JTABLE, TABLEMODEL Y RENDIMIENTO

Francesc Rosés / lulio 2004
v. 1.1 (23/09/04)

Con frecuencia, nos vemos obligados a visualizar una gran cantidad de datos mediante una `JTable`. El resultado es una carga lenta, un *scroll* penoso y un consumo de RAM intolerable.

Ciertamente, hay alternativas. Uno siempre intenta minimizar los datos que carga e incluso se inventa algún tipo de caché que necesita un montón de pruebas hasta que se da por aceptable.

La mayor parte de los datos en las cargas masivas suelen tener su origen en una base de datos a la que accedemos mediante un driver JDBC. En este artículo hago una propuesta de un uso racional tanto del `TableModel` como de las posibilidades que nos ofrece JDBC 2.0 (y posteriores) para reducir a prácticamente nada el coste de la representación de grandes volúmenes de datos mediante `JTables`.

¿QUÉ HACEMOS MAL?

Antes de exponer mi propuesta, creo que es conveniente analizar algunas de las acciones más frecuentes que nos llevan a la ineficiencia.

Usar lo que ya está hecho

Bien, usar lo que ya está hecho no es malo, si lo que está hecho funciona como es debido. Este no es el caso, por ejemplo, de una de las subclases de `JTableModel`: `DefaultTableModel`. Los más comodones nos limitamos a usarla directamente, y los que lo son menos, se permiten subclasearla. Sin embargo, `DefaultTableModel` es el final de una cadena de dos elementos: la interficie `JTableModel` y la clase abstracta que la implementa: `AbstractTableModel`. La clase `DefaultTableModel` no es más que un ejemplo práctico de extensión de `AbstractTableModel`.

No leer con detalle las APIs que tocan

En realidad, nuestra primera aproximación se basaba en un “copy/paste” del código de un compañero que en cinco minutos, mientras tomábamos un café, nos contó que eso de las tablas y el JDBC era muy sencillito, pero que el *Swing*, ya se sabe, es más lento que el caballo del malo.

En definitiva, sin ver un *Javadoc* ni por el forro, nos hemos lanzado a escribir nuestra primera versión de la aplicación.

Invito al lector a hacer una prueba. Pregunta a tus compañeros (y a ti mismo, claro está) si tienen una *bookmark* de las APIs de Java en su explorador. Verás qué poquitos la tienen.

¿POR QUÉ ESTÁ MAL LO QUE HACEMOS?

Usar lo que ya está hecho: DefaultTableModel

La clase `DefaultTableModel` extiende `AbstractTableModel` y, si bien para algunos casos simples y con poco volumen de datos puede ser útil, tiene varios problemas. El primero es el uso

exhaustivo de la clase `Vector`. No entraremos en detalles, sólo decir que `Vector` tiene todos sus métodos sincronizados lo que es bastante caro y, en la mayor parte de los casos, innecesario. Sería más conveniente usar una `ArrayList` para el almacenamiento de datos. Es equivalente a `Vector`, pero no tiene sus métodos sincronizados.

Para aquellos casos simples en los que puede ser interesante usar un modelo con las funcionalidades de `DefaultTableModel`, he desarrollado un nuevo modelo con la misma funcionalidad que `DefaultTableModel` pero basado en una `ArrayList`: `ArrayListTableModel`.

Si bien `ArrayListTableModel` da un mayor rendimiento que `DefaultTableModel` (un 24% aproximadamente), no es la panacea. Las pruebas se han realizado con una tabla de 68.719 registros que ocupa, aproximadamente, 3,16 MB. Esto significa que, en ambos casos, hemos de tener cargados en RAM una `List` de, al menos, 3,16 MB.

Bueno, hemos conseguido mejorar el rendimiento, pero no los requisitos de memoria.

No leer con detalle las APIs que tocan

Sé que no se puede leer todo con detalle, pero es conveniente hacer una lectura secuencial de los *Javadoc* de las APIs involucradas en nuestro problema y mantener apuntadores a lo que nos ha parecido interesante (aunque no le hayamos visto una aplicación inmediata).

