7 Saving data on disc

Most programs require data to be stored on disc, so that it can be reloaded when the program is re-run at a later date.

The simplest way in which Java can store data is as a text file. We will begin by creating a program to save a text file containing data for an employee in a company: the employee's name, department, age and salary.

Set up a new project in the standard way:

Close all projects, then set up a ***New Project***. Give this the name ***staffRecord***, and ensure that the ***Create Main Class*** option is not selected.

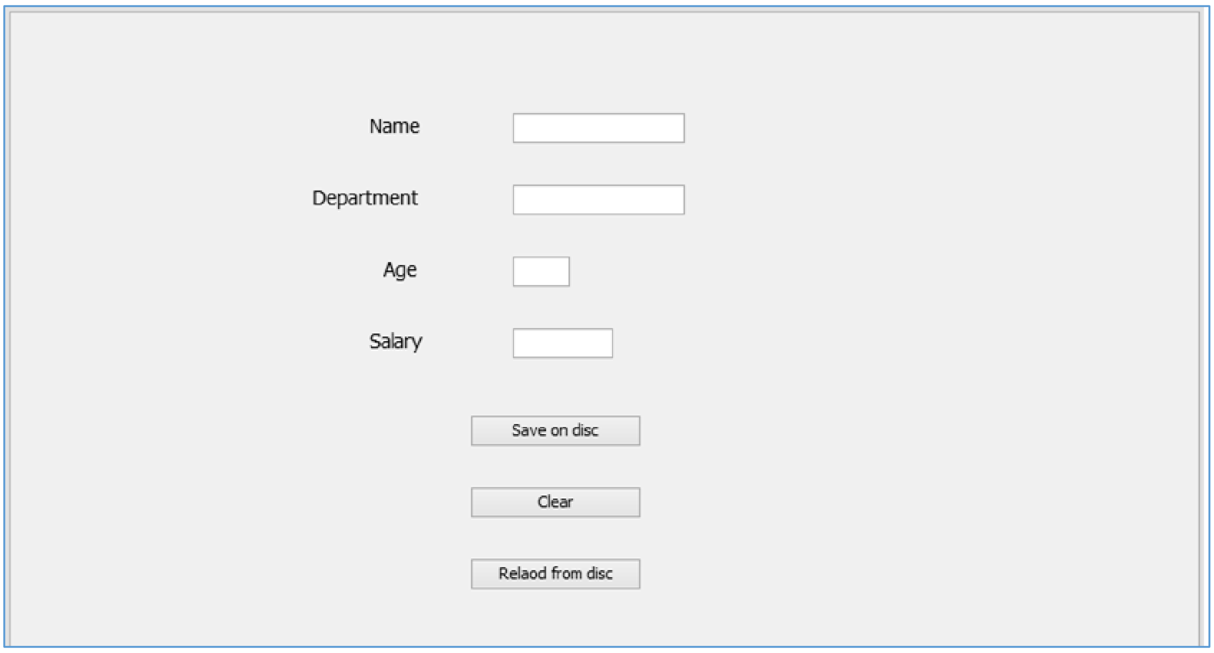
Return to the NetBeans editing page. Right-click on the ***staffRecord*** project, and select ***New / JFrame Form***. Give the ***Class Name*** as ***staffRecord***, and the ***Package*** as ***staffRecordPackage***:

Return to the NetBeans editing screen.

- Right-click on the ***form***, and select ***Set layout*** / ***Absolute layout***.
- Go to the ***Properties*** window on the bottom right of the screen and click the ***Code*** tab. Select the option: ***Form Size Policy*** / ***Generate pack()*** / ***Generate Resize code***.
- Click the Source tab above the design window to open the program code. Locate the main method. Use the + icon to open the program lines and change the parameter "***Nimbus***" to "***Windows***".

Run the program and accept the ***main*** class which is offered. Check that a blank window appears and has the correct size and colour scheme. Close the program and return to the editing screen. Click the ***Design*** tab to move to the form editing screen.

Add labels and text fields to the form to create an input screen for the employee data. Rename the text fields as: ***txtName***, ***txtDepartment***, ***txtAge***, and ***txtSalary***:



We will add three ***buttons*** to the form to test the data saving procedure:

- A button to save the employee data. Give this the caption "***Save on disc***" and the name ***btnSave***.
- A button to clear the four text fields. Give this the caption "***Clear***" and the name ***btnClear***.
- A button to reload the employee data from the disc file and re-display it in the text fields. Give this the caption "***Reload from disc***" and the name ***btnReload***.

Use the Source tab to change to the program code page. A series of Java modules will be needed to save and load the data, and to deal with any file errors which might occur. Add these modules at the start of the program.

```
package staffRecordPackage;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import javax.swing.JOptionPane;

public class staffRecord extends javax.swing.JFrame {

    public staffRecord() {
        initComponents();
    }
```

We must now give a name for the data file which will be created on disc. Add a line of code to do this. We have created our own file type extension (.DAT) to identify a data file.

```
package staffRecordPackage;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import javax.swing.JoptionPane;

public class staffRecord extends javax.swing.JFrame {

    static String fileLocation="staff.dat";

    public staffRecord() {
        initComponents();
    }
```

Please note that if just a file name is given, such as:

"staff.dat"

then the file will be stored in the root of the current project folder, ***staffRecord***. Alternatively, a full pathway can be given to a location where you want the file to be stored, for example:

"C:\program data\staff.dat"

We will now work on the ***save*** procedure for the data file. Use the ***Design*** tab to return to the form view, then double click the "***Save on disc***" button to create a ***method***. Add lines of code to produce a ***TRY … CATCH*** block. This will display a message box if an error occurs during file saving.

```
private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
   try
   {

   }
   catch (IOException e)
   {
       JOptionPane.showMessageDialog(staffRecord.this, "File error");
   }
}
```

A red error symbol will appear alongside the program listing. Ignore this for the time being; we will be adding code shortly which will solve the problem.

The next step is to collect the data items from the text fields and store these temporarily as variables.

```
private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
   try
   {
       String name=txtName.getText();
       String department=txtDepartment.getText();
       String age=txtAge.getText();
       String salary=txtSalary.getText();
   }
   catch (IOException e)
```

We are now going to combine the data items to create a ***record***. We will separate the individual data fields with commas, for example:

Dafydd Jones, Engineering, 23, 21500

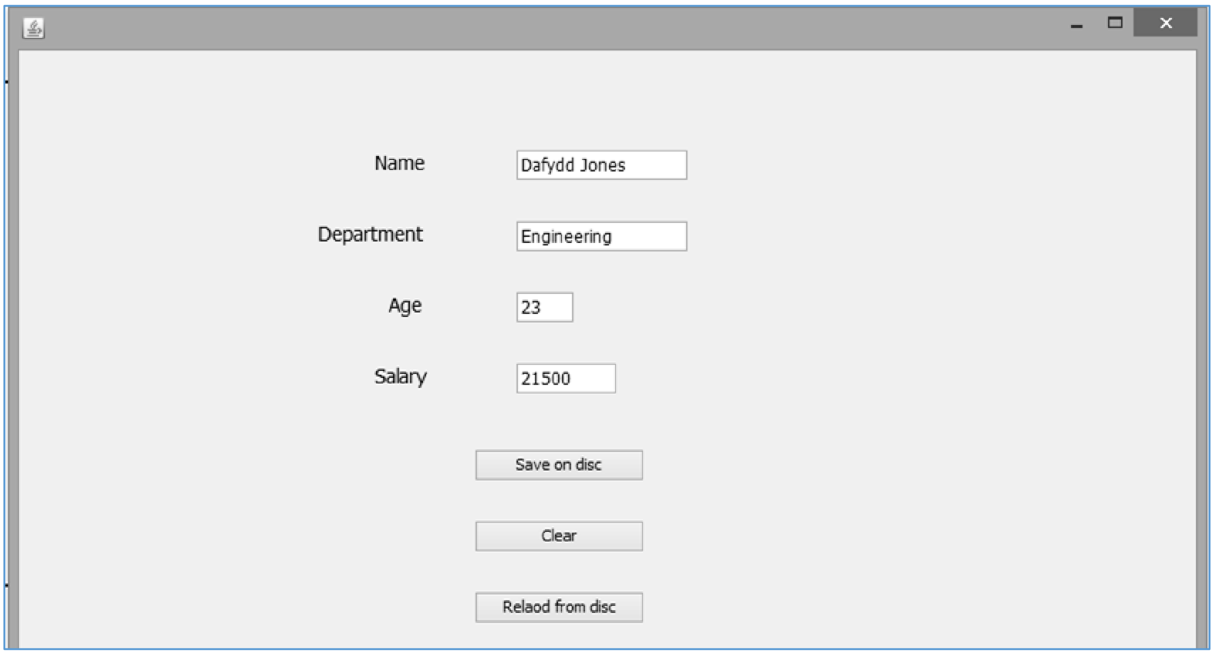
To do this, set up the string variable **s**, then add each of the data items in turn to build up the record:

```
   try
   {
       String name=txtName.getText();
       String department=txtDepartment.getText();
       String age=txtAge.getText();
       String salary=txtSalary.getText();
       String s = name + ",";
       s += department + ",";
       s += age + ",";
       s += salary;
   }
   catch (IOException e)
   {
```

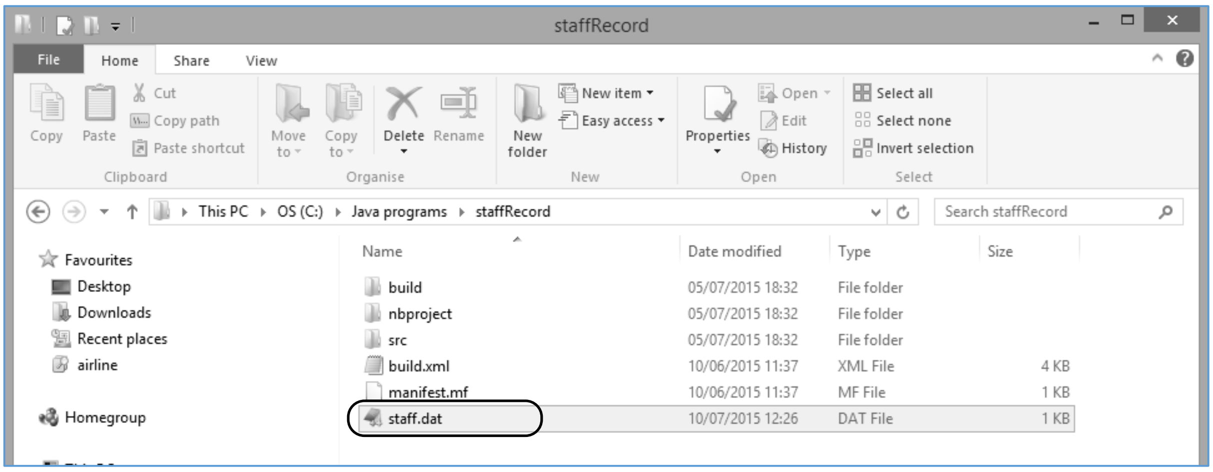
The final step is to save the record on disc. Add lines of code which will open a file with the name and location which we specified earlier, save the record into the file, then close the file.

```
            String s = name+",";
            s += department+",";
            s += age+",";
            s += salary;

            FileWriter w = new FileWriter(fileLocation);
            BufferedWriter writer = new BufferedWriter(w);
            writer.write(s);
            writer.close();

        }
        catch (IOException e)
        {
```

Run the program, enter test data for an employee, then click the "***Save on disc***" button.

