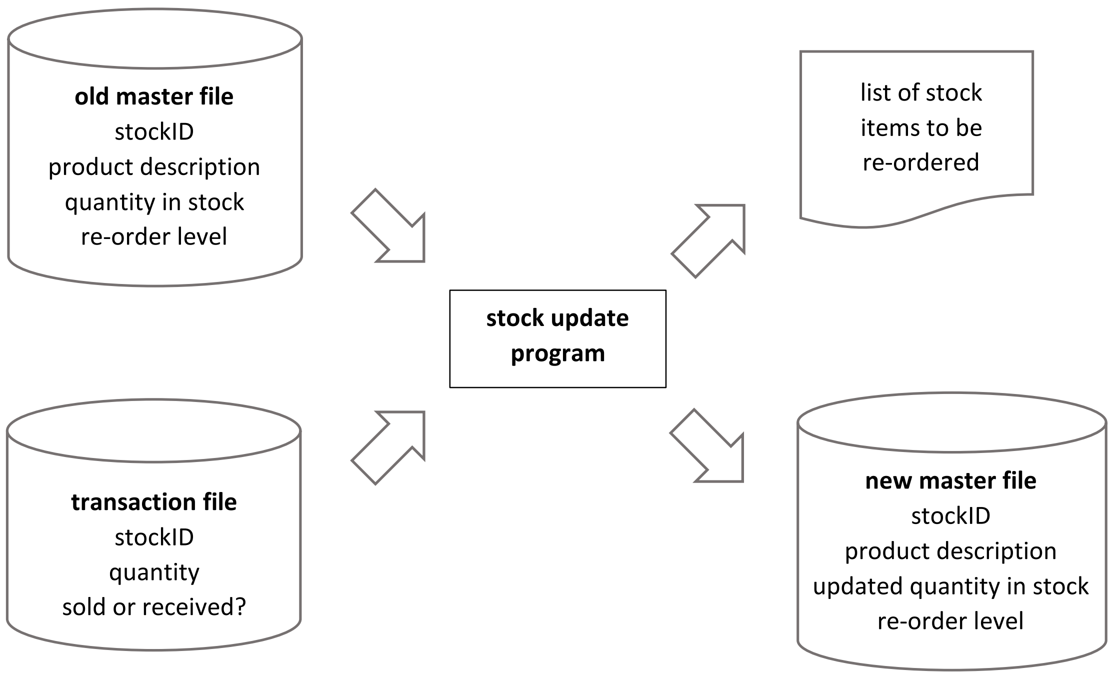
# 12 Batch processing

A number of important computing applications operate by ***batch processing***, for example: the production of gas and electricity bills, bank statements, and even the printing of personalised examination timetables and examination results letters for students.

The characteristics of a batch processing system are that the program and all necessary data are prepared beforehand, then the data is processed automatically without user intervention. Typically, a batch processing system will make use of a ***master file*** which contains data about particular customers or products, and will combine this with a ***transaction file*** to produce the required output.

In an earlier chapter, you produced a ***stock control*** system to keep records of the quantities of products in a shop. The records would need to be updated regularly with changes to the stock totals if products are sold to customers, or new stocks are received from suppliers.

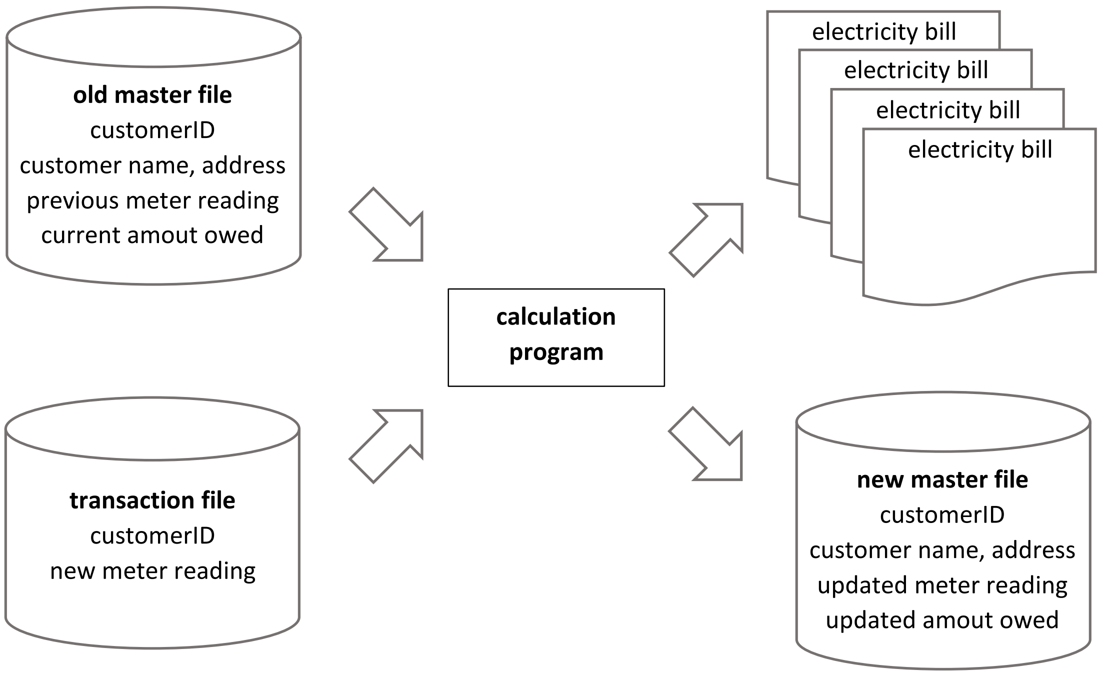
Stock control can be carried out by ***batch processing***. Current details of each product are held in a ***master file***. During each day, the quantities of goods sold or received are recorded in a ***transaction file***. At the end of the day, a program can be run to update the master file and provide a list of stock items which need to be reordered.



In this chapter we will create a complete ***batch processing system***. The application chosen is the production of ***electricity bills***. As well as developing an algorithm for calculating the bills, there will be an opportunity to practice a range of techniques covered in earlier programs, particularly: the use of ***classes*** and ***objects***, ***sorting*** data, and handling ***fixed length records***.

The system to produce electricity bills will make use of two files:

- A ***master file***. This will contain the contact information for each customer, their meter reading at the time that the previous electricity bill was produced, and any amount of money which they currently owe to the electricity company.
- A ***transaction file***. This is produced by the electricity company staff who visit customers' homes to read the meters. The file contains information to identify the customer, plus the new meter reading.



When the meter readings have been collected, a calculation program will work through each customer in turn:

- Information will be collected from the ***master file***.
- The amount of electricity used by the customer will be found by comparing the previous meter reading held in the ***master file*** with the current meter reading held in a ***transaction file***.
- The cost of the electricity can then be calculated, a bill printed, and the customer's record updated with the new meter reading and amount owed.
- Updated records are stored in a ***new master file***.

We will create a program containing three forms:

The first form will allow us to set up **master records** for a series of customers and store these on disc.

The second form will allow new meter readings to be entered for customers, creating a ***transaction file***.

The third form will carry out the ***batch processing*** to produce and display the electricity bills, and will update the master file with the new meter readings and amounts owed by customers.