¿Qué APIs están involucradas en nuestro problema? Básicamente, nueve:

- `javax.swing.JTable`
- `javax.swing.table.JTableModel`
- `javax.swing.table.AbtractTableModel`
- `javax.swing.table.TableCellRenderer`
- `java.sql.Connection`
- `java.sql.DatabaseMetaData`
- `java.sql.Statement`
- `java.sql.ResultSet`
- `java.sql.ResultSetMetaData`

Todas aparecen en la solución propuesta y comentaremos con más detalle, en su momento, los aspectos que nos interesan de cada una de ellas.

SOLUCIÓN CON ArrayList

Se recomienda utilizar `ArrayList` en vez de `Vector` siempre que podamos. Hay que tener en cuenta que `Vector` existe desde el JDK 1.0, mientras que `ArrayList` aparece en JDK 1.2. Algo debe aportar...

He comentado más arriba, que el factor determinante es que `ArrayList` no tiene métodos `synchronized`. Esto implica que si otro *thread* modifica nuestra `ArrayList`, hay que hacer la sincronización a mano. Si consideramos que este hecho no es importante para nuestra aplicación, y creo que en el 90% de los casos de modelos de `JTable` no lo es, podemos usar un modelo basado en `ArrayList`s en nuestro modelo, Si lo es, podemos seguir usando el consabido `DefaultTableModel`.

El problema es que Sun no proporciona ningún modelo basado en `ArrayList` y nos tocará escribirlo desde cero. Yo he escrito un nuevo modelo basado en `ArrayList`, `ArrayListTableModel`, que se supone que debe funcionar exactamente igual que `DefaultTableModel`. Está a disposición del lector.

Mis pruebas de rendimiento, como he comentado más arriba, me dan un 24% de mejora de rendimiento respecto a `DefaultTableModel`. Hay que tener presente, sin embargo, que este modelo no nos libera de necesidad de tener todos los registros en RAM.

Como es lógico, `ArrayListTableModel` extiende `AbstractTableModel`. Si el lector se encuentra en la necesidad que escribir su propio modelo, debe extender siempre `AbstractTableModel`, no `DefaultTableModel`.

SOLUCIÓN CON JDBC Y SCROLLABLE CURSORS

Ésta es, para mí, la solución óptima ya que cumple con los dos requisitos fundamentales:

1. Mejora el rendimiento
2. Utiliza poca memoria

La solución que propongo es sencilla de implementar y, además, muy eficiente. Podría limitarme a explicar someramente su implementación y dar el código, pero como no creo que sea una solución definitiva, y es posible que a más de uno se le ocurra algo mejor, escribiré un poco más e intentaré explicar cómo usa `JTable` los `TableModel`.

Es evidente que la solución propuesta no es la panacea. Como podrá observar el lector, el trabajo lo hace el gestor de bases de datos, lo que supone una conexión abierta con la base de datos mientras se esté ejecutando nuestra aplicación y una carga para el servidor.

El paradigma MVC (Model View Controller)

Supongo, aquí, que el lector conoce, más o menos, cómo funciona este paradigma, así que no entraré en detalles introductorios y me centraré en lo que nos interesa, simplificando lo que sea necesario en aras de una mayor claridad.

Swing no utiliza exactamente el MVC, sino una variante: MD (*Model Delegate*) en la que el *Delegate* incluye *View* y *Controller*. Además, `JTable` es un componente que está formado por varios subcomponentes y cada uno de ellos con su modelo y su *delegate*.

JTable y MVC: un ejemplo sencillo

Simplificando mucho, diremos que una `JTable` muestra unos datos que se encuentran en el modelo. Una `JTable` tiene unas dimensiones determinadas en las que cabe un número concreto de registros mostrables. `JTable` lo sabe y le pide al modelo sólo aquellos registros que puede mostrar.

En la mayor parte de los casos, nuestras `JTables` se encontrarán dentro de un `JScrollPane` ya que no disponemos de espacio suficiente para mostrar todos los registros.

Así, cada vez que movemos la barra de desplazamiento vertical del `JScrollPane`, nuestra `JTable` le pide más datos al modelo para poderlos mostrar. La solución es óptima ya que sólo le pide unos pocos datos: aquéllos que puede representar.

El método que utiliza para pedir datos al modelo es `getValueAt(int row, int column)`.

Imaginemos que tenemos una `JTable` con dos columnas; la primera muestra los números de `1` a `100` y la segunda, su doble (el valor de la primera columna, multiplicado por dos).