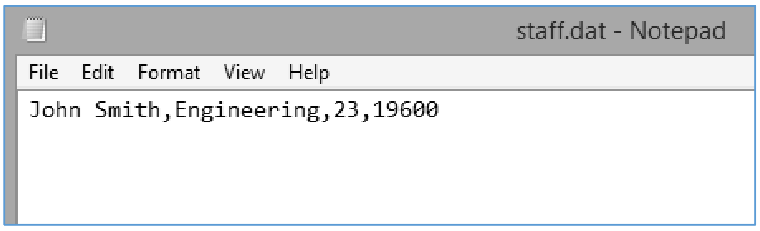

Name: Dafydd Jones
Department: Engineering
Age: 23
Salary: 21500

To check that the data has been saved correctly, use ***Windows Explorer*** to locate the ***staff.dat*** file in the ***staffRecord*** folder:


This PC › OS (C:) › Java programs › staffRecord

Name	Date modified	Type	Size
build	05/07/2015 18:32	File folder	
nbproject	05/07/2015 18:32	File folder	
src	05/07/2015 18:32	File folder	
build.xml	10/06/2015 11:37	XML File	4 KB
manifest.mf	10/06/2015 11:37	MF File	1 KB
staff.dat	10/07/2015 12:26	DAT File	1 KB

Use a text editor such as ***Notepad*** or ***Wordpad*** to open the ***staff.dat*** file. The record should be displayed, with the individual data fields separated by commas:



```
John Smith,Engineering,23,19600
```

Close the program window and return to the NetBeans editing screen. Use the ***Design*** tab to display the form view, then double click the "***Clear***" button to create a ***method***. Add lines of code to clear the entries in the four text fields:

```
private void btnClearActionPerformed(java.awt.event.ActionEvent evt) {

    txtName.setText("");
    txtDepartment.setText("");
    txtAge.setText("");
    txtSalary.setText("");

}
```

Select the ***Design*** tab again to return to the form view, then double click the "***Reload from disc***" button to create a ***method***. Begin by producing a TRY … CATCH block to display an error message when required.

```
private void btnReloadActionPerformed(java.awt.event.ActionEvent evt) {

    try
    {

    }
    catch (IOException e)
    {
         JOptionPane.showMessageDialog(staffRecord.this, "File error");
    }

}
```

When the "***Reload***" button is clicked, the data file must first be opened, the record read into the program, then the file closed. Add lines of code to do this:

```
    try
    {

        FileReader r = new FileReader(fileLocation);
        BufferedReader reader = new BufferedReader(r);
        String s=reader.readLine();
        reader.close();

    }
    catch (IOException e)
```

The next step is to split the record into the separate fields, then the data items displayed in the text fields on the form. We use the ***split*** command in Java:

```
    private void btnReloadActionPerformed(java.awt.event.ActionEvent evt) {
        try
        {
            FileReader r = new FileReader(fileLocation);
            BufferedReader reader = new BufferedReader(r);
            String s=reader.readLine();
            reader.close();

            String dataItem[] = s.split(",");
            txtName.setText(dataItem[0]);
            txtDepartment.setText(dataItem[1]);
            txtAge.setText(dataItem[2]);
            txtSalary.setText(dataItem[3]);

        }
        catch (IOException e)
        {
```

We tell the ***split*** command which character should be used to separate the string **s** into individual fields. We have used a comma:

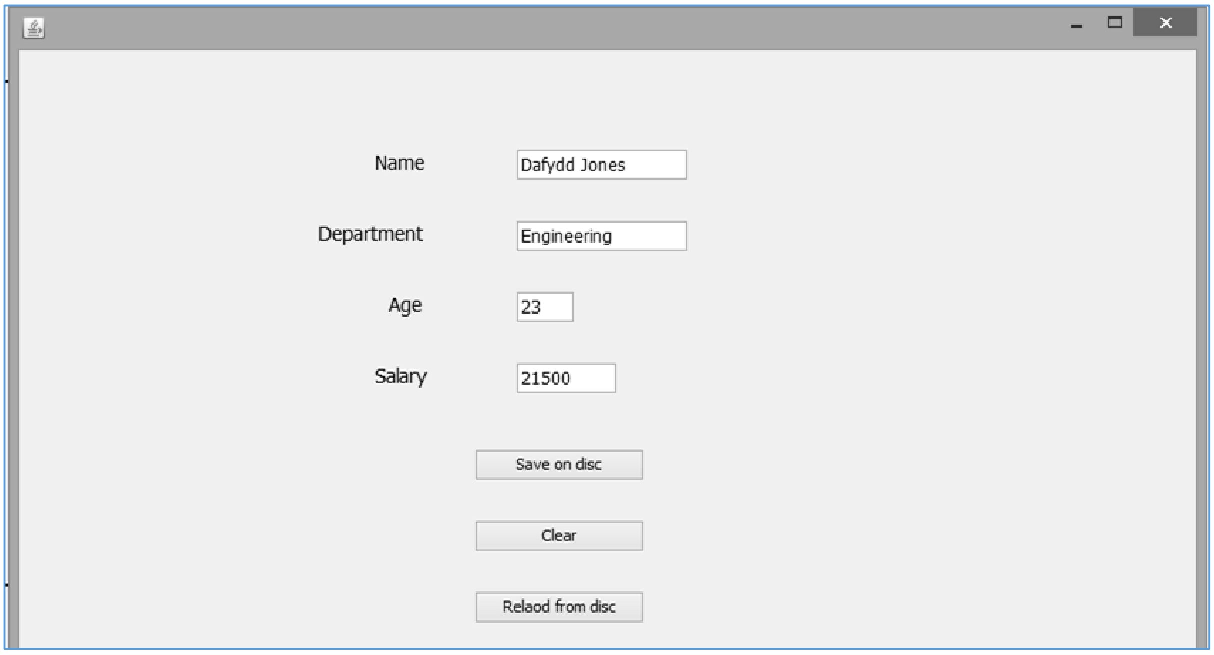
s.split(",")

The command will then create an array containing each of the individual data items:

- ***dataItem[0]*** will contain the employee name
- ***dataItem[1]*** will contain their department
- ***dataItem[2]*** will contain their age
- ***dataItem[3]*** will contain their salary

This data can then be displayed in the text fields on the form.

Run the program. Enter and save a staff record. Click the "***Clear***" button, then check that the data can be reloaded correctly:



Name	Dafydd Jones
Department	Engineering
Age	23
Salary	21500

In the next program, we will see how more than one record can be added to a text file, then re-displayed on screen.

An airport requires a program which will input information about departing flights, then display this information for passengers in a data table.

For each flight, the information required is:

- Flight ID
- Destination
- Airline
- Departure time

We will use similar methods to the previous program for storing and reloading the flight departure records.

Set up a new project in the standard way. Close all projects, then set up a ***New Project***. Give this the name ***airlineFlights***, and ensure that the ***Create Main Class*** option is not selected.

Return to the NetBeans editing page. Right-click on the ***airlineFlights*** project, and select ***New*** / ***JFrame Form***. Give the ***Class Name*** as ***airlineFlights***,and the ***Package*** as ***airlineFlightsPackage***:

Return to the NetBeans editing screen.

- Right-click on the ***form***, and select ***Set layout*** / ***Absolute layout***.
- Go to the ***Properties*** window on the bottom right of the screen and click the ***Code*** tab. Select the option: ***Form Size Policy*** / ***Generate pack()*** / ***Generate Resize code***.
- Click the Source tab above the design window to open the program code. Locate the main method. Use the + icon to open the program lines and change the parameter "***Nimbus***" to "***Windows***".

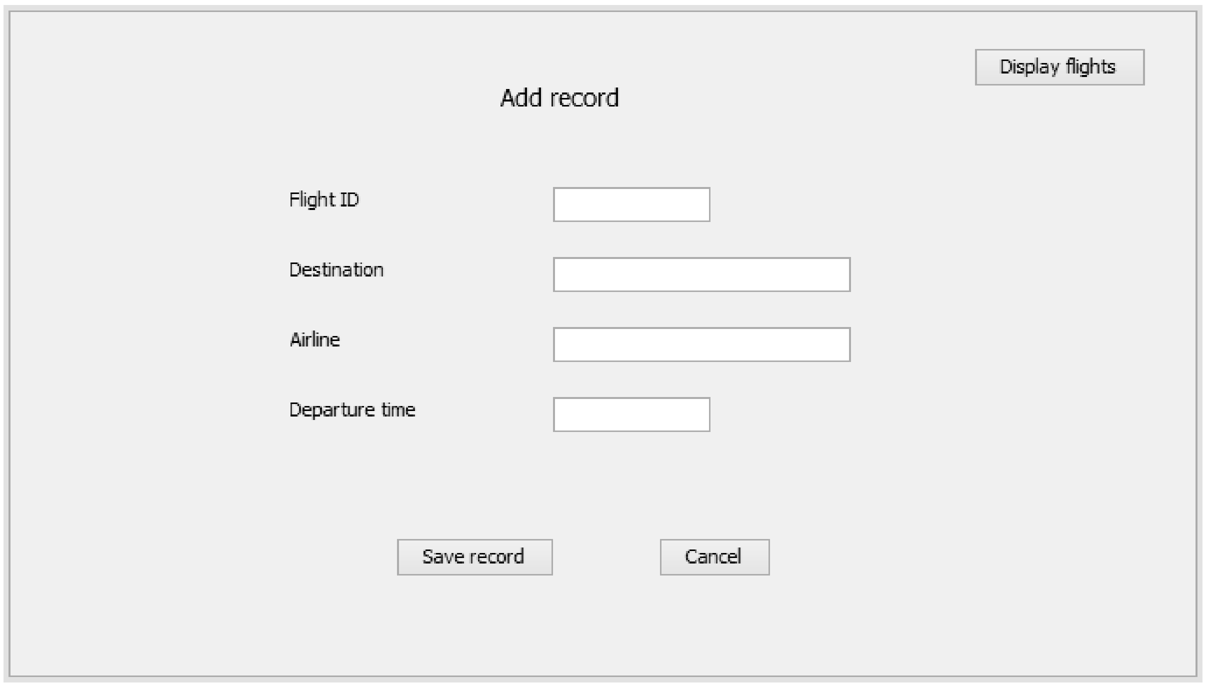
Run the program and accept the ***main*** class which is offered. Check that a blank window appears and has the correct size and colour scheme. Close the program and return to the editing screen. Click the Design tab to move to the form layout view.

Set up the input form by adding ***labels***, ***text fields*** and ***buttons*** as shown in the illustration on the next page. Rename the text fields as:

txtFlightID
txtDestination
txtAirline
txtDepartureTime

Rename the buttons and add captions:

btnDisplayFlights	***"Display flights"***
btnSave	***"Save record"***
btnCancel	***"Cancel"***



Use the ***Source*** tab to open the program code screen.

Go to the start of the program listing and add Java modules for saving data and handling file errors. We will also specify a file name "***flights.dat***" for saving the flight records.

```
package airlineFlightsPackage;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import javax.swing.JOptionPane;

public class airlineFlights extends javax.swing.JFrame {

    static String fileLocation="flights.dat";

    public airlineFlights() {
        initComponents();
    }
```

It will be useful to produce a ***method*** to clear the text fields. Add this below ***airlineFlights( )***:

```
    public airlineFlights() {
        initComponents();
    }

    private void clear()
    {
       txtFlightID.setText("");
       txtDestination.setText("");
       txtAirline.setText("");
       txtDepartureTime.setText("");
    }
```

Use the ***Design*** tab to return to the form design screen.

Double click the "***Cancel***" button to create a method. Add a line of code to call the ***clear( )*** method which we have just written:

```
private void btnCancelActionPerformed(java.awt.event.ActionEvent evt) {
    clear();
}
```

Run the program. Type some data into the text fields, then check that this can be cleared by clicking the "***Cancel***" button.

Close the program window and return to the editing screen. Use the ***Design*** tab to move to the form design page, then double click the "***Save record***" button to create a method.