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

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

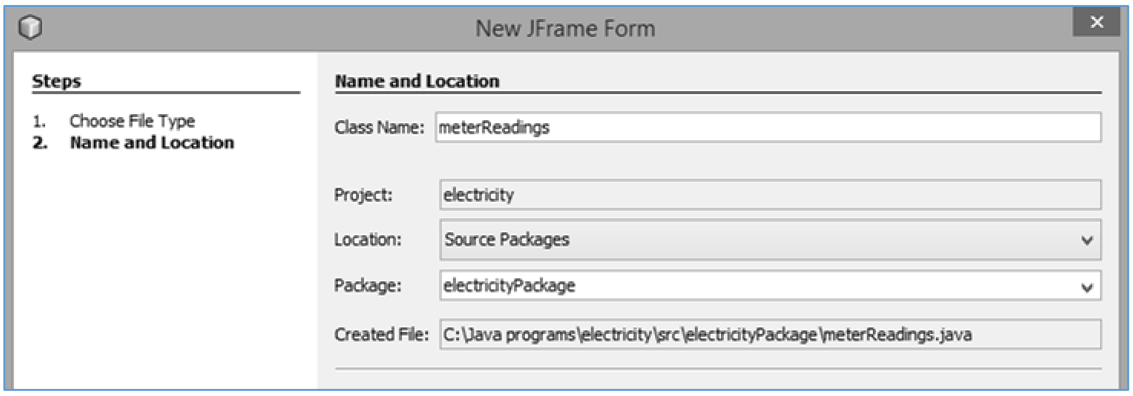
Return to the NetBeans editing screen.

- 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.

It will be best to create the two additional forms before beginning the programming:

Locate ***electricityPackage*** in the Project window at the top left of the screen. Right-click on ***electricityPackage*** and select ***New*** / ***JFrame Form***. Set the ***Class Name*** as ***meterReadings***. Leave the ***Package*** name as ***electricityPackage***.

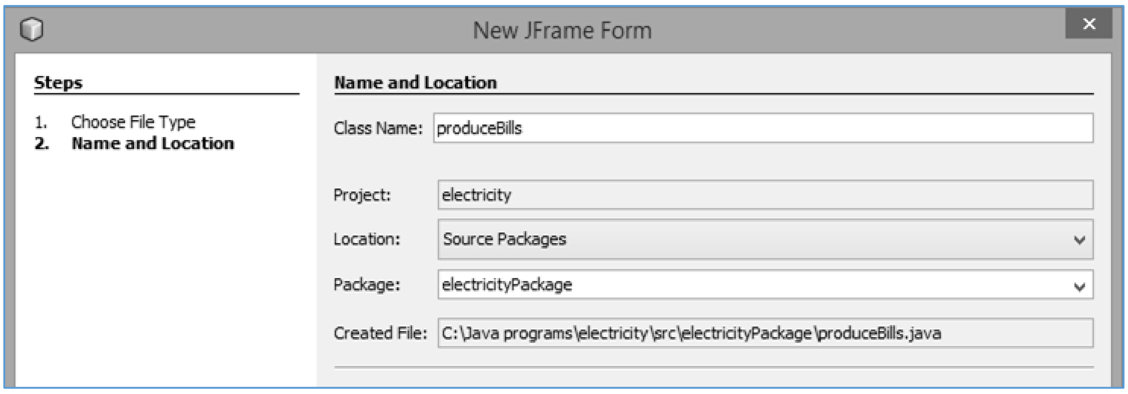


| | |
|---|---|
| Class Name: | meterReadings |
| Project: | electricity |
| Location: | Source Packages |
| Package: | electricityPackage |
| Created File: | C:\Java programs\electricity\src\electricityPackage\meterReadings.java |

Return to the NetBeans editing screen.

- Right-click on the ***meterReadings*** 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***.

Repeat the process to create another form. Set the ***Class Name*** as ***produceBills***.



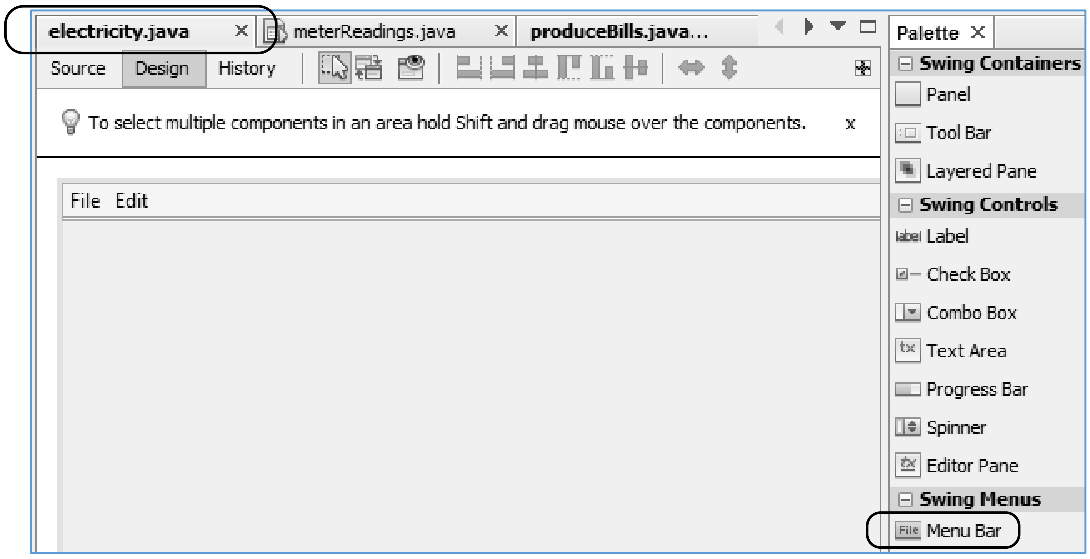
| | |
|---|---|
| Class Name: | produceBills |
| Project: | electricity |
| Location: | Source Packages |
| Package: | electricityPackage |
| Created File: | C:\Java programs\electricity\src\electricityPackage\produceBills.java |

Return to the NetBeans editing screen and again set the required properties:

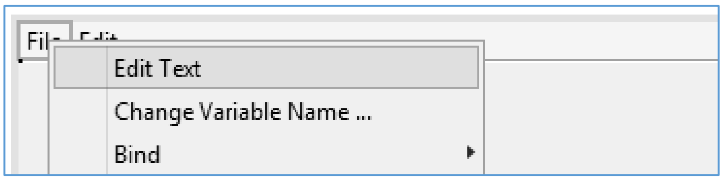
- Right-click on the ***produceBills*** form, and select ***Set layout*** / ***Absolute layout***.
- Go to the ***Properties*** window on the bottom right of the screen and click the ***Code*** tab. Select the option: ***Form Size Policy*** / ***Generate pack()*** / ***Generate Resize code***.

We will now produce a menu system to link the three forms.

Select the ***electicity.java*** tab to open the first form. Click the ***Design*** tab to move to the form layout view. Locate the ***Menu Bar*** component in the ***Palette*** and drag this onto the form.

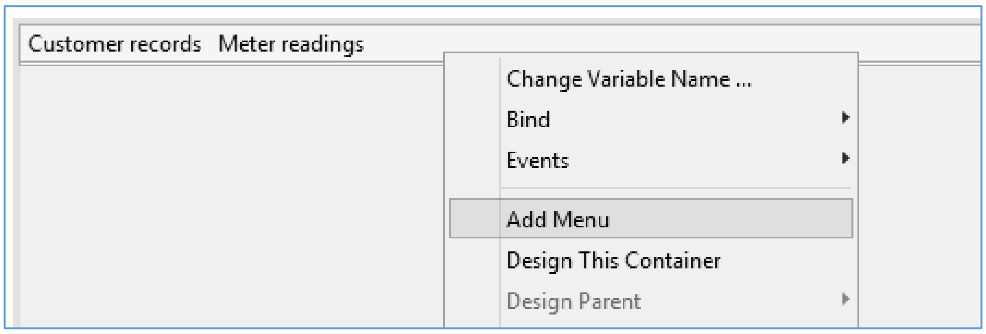


Right-click on the ***File*** menu item and select ***Edit Text***.