Una utilización no recomendable de nuestro modelo mantendría una matriz con los valores de los 100 primeros números y su doble:

```
int[][] data = new int[100][2];
[...]
private void fillData() {
  for (int i = 1; i <= 100; i++) {
    data[i-1][0] = i;
    data[i-1][1] = i * 2;
  }
}
```

En nuestro modelo, declaramos una matriz de enteros de `100x2` y la llenamos mediante el método `fillData()`.

Para pedir datos a nuestro modelo, nuestra `JTable` usa los siguientes métodos del modelo:

```
public int getColumnCount() {
  return data[0].length;
}

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

El primero le sirve para acotar el número de columnas. Si tiene 2 columnas, no preguntará por la onceava...

El segundo le sirve para acotar el número de registros disponibles.

Finalmente, una vez tiene claro cuántas columnas y cuántos registros tiene el modelo de datos, pregunta por el valor concreto de una celda usando el método `Object getValueAt(int rowIndex, int columnIndex)`.

En nuestro caso, lo podríamos implementar fácilmente así:

```
public Object getValueAt(int rowIndex, int columnIndex) {
  return new Integer(data[rowIndex-1][columnIndex]);
}
```

Observemos que ésta es una implementación sumamente ineficiente. Hay que llenar una matriz y después consultarla. Esto puede no ser caro para los 100 primeros números, pero si queremos trabajar, por ejemplo, con los 500.000 primeros números, nuestra implementación empezaría a renquear.

Veamos una implementación más eficiente:

```
public int getColumnCount() {
  return 2;
}

public int getRowCount() {
  return 500000;
}

public Object getValueAt(int rowIndex, int columnIndex) {
  if (columnIndex == 0) {
    return new Integer(rowIndex);
  }
  return new Integer((rowIndex*2));
```

```
}
```

En esta segunda implementación optimizamos al máximo los recursos de memoria ya que no almacenamos ningún valor. Los valores se calculan en tiempo de ejecución.

JTable y MVC: un mal ejemplo con JDBC

Si bien el ejemplo anterior es bastante ilustrativo y nos puede venir bien para implementar algunos de nuestros modelos, no ejemplifica el caso más típico: los datos a representar provienen de una consulta a la base de datos.

Llegados a este punto, optamos por una solución típicamente ineficiente como ésta:

```
Statement stmt = null;
ResultSet rs   = null;
Vector rows = new Vector();
String select = “SELECT NOMBRE, APELLIDO1, APELLIDO2 FROM PERSONAS”;
try {
  stmt = connection.createStatement();
  rs = stmt.executeQuery(select);
  while (rs.next()) {
    String nombre    = rs.getString(1);
    String apellido1 = rs.getString(2);
    String apellido2 = rs.getString(3);

    Vector row = new Vector();
    row.addElement(nombre);
    row.addElement(apellido1);
    row.addElement(apellido2);
    rows.addElement(row);
  }
} catch (SQLException e) {
    e.printStackTrace();
  }
  finally {
    if (stmt != null) {
      try {
        stmt.close();
      } catch (SQLException e) {
        }
    }
  }

Vector colNames = new Vector(3);
colNames.addElement(“Nombre”);
colNames.addElement(“Primer Apellido”);
colNames.addElement(“Segundo Apellido”);