We will begin by adding a ***TRY … CATCH*** block to handle any file errors.

```
private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
    try
    {

    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(airlineFlights.this, "File error");
    }
}
```

The next step is to collect the data values for the text fields and combine them into a record, separated by commas:

```
    try
    {
        String flightID=txtFlightID.getText();
        String destination=txtDestination.getText();
        String airline=txtAirline.getText();
        String departureTime=txtDepartureTime.getText();
        String s = flightID + ",";
        s += destination + ",";
        s += airline + ",";
        s += departureTime;
    }
    catch (IOException e)
    {
```

The record can now be saved into the ***flights.dat*** file. If all is well, we can display a message that the record has been saved and call the ***clear( )*** method to leave the text fields empty, ready for entry of the next record.

```
        String s = flightID + ",";
        s += destination + ",";
        s += airline + ",";
        s += departureTime;

        FileWriter w = new FileWriter(fileLocation,true);
        BufferedWriter writer = new BufferedWriter(w);
        writer.write(s);
        writer.newLine();
        writer.close();
        JOptionPane.showMessageDialog(airlineFlights.this, "Record saved");
        clear();

    }
    catch (IOException e)
    {
```

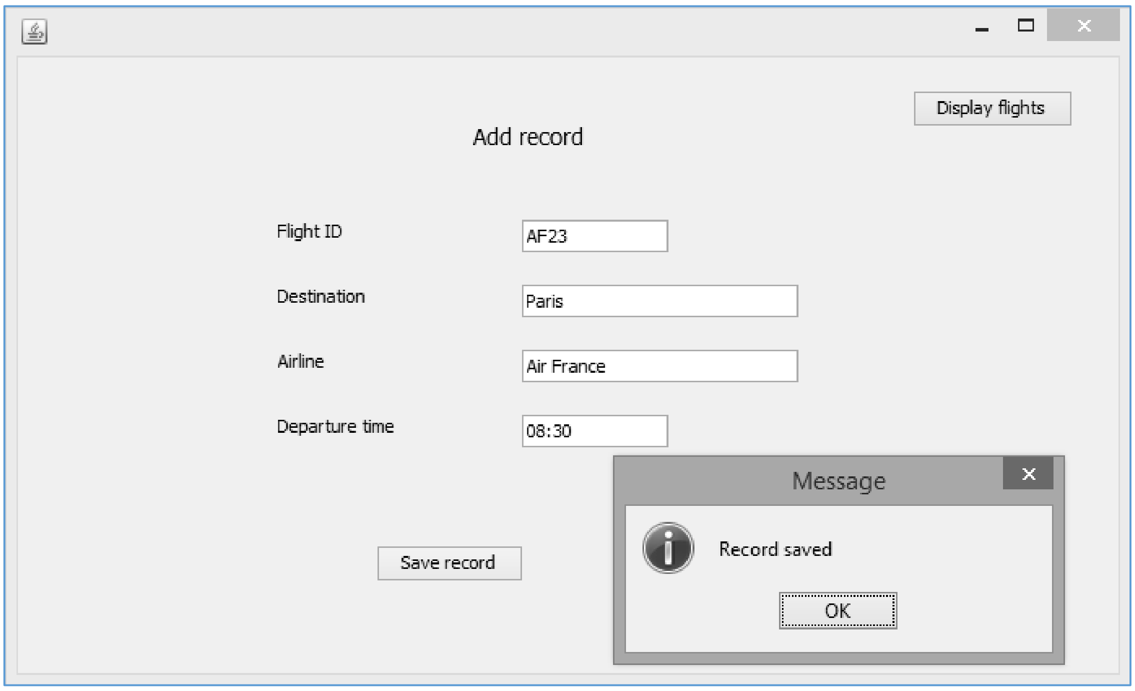
Notice that we have made a small change to the program code, compared to the ***staff record*** program earlier in this chapter. In that program, we used a line:

FileWriter w = new FileWriter(fileLocation);

to instruct the computer to create a new file, ready to save records. However, in this case we only want a new file to be created for the first record, then further records are to be added to the existing file. This can be done by adding another parameter "***true***" to the command:

FileWriter w = new FileWriter(fileLocation, true);

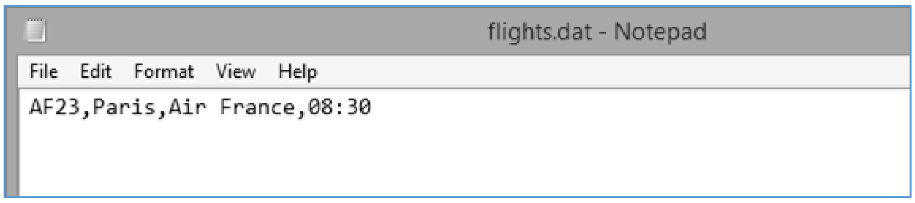
Run the program. Enter a flight record then click the "***Save record***" button. The message "***Record saved***" should appear.



Flight ID AF23
Destination Paris
Airline Air France
Departure time 08:30

Record saved

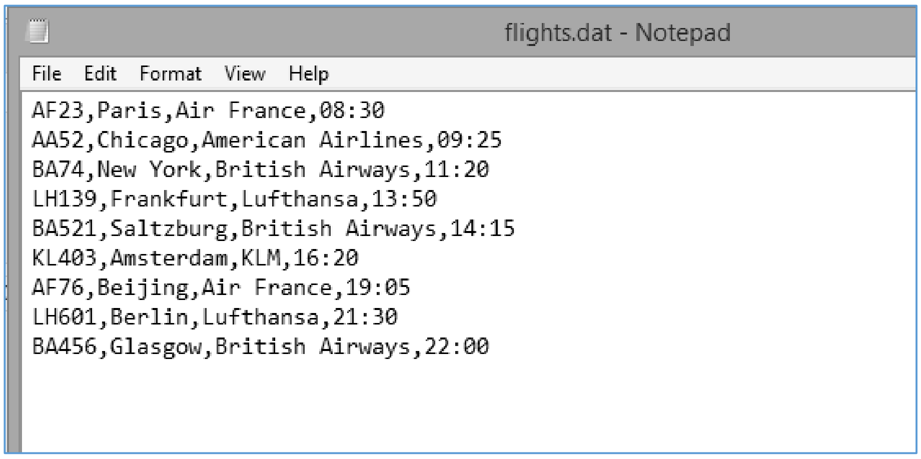
Use ***Windows Explorer*** to locate the ***airlineFlights*** project folder. This should now contain the data file ***flights.dat***. Open the data file with a text editing program such as Notepad. Check that the flight record has been saved correctly:



```
AF23,Paris,Air France,08:30
```

Return to the airlineFlights program window. Enter and save data for several more flights.

Again open the ***flights.dat*** file in a text editor program and check that the records have been added.

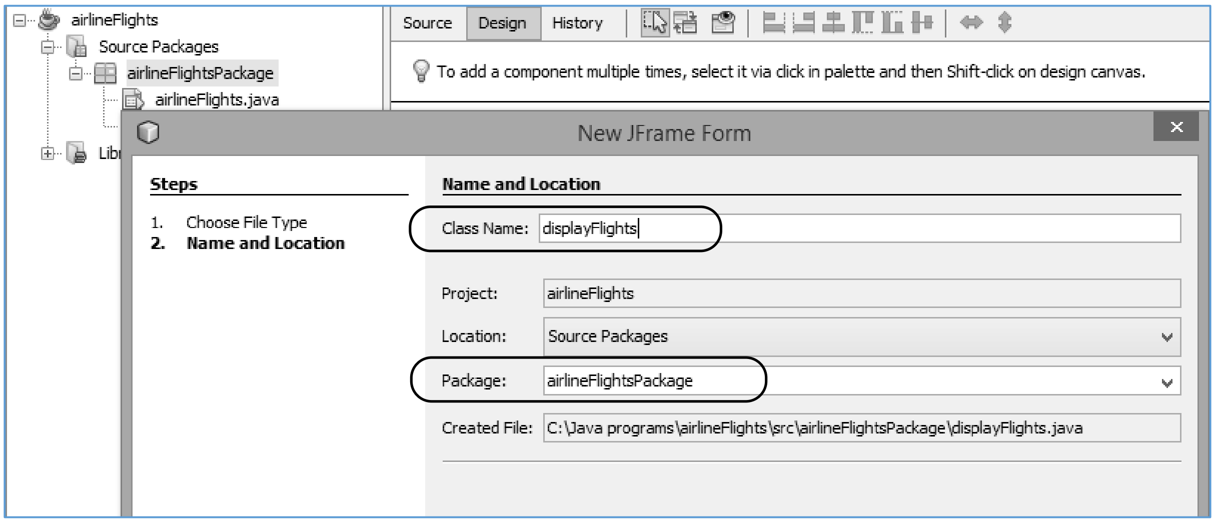


```
AF23,Paris,Air France,08:30
AA52,Chicago,American Airlines,09:25
BA74,New York,British Airways,11:20
LH139,Frankfurt,Lufthansa,13:50
BA521,Saltzburg,British Airways,14:15
KL403,Amsterdam,KLM,16:20
AF76,Beijing,Air France,19:05
LH601,Berlin,Lufthansa,21:30
BA456,Glasgow,British Airways,22:00
```

Close the program window and return to the NetBeans editing screen.

We have now completed the input section of the program and can work on the output of flight information using a data table component. This can be done on a separate form.

Go to the ***Projects*** window and locate ***airlineFlightsPackage***. Right-click to open the menu, then select ***New / JFrame Form***. Give the ***Class Name*** as ***displayFlights***, but leave the ***Package*** name as ***airlineFlightsPackage***.



To add a component multiple times, select it via click in palette and then Shift-click on design canvas.

Class Name: displayFlights

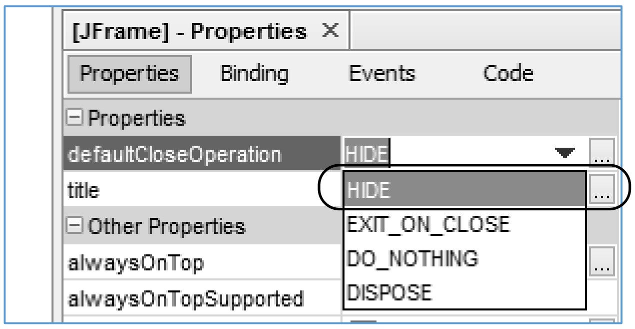
Project: airlineFlights

Location: Source Packages

Package: airlineFlightsPackage

Created File: C:\Java programs\airlineFlights\src\airlineFlightsPackage\displayFlights.java

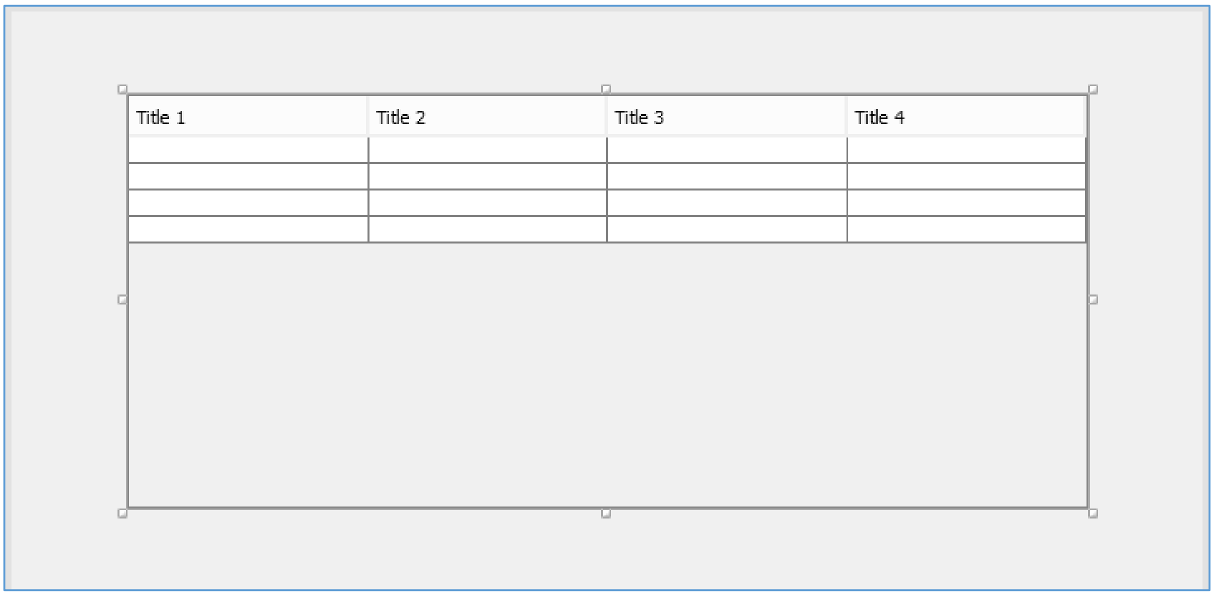
The new blank form will open. Go to the ***Properties*** window and set the ***defaultCloseOperation*** to '***HIDE***'. This will allow the user to close the display form and return to the flight input screen without the whole program closing.