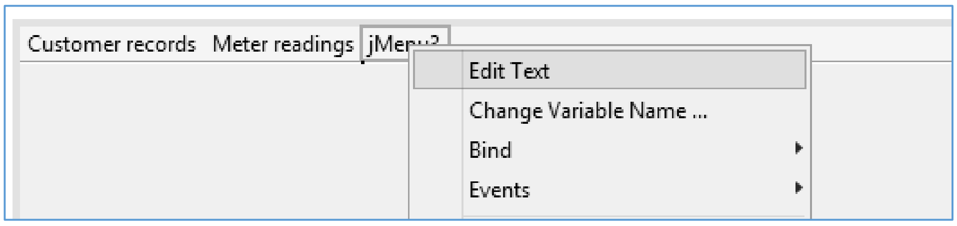


Change the caption to '***Customer records***'. In a similar way, change the Edit menu item to '***Meter readings***'.

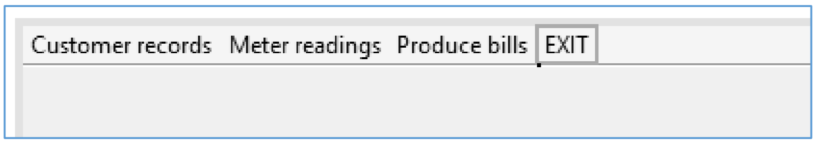
Right-click on an empty area of the menu bar, then select ***Add Menu***.



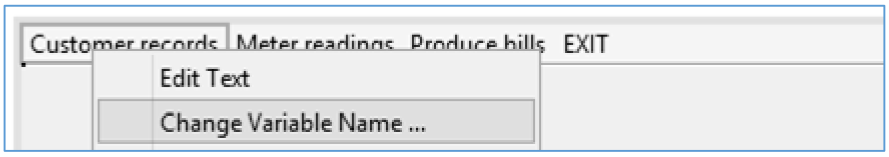
Right-click the new menu item and select Edit Text. Change the caption to ***Produce bills***.



Repeat the procedure to create a fourth menu item '***EXIT***':



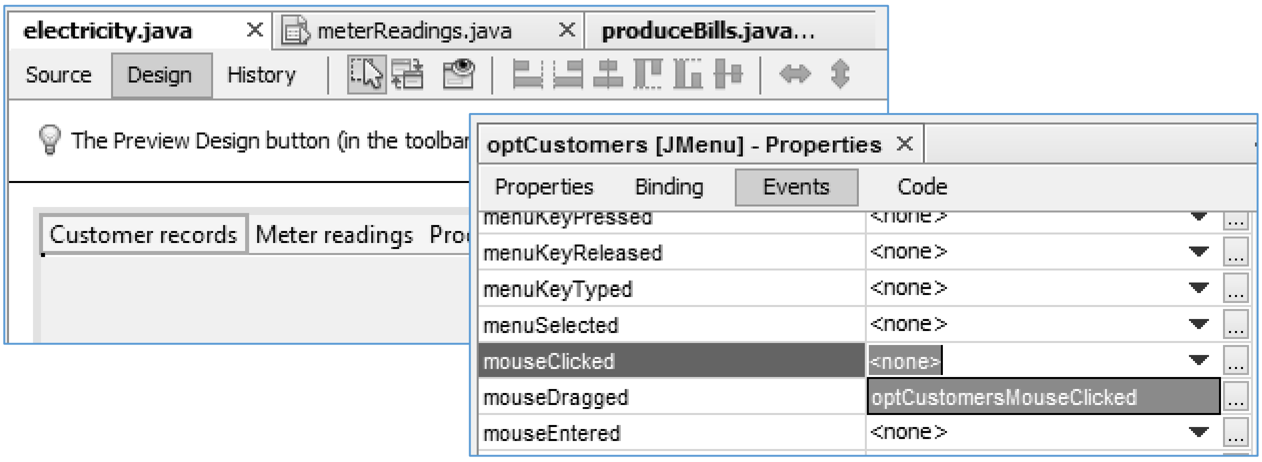
Right-click each of the menu items in turn, and select the ***Change Variable Name*** option:



Change the names of the sequence of menu items to:

***optCustomers***
***optReadings***
***optBills***
***optExit***

Select the ***Customer records*** menu item. Go to the ***Events*** tab in the Properties window and locate the ***mouseClicked*** event. Select ***optCustomersMouseClicked*** from the drop down list.



| | |
|---|---|
| menuKeyPressed | <none> |
| menuKeyReleased | <none> |
| menuKeyTyped | <none> |
| menuSelected | <none> |
| mouseClicked | <none> |
| mouseDragged | optCustomersMouseClicked |
| mouseEntered | <none> |

Add a line of code to the button click method.

```
    private void optCustomersMouseClicked(java.awt.event.MouseEvent evt) {

        new electricity().setVisible(true);

    }
```

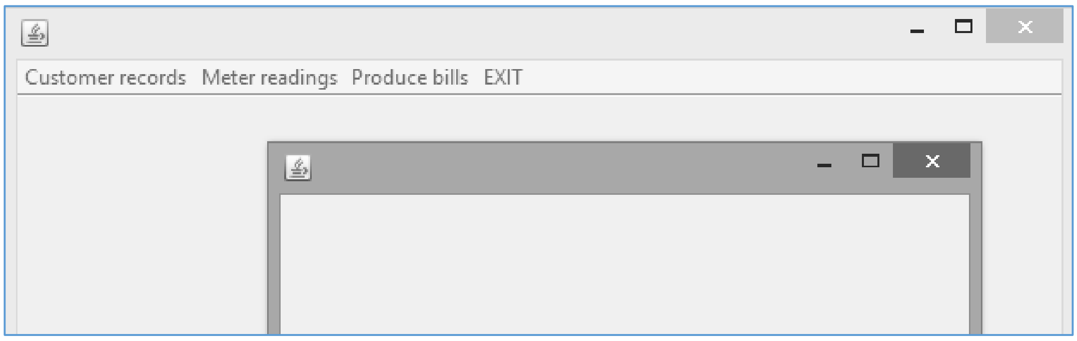
Return to the ***Design*** screen. Create a ***mouseClicked*** method in a similar way for the ***Meter readings*** menu option. Add a line of code to the method.

```
private void optReadingsMouseClicked(java.awt.event.MouseEvent evt) {
   new meterReadings().setVisible(true);
}
```

Again return to the Design screen. Create a ***mouseClicked*** method for the ***Produce bills*** menu option and add a line of code to the method.

```
private void optBillsMouseClicked(java.awt.event.MouseEvent evt) {
   new produceBills().setVisible(true);
}
```

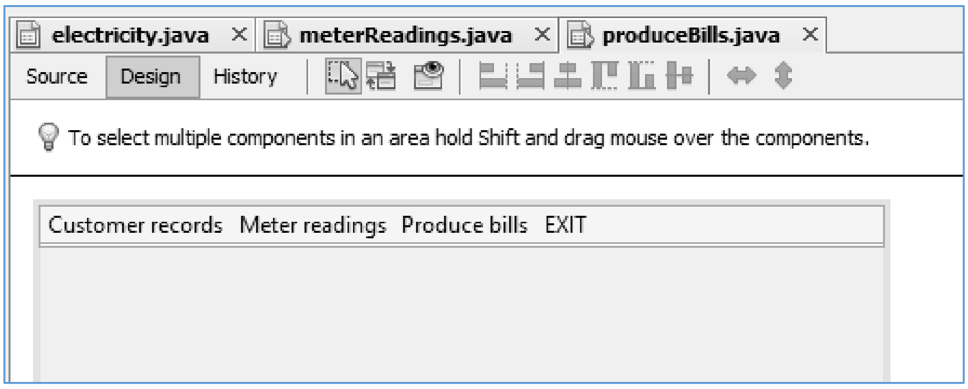
Run the program. Check that new windows are opened when the menu options are clicked.