DefaultTableModel model = new DefaultTableModel(rows, colNames);
miTabla.setModel(model);
```

Como la lista de personas sea la de la guía telefónica de España, no hay `JTable` que lo soporte.

Éste suele ser el problema al que nos enfrentamos. Cargar toda la tabla en memoria es realmente costoso.

Análisis del problema

Fijémonos en el primer ejemplo. El modelo, en realidad, no contiene datos. Los calcula. Es eficiente porque no tiene un lento proceso de carga y porque prácticamente no usa memoria. Simplemente, proporciona a la `JTable` lo que ésta le pide.

En nuestro ejemplo JDBC, el modelo contiene todos los datos. Hay que cargarlos y almacenarlos en memoria. No es, pues, eficiente si los datos son muchos.

Observemos, sin embargo, que los datos ya están en la base de datos. Si esto es así, ¿por qué no "calculamos" los datos que nos pide la `JTable` pidiéndoselos a nuestra base de datos y simplemente le retornamos lo que necesita?

Un modelo proporciona datos al *delegate* siguiendo un protocolo concreto, no es necesario que él tenga los datos. Basta con que los suministre (calculándolos u obteniéndolos de un tercero bajo demanda).

UNA POSIBLE SOLUCIÓN: ScrollableTableModel

Hasta la aparición de JDBC 2.0, no podíamos hacer otra cosa que cargar los datos resultantes de nuestra consulta en memoria. El `ResultSet` sólo podía moverse en una dirección: hacia adelante. No podía, pues, dar respuesta a los requerimientos de una tabla capaz de moverse hacia adelante y hacia atrás. Pero a partir de JDBC 2.0, disponemos de `ResultSet`s que pueden moverse libremente por el conjunto de resultados de una consulta. El modelo, pues, puede limitarse a "calcular" los datos que debe devolver y extraerlos del `ResultSet`.

Determinar si nuestro driver JDBC soporta scrollable cursors

Lamentablemente, no todos los drivers JDBC soportan scrollable cursors. Para determinar si nuestro driver los soporta, hemos de consultar la *metadata* de la base de datos:

```
private boolean supportsScrollInsensitive(Connection con) {
  DatabaseMetaData md = null;
  try {
    md = con.getMetaData();
  } catch (SQLException e) {
      String errMsg = "Error getting database metadata.";
      throw new ScrollableTableModelException(errMsg, e);
    }
  try {
    return md.supportsResultSetType(
                                  ResultSet.TYPE_SCROLL_INSENSITIVE);
  } catch (SQLException e) {
      String errMsg = "Error getting database metadata.";
      throw new ScrollableTableModelException(errMsg, e);
    }
} // supportsScrollInsensitive()
```

Uso de scrollable cursors

Como hemos visto, para que nuestro modelo funcione debemos disponer de *scrollable cursors*. Suponiendo que `suportsScrollInsensitive()` nos devuelva `true`, podemos definir un `Statement` que use *scrollable cursors*:

```
stmt = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                  ResultSet.CONCUR_READ_ONLY);
```

No entraré aquí en detalles sobre la sintaxis de este método, sólo diré que el primer parámetro establece el tipo de *scroll* y el segundo, el nivel de aislamiento del cursor.

Obtención y retorno de los datos

Una vez disponemos de nuestro `ResultSet` *scrollable*, hemos de utilizar sus posibilidades sobrescribiendo el método `getValueAt()`:

```
public Object getValueAt(int rowIndex, int columnIndex) {
  int rowNdx = rowIndex + 1;
  int colNdx = columnIndex + 1;
  try {
    resultSet.absolute(rowNdx);
    return resultSet.getObject(colNdx);
  } catch (SQLException e) {
      String errMsg = "Error getting value at " +
                      rowIndex + ", " + columnIndex;
      throw new ScrollableTableModelException(errMsg, e);
    }
}
```

Lo primero que hace nuestro método es traducir las coordenadas de `JTable` a coordenadas de JDBC. En JDBC se empieza a contar desde 1, y en el resto del mundo Java, se empieza a contar desde 0. Así, si `JTable` nos pide los datos del registro 0, columna 0, hemos de traducirlo a JDBC como registro 1, columna 1.

A continuación situamos el cursor en el registro JDBC solicitado:

```
resultSet.absolute(rowNdx);
```

Una vez situado el cursor, obtenemos los datos almacenados en la columna JDBC que se nos pide:

```
resultSet.getObject(colNdx);
```

Obsérvese que pedimos y devolvemos un `Object`.

Las clases de las columnas

`JTable` es capaz de mostrar nuestros datos en función de su tipo. Para ello utiliza clases que implementan la interfície `TableCellRenderer`. Pero, como es lógico, alguien tiene que decirle qué clase de objeto recibe. Si no se le dice, utiliza el método `toString()` del objeto de datos para mostrar su contenido.

En nuestro caso, devolvemos un `Object` y, si no le proporcionamos más datos, nos mostrará lo que devuelva el método `toString()` del objeto que devolvemos.

Si no hay un *renderer* para un tipo de objeto concreto, también utiliza el método `toString()` de dicho objeto.

Es conveniente, pues, indicarle qué clase de objeto le estamos devolviendo para cada columna. Esto se lleva a cabo implementando el método abstracto `getColumnClass()` de `AbstractTableModel`. Lo cierto es que, por ejemplo, `DefaultTableModel` no implementa este método. Un motivo más para utilizarlo con precaución.

La implementación para nuestro modelo podría ser la siguiente:

```
List colClasses = null;
[...]
public Class getColumnClass(int columnIndex) {
  if (colClasses == null) {
    colClasses = new ArrayList();
    ResultSetMetaData md = null;
    try {
      md = resultSet.getMetaData();
```

```
        int colCount = md.getColumnCount();
        for (int i = 0; i < colCount; i++) {
          try {
            String className = md.getColumnClassName(i + 1);
            Class c = Class.forName(className);
            colClasses.add(c);
          } catch (ClassNotFoundException e) {
              String errMsg = "Error getting column classes.";
              throw new ScrollableTableModelException(errMsg, e);
            }
        }   // for i
      } catch (SQLException e) {
          String errMsg = "Error getting column classes.";
          throw new ScrollableTableModelException(errMsg, e);
        }
    }

    Class c = (Class)colClasses.get(columnIndex);

    return c;
  }
```

Como puede verse, la información sobre el tipo del objeto retornado proviene de `ResultSetMetaData`. Para cada una de las columnas, preguntamos cuál es el nombre de la clase (`getColumnClassName(i + 1)`), una vez más traduciendo coordenadas de `JTable` a JDBC.

A partir del nombre de la clase, obtenemos la `Class`: `Class c = Class.forName (className);` que es lo que devolvemos a la JTable. Ella ya se encargará de ver cómo muestra los datos de esa clase.

Obtención de los nombres de las columnas

Como hemos visto, una de las cosas que `JTable` pide al modelo son los nombres de las columnas. Podemos obtener estos nombres directamente del `ResultSet` utilizando la clase `ResultSetMetaData`:

```
ArrayList colNames = null;
[...]

ResultSetMetaData rsmd = null;
try {
  rsmd = resultSet.getMetaData();
  int colCount = rsmd.getColumnCount();
  if (colCount == 0) {
    // TODO: No hay columnas!
  }
  this.colNames = new ArrayList();
  for (int i = 0; i < colCount; i++) {
    String colLabel = rsmd.getColumnLabel(i+1);
    this.colNames.add(colLabel);
   }
} catch (SQLException e) {
    e.printStackTrace();
    String errMsg = "Error getting ResultSetMetadata";
    throw new ScrollableTableModelException(errMsg, e);
  }
```

Obsérvese que utilizo el método `getColumnLabel()` para obtener el nombre de la columna, cuando pareciera más lógico usar el método `getColumnName()`. Lo cierto es que `getColumnLabel()` devuelve lo mismo que `getColumnName()`, excepto si especificamos

una etiqueta específica en nuestra consulta SQL:

```
SELECT N AS “Nombre”, A AS “Apellidos” FROM PERSONAS
```

En este ejemplo, `getColumnName()` devolvería `N` y `A`, mientras que `getColumnLabel()` devolvería `Nombre` y `Apellidos`.

Obviamente, nuestro modelo debe ofrecer la posibilidad de especificar una `ArrayList` con los nombres de las columnas.

Lógicamente, habrá que sobrescribir el método `getColumnName()`:

```
public String getColumnName(int column) {
  return (String)colNames.get(column);
}
```

¿Cuántas columnas?

Como hemos visto más arriba, una de las cosas que `JTable` necesita saber es el número de columnas del modelo. Para ello usa el método `getColumnCount()` que podemos implementar como sigue:

```
public int getColumnCount() {
  return colNames.size();
}
```

¿Cuántos registros?

Tal como hemos comentado, `JTable` necesita saber también cuántos registros tiene el modelo para poder acotar los parámetros pasados al método `getValueAt(int rowIndex, int columnIndex)`. Para ello utiliza el método `getRowCount()`, que podría ser implementado, en nuestro caso, de la siguiente manera:

```
int rowCount = -1;
[...]