Still within the Properties window, click the ***Code*** tab. Select the option: ***Form Size Policy*** / ***Generate pack()*** / ***Generate Resize code***.

Return to the blank form. Right-click and select ***Set layout*** / ***Absolute layout***.

Locate the Table component in the palette. Drag and drop this on the form. Rename the table as ***tblDepartures***.

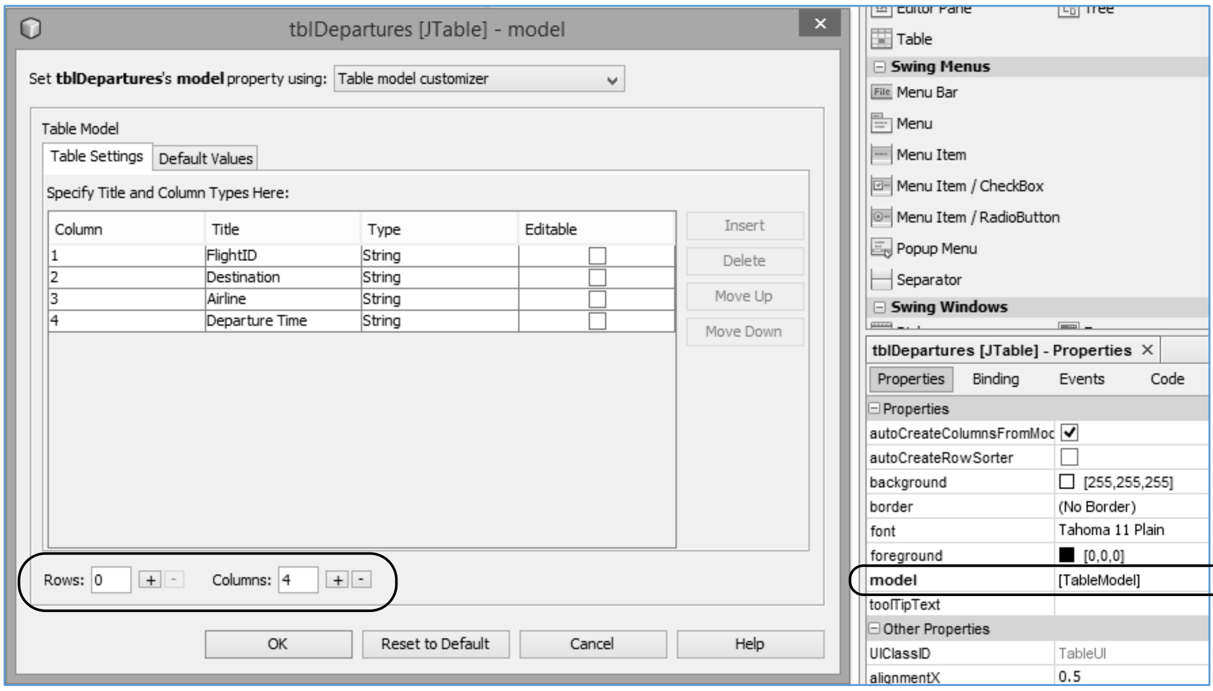


Go to the Properties window and locate the ***model*** property. Click in the right hand column to open an editing window, as shown below.

Set the number of ***Rows*** to **0** and the number of ***Columns*** to **4**. Set up titles and data types for the four columns of the table:

FlightID	***String***
Destination	***String***
Airline	***String***
Departure Time	***String***

Remove the ticks from the Editable property for each field, as shown below.



tblDepartures [JTable] - model

Set **tblDepartures**'s **model** property using: Table model customizer

Table Model

Table Settings | Default Values

Specify Title and Column Types Here:

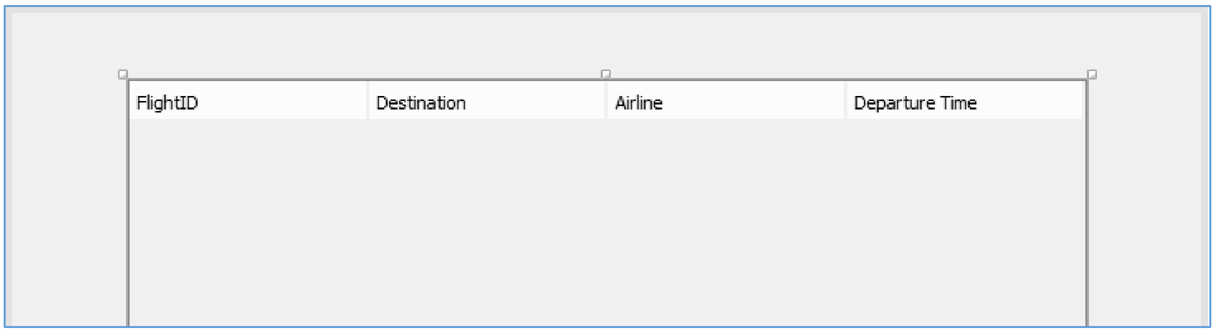
Column	Title	Type	Editable
1	FlightID	String	☐
2	Destination	String	☐
3	Airline	String	☐
4	Departure Time	String	☐

Rows: 0 Columns: 4

tblDepartures [JTable] - Properties

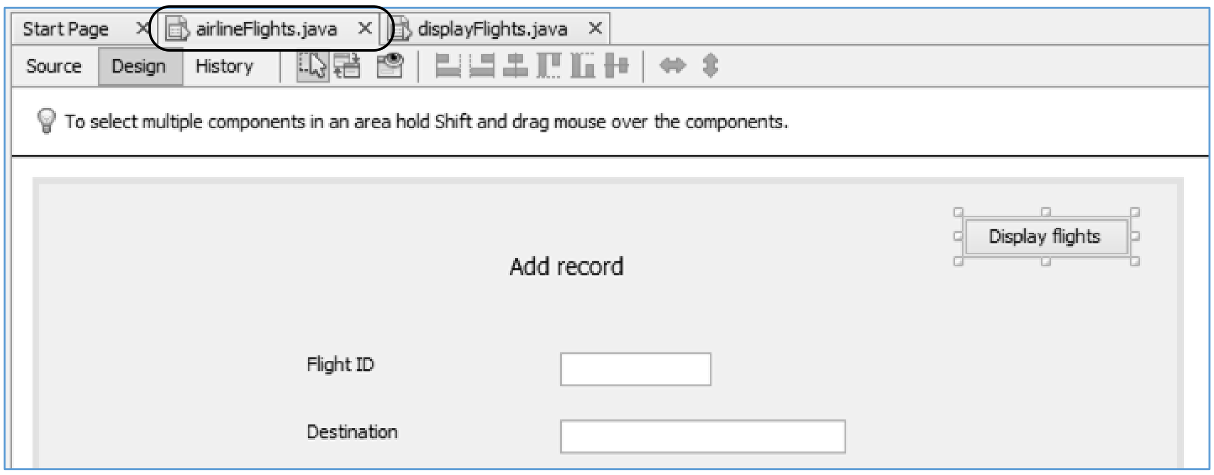
autoCreateColumnsFromModel	✓
autoCreateRowSorter	☐
background	[255,255,255]
border	(No Border)
font	Tahoma 11 Plain
foreground	[0,0,0]
model	[TableModel]
toolTipText	
UIClassID	TableUI
alignmentX	0.5

Click the ***OK*** button to return to the form. The table headings should now be displayed.



Before going further, we should link the ***displayFlights*** form to the main program, and check that this form can be opened correctly when the program is running.

Return to the ***airlineFlights*** form and double click the '***Display flights***' button to create a method.

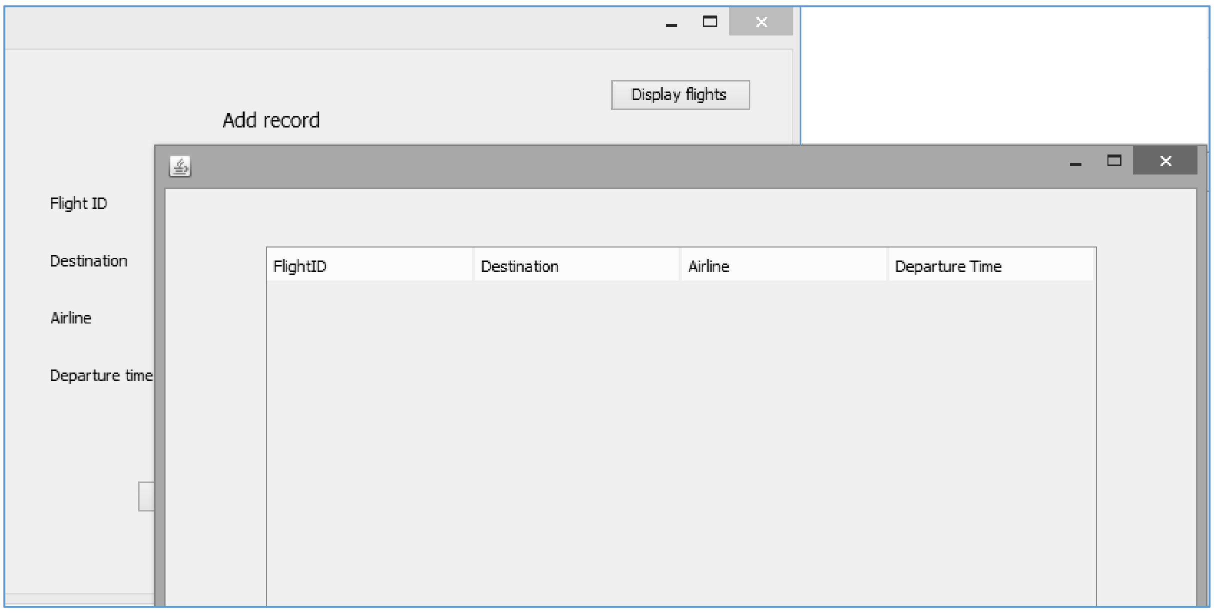


To select multiple components in an area hold Shift and drag mouse over the components.

Add the line of code needed to open the ***displayFlights*** form.

```
    private void btnDisplayFlightsActionPerformed(java.awt.event.ActionEvent evt) {
        new displayFlights().setVisible(true);
    }
```

Run the program. Check that the ***displayFlights*** form opens correctly when the button is clicked. It should be possible to close the ***displayFlights*** form and return to the ***Add record*** page with the program still running.