Return to the NetBeans editing screen and change to the Design view. Create a ***mouseClick*** method for the ***Exit*** menu option and add a line of code.

```
private void optExitMouseClicked(java.awt.event.MouseEvent evt) {
   System.exit(0);
}
```

Change to the ***Design*** view. Select the complete menu bar, then ***copy*** it. This can be done either by right-clicking and selecting ***Copy*** from the drop down menu, or by using the short-cut keys ***Ctrl-C***. Move to each of the other forms and ***paste*** a copy of the menu bar. Select the form, right-click to display a drop-down menu and select ***Paste***, or use the short-cut keys ***Ctrl-V***.



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

Run the program. Check that new windows can be opened with the menu options, and that each window runs the menu bar options correctly. A slight problem is that multiple copies of each of the forms can be created. It would be neater if each window closes as the next window opens.

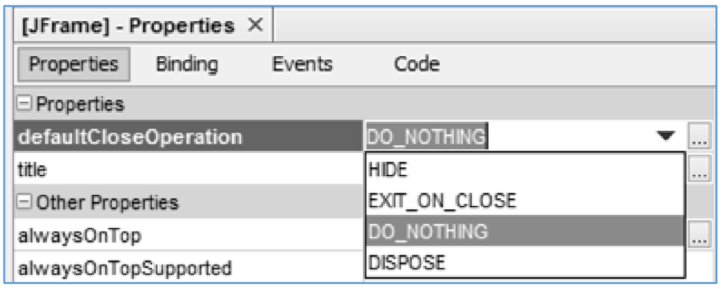
Return to the NetBeans editing screen and change to the ***Source*** program code view for the ***electricity.java*** form. Add a command to the ***mouseClick*** options to close the current form when a new form opens.

```
private void optCustomersMouseClicked(java.awt.event.MouseEvent evt) {

    this.setVisible(false);

    new electricity().setVisible(true);
}

private void optReadingsMouseClicked(java.awt.event.MouseEvent evt) {

    this.setVisible(false);

    new meterReadings().setVisible(true);
}

private void optBillsMouseClicked(java.awt.event.MouseEvent evt) {

    this.setVisible(false);

    new produceBills().setVisible(true);
}
```

These changes need to be applied to the menu bars of the other two forms. It is probably quickest to delete the menu bars from the ***meterReadings*** and ***produceBills*** forms, then recopy the menu bar from the ***electricity.java*** form.

Run the program. Check that the windows now close correctly as each new menu option is selected, and the program ends when the ***Exit*** option is selected.

One final problem is that the whole program ends if the user clicks the cross icon at the top of a form. This should be disabled. Go to the ***Design*** view for ***electricity.java*** and click to select the blank area of form. In the Properties window, set the ***defaultCloseOperation*** property to ***DO_NOTHING***.



defaultCloseOperation DO_NOTHING
title
alwaysOnTop
alwaysOnTopSupported

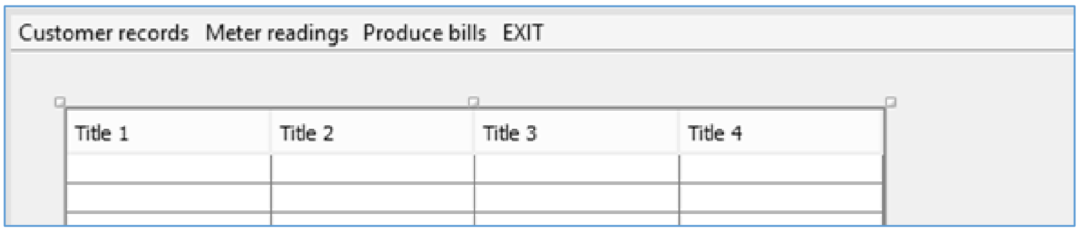
Disable the ***close*** icons for the other two forms in a similar way.

Run the program. Check that it is only possible to close the program by means of the ***Exit*** menu option.

We can now begin work on the input screen for customer records. For simplicity, we will only store five pieces of information:

| | |
|---|---|
| • CustomerID | three digit integer number |
| • Customer surname | string field of 24 characters |
| • Town | string field of 24 characters |
| • Previous meter reading | five digit integer number |
| • Amount owing | decimal (real) number |

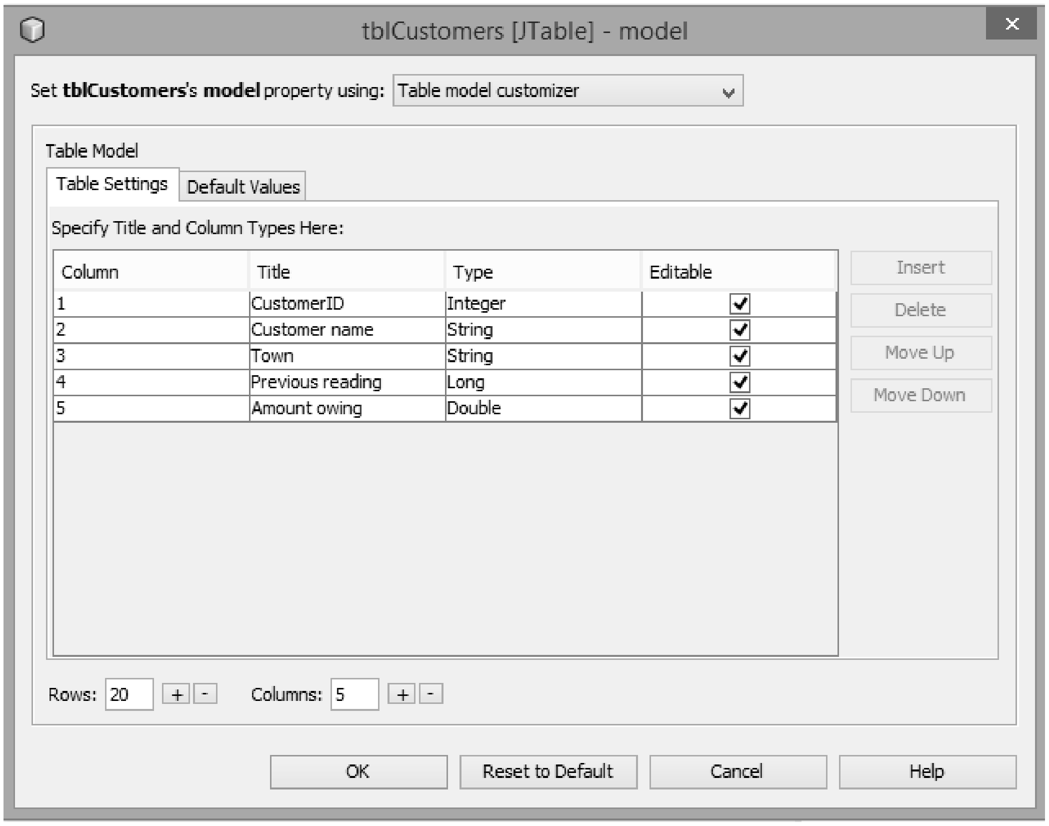
Go to the Design view for the electricity.java form. Add a ***Table*** component. Right-click on the table and change the name to ***tblCustomers***.



Go to the Properties window and locate the model property. Click in the right column to open the table editing window. Set ***Rows*** to **20** and ***Columns*** to **5**. Give the Titles and Data Types as shown below:

| | |
|---|---|
| ***CustomerID*** | ***Integer*** |
| ***Customer Name*** | ***String*** |
| ***Town*** | ***String*** |
| ***Previous reading*** | ***Long*** |
| ***Amount owing*** | ***Double*** |

We will use ***long*** as the data type for the meter readings, as it allows much larger numbers to be stored than a normal integer.



