,
                                    List<DetalleBoletaBean> lstDetalle){
            int contador = -1;
            Connection conn = null;
            PreparedStatement pstm1 = null, pstm2= null,pstm3= null;

             try {
                        conn = new ConexionDB().getConexion();
                        //Se anula el auto envío
                        conn.setAutoCommit(false);

                        //se crea el sql de la cabecera
                        String sql1 ="insert into boleta
                        values(null,?,?,?)"; pstm1 =
                        conn.prepareStatement(sql1);
                        pstm1.setDate(1, boletaBean.getFecha());
                        pstm1.setInt(2,
                        boletaBean.getIdCliente());
                        pstm1.setInt(3,
                        boletaBean.getIdUsuario());
                        pstm1.executeUpdate();

                        //se obtiene el idBoleta insertado
                        String sql2 ="select max(idBoleta) from boleta";
                        pstm2 =   conn.prepareStatement(sql2);
                        ResultSet rs = pstm2.executeQuery();
                        rs.next();
                        int idBoleta = rs.getInt(1);
 values(?,?,?,?)";
                        //se inserta el detalle de boleta
                        String sql3 ="insert into producto_has_boleta

                        pstm3 =   conn.prepareStatement(sql3);
                     for (DetalleBoletaBean aux : lstDetalle) {
                           pstm3.setInt(1, aux.getIdProducto());
                           pstm3.setInt(2, idBoleta);
                           pstm3.setDouble(3, aux.getPrecio());
                           pstm3.setInt(4, aux.getCantidad());
                           pstm3.executeUpdate();
          }

                        //se ejecuta todos los SQL en la base de
                        datos conn.commit();
                  } catch (Exception e) {
                        try {
                              conn.rollback();
                              //se vuelva a un inicio
                              //No permite un SQL por separado
                        } catch (SQLException
                        e1) {}
                        System.out.println(e);
                  } finally{
```

```
                                try {
                                          conn.close();

                                       pstm1.close();
                                       pstm2.close();

                                pstm3.close();

                          } catch (SQLException e) {
                          }
                    }
                    return contador;
              }
}
```

# Resumen

1 Evita el envio automático

```
conn.setAutoCommit(false);
```

2 Permite el envio de todas las sentencias sqls