Close the program and return to the NetBeans editing screen. Use the ***displayFlights.java*** tab above the program code window to move to the ***displayFlights*** source code page.

We will again add Java modules at the start of the program listing. These are required for the loading of data and the display of records in the table, along with the processing of any errors which might occur.

```
package airlineFlightsPackage;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Vector;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;

public class displayFlights extends javax.swing.JFrame {
```

We will add a line of code to give the name of the data file, "***flights.dat***".

It will be convenient to create a ***loadData( ) method*** to read the records from the disc file and display them in the table. This method can be called from ***displayFlights( )***, which is the first method to operate when the form is opened.

```
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;

public class displayFlights extends javax.swing.JFrame {

    static String fileLocation="flights.dat";

    public displayFlights() {
        initComponents();

        loadData();

    }

    private void loadData()
    {

    }
```

We will add ***TRY .. CATCH*** blocks to the ***loadData( )*** method to handle any errors which occur when loading data from disc. We will also need a string variable **s** to temporarily store the records as they are being loaded.

Within the ***TRY*** block we will add lines of code to open the file ready for input of data, then close the file when data loading is completed.

```
private void loadData()
{
    String s;
    try
    {
        FileReader r = new FileReader(fileLocation);
        BufferedReader reader = new BufferedReader(r);

        reader.close();
    }
    catch (IOException e)
    {
         JOptionPane.showMessageDialog(displayFlights.this, "File error");
    }
}
```

This program differs from the ***staff record*** project earlier in the chapter, as there will be more than one record to load. We do not know in advance how many records are present in the file, so a ***WHILE*** loop will be used:

while((s = reader.readLine()) != null)

This line will attempt to read a record from the file and store it temporarily as the variable **s**. If no more records are available, then **s** will be left ***empty***; this is known as a ***NULL*** condition. The logical operator " ***!=*** " means ***NOT***. The effect of the line of code is therefore:

"Try to read a record from the file and store it in the variable **s**. Keep doing this as long as another record can be loaded and the variable **s** is not empty."

Add lines of code to set up the loop.

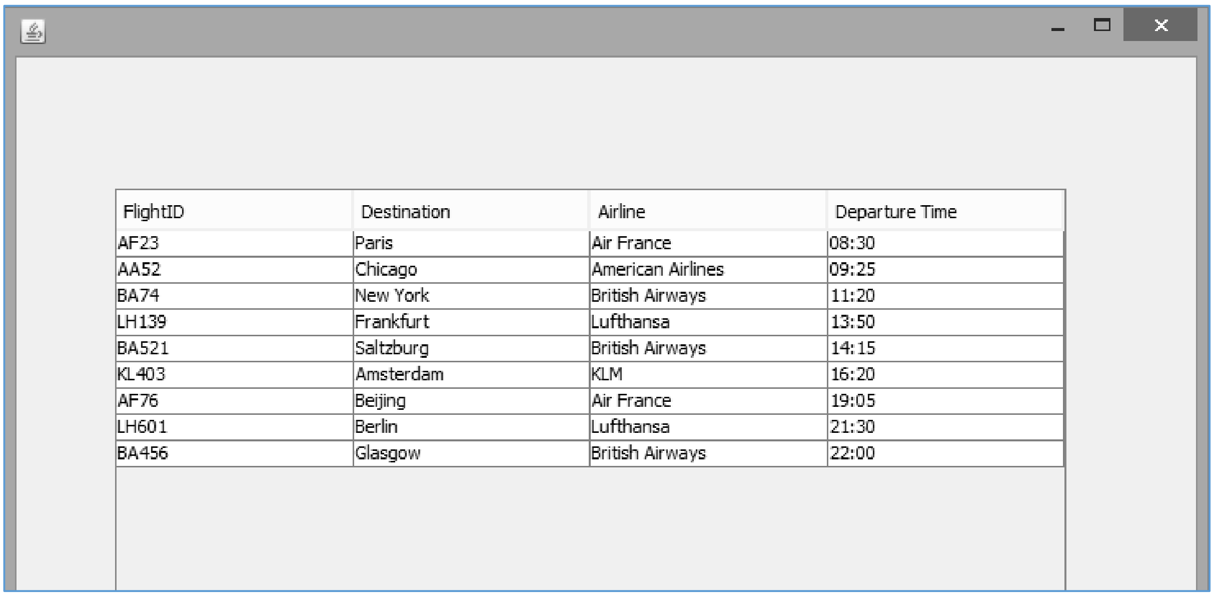
```
    try
    {
        FileReader r = new FileReader(fileLocation);
        BufferedReader reader = new BufferedReader(r);

        while((s=reader.readLine())!=null)
        {

        }

        reader.close();
    }
    catch (IOException e)
    {
```

The final stage is to split the record into separate fields at the positions of the commas by using the ***split(",")*** command. The individual field values will be stored in an array called ***dataItem[ ]***. A new ***row*** is added to the table for each record, and the data items copied into that row.

```
try
{
    FileReader r = new FileReader(fileLocation);
    BufferedReader reader = new BufferedReader(r);
    while((s=reader.readLine())!=null)
    {

        if (s.length()>0)
        {
            String dataItem[] = s.split(",");
            DefaultTableModel model =(DefaultTableModel) tblDepartures.getModel();
            Vector row = new Vector();
            row.add(dataItem[0]);
            row.add(dataItem[1]);
            row.add(dataItem[2]);
            row.add(dataItem[3]);
            model.addRow(row);
        }

    }
    reader.close();
```

Run the program. Click the "***Display flights***" button and check that the flights previously entered are shown correctly in the table.

Close the ***Display flights*** form to return to the ***Add record*** page. Enter a couple more flight records, then check that these now appear in the display table.



FlightID	Destination	Airline	Departure Time
AF23	Paris	Air France	08:30
AA52	Chicago	American Airlines	09:25
BA74	New York	British Airways	11:20
LH139	Frankfurt	Lufthansa	13:50
BA521	Saltzburg	British Airways	14:15
KL403	Amsterdam	KLM	16:20
AF76	Beijing	Air France	19:05
LH601	Berlin	Lufthansa	21:30
BA456	Glasgow	British Airways	22:00

Fixed and variable length records

The two programs which we have produced so far in this chapter have used ***variable length records***. By this, we mean that the size of the records stored in the disc file will vary according to the size of the data items. For example, the record:

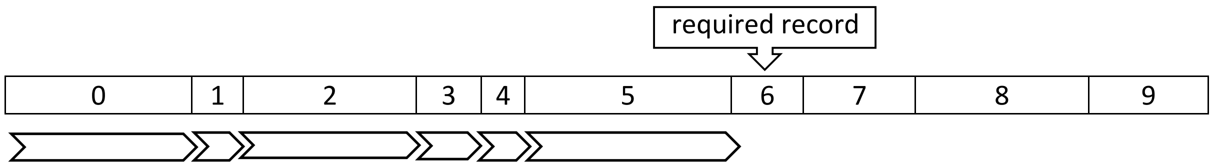
AF23,Paris,Air France,08:30

takes less storage space than the record:

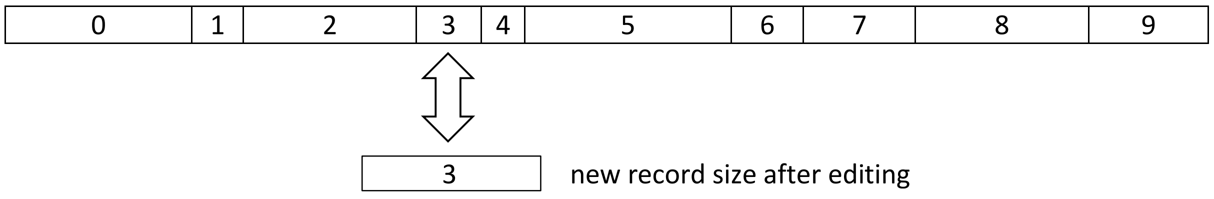
BA521,Saltzburg,British Airways,14:15

This strategy has the advantage that the minimum amount of space will be needed to store the data on disc, but there are some disadvantages.

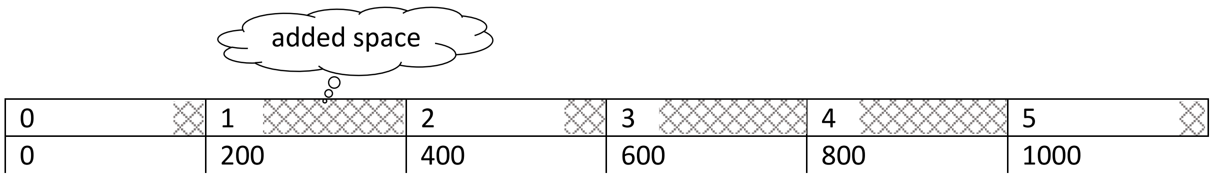
Since each record is of an unknown length, there is no easy way to find a particular record in the file without reading through every previous record first.



A further problem with fixed length records is that it is difficult to edit the data. It is likely that the size of a record will change when it is edited, and the whole file will have to be rebuilt.



Both of these problems can be solved by an alternative approach to storing data which uses ***fixed length records***. In this system, a record size is chosen which will be sufficient to hold the largest of the data items for each field. All records are then adjusted to the specified size by adding blank space if necessary. For example, each record may be given a size of 200 bytes:



The location of any record within the file can now be calculated using the formula:

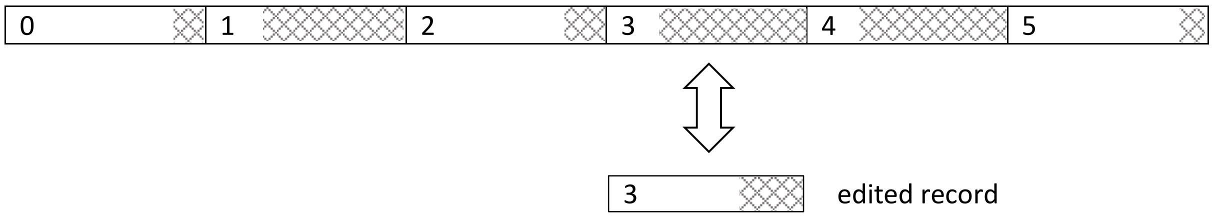
(record location) = (record sequence number) * (record size)

For example, the location of record 4 in this file is:

(4 * 200) = 800 bytes from the start of the file

It is therefore possible to go straight to the required record without reading any previous records first, which can greatly improve access speed for a large database.

Fixed length records also offer the advantage that they can be edited and reinserted into the same place in the file without affecting any other records, even if the amount of data in the edited record is changed:



The decision whether to use ***variable length records*** or ***fixed length records*** will depend on weighing up the advantages and disadvantages for any particular program:

- If very fast access is important, or if the data is likely to be changed frequently, the programmer may choose ***fixed length records***. The speed advantages may outweigh the cost of the extra storage space required.
- If the individual records are likely to be very different in size, the programmer may decide to use ***variable length records*** to avoid wasting large amounts of storage space. Slower access times may then have to be accepted.

In the next program, we will examine how fixed length records can be used in a Java program.

An office supplies company requires a database to keep records of its stock. The records will have to be updated regularly as quantities of stock and prices change, so a fixed length record system is to be used.

The fields required for each stock record are:

• Stock ID	4 character code
• Description	maximum of 60 characters
• Quantity in stock	maximum of 4 characters
• Price	maximum of 8 characters

For simplicity, we will treat all the data as text strings. Each text character A-Z, 0-9 and other keyboard symbols can be represented by one byte of data. The structure of a record is therefore:

StockID	Description	Quantity	Price
4 bytes	60 bytes	4 bytes	8 bytes

making a total size of 76 bytes for each record.