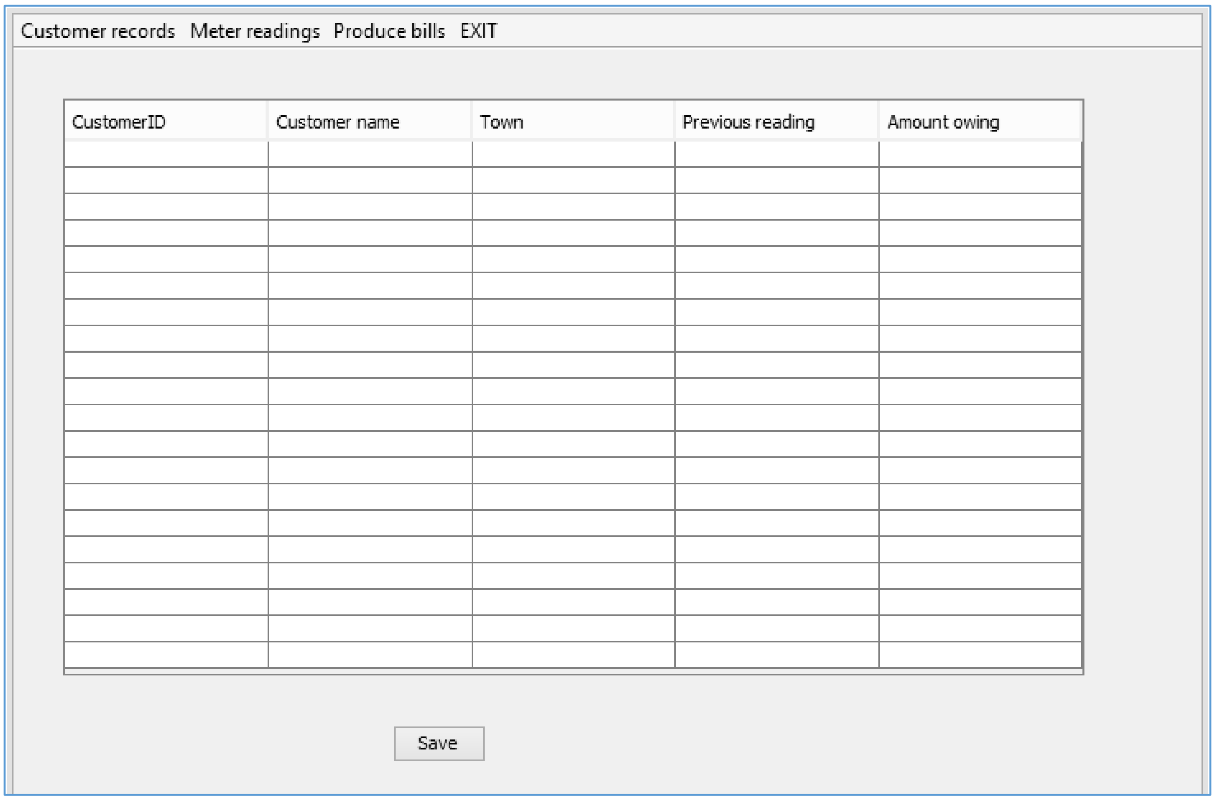
Set **tblCustomers**'s **model** property using: Table model customizer

Specify Title and Column Types Here:

| Column | Title | Type | Editable |
|---|---|---|---|
| 1 | CustomerID | Integer | ✓ |
| 2 | Customer name | String | ✓ |
| 3 | Town | String | ✓ |
| 4 | Previous reading | Long | ✓ |
| 5 | Amount owing | Double | ✓ |

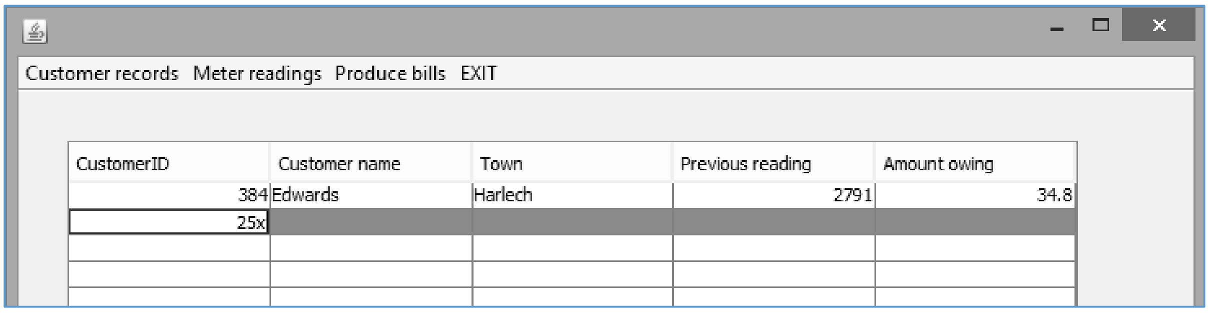
Rows: 20 Columns: 5

Click ***OK*** to return to the ***Design*** view and check that the table headings are shown correctly. Add a button below the table, with the caption '***Save***'. Rename the button as ***btnSave***.



| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|

Run the program. Check that errors can be detected when data is entered in the cells of the table. The ***CustomerID*** and ***Previous reading*** columns will only accept ***integer*** numbers, and the ***Amount owing*** column will only accept ***real*** numbers. If data is entered in an incorrect format, a red outline appears around the cell, and the error must be corrected before further data can be entered.



| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 384 | Edwards | Harlech | 2791 | 34.8 |
| 25x | | | | |

Close the program by clicking the EXIT menu option, and return to the NetBeans editing screen.

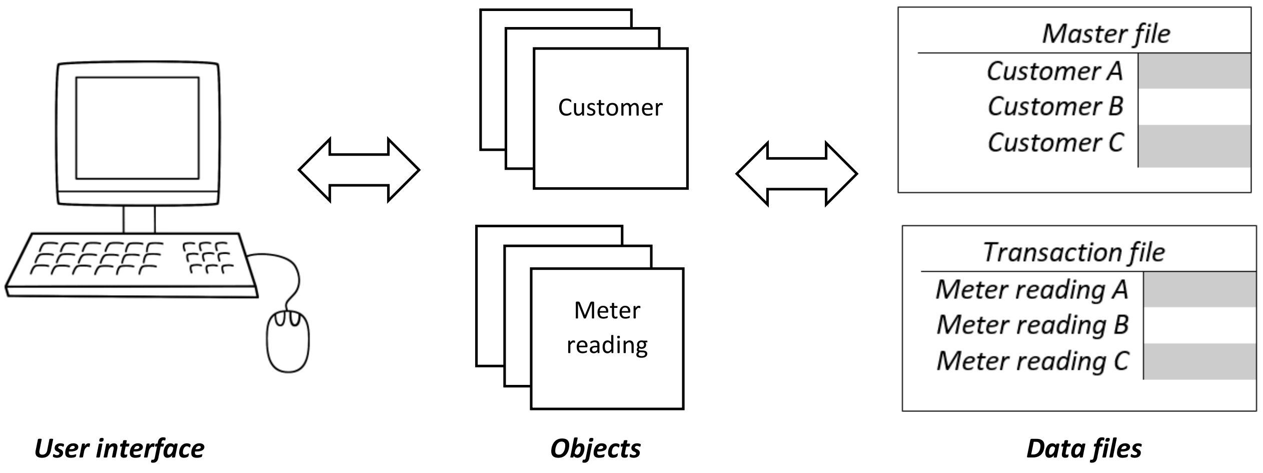
The next step is to save the records from the ***Customer*** table to create the ***master file*** for the batch processing system.

For this project, we use an ***object oriented*** approach. Most large projects are now programmed using classes of objects, as this makes the software more modular in its structure, easier to maintain, and reduces the likelihood of programming errors.