public int getRowCount() {
  if (this.rowCount == -1) {
    try {
      resultSet.last();
      this.rowCount = resultSet.getRow();
    } catch (SQLException e) {
        String errMsg = "Error scrolling to latest row.";
        throw new ScrollableTableModelException(errMsg, e);
      }
}
```

Disponemos de la variable `rowCount` inicializada a `-1`, lo que nos indicará que todavía no hemos calculado su valor.

El el método `getRowCount()`, pasamos a calcular su valor siempre que éste valga `-1` (es decir, siempre que todavía no lo hayamos inicializado). Para ello, situamos el cursor en el último registro (`resultSet.last()`) y asignamos a `rowCount` el valor devuelto por el método `getRow()`; esto es, el número del último registro que, en notación JDBC equivale al número de registros.

Rendimientos

Entramos aquí en el apartado más práctico de todos: *¿qué gano en cada caso?*

Hemos comentado ya que el modelo `ArrayListTableModel` suponía una mejora de rendimiento del 24% respecto al modelo `DefaultTableModel`. Las pruebas realizadas sobre la misma base de datos con `ScrollableTableModel` me indican que la mejora de rendimiento respecto a `DefaultTableModel` es, aproximadamente, del **600%**.

LA CLASE ScrollableTableModel

Como he comentado más arriba, pongo a disposición pública la clase `ScrollableTableModel`. Si bien, hasta ahora he comentado algunos aspectos de la implementación, no he hablado todavía del uso. Vamos, pues, a ello.

Constructores

public ScrollableTableModel(Connection con, String select)

Parámetro	*Comentario*
con	Una conexión abierta con la base de datos
select	La instrucción `SELECT` necesaria para obtener los datos

Los nombres de las columnas se obtienen a partir del `ResultSetMetaData`, tal como he comentado más arriba.

ScrollableTableModel(Connection con, String select, List colNames)

Parámetro	*Comentario*
con	Una conexión abierta con la base de datos
select	La instrucción `SELECT` necesaria para obtener los datos
colNames	Una `java.util.List` (p.e. `ArrayList`) con los nombres de las columnas

El número de elementos de `colNames` deberá coincidir con el número de columnas devuelto por la consulta.

public ScrollableTableModel(ResultSet rs)

Parámetro	*Comentario*
rs	Un `ResultSet` configurado con `Scroll Intensive`.

Recordemos que para que el `ResultSet` soporte *scroll*, debemos especificarlo en el momento de creación del `Statement`. Por ejemplo:

```
stmt = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                  ResultSet.CONCUR_READ_ONLY);
```

Los nombres de las columnas se obtienen a partir del `ResultSetMetaData`, tal como he comentado más arriba.

public ScrollableTableModel(Statement stmt)

Parámetro	*Comentario*
stmt	Un `Statement` configurado con `Scroll Intensive` conteniendo un `ResultSet`.

Los nombres de las columnas se obtienen a partir del `ResultSetMetaData`, tal como he comentado más arriba.

public ScrollableTableModel(ResultSet rs, List colNames)

Parámetro	*Comentario*
rs	Un `ResultSet` configurado con `Scroll Intensive`.
select	La instrucción `SELECT` necesaria para obtener los datos
colNames	Una `java.util.List` (p.e. `ArrayList`) con los nombres de las columnas

public ScrollableTableModel(Statement stmt, List colNames)

Parámetro	*Comentario*
Stmt	Un `Statement` configurado con `Scroll Intensive` conteniendo un `ResultSet`.
select	La instrucción `SELECT` necesaria para obtener los datos
colNames	Una `java.util.List` (p.e. `ArrayList`) con los nombres de las columnas

Ejemplo de uso

El siguiente código muestra un sencillo ejemplo de uso de `ScrollableTableModel`, usando tan solo una conexión a base de datos y un select:

```
public class ScrollableTableModelTest {
  public static void main(String[] args) {
    Connection con = null;

    [... Carga del Driver JDBC y connexión a BD ...]

    JFrame f = new JFrame();
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JTable tabResults = new JTable();
    JscrollPane spTabResults = new JscrollPane(tabResults);
    f.getContentPane().add(spTabResults, BorderLayout.CENTER);

    String sel = “SELECT * FROM PERSONAL”;
    ScrollableTableModel model = new  ScrollableTableModel(con, sel);
    tabResults.setModel(model);

    f.pack();

    // Ventana centrada en la pantalla
    f.setLocationRelativeTo(null);
    f.setVisible(true);
  }   // main()
}
```