Set up a new project in the standard way. Close all projects, then set up a ***New Project***. Give this the name ***officeSupplies***, and ensure that the ***Create Main Class*** option is not selected.

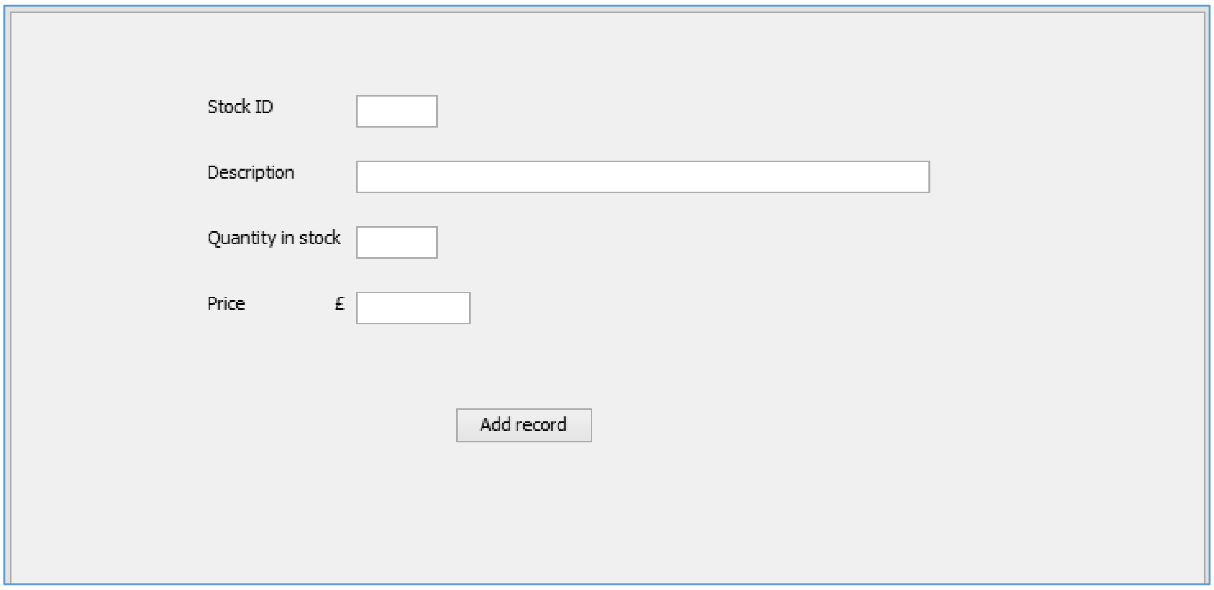
Return to the NetBeans editing page. Right-click on the ***officeSupplies*** project, and select ***New*** / ***JFrame Form***. Give the ***Class Name*** as ***officeSupplies***,and the ***Package*** as ***officeSuppliesPackage***:

Return to the NetBeans editing screen.

- Right-click on the ***form***, and select ***Set layout*** / ***Absolute layout***.
- Go to the ***Properties*** window on the bottom right of the screen and click the ***Code*** tab. Select the option: ***Form Size Policy*** / ***Generate pack()*** / ***Generate Resize code***.
- Click the Source tab above the design window to open the program code. Locate the main method. Use the + icon to open the program lines and change the parameter "***Nimbus***" to "***Windows***".

Run the program and accept the ***main*** class which is offered. Check that a blank window appears and has the correct size and colour scheme. Close the program and return to the editing screen. Click the Design tab to move to the form layout view.

Set up the input form by adding ***labels***, ***text fields*** and a ***button*** as shown in the illustration on the next page. Rename the text fields as: ***txtStockID***, ***txtDescription***, ***txtQuantity*** and ***txtPrice***. Set the button caption to '***Add record***' and rename the button as ***btnSave***.



Use the Source tab to move to the program code screen. Add Java modules which will be needed by the program, and give the filename "stock.dat".

```
package officeSuppliesPackage;

import java.io.IOException;
import java.io.RandomAccessFile;
import javax.swing.JOptionPane;

public class officeSupplies extends javax.swing.JFrame {

    static String filename = "stock.dat";

    public officeSupplies() {
        initComponents();
    }
```

It will be useful to set up a ***clear( )*** method to reset the text fields. Add this after the ***officeSupplies( )*** method.

```
    public officeSupplies() {
        initComponents();
    }

    private void clear()
    {
        txtStockID.setText("");
        txtDescription.setText("");
        txtQuantity.setText("");
        txtPrice.setText("");
    }
```

Return to the ***Design*** screen and double click the "***Add record***" button to create a method.

Add lines of code to collect the input data from the text fields. Notice that a ***trim( )*** function has been included on each line. This will remove any blank spaces at the beginning or end of the string, leaving only the characters that were actually entered by the user.

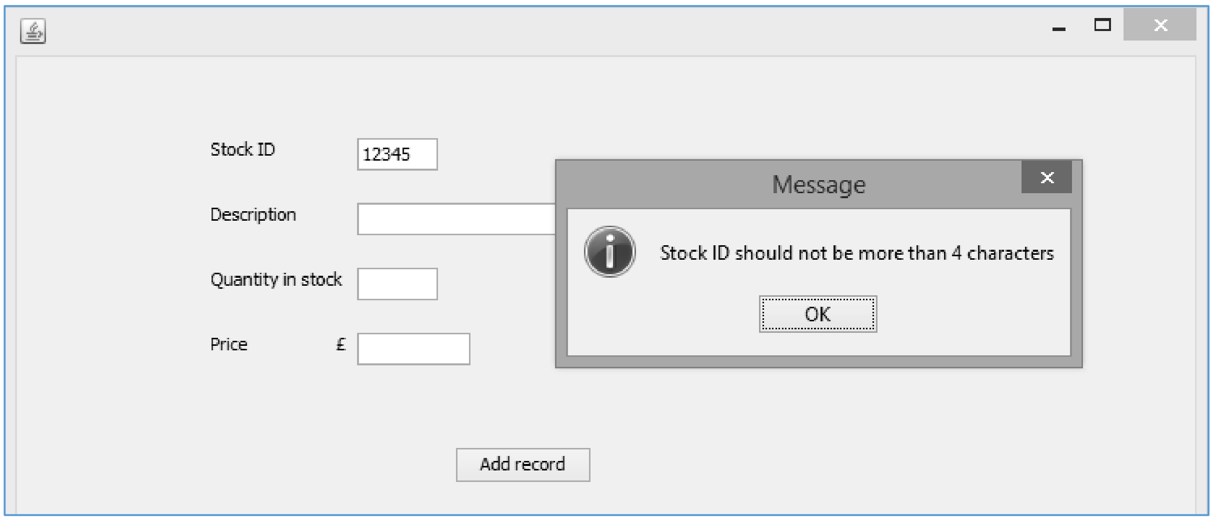
```
    private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
        String stockID=txtStockID.getText().trim();
        String description=txtDescription.getText().trim();
        String quantity=txtQuantity.getText().trim();
        String price=txtPrice.getText().trim();
    }
```

We will be saving ***fixed length records***, so it is important to check that the user does not enter data items which exceed the allocated space. Add lines of code to check the lengths of the ***stockID*** and ***description*** entries, and display an error message if too many characters have been entered.

(Please note that the ***showMessageDialog*** commands should be entered as single lines of code with no line break.)

```
private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
        String stockID=txtStockID.getText().trim();
        String description=txtDescription.getText().trim();
        String quantity=txtQuantity.getText().trim();
        String price=txtPrice.getText().trim();
        if (stockID.length()>4)
        {
           JOptionPane.showMessageDialog(officeSupplies.this,
                        "Stock ID should not be more than 4 characters");
        }
        else
        {
            if(description.length()>60)
            {
                JOptionPane.showMessageDialog(officeSupplies.this,
                        "Description should not be more than 60 characters");
            }
            else
            {

            }
        }
}
```

Run the program. Check that error messages are displayed correctly if the entries for ***Stock ID*** or ***Description*** exceed the allowed lengths.



Stock ID should not be more than 4 characters

Close the program and return to the program code screen.

The next step is to create the record, ready to save into the disc file. We do this be adding a series of lines similar to:

description=String.format("%-60s", description);

This line takes the variable ***description***, and adds blank space at the end to create a string with a length of 60 characters.

```
    if(description.length()>60)
    {
         JOptionPane.showMessageDialog(officeSupplies.this,
                        "Description should not be more than 60 characters");
    }
    else
    {
        stockID=String.format("%-4s", stockID);
        description=String.format("%-60s", description);
        quantity=String.format("%-4s", quantity);
        price=String.format("%-8s", price);
        String s = stockID + description + quantity + price + "***";
    }
```

The final line of this block:

String s = stockID + description + quantity + price + "";***

combines the fields to create the record, which is stored temporarily as the variable **s**. Notice that three asterisk ("*") characters have been added as an ***end of record marker***. This is not strictly necessary, but it will make it easier to identify individual records when we examine the data file.

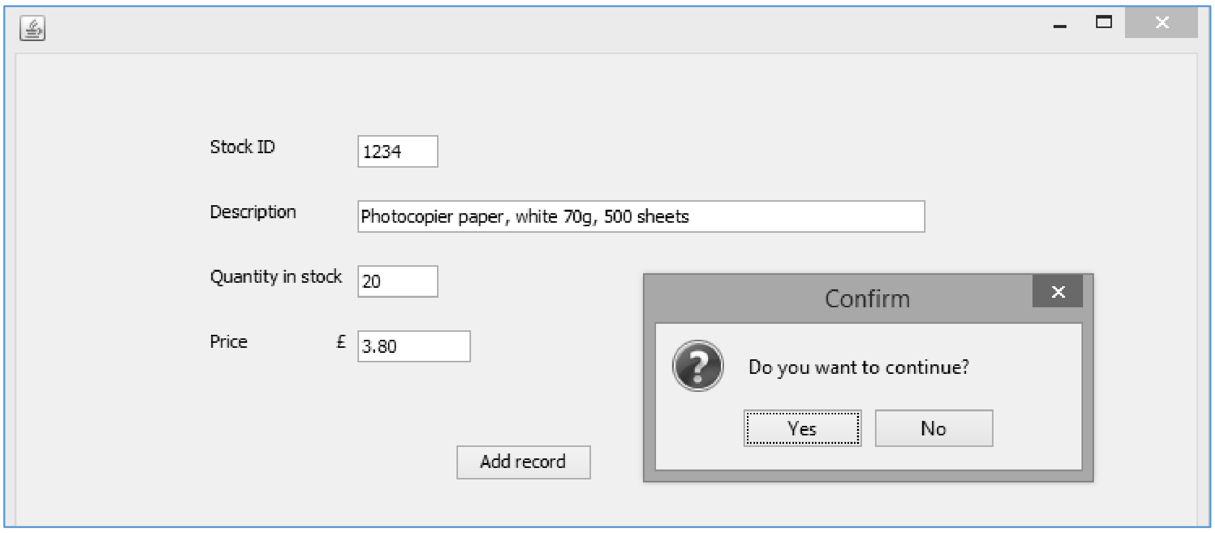
When saving or editing data, it is often a good idea to ask the user to confirm their decision before changes are made to the data file. We can do this by adding a ***showConfirmDialog*** function. Note that the command:

int response = JOptionPane.showConfirmDialog(... JOptionPane.QUESTION_MESSAGE);

should be entered as a single line of code without line breaks.

```
else
{
    stockID=String.format("%-4s", stockID);
    description=String.format("%-60s", description);
    quantity=String.format("%-4s", quantity);
    price=String.format("%-8s", price);
    String s = stockID + description + quantity + price + "***";

    int response = JOptionPane.showConfirmDialog(null,
             "Do you want to continue?", "Confirm", JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE);
    if (response == JOptionPane.YES_OPTION)
    {

    }
 }
```

Run the program. Enter some test data and click the "***Add record***" button. Check that the "***Confirm***" dialog box appears.