The project will require two classes of objects:

- ***Customer objects***, which will be used to create the ***master file***,
- ***Meter reading objects***, which will be used to create the ***transaction file***.

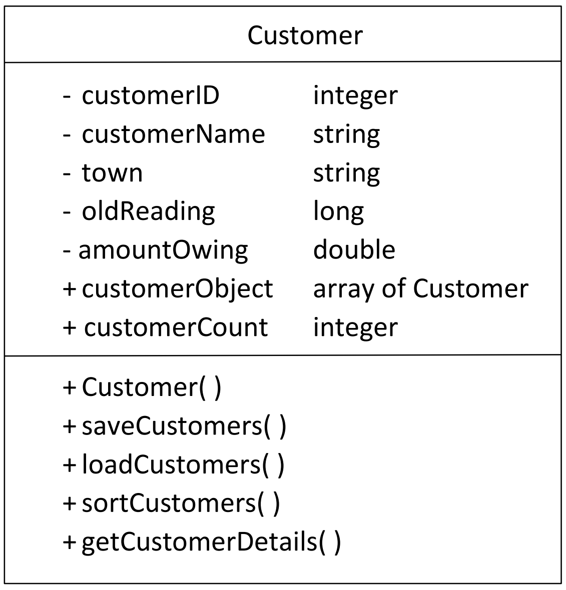
It is an important principle of object oriented programming that the user interface has no direct communication with files stored on disc. This reduces the risk of files being corrupted, or records being accidentally altered or deleted.



All file operations are carried within the object classes, which will contain methods to save and load objects to and from the disc files.

The user interface can display the objects held in the computer memory, and can be used to input data to create additional objects.

We can produce a ***class diagram*** to illustrate the requirements for the ***Customer*** class. ***Properties*** are shown in the box below the object name, and ***methods*** in the bottom box. A ***minus symbol*** indicates ***private*** properties which can only be accessed by methods belonging to this object class. A ***plus symbol*** indicates properties and methods which are ***public*** and can be accessed from other parts of the program.



| Customer | |
|---|---|
| - customerID | integer |
| - customerName | string |
| - town | string |
| - oldReading | long |
| - amountOwing | double |
| + customerObject | array of Customer |
| + customerCount | integer |
| + Customer( ) | |
| + saveCustomers( ) | |
| + loadCustomers( ) | |
| + sortCustomers( ) | |
| + getCustomerDetails( ) | |

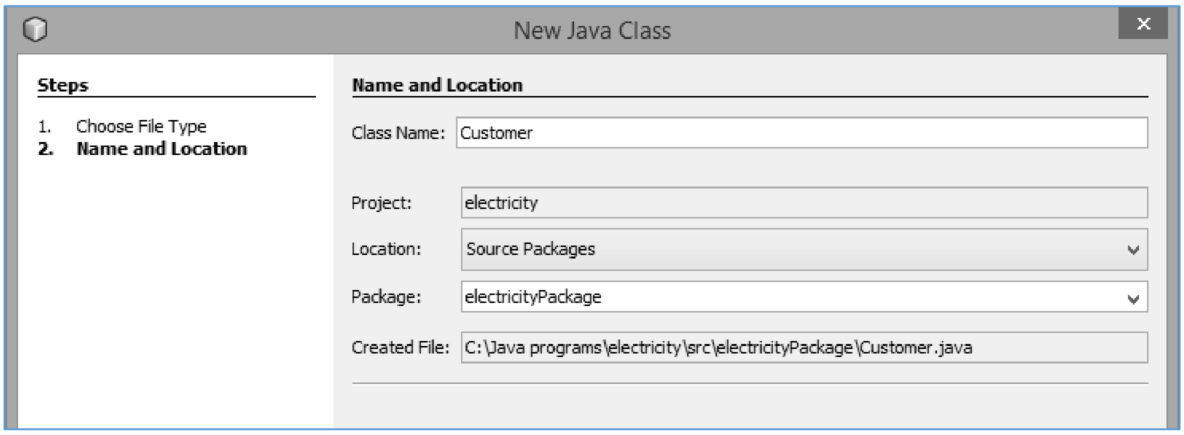
Apart from the five private object properties, we will also use

- a ***customerObject array*** to record the memory locations of Customer objects when they are created,
- an integer variable ***customerCount*** to record the number of Customer objects which have been created.

The Customer class will require methods to carry out various tasks:

- ***Customer( )*** is the ***constructor*** method used to create objects,
- ***saveCustomers( )*** and ***loadCustomers( )*** will copy the Customer objects to and from the disc file,
- ***sortCustomers( )*** will sort the Customer objects into order of ***CustomerID*** number.
- ***getCustomerDetails( )*** will allow the program to access the data in the Customer objects, so that it can be displayed on screen or used in calculating customer bills.

We will now set up the Customer class. Locate ***electricityPackage*** in the Project window at the top left of the screen. Right-click on ***electricityPackage*** and select ***New*** / ***Java Class***. Set the ***Class Name*** as ***Customer***. Leave the ***Package*** name as ***electricityPackage***.



Class Name: Customer
Project: electricity
Location: Source Packages
Package: electricityPackage
Created File: C:\Java programs\electricity\src\electricityPackage\Customer.java

The ***Customer.java*** class file will open. Begin by adding the properties specified in the ***class diagram***. Notice that ***customerCount*** and ***Customer[ ]*** are marked as ***static***, as only ***one copy*** of these variables will exist for the whole class. This is in contrast to the properties relating to individual customers, such as ***customerName*** and ***town***, which are created multiple times as each new ***Customer object*** is added.

```
package electricityPackage;

public class Customer {

    private int customerID;
    private String customerName;
    private String town;
    private long oldReading;
    private double owing;

    public static int customerCount=0;
    public static Customer[] customerObject=new Customer[20];

}
```

Add a filename for the ***master file*** which will store the Customer objects. We can then produce a ***Customer( )*** constructor method . Please note that the header:

***public Customer( …. )***

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

```
    public static int customerCount=0;
    public static Customer[] customerObject=new Customer[20];

    static String filename = "master.dat";

    public Customer(int tCustomerID, String tCustomerName, String tTown,
                                          long tOldReading, double tOwing)
    {
        customerID = tCustomerID;
        customerName = tCustomerName;
        town=tTown;
        oldReading=tOldReading;
        owing=tOwing;
    }

}
```

Return to the ***electricity.java*** form. Select the Source tab to open the program code view. Go to the start of the program and add two Java modules which will be needed for editing the table, and to display error messages.

```
package electricityPackage;

import javax.swing.JOptionPane;
import javax.swing.table.TableCellEditor;

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

Change now to the ***Design*** view. Double click the '***Save***' button below the table to create a button click method.

We will set up a ***makeCustomerObjects( )*** method to read the customer data from the table and create a set of ***Customer objects***. Call this method from the '***Save***' button click method.

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

    makeCustomerObjects();

}

private void makeCustomerObjects()
{

}
```

We will begin the ***makeCustomerObjects( )*** method with definitions for the variables which will be required, then add lines of code to stop the ***table editor*** so that all the data in the table is available for processing.

```
private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
     makeCustomerObjects();
}

private void makeCustomerObjects()
{
    int customerCount=0;
    Integer customerID;
    String customerName;
    String town;
    Long oldReading;
    Double owing;
    TableCellEditor editor = tblCustomers.getCellEditor();
    if (editor != null)
    {
        editor.stopCellEditing();
    }
}
```

Set up a ***TRY … CATCH*** block to detect errors. Inside the ***TRY*** block, we will use a loop to check each of the twenty rows of the table, to see whether a customer record has been entered on that line. The program will know that a record is present if data has been entered in the customerID cell, so this cell value is not ***null***. Note that the line:

***JOptionPane.showMessageDialog( ... )***

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

```
    if (editor != null)
    {
        editor.stopCellEditing();
    }
    try
    {
        for (int i=0; i<20; i++)
        {
            if (!(tblCustomers.getModel().getValueAt(i,0)==null))
            {

            }
        }
    }
    catch(NullPointerException e)
    {
        JOptionPane.showMessageDialog(electricity.this,
                           "Not all the required data has been entered");
    }
}
```

We will now add lines of code to the loop to collect the customer data from the table row, and use this to create a Customer object.