```
conn.commit();
```

3 Permite la cancelación de las sentencias sqls

```
conn.rollback();
```

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- ✓ http://www.tutorialspoint.com/jdbc/jdbc-transactions.htm
  Transacciones

- ✓ http://www.java2s.com/Code/Java/Database-SQL-JDBC/CommitorrollbacktransactioninJDBC.htm
  Ejemplos de transacciones

# 5

# APLICACIONES DE ESCRITORIO EN JAVA CON CONEXIÓN A BASE DE DATOS

**LOGRO DE LA UNIDAD DE APRENDIZAJE**

Al término de la unidad, el alumno crea aplicaciones que utilizan una fuente de datos Mysql y podrá realizar mantenimientos, consultas y transacciones utilizando las clases Connection, PreparedStatement, etc; del JDBC. Además, podrá acceder a procedimientos almacenados.

**TEMARIO**

**5.6 Tema 12 : Reportes**

5.6.1 : IReport

5.6.2 : Generación de reportes

**ACTIVIDADES PROPUESTAS**

- Crear un reporte

## 5.6. Reportes

### 5.6.1. IReporte

**iReport es el diseñador de informes Open Source para JasperReports,** facilita de forma visual, la construcción y edición de los reportes.

iReport genera un archivo de tipo “jrxml”, en el cual se encontrará el diseño y estilo del reporte, el cual al compilarse genera el archivo de tipo "jasper".

**Jasper Reports**

Jasper Reports es un lenguaje para generación de reportes en Java.

Se diseña el reporte en un archivo .xml y se compila a un archivo . Jasper.

Este archivo se interpreta junto a la BD generando el reporte .PDF.

### 5.6.2 Generación de Reportes

Realizar el reporte de la tabla Productos

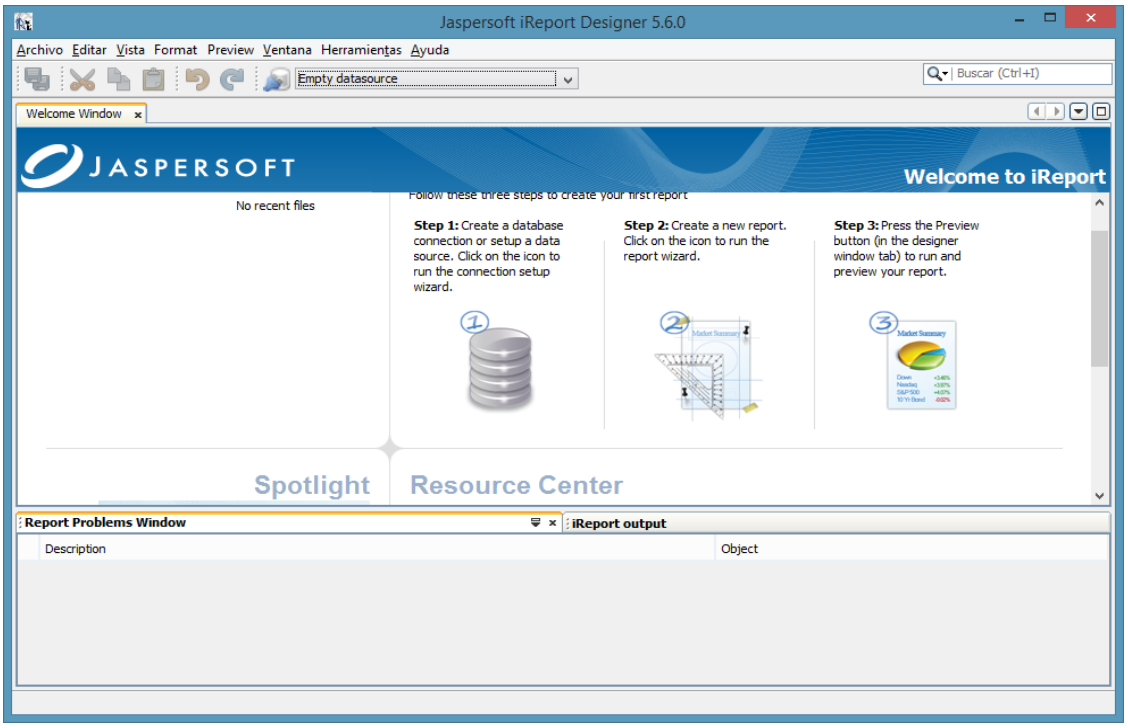

**Criterios para crear el reporte**

- Abrir el programa iReports
- Configurar la BD a utilizar

- Diseñar el reporte

## Abrir iReports

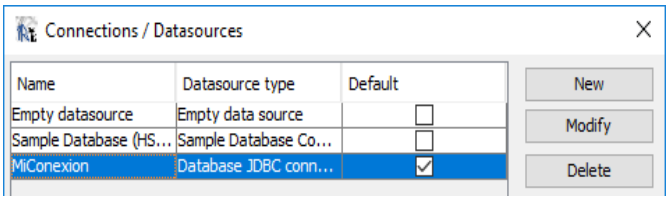
Ejecuta el icono


## Configurando la BD

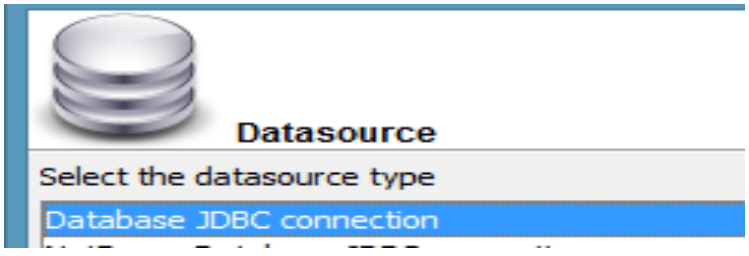
Para especificar los datos a mostrar debemos:


- **Definir** el Datasource (origen de datos):
- Selecciona una conexión existente, sino crear una **nueva**



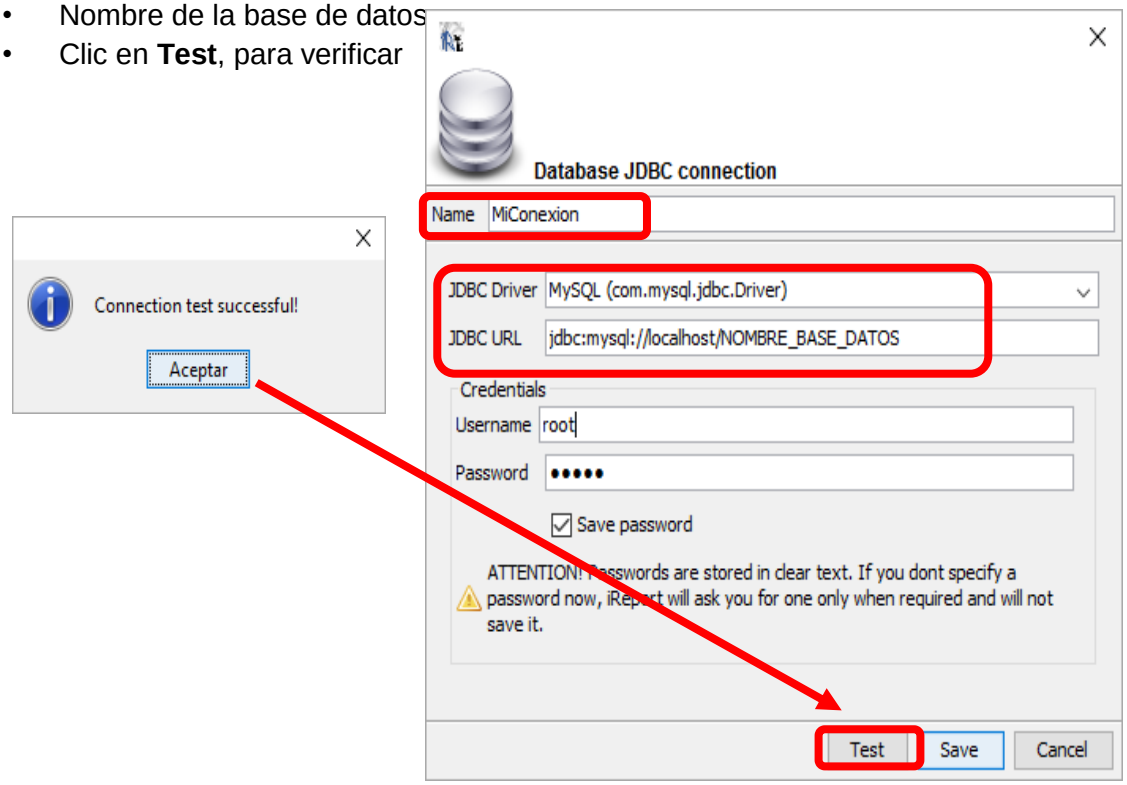


- Tipo **Database JDBC Connection**



Ingrese los siguientes datos:

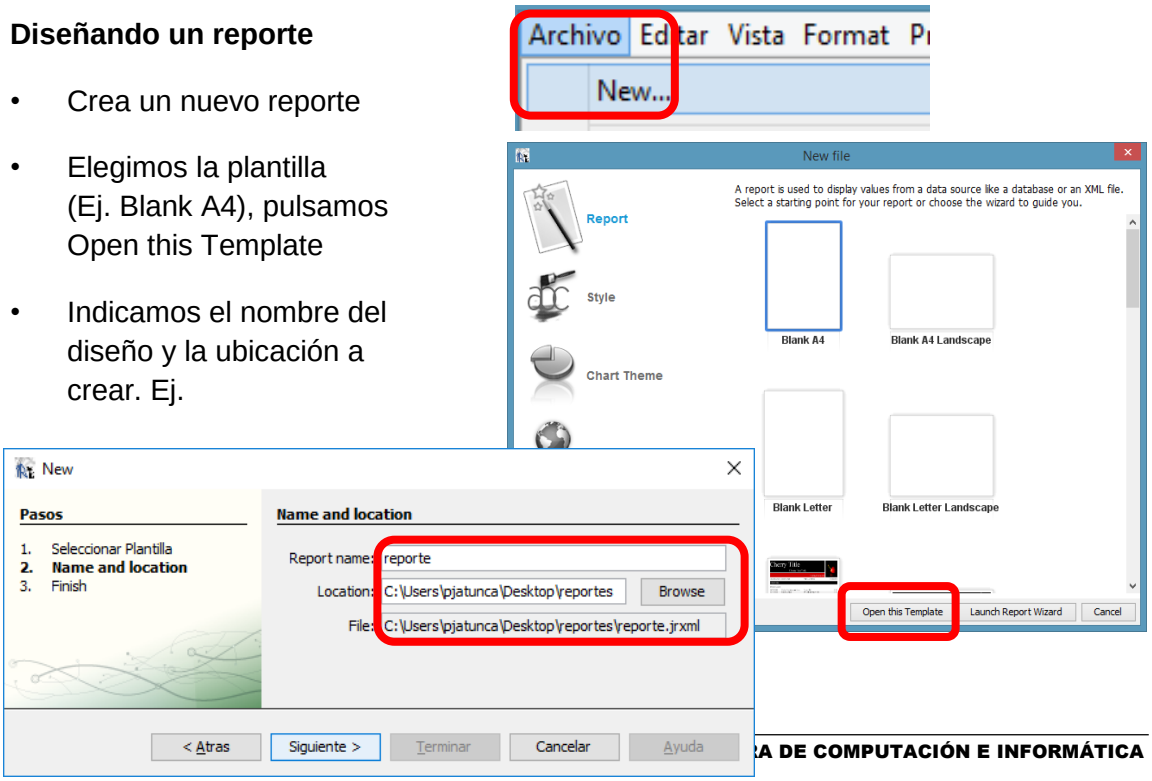
- Nombre para la conexión
- Elija el Tipo de Conexión
- Nombre de la base de datos
- Clic en **Test**, para verificar



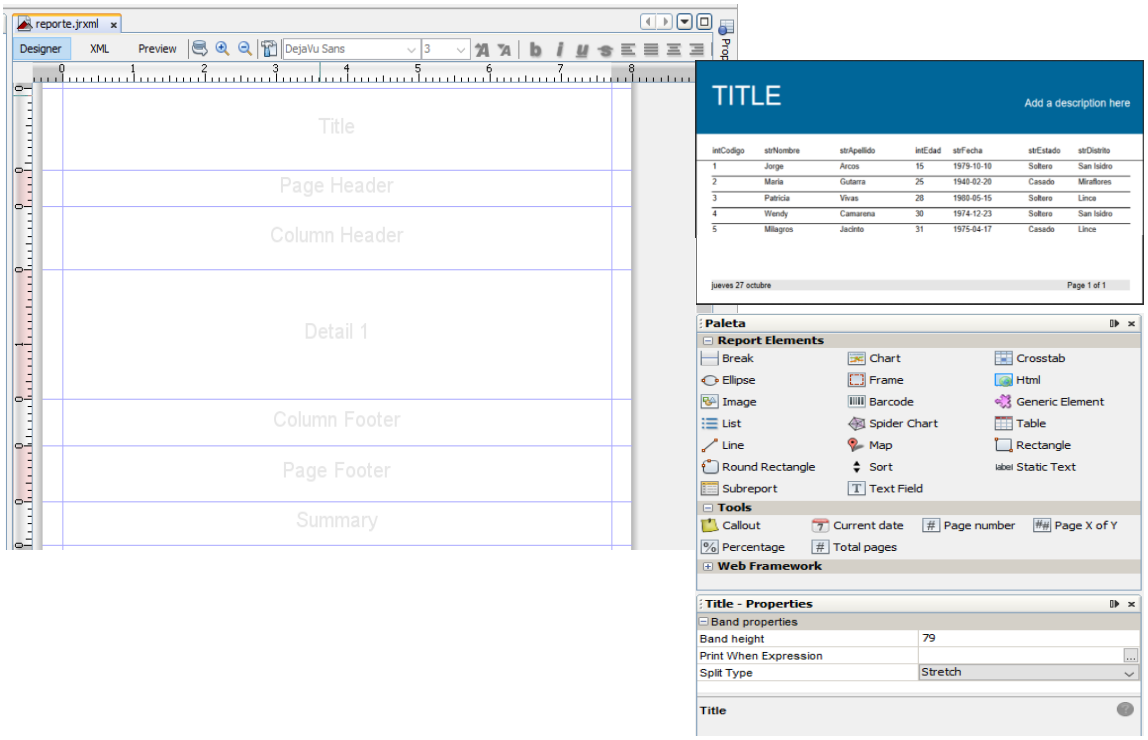

- De haber error, revisar si ya se creó la base de datos
- Finalmente Grabar y cerrar la ventana

**Diseñando un reporte**

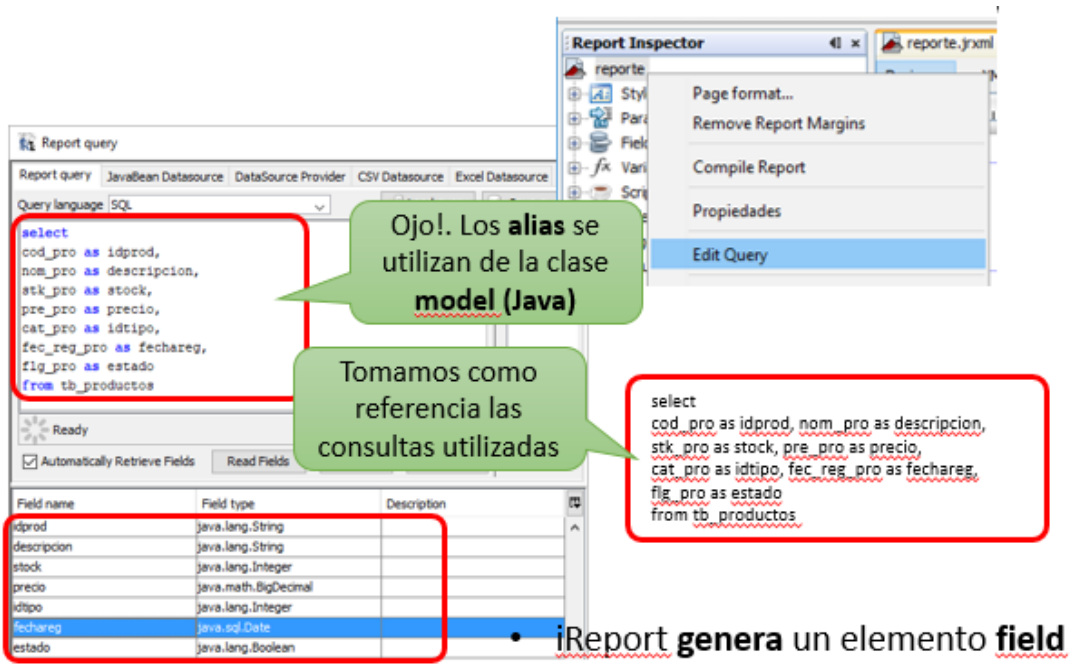
- Crea un nuevo reporte
- Elegimos la plantilla (Ej. Blank A4), pulsamos Open this Template
- Indicamos el nombre del diseño y la ubicación a crear. Ej.



Identifica las zonas a utilizar y las herramientas

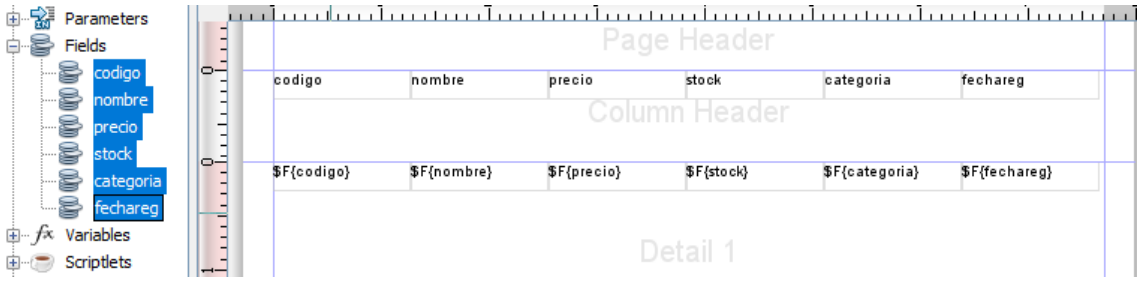



- Definimos la consulta

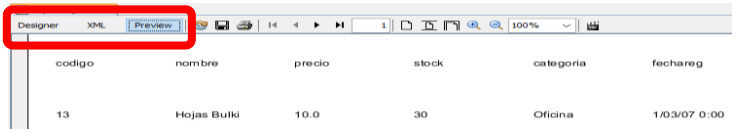



- iReport **genera** un elemento **field** por cada columna, mapeando el tipo de dato con su **equivalente** en Java.

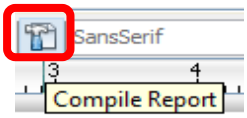
- Para colocar los campos del reporte, se arrastra los campos.




- Visualizamos el resultado




- Mejoramos el diseño y
- Generamos el .jasper



# Resumen

1 Es un lenguaje para generación de reportes en Java

**Jasper Reports**

2 Es el diseñador de informes Open Source para JasperReports

**iReports**

Pueden revisar los siguientes enlaces para ampliar los conceptos vistos en esta unidad:

- https://jonathanmelgoza.com/blog/como-crear-reportes-en-java/
  Ejemplos para crear reportes en java

- http://www.cpxall.com/2013/01/mi-primer-reporte-en-java-con-eclipse-ireport-mysql.htm
  Reporte en java

# 6

# GENÉRICO Y COLECCIONES

## LOGRO DE LA UNIDAD DE APRENDIZAJE

Al término de la unidad, el alumno crea aplicaciones que utilicen eficientemente las clases e interfaces del Framewok Collections tales como Set, List y Map, manipulando arreglos para ordenarlos o implementar búsquedas binarias basadas en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

## TEMARIO

**6.1 Tema 13 : Colecciones**

6.1.1 : Framework Collections. Principales clases e Interfaces.
6.1.2 : Clase ArrayList. Principales métodos.
6.1.3 : Ordenamiento de Collections y arreglo
6.1.4 : Set y tipos.
6.1.5 : Map y tipos.

## ACTIVIDADES PROPUESTAS

- Crear ejemplos de listas
- Crear ejemplos de mapas

# 6.1. COLECCIONES

## 6.1.1. Framework Collections. Principales clases e Interfaces.

Las colecciones en Java, se empiezan a utilizar a partir de la versión 1.2, se amplía luego su funcionalidad en la versión 1.4 mejorándose aún más en la versión 5.

¿Qué podemos hacer con una colección?

Existen diversas operaciones básicas que podremos realizar con las colecciones. A continuación, se mencionan las principales:

- Añadir objetos a la colección
- Eliminar objetos de la colección
- Determinar si un objeto (o grupo de objetos) se encuentra en la colección
- Recuperar un objeto de la colección (sin eliminarlo)

La API collections de Java involucra un grupo de interfaces y también un amplio número de clases concretas. Las principales interfaces de este Framework son las siguientes:

| | | |
|---|---|---|
| Collection | Set | SortedSet |
| List | Map | SortedMap |
| Queue | | |

Las principales clases del Framework son las siguientes:

| Maps | Sets | Lists | Queues | Utilities |
|---|---|---|---|---|
| HashMap | HashSet | ArrayList | PriorityQueue | Collections |
| Hashtable | LinkedHashSet | Vector | | Arrays |
| TreeMap | TreeSet | LinkedList | | |
| LinkedHashMap | | | | |

El término colección, dentro del contexto de Java, posee tres significados, los cuales detallamos a continuación:

**colección** (con “c” minúscula) representa cualquiera de las estructuras de datos en la que se almacenan los objetos y pueden ser iterados.

**Colección** (con “C” mayúscula), es en realidad la interfaz java.util.Collection a partir de

la cual heredan los componentes Set, List y Queue.

**Colecciones** (con “C” mayúscula y terminando con “s”). Es la clase java.util.Collections, la cual nos proporciona métodos estáticos de utilidad para la gestión de colecciones.

**Tipos básicos de una colección:**

Las colecciones pueden ser de cuatro tipos básicos:

**Lists:** Listas de elementos (clases que implementan la interface List).

**Sets:** Elementos únicos (clases que implementan la interface Set).

**Maps:** Elementos con un único ID (Clases que implementan la interface Map).

**Queues:** Elementos ordenados en base al “orden” en que fueron procesados.

Dentro de estos cuatro tipos básicos existen los siguientes sub tipos:

Sorted Unsorted Ordered Unordered

Una clase que implemente una interface de la familia Collection, puede ser desordenada y no estar clasificada (unsorted y unordered), clasificada pero no ordenada, o ambas, clasificada y ordenada.

Una implementación nunca podrá ser ordenada, pero no clasificada, porque el ordenamiento es un tipo específico de clasificación. Por ejemplo, un HashSet es un conjunto no clasificado y desordenado, mientras que un LinkedHashSet es un conjunto clasificado (pero no ordenado) que mantiene el orden en el cual los objetos fueron insertados.

**Colecciones clasificadas**. Una colección clasificada significa que podemos iterar a través del collection en un orden específico (not-random). Un Hashtable no está clasificado. Aunque el Hashtable en sí mismo tiene una lógica interna para para determinar el orden (basado en hashcodes y en la implementación del collection en sí misma), no encontraremos ningún orden cuando iteremos sobre un Hashtable. Un ArrayList, por otro lado, mantiene el orden establecido por la posición del índice de cada elemento. Un objeto LinkedHashSet keeps mantiene el orden establecido por la inserción.

**Colecciones ordenadas.** Una colección ordenada significa que el orden en el collection está determinado por alguna regla o reglas conocidas como “sort order”. Un sort order no tiene nada que hacer con el hecho de agregar un objeto a un collection, cuándo fue la última vez que fue accesado o en qué posición fue agregado. El ordenamiento se hace sobre la base de las propiedades de los objetos en sí mismos.Cuando colocamos objetos en un collection, el collection asumirá el orden en que lo colocamos, basado en el sort order. Una colección que mantiene un orden (Como cualquier List, los

cuales usan el orden de inserción) no es considerada realmente ordenada, a menos que el collection ordene sus elementos utilizando algún tipo de sort order. El sort order más comúnmente utilizado es uno llamado **natural order**.

¿Qué signfica ésto?

Para un collection de objetos de tipo String, el orden natural es alfabético. Para una colección de enteros, el orden natural es por valor numérico, 1 antes que 2 y así sucesivamente. Para objetos creados por el usuario, no existe un orden natural, a menos que o hasta que le proporcionemos uno a través de una interface (Comparable) que define como las instancias de una clase pueden ser comparadas unas con otras. Si el desarrollador decide que los objetos deben ser comparados usando el valor de alguna variable de instancia, entonces una colección ordenada ordenará los objetos sobre la base de las reglas en dicha clase definidas para hacer ordenamiento sobre la base de una variable específica. Evidentemente, este objeto podría también heredar un orden natural de una super clase.

**Importante:**

Tenga en cuenta que un sort order (incluyendo el orden natural), no es lo mismo que el orden dado por la inserción, acceso o un índice.

### 6.1.2. Clase ArrayList. Principales métodos.

Se orienta al manejo de indices. Cuenta con un conjunto de métodos relacionados con la gestión de los índices. Así tenemos, por ejemplo, los métodos:
get(int index),
indexOf(Object o),
add(int index, Object obj), etc.

Las tres implementaciones de la interface List están ordenadas por la posición del índice, una posición que determinamos ya sea cargando un objeto en un índice específico o agregándolo sin especificar posición. En este caso, el objeto es agregado al final. A continuación, se describen las implementaciones de esta interface:

**ArrayList** es una lista dinámica que nos proporciona una iteración rápida y un rápido acceso aleatorio. Es una colección clasificada (por índice) pero no ordenada. Implementa la interface RandomAccess, la cual permite que la lista soporte un rápido acceso aleatorio. Es ideal y superior a LinkedList para iteraciones rápidas, mas no para realizar muchas operaciones de insert o delete.

**Vector** es básicamente igual que un ArrayList, pero sus métodos se encuentran sincronizados para para soportar thread safety. Vector es la única clase junto con ArrayList que cuentan con acceso aleatorio.

**LinkedList** Este objeto está clasificado por posición del índice como un ArrayList, con la excepción de que sus elementos se encuentran doblemente enlazados uno con otro. Este esquema permite que contemos con nuevos métodos para agregar y eliminar desde el inicio o desde el final, lo que lo hace una opción interesante para implementar pilas o colas. Puede iterar más lentamente que un ArrayList, pero es la major opción para hacer inserts o deletes rápidamente. A partir de la versión 5, implementa la interface java.util.Queue, por lo que soporta los métodos relacionados con colas: : peek(), poll(), and offer().

### 6.1.3. Ordenamiento de Collections y arreglo

**Ordenando Collections**

A continuación se muestra un ejemplo de aplicación:

```
import java.util.*;
class TestSortl {
  public static void main(String[] args) {
    ArrayList<String> stuff = new ArrayList<String>(); // #1
    stuff.add("Denver");
    stuff.add("Boulder");
    stuff.add("Vail") ;
    stuff.add("Aspen"); stuff.add("Telluride");
    System.out.println("unsorted " + stuff);
    Collections.sort(stuff);                                    // #2
    System.out.println("sorted " + stuff);
  }
}
```

Obtendremos la siguiente salida:

Salida desordenada: [Denver, Boulder, Vail, Aspen, Telluride] Salida ordenada: [Aspen, Boulder, Denver, Telluride, Vail]

En la línea 1 declaramos un ArrayList de Strings, y en la línea 2 ordenamos el ArrayList alfabéticamente.

**La interface Comparable**

La interface Comparable es usada por los métodos Collections.sort() y java.utils.Arrays.sort() para ordenar listas y arreglos de objetos respectivamente. Para implementar Comparable, una clase debe implementar el método compareTo():

int x = thisObject.compareTo(anotherObject);

El método compareTo() retorna un valor entero con las siguientes características:

**Negative**

Si este objeto < otro Object

**zero**

Si este objeto == otro Objeto

**Positive**

If este objeto > otro Objeto

El método sort() usa compareTo() para determinar cómo la lista o arreglo de objetos debe ser ordenada. Dado que podemos implementar el método compareTo() para nuestras propias clases, podremos usar cualquier criterio de ordenamiento para nuestras clases:

```
class DVDInfo implements Comparable<DVDInfo> { // #1
  // existing code
  public int compareTo(DVDInfo d) {
    return title.compareTo(d.getTitle());              // #2
} }
```

En la línea 1, declaramos que la clase DVDInfo implementa Comparable de tal manera que los objetos de tipo DVDInfo pueden ser comparados con otros objetos de su mismo tipo. En la línea 2, implementamos compareTo() para comparar los títulos de los objetos DVDInfo. Dado que sabemos que los títulos son de tipo Strings y los String implementan Comparable, esta será una manera sencilla de ordenar objetos de tipo DVDInfo por título. Antes de la versión 5 de Java, debíamos implementar Comparable con un código similar al siguiente:

```
class DVDInfo implements Comparable {
  // existing code
  public int compareTo(Object o) { // takes an Object rather
                                   // than a specific type
                                    DVDInfo d --- (DVDInfo)o;
 return title.compareTo(d.getTitle());
            } }
```

Este código es válido, pero puede ser riesgoso dado que debemos de hacer un casteo y debemos de asegurarnos de que este casteo no fallará antes de que lo utilicemos.

**Importante:**

Cuando sobreescribimos el método equals(), debemos recibir un argumento de tipo Object, pero cuando sobreescribmos compareTo() tomaremos un argumento del tipo que estemos ordenando.

## Ordenando con Comparator

La interface Comparator nos proporciona la capacidad de ordenar un collection de muchas formas diferentes. Podemos, también, usar esta interface para ordenar instancias de cualquier clase, incluso clases que no podamos modificar, a diferencia de la interface Comparable, la cual nos obliga a modificar la clase de cuyas instancias queremos ordenar. La interface Comparator es también muy fácil de implementar, sólo tiene un método: compare(). A continuación, se muestra un ejemplo de aplicación:

```
import java.util.*;
class GenreSort implements Comparator<DVDInfo> {
public int compare(DVDInfo one, DVDInfo two) {
return one.getGenre().compareTo(two.getGenre());
  }
}
```

El método Comparator.compare() retorna un valor entero cuyo significado es el mismo que el valor retornado por Comparable.compareTo(). A continuación, se muestra un ejemplo de aplicación de ambos métodos:

```
import java.util.*;
import java.io.*;                    // populateList() needs this public class
TestDVD {
  ArrayList<DVDInfo> dvdlist = new ArrayList<DVDInfo>();
  public static void main(String[] args) {
    new TestDVD().go();
  }
  public void go() { populateList();
    System.out.println(dvdlist);              // output as read from file
    Collections.sort(dvdlist);
    System.out.println(dvdlist);              // output sorted by title

    GenreSort gs = new GenreSort(); Collections.sort(dvdlist, gs) ;
    System.out.println(dvdlist);              // output sorted by genre
  }

  public void populateList() {
     // read the file, create DVDInfo instances, and
     // populate the ArrayList dvdlist with these instances
  }
}
```

Se muestra, a continuación, una tabla con la comparación de ambas interfaces:

| java.lang.Comparable | java.util.Comparator |
|---|---|
| int objOne.compareTo(objTwo) | int compare(objone, objTwo) |
| Retorna:<br>Negativo si obj One < objTwo<br>Cero si objOne == objTwo<br>Positivo si objOne > objTwo | El mismq que Comparable |
| Debemos modificar la clase cuyas instancias queremos ordenar. | Debemos construir una clase independiente de la clase que queremos ordenar. |
| Solo podemos crear una secuencia de ordenamiento. | Podemos crear muchas secuencias de ordenamiento |
| Implementada frecuentemente por la API java: String, Wrapper classes, Date, Calendar… | Creada para ser implementada por instancias de Terceros. |

## Ordenando con la clase Arrays

Ordenar arreglos de objetos es igual que ordenar collections de objetos. El método Arrays.sort() es sobreescrito de la misma manera que el método Collections.sort().

**Arrays.sort(arrayToSort) Arrays.sort(arrayToSort, Comparator)**

Recordemos que los métodos sort() de las clases Collections y Arrays son métodos estáticos, y ambos trabajan directamente sobre los objetos que están ordenando, en vez de retornar un objeto diferente ordenado.

## Convirtiendo arreglos a listas y listas a arreglos

Existen dos métodos que permiten convertir arreglos en listas y listas en arreglos. Las clases List y Set tienen los métodos toArray(), y las clases Arrays, tienen un método llamado asList().

El método Arrays.asList() copia un arreglo en un objeto tipo List:

```
String [] sa = {"one", "two", "three", "four"};
List sList = Arrays.asList(sa);                    // make a List
System.out.println("size " + sList.size());
System.out.println("idx2 " + sList.get(2));

sList.set(3,"six");                                // change List
sa[l] = "five";                                    // change array for(String s : sa)
  System.out.print(s + " ");
```

```
System.out.println("\nsl[1] " +
sList.get(1));
```

Obtendremos la siguiente salida:

```
size 4
idx2 three
one five
three six
sl[1] five
```

A continuación veamos el uso del método toArray(). Existen dos versiones de este método. La primera, retorna un nuevo objeto de tipo Array. La segunda, usa como destino un arreglo pre existente.

```
List<Integer> iL = new
ArrayList<Integer>(); for(int x=0; x<3;
x++)
  iL.add(x);
Object[] oa = iL.toArray();                    //
create an Object array Integer[] ia2 = new
Integer[3];
ia2 = iL.toArray(ia2);                         // create an Integer array
```

### 6.1.4. Set y tipos

La interface Set no permite objetos duplicados. El método equals () determina si dos objetos son idénticos (en cuyo caso sólo uno puede ser parte del conjunto).

**HashSet.** Es un conjunto sin clasificar y sin ordenar. Se utiliza el valor hashCode del objeto que se inserta como mecanismo de acceo. Esta clase es ideal cuando requerimos un Collection que no tenga objetos duplicados y no nos importe como iteraremos a través de ellos.

**LinkedHashSet**. Versión mejorada de HashSet que mantiene una lista doble través de todos sus elementos. Es conveniente utilizar esta clase en vez de HashSet cuando importa el orden en la iteración: iteraremos a través de los elementos en el orden en que fueron insertados.

**Importante:**

Cuando utilicemos objetos HashSet, o LinkedHashSet, los objetos que agreguemos a estos collections deben sobreescribir el método hashCode (). Si no lo hacemos, el método hashCode() que existe por defecto permitirá agregar múltiples objetos que podríamos considerar iguales a nuestra conjunto de objetos “únicos”.

**TreeSet.** Un TreeSet es uno de los dos collections ordenados con que se cuentan dentro

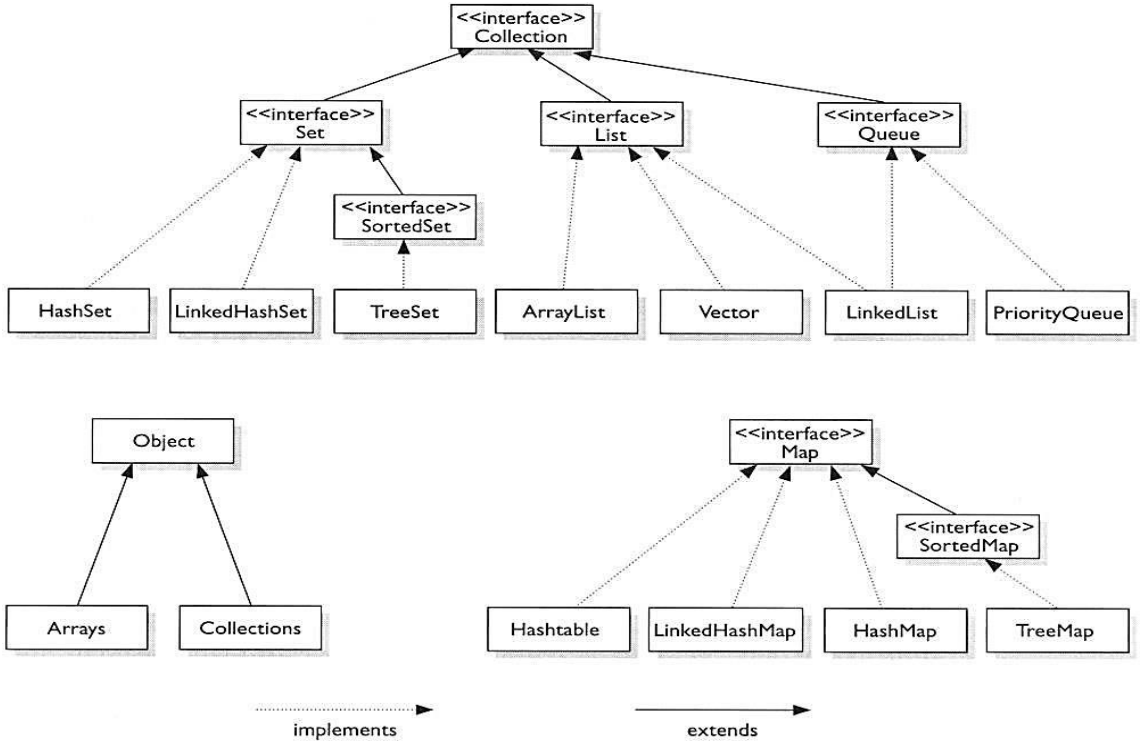
del framework (el otro es TreeMap). Esta estructura garantiza que los elementos se encontrarán en orden ascendente de acuerdo con el orden natural. Opcionalmente, podremos construir un TreeSet utilizando un constructor que le indique una regla de ordenamiento específica.

### 6.1.5. Map y tipos.

Un objeto Map permite solo identificadres únicos. Debemos asociar una key única con un valor específico, donde ambos, la key y el valor, son objetos. Las implementaciones de la interface Map nos permitirán hacer operaciones tales como buscar valores basados en una key, preguntar por sólo los valores de un collection, o sólo por sus keys. De la misma manera que los conjuntos, un Map utiliza el método equals() para determinar si dos keys son las mismas o diferentes.

# Resumen

1. Clases de las colecciones




- http://stackoverflow.com/questions/3317381/what-is-the-difference-between-collection-and-list-in-java
  Colecciones en java

- http://docs.oracle.com/javase/tutorial/collections/index.html
  Collections

# Anexo

# GENÉRICOS Y COLECCIONES

## LOGRO DE LA UNIDAD DE APRENDIZAJE

Al término de la unidad, el alumno crea aplicaciones que utilicen eficientemente las clases e interfaces del Framewok Collections tales como Set, List y Map, manipulando arreglos para ordenarlos o implementar búsquedas binarias basadas en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

## TEMARIO

**6.2 Tema 14 : Genéricos**

- 6.2.1 : Polimorfismo y tipos genéricos
- 6.2.2 Métodos genéricos
- 6.2.3 Declaraciones genéricas

## ACTIVIDADES PROPUESTAS

- Crear aplicaciones usando genericos

# 6.2. GENÉRICOS

## 6.2.1. Polimorfismo y tipos genéricos

Las colecciones genéricas nos proporcionan los mismos beneficios de seguridad en los tipos que los arreglos, aunque existen algunas diferencias fundamentales que están relacionadas con el uso de polimorfismo en Java.

El Polimorfismo se aplica al tipo base del collection:

```
List<Integer> myList = new ArrayList<Integer>();
```

En otras palabras, es válido asignar un ArrayList a un objeto List porque List es clase padre de ArrayList.

Veamos ahora el siguiente ejemplo: class Parent { }

```
class Child extends Parent { }
List<Parent> myList = new ArrayList<Child>();
```

¿Cuál será el resultado?

La respuesta es No funcionará. Existe una regla básica para el ejemplo anterior: El tipo de una declaración de variable debe corresponder al tipo que que pasamos en el objeto actual. Si declaramos List<Foo> foo, entonces lo que asignemos a la variable foo debe ser un tipo genérico<Foo>. No un subtipo de <Foo>. Tampoco un supertipo de <Foo>. Sólo <Foo>.

**Esto sería un error:**

```
List<object> myList = new ArrayList<JButton>(); // NO!
List<Number> numbers = new ArrayList<Integer>(); // NO!
// remember that Integer is a subtype of Number
```

**Esto sería correcto:**

```
List<JButton> myList = new ArrayList<JButton>(); // yes
List<Object> myList = new ArrayList<Object>(); // yes
List<Integer> myList = new ArrayList<Integer>(); // yes
```

## 6.2.2. Métodos genéricos

Uno de los mayors beneficios del polimorfismo es que podemos declarar el parámetro de un método como de un tipo en particular y, en tiempo de ejecución, hacer que este parámetro referencie a cualquiera de sus subtipos.

Por ejemplo, imaginemos un ejemplo clásico de polimorfismo: La clase AnimalDoctor con el método checkup(). Tenemos, también, tres subtipos de la clase Animal: Dog, Cat, y Bird. A continuación, se muestran las implementaciones del método abstracto checkup() de la clase Animal:

```
abstract class Animal {
    public abstract void checkup();
}
class Dog extends Animal {
    public void checkup() { // implement Dog-specific code
      System.out.println("Dog checkup");
    }
}
class Cat extends Animal {
    public void checkup() { // implement Cat-
      specific code System.out.println("Cat
      checkup");
    }
}
class Bird extends Animal {
    public void checkup() { // implement Bird-specific code
      System.out.println("Bird checkup");
} }
```

Si requerimos que la clase AnimalDoctor invoque al método checkup() adecuado correspondiente al animal específico que se está atendiendo, podríamos tener el siguiente código:

```
public void checkAnimal(Animal a) {
    a.checkup(); // no importará el subtipo de animal que se reciba
                 // siempre invocaremos al método checkup() adecuado.
}
```

Si ahora requerimos atender a un conjunto variado de animals diversos, los cuales son recibidos como un arreglo de objetos de la clase Animal[], podríamos tener un código similar al siguiente:

```
public void checkAnimals(Animal[] animals) {
    for(Animal a :
      animals) {
      a.checkup();
    }
}
```

A continuación, se muestra el ejemplo completo en el que probamos el uso de polimorfismo a través de arreglos de subtipos de la clase Animal:

```
import java.util.*;
abstract class Animal {
    public abstract void checkup();
}
class Dog extends Animal {
   public void checkup() { // implement Dog-specific
     code System.out.println("Dog checkup");
   }
}
```

```
class Cat extends Animal {
      public void checkup() {                    //
        implement Cat-specific code
        System.out.println("Cat checkup");
      }
   }

class Bird extends Animal {
  public void checkup() {                      // implement
    Bird-specific code System.out.println("Bird
    checkup");
  }
}
public class AnimalDoctor  {
   // method takes an array of any animal
   subtype public void checkAnimals(Animal[]
   animals) { for(Animal a : animals) {
      a.checkup();
    }
   }
public static void main(String[] args)                    {
   // test it
  Dog[] dogs = (new Dog(), new Dog()};
  Cat[] cats = (new Cat(), new Cat(), new
  Cat()); Bird[] birds = (new Bird());

  AnimalDoctor doc = new AnimalDoctor();
  doc.checkAnimals(dogs); // pass the Dog[]
  doc.checkAnimals(cats); // pass the Cat[]
  doc.checkAnimals(birds); // pass the Bird[]
 }
}
```

El ejemplo anterior trabaja correctamente; sin embargo, debemos tener presente que este enfoque no es válido si trabajamos con collections bajo el esquema de manera segura (type safe collections).

Si un método, por ejemplo, recibe un collection de tipo ArrayList<Animal>, no podrá aceptar una colección constituída por cualquier subtipo de la clase Animal. Eso significa que, por ejemplo, no podremos recibir un Collection de tipo ArrayList<Dog> en un método que soporta argumentos de tipo ArrayList<Animal>.

Como podemos observar, el ejemplo anterior, evidencia las diferencias entre el uso de arreglos de un tipo específico versus collections de un tipo específico.

Anque sabemos que no se ejcutará correctamente, haremos los siguientes cambios a la clase AnimalDoctor para que utilice objetos genéricos en vez de arreglos:

```
public class AnimalDoctorGeneric  {

   // change the argument from Animal[] to
   ArrayList<Animal> public void
   checkAnimals(ArrayList<Animal> animals) { for(Animal
   a : animals) {
      a.checkup();
      }
                  }

public static void main(String[] args) {
   // make ArrayLists instead of arrays for Dog, Cat, Bird
   List<Dog> dogs = new ArrayList<Dog>();
   dogs.add(new Dog());
   dogs.add(new Dog());
   List<Cat> cats = new ArrayList<Cat>();
   cats.add(new Cat());
   cats.add(new Cat());
   List<Bird> birds = new ArrayList<Bird>();
   birds.add(new Bird());

   // this code is the same as the Array version
   AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
   // this worked when we used arrays instead of ArrayLists
   doc.checkAnimals(dogs); // send a List<Dog>
   doc.checkAnimals(cats); // send a List<Cat>
   doc.checkAnimals(birds); // send a List<Bird>
  }
}
```

Si ejecutamos la siguiente línea:  javac
AnimalDoctorGeneric.Java

**Obtendremos:**

```
AnimalDoctorGeneric.Java:51: checkAnimals(Java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
```