Stock ID: 1234
Description: Photocopier paper, white 70g, 500 sheets
Quantity in stock: 20
Price £ 3.80

Confirm
Do you want to continue?

Close the program and return to the code editing screen.

The final step, if the user confirms that they wish to continue, is to save the record into the data file. Add lines of code to the block beginning:

if (response == JOptionPane.YES_OPTION)

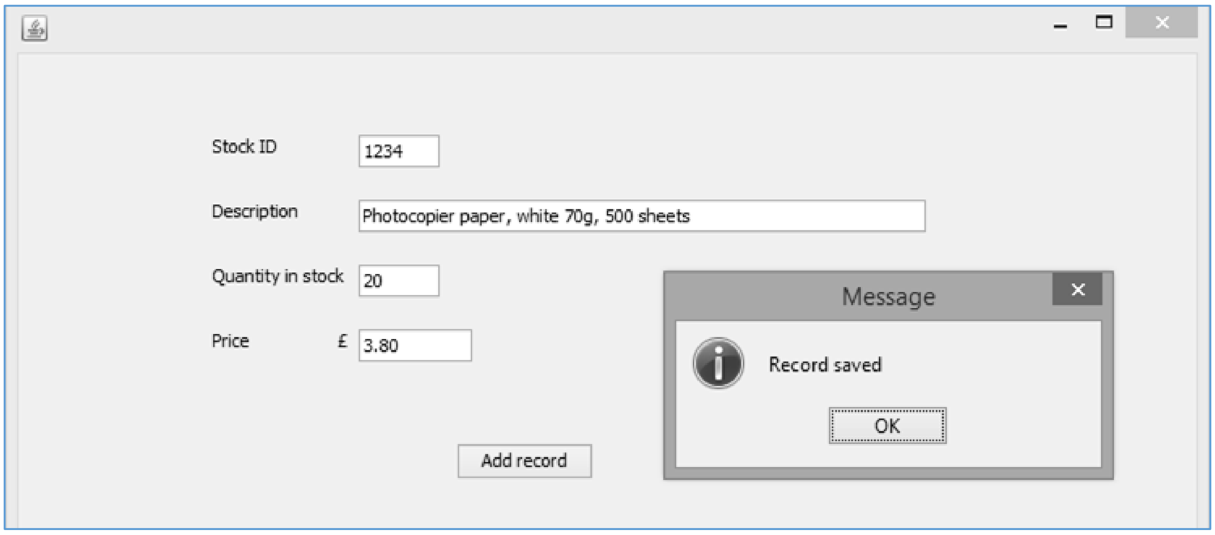
This block of code will only operate if the user clicks the "***Yes***" button.

```
if (response == JOptionPane.YES_OPTION)
{
    try(RandomAccessFile file = new RandomAccessFile(filename, "rw"))
    {
        int position=(int) file.length();
        file.seek(position);
        file.write(s.getBytes());
        JOptionPane.showMessageDialog(officeSupplies.this, "Record saved");
        clear();
    }
    catch(IOException e)
    {
        JOptionPane.showMessageDialog(officeSupplies.this, "File error");
    }
}
```

Quite a lot is happening here:

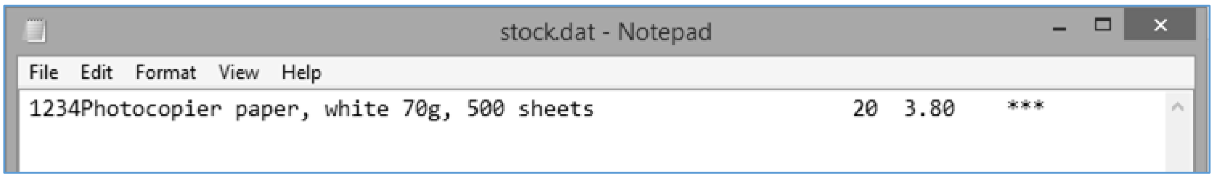
- The program tries to open the file ***stock.dat***. It will create a new file called ***stock.dat*** if the file is not found.
- A ***seek*** pointer is moved to the end of the file, so that the new record will be added after any existing records.
- The record, currently stored as the string variable **s**, is converted into a binary form. Each character will be represented by one byte.
- The record is saved into the file. If this operation is successful, a confirmation message is given to the user, otherwise a file error message is displayed.
- After saving the record, the text fields are cleared, ready for the next data entry.

Run the program and enter a stock record. Click the "***Add record***" button and confirm to save:


Stock ID: 1234
Description: Photocopier paper, white 70g, 500 sheets
Quantity in stock: 20
Price £ 3.80

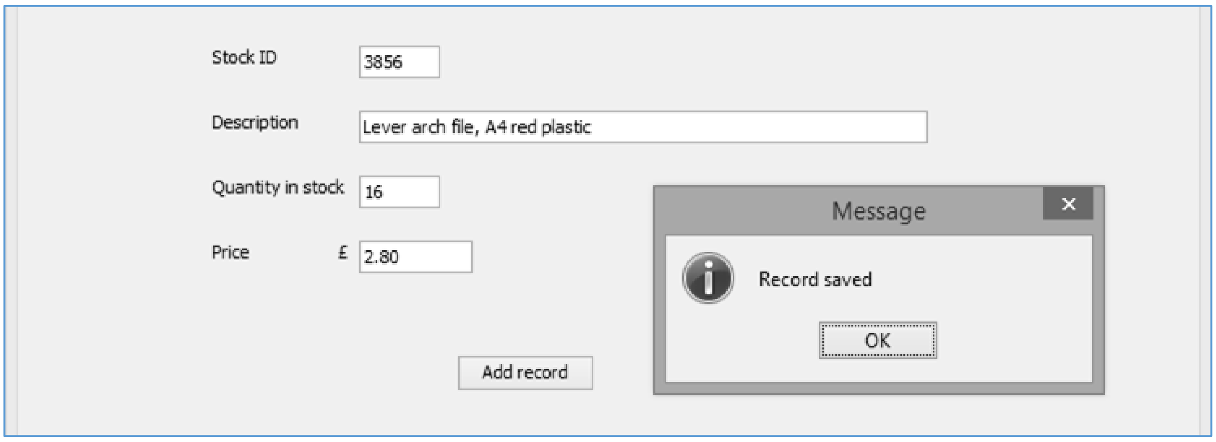
Message
Record saved

Use ***Windows Explorer*** to locate the file ***stock.dat*** in the ***officeSupplies*** project folder. Open the file in a text editing program such as ***Notepad***. The record should be displayed. Notice that blank space has been added to create the correct field lengths, and the record ends with the marker " ***** ".



```
1234Photocopier paper, white 70g, 500 sheets                    20  3.80     ***
```

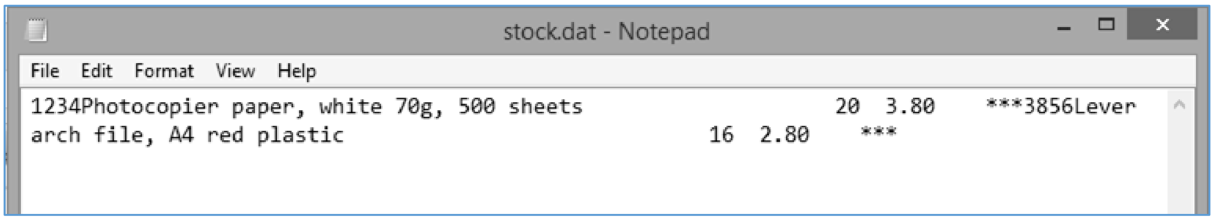
Return to the program and enter another record:



Stock ID: 3856
Description: Lever arch file, A4 red plastic
Quantity in stock: 16
Price £ 2.80

Record saved

Re-open the ***stock.dat*** file and check that the second record has been added. Notice that no line breaks occur between the records.



```
1234Photocopier paper, white 70g, 500 sheets                    20  3.80     ***3856Lever
arch file, A4 red plastic                                 16  2.80     ***
```

Enter several more records, then close the program window and return to the program editing page.

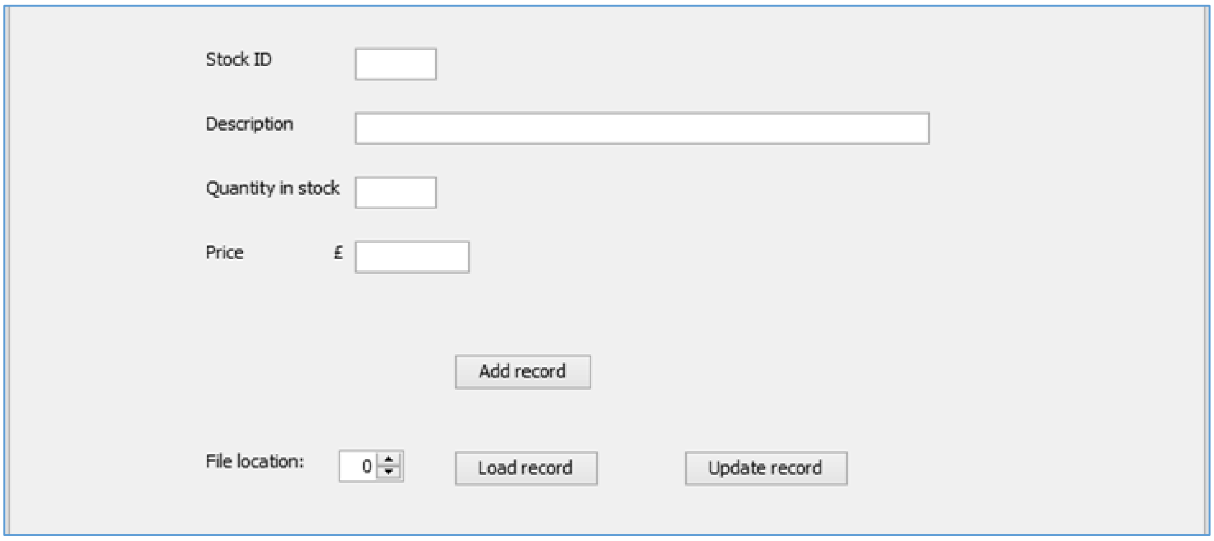
From the earlier discussion, you may remember that two advantages of ***fixed length records*** are:

- an individual record can be accessed directly by calculating its position in the file,
- an edited record can be inserted back into the file without affecting other records.

We will now develop methods in the officeSupplies project to demonstrate these functions.

Use the Design tab to move to the form display, then add a label, spinner, and two buttons as shown below. Rename the spinner as ***spinCount***. Set the variable names and text captions for the buttons as:

btnLoad	***Load record***
btnUpdate	***Update record***



Double click the "***Load record***" button to create a method.