***Tables*** in Java NetBeans return numbers in special formats written as ***Integer***, ***Long*** and ***Double***, rather than the number formats ***int***, ***long*** and ***double*** which we normally use. To access the actual number values, it is necessary to add the functions ***intValue( )***, ***longValue( )*** or ***doubleValue( )*** to the variable names, for example: ***customerID.intValue( )***.

Please note that the command:

***Customer.customerObject[customerCount] = new Customer ( ... );***

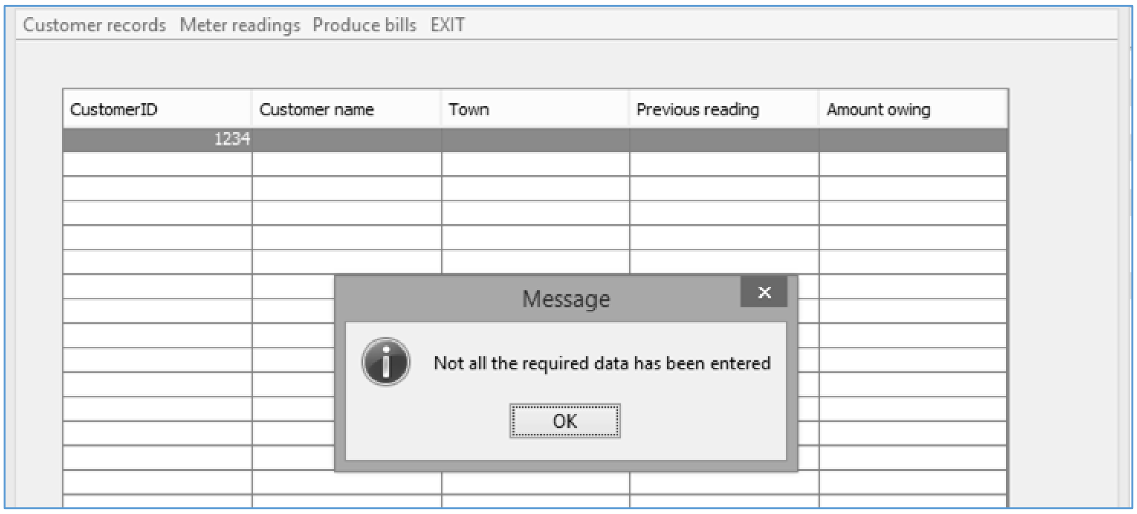
and lines beginning:

***JOptionPane.showMessageDialog(…***

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

```
try
{
    for (int i=0; i<20; i++)
    {
        if (!(tblCustomers.getModel().getValueAt(i,0)==null))
        {
            customerID = (Integer) tblCustomers.getModel().getValueAt(i,0);
            customerName=(String) tblCustomers.getModel().getValueAt(i,1);
            town=(String) tblCustomers.getModel().getValueAt(i,2);
            oldReading=(Long) tblCustomers.getModel().getValueAt(i,3);
            owing=(Double) tblCustomers.getModel().getValueAt(i,4);
            if (customerID<100 || customerID>999)
            {
                JOptionPane.showMessageDialog(electricity.this,
                                        "CustomerID must be three digits");
                tblCustomers.getModel().setValueAt("",i,0);
            }
            else
            {
                if (oldReading<10000 || oldReading>99999)
                {
                    JOptionPane.showMessageDialog(electricity.this,
                                        "Previous reading must be five digits");
                    tblCustomers.getModel().setValueAt("",i,3);
                }
                else
                {
                    Customer.customerObject[customerCount] =
                            new Customer(customerID.intValue(), customerName,
                            town, oldReading.longValue(), owing.doubleValue());
                    customerCount++;
                }
            }
        }
    }
    Customer.customerCount=customerCount;
}
catch(NullPointerException e)
```

Run the program. Check the error trapping. Enter partial data for a customer, then click the '***Save***' button. An error message should appear, to indicate that not all the required data has been entered.

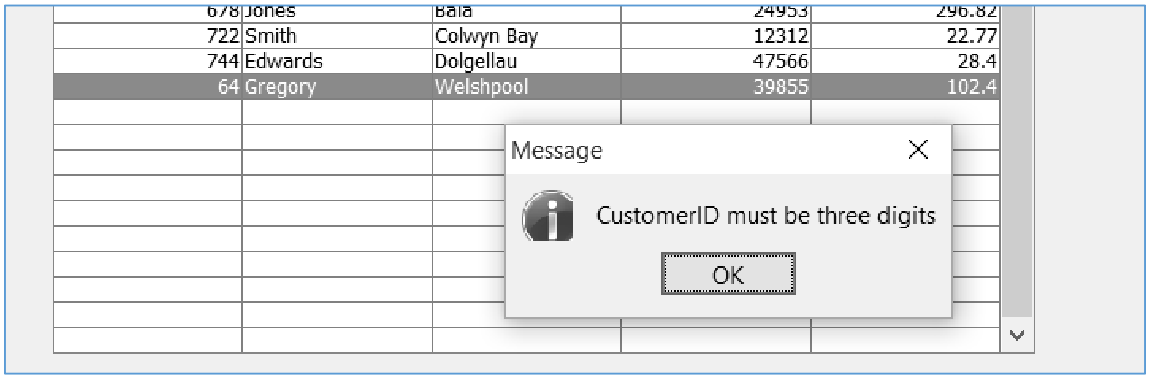


| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 1234 | | | | |

Message

Not all the required data has been entered

The ***CustomerID*** should be three digits in length. Enter a CustomerID number which is less or more than three digits, and check that an error message is displayed.

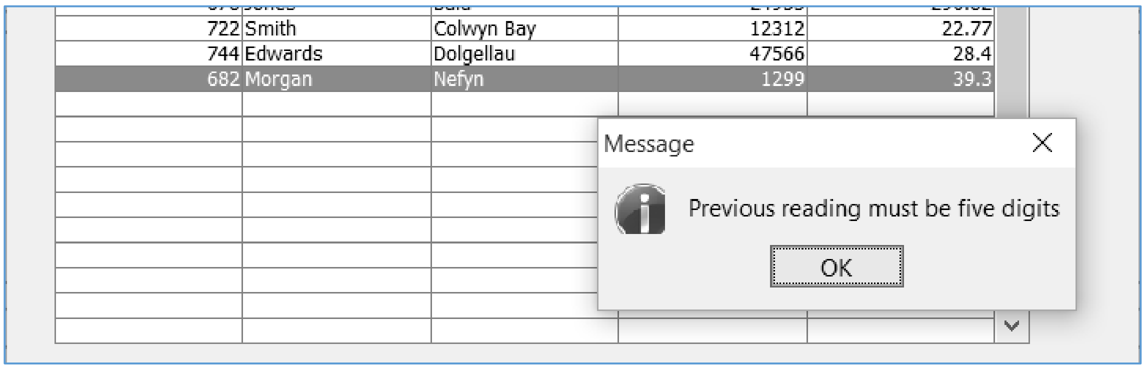


| | | | | |
|---|---|---|---|---|
| 678 | Jones | Bala | 24953 | 296.82 |
| 722 | Smith | Colwyn Bay | 12312 | 22.77 |
| 744 | Edwards | Dolgellau | 47566 | 28.4 |
| 64 | Gregory | Welshpool | 39855 | 102.4 |

Message

CustomerID must be three digits

The ***Previous reading*** should be five digits in length. Enter a Previous reading which is less or more than five digits, and check that an error message is displayed.



| | | | | |
|---|---|---|---|---|
| 722 | Smith | Colwyn Bay | 12312 | 22.77 |
| 744 | Edwards | Dolgellau | 47566 | 28.4 |
| 682 | Morgan | Nefyn | 1299 | 39.3 |

Message

Previous reading must be five digits

A complete record with correct data in each field should be accepted without an error message.