```
(java.util.List<Dog>)
      doc.checkAnimals(dogs);
        ^
AnimalDoctorGeneric.Java: 52: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Cat>)
      doc.checkAnimals(cats);
        ^
AnimalDoctorGeneric.Java:53: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Bird>)
      doc.checkAnimals(birds);
        ^
        3 errors
```

El compilador generó tres errores debido a que no podemos asignar ArrayLists de subtipos de Animal (<Dog>, <Cat>, or <Bird>) a un ArrayList del supertipo <Animal>.

Ante esta situación ¿Cuál podría ser una salida válida utilizando objetos genéricos?

**Podríamos hacer lo siguiente:**

```
List<Animal> animals = new ArrayList<Animal>();
animals.add(new Cat()); // OK
animals.add(new Dog()); // OK
```

El código anterior es válido tanto para arreglos como para collections genéricos, podemos agregar una instancia de un subtipo en un arreglo o colección declarados con un supertipo.

Por lo tanto, y bajo ciertas circunstancias, el código mostrado a continuación sería válido:

```
public void addAnimal(ArrayList<Animal> animals) {
animals.add(new Dog()); // sometimes allowed...
}
```

El código mostrado en líneas anteriores, compilará satisfactoriamente, siempre y cuando el parámetro que que le pasemos al método sea de tipo ArrayList<Animal>.

**importante:**

El único objeto que podremos pasar a un método con un argumento de tipo ArrayList<Animal> es un un objeto de tipo ArrayList<Animal>!

Existe en Java, también, un mecanismo para indicar al compilador que aceptaremos

cualquier subtipo genérico del argumento declarado en un método. Es importante estar seguros que no agregaremos cualquier elemento (no válido) a el collection, el mecanismo es el uso de <?>.

La firma del método cambiará de:

**public void addAnimal(List<Animal> animals)**

por:

**public void addAnimal(List<? extends Animal> animals)**

El decir **<? extends Animal>**, nos permite indicar al compilador que podremos asignar un collection que sea un subtipo de List y de tipo <Animal> o cualquier otro tipo que herede de Animal.

Sin embargo, el método addAnimal() no funcionará debido a que el método agrega un elemento al collection.

```
public void addAnimal(List<? extends Animal> animals) {
    animals.add(new Dog()); // NO! Can't add if we
                            // use <? extends Animal>
}
```

El error que obtendríamos sería el siguiente: javac AnimalDoctorGeneric.java

```
AnimalDoctorGeneric.java:38: cannot find symbol
symbol : method add(Dog)
location: interface java.util.List<capture of ? extends
    Animal> animals.add(new Dog());
        ^
1 error
```

Si cambiamos el método para que no agregue un elemento, funcionará correctamente el código anterior.

### 6.2.4. Declaraciones genéricas

Veamos el siguiente ejemplo:

public interface **List<E>**

El identificador <E> representa, de manera “general”, el tipo que pasaremos a la lista. La interface List se comporta como una plantilla genérica, de modo que cuando creemos nuestro código, podamos hacer declaraciones del tipo: List<Dog> or List<Integer>, and

so on.

El uso de **"E",** es solo una convención. Cualquier identificador válido en Java funcionará adecuadamente. Se escogió E porque se puede interpretar como "Elemento". La otra convención básica es el uso de **T** (que signfica **Type**), utilizada para tipos que no son collections.

**Ejercicios de aplicación:**

1. Dada la siguiente clase:

```
import java.util.*;
class Test {
  public static void main(String[] args) {
    // insert code here
    x.add("one");
    x.add("two");
    x. add ("TWO");
    System.out.printIn(x.poll());
  }
}
```

¿Cuáles de las siguientes líneas de código, al ser insertadas en lugar de:

// insert code here, compilarán?

a) List<String> x = new LinkedList<String>();

b) TreeSet<String> x = new TreeSet<String>();

c) HashSet<String> x = new HashSet<String>();

d) Queue<String> x = new PriorityQueue<String>();

e) ArrayList<String> x = new ArrayList<String>();

f) LinkedList<String> x = new LinkedList<String>();

2. Dadas las siguientes líneas de código:

```
10.  public static void main(String[] args) {
11.      Queue<String> q = new LinkedList<String>();
12.      q.add("Veronica");
13.      q.add("Wallace");
14.      q.add("Duncan");
15.      showAll(q);
16.  }
17.
18.  public static void showAll(Queue q) {
19.      q.add(new Integer(42));
20.      while (!q.isEmpty ( ) )
21.          System.out.print(q.remove( ) + "  ");
22.  }
```

¿Cuál será el resultado obtenido?

a) Veronica Wallace Duncan

b) Veronica Wallace Duncan 42

c) Duncan Wallace Veronica

d) 42 Duncan Wallace Veronica

e) Compilation fails.

f) An exception occurs at runtime.

3. Dadas las siguientes clases:

```
import java.util..*;
class MapEQ {
  public static void main(String[] args) {
    Map<ToDos, String> m = new HashMap<ToDos, String>();
    ToDos tl = new ToDos("Monday");
    ToDos t2 = new ToDos("Monday");
    ToDos t3 = new ToDos("Tuesday");
    m.put(tl, "doLaundry");
    m.put(t2, "payBills");
    m.put(t3, "cleanAttic");
    System.out.printIn(m.size());
  }
}
class ToDos{
  String day;
  ToDos(String d) { day = d; }
  public boolean equals(Object o) {
    return ((ToDos)o).day == this.day;
  }
  // public int hashCode() ( return 9; }
}
```

¿Cuáles de las afirmaciones mostradas a continuación son válidas?

a) As the code stands it will not compile.

b) As the code stands the output will be 2.

c) As the code stands the output will be 3.

d) If the hashCode() method is uncommented the output will be 2.

e) If the hashCode() method is uncommented the output will be 3.

f) If the hashCode() method is uncommented the code will not compile

# Resumen

1 Dentro del contexto de las colecciones, las asignaciones que usan polimorfismo son válidas para el tipo base, no para el tipo específico del Collection. Será válido decir:

List<Animal> aList = new ArrayList<Animal>(); // yes

Será inválido:

List<Animal> aList = new ArrayList<Dog>(); // no

2 El uso de caracteres comodín dentro de un método genérico, nos permite aceptar subtipos (o supertipos) del tipo declarado en el argumento del método:

void addD(List<Dog> d) {} // can take only <Dog>
void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>

3 La palabra clave extends es usada para significar "extends" o "implements". Por lo tanto, en el ejemplo <? extends Dog>, Dog puede ser una clase o una interface.

4 Cuando usamos una definición del tipo: List<? extends Dog>, la colección puede ser accesada pero no modificada por la aplicación.

5 Las convenciones para la declaración de tipos genéricos sugieren utilizar T para tipos y E para elementos:

public interface List<E> // API declaration
for List boolean add(E o) // List.add() declaration

Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

- http://javabasico.osmosislatina.com/curso/polimorfismo.htm

  Aquí hallará un interesante artículo sobre el uso de polimorfismo y arreglos en java.

- http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

  En esta página, hallará información oficial y detallada sobre el uso de genéricos en java.