The first step is to calculate the position in the file where the required record begins. We can make use of the formula:

(record location) = (record sequence number) * (record size)

The record sequence number is given by the spinner value. The size of the four fields of the stock record adds up to 76 bytes. With the additional three characters for the end of record marker, the total record size becomes 79 bytes.

```
private void btnLoadActionPerformed(java.awt.event.ActionEvent evt) {

    int recordWanted=(int) spinCount.getValue();
    int position=recordWanted * 79;

}
```

Add a ***TRY … CATCH*** blocks and lines of code to read the required record from the file:

```
private void btnLoadActionPerformed(java.awt.event.ActionEvent evt) {

    int recordWanted=(int) spinCount.getValue();
    int position=recordWanted * 79;

    try
    {
        RandomAccessFile file = new RandomAccessFile(filename, "r");
        file.seek(position);
        byte[] bytes = new byte[79];
        file.read(bytes);
        file.close();
        String s=new String(bytes);
        System.out.println(s);
    }
    catch(IOException e)
    {
        JOptionPane.showMessageDialog(officeSupplies.this, "File error");
    }

}
```

A series of actions are carried out by this code:

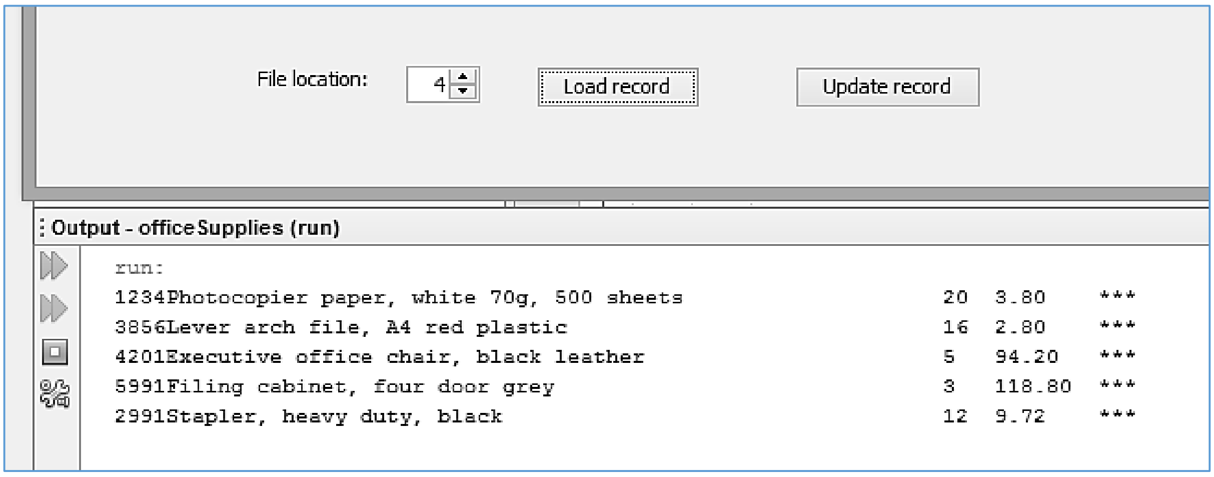
- The program attempts to open the file ***stock.dat***
- The ***seek*** pointer moves to the calculated position in the file where the required record begins.
- The 79 bytes of the record are read from the file, and the file is closed.
- The binary data is then converted into a text string **s**.

An additional line of code has been added for testing purposes:

System.out.println(s);

This will display the string variable **s** in an output window at the bottom of the NetBeans editing screen while the program is running.

Run the program. Use the spin component to select each of the records in turn and click the "***Load record***" button. Remember that the first record is at ***location 0*** in the file. The complete records should be displayed in the ***Output*** window at the bottom of the NetBeans editing screen.



```
run:
1234Photocopier paper, white 70g, 500 sheets                  20   3.80     ***
3856Lever arch file, A4 red plastic                           16   2.80     ***
4201Executive office chair, black leather                     5    94.20    ***
5991Filing cabinet, four door grey                            3    118.80   ***
2991Stapler, heavy duty, black                                12   9.72     ***
```

Close the program window and return to the code editing page.

The next step is to split the record into its separate fields and display the data in the text fields on the form. Add lines of code to do this:

```
                file.read(bytes);
                file.close();
                String s=new String(bytes);
                System.out.println(s);

                String stockID=s.substring(0,4);
                s=s.substring(4);
                String description=s.substring(0,60);
                s=s.substring(60);
                String quantity=s.substring(0,4);
                s=s.substring(4);
                String price=s.substring(0,8);
                txtStockID.setText(stockID);
                txtDescription.setText(description);
                txtQuantity.setText(quantity);
                txtPrice.setText(price);

            }
            catch(IOException e)
```

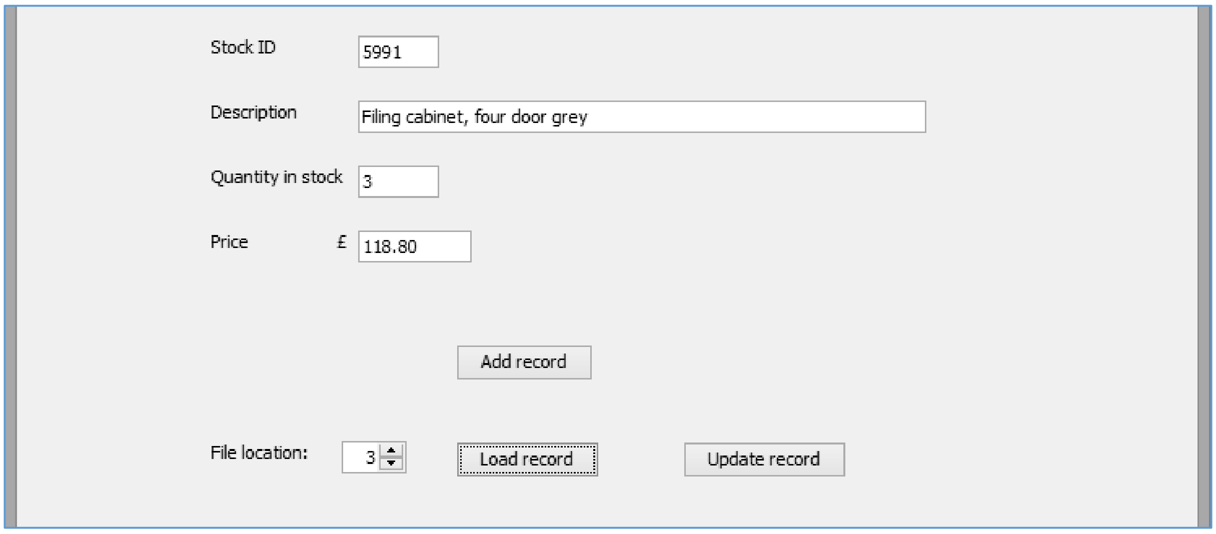
A series of ***substring*** commands are used:

- The required number of characters are copied from the start of the string **s** to create a field variable. For example, four characters are copied to produce a ***stockID*** such as:
  2991
- The corresponding number of characters are then deleted from the start of the string **s**. For example:
  2991Stapler, heavy duty, black   12 9.72   ***

  becomes:
  Stapler, heavy duty, black   12 9.72   ***

The process is repeated until all field variables have been created, then these are displayed in the text fields.

Run the program. Check that each of the records can be selected and displayed on the form. You may need to make the ***txtDescription*** field longer so that all the text output is visible.



Stock ID: 5991
Description: Filing cabinet, four door grey
Quantity in stock: 3
Price £ 118.80
File location: 3

Close the program and return to the editing screen. If all is working correctly, the test line:

System.out.println(s);

can now be deleted.

The final method to implement is the ***updating*** of a record. This will be very similar to the method for saving a new record. Go to the ***Design*** screen and double click the "***Update record***" button. Add code to the method as shown on the next page. Please note that the ***showMessageDialog*** and ***showConfirmDialog*** commands should be entered as single lines of code without line breaks.

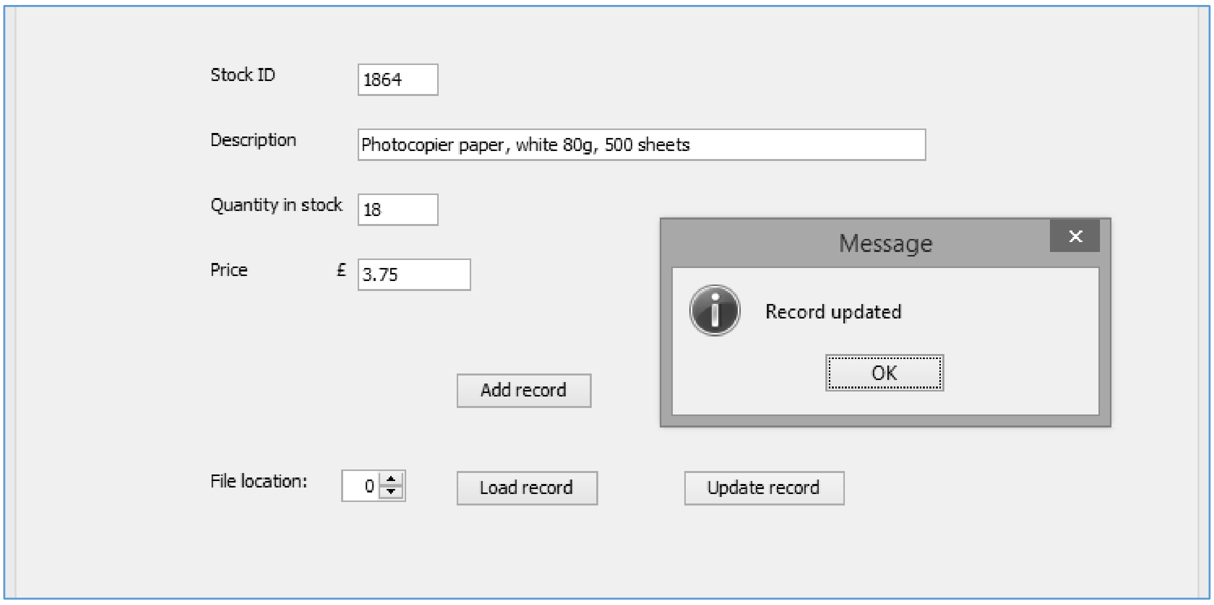
The stages of the update method are:

- Calculate the position of the current record which is being updated using the formula:
  (record location) = (record sequence number) * (record size)
- Collect the data values from the text fields, removing any blank spaces.
- Check the variable lengths for ***stockID*** and ***description***, and give an error message if the data has too many characters.
- Add spaces to the field variables as necessary, then construct the ***fixed length record***.
- Obtain confirmation from the user to update the record.
- Use the ***seek*** pointer to move to the correct position in the file, then overwrite the previous version of the record with the updated version.

```
private void btnUpdateActionPerformed(java.awt.event.ActionEvent evt) {

    int recordWanted=(int) spinCount.getValue();
    int position=recordWanted*79;

    String stockID=txtStockID.getText().trim();
    String description=txtDescription.getText().trim();
    String quantity=txtQuantity.getText().trim();
    String price=txtPrice.getText().trim();

    if (stockID.length()>4)
    {
       JOptionPane.showMessageDialog(officeSupplies.this,
                      "Stock ID should not be more than 4 characters");
    }
    else
    {
        if(description.length()>60)
        {
            JOptionPane.showMessageDialog(officeSupplies.this,
                    "Description should not be more than 60 characters");
        }
        else
        {
            stockID=String.format("%-4s", stockID);
            description=String.format("%-60s", description);
            quantity=String.format("%-4s", quantity);
            price=String.format("%-8s", price);
            String s = stockID + description + quantity + price + "***";

            int response = JOptionPane.showConfirmDialog(null,
                     "Do you want to continue?", "Confirm",
                     JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);

            if (response == JOptionPane.YES_OPTION)
            {
                try(RandomAccessFile file = new RandomAccessFile(filename, "rw"))
                {
                    file.seek(position);
                    file.write(s.getBytes());
                    JOptionPane.showMessageDialog(officeSupplies.this,
                                                     "Record updated");
                }
                catch(IOException e)
                {
                    JOptionPane.showMessageDialog(officeSupplies.this,
                                                     "File error");
                }
            }
        }
    }
}
```

Run the program.

Edit the data for any of the records, then click the "***Update record***" button.



Stock ID 1864

Description Photocopier paper, white 80g, 500 sheets

Quantity in stock 18

Price £ 3.75

File location: 0

Message

Record updated

Display other records, then return to the updated record. Check that the revised data is displayed correctly.