Use the ***Exit*** menu option to return to the NetBeans editing page.

We now need to carry out the opposite task of reading the set of Customer objects and displaying them in the table. Begin by going to the ***Customer.java*** class file and adding methods to make the data values available.

```
        public Customer(int tCustomerID, String tCustomerName, String tTown,
                                                 long tOldReading, double tOwing)
        {
            customerID = tCustomerID;
            customerName = tCustomerName;
            town=tTown;
            oldReading=tOldReading;
            owing=tOwing;
        }

        public int getCustomerID()
        {
           return customerID;
        }

        public String getCustomerName()
        {
           return customerName;
        }

        public String getTown()
        {
           return town;
        }

        public long getOldReading()
        {
           return oldReading;
        }

        public double getOwing()
        {
           return owing;
        }

    }
```

Return to the electricity.java program code page. Immediately after the ***makeCustomerObjects( )*** method, add a ***displayCustomers( )*** method. Include the variables which will be needed.

```
        private void displayCustomers()
        {
            int customerID;
            String customerName;
            String town;
            long oldReading;
            double owing;
        }
```

Add loops to clear the table by resetting each of the cell values to ***null***.

```
        String town;
        long oldReading;
        double owing;

         for (int i=0; i<20; i++)
         {
             for (int j=0; j<5; j++)
             {
                 tblCustomers.getModel().setValueAt(null,i,j);
             }
         }

    }
```

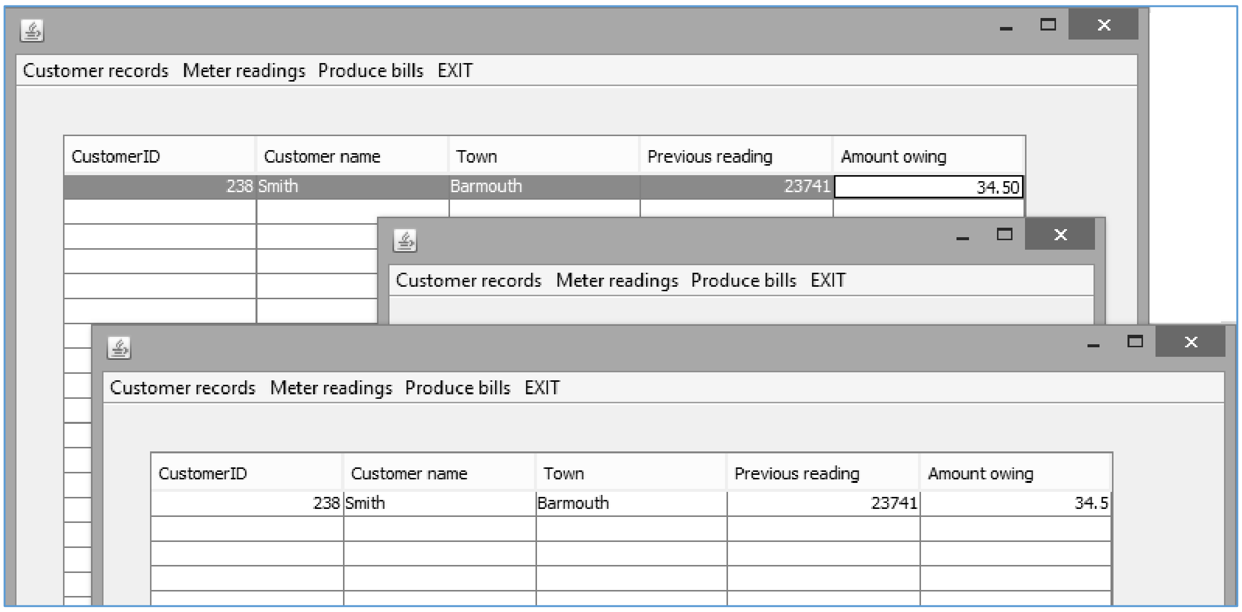
We will now use a loop to access each of the ***Customer objects*** and display the field values in the table.

```
        for (int i=0; i<20; i++)
        {
            for (int j=0; j<5; j++)
            {
                tblCustomers.getModel().setValueAt(null,i,j);
            }
        }

        if (Customer.customerCount>0)
        {
            for (int i=0; i<Customer.customerCount; i++)
            {
                customerID=Customer.customerObject[i].getCustomerID();
                tblCustomers.getModel().setValueAt(customerID,i,0);

                customerName = Customer.customerObject[i].getCustomerName();
                tblCustomers.getModel().setValueAt(customerName,i,1);

                town = Customer.customerObject[i].getTown();
                tblCustomers.getModel().setValueAt(town,i,2);

                oldReading = Customer.customerObject[i].getOldReading();
                tblCustomers.getModel().setValueAt(oldReading,i,3);

                owing = Customer.customerObject[i].getOwing();
                tblCustomers.getModel().setValueAt(owing,i,4);
            }
        }

    }
```

We will call the ***displayCustomers( )*** method when the form opens. Go to the start of the program listing and locate the ***electricity( )*** method, then add the line of code.

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

    public electricity() {
        initComponents();

        displayCustomers();

    }
```

Run the program. Enter a customer record, then click the '***Save***' button. Use the menu options to move to different pages, then return to the ***Customer records*** page. A new form will open. Check that the record entered earlier is correctly displayed.



| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 238 | Smith | Barmouth | 23741 | 34.50 |

| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 238 | Smith | Barmouth | 23741 | 34.5 |

Use the ***Exit*** menu option to return to the NetBeans editing screen.

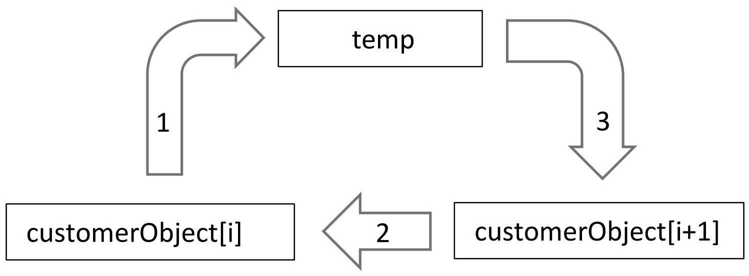
As we will see later in this chapter, it is necessary for both the ***master records*** and the ***transaction records*** to be ***sorted*** into order of ***customerID*** before the electricity bills are produced. We will now set up a method to sort the ***Customer objects***.

Go to the ***Customer*** class file and create a ***sort( )*** method below the ***getCustomerID( )*** method. Add a line to produce an empty Customer object called ***temp*** which we will use in the sort procedure.

```
    public int getCustomerID()
    {
       return customerID;
    }

    public static void sort()
    {
        Customer temp=new Customer(0,"","",0,0);
    }
```

We will sort the Customer objects using a ***bubble sort***. We can decide whether any two objects need to be swapped by comparing the ***customerID*** values. The objects should be sorted into ascending order of ***customerID*** number.

Complete objects, including all of their properties, can be copied in a single operation. The customer objects are held in the ***customerObject[ ]*** array. If we find that the objects at ***position [i]*** and ***position [i+1]*** need to be swapped, this can be done using a ***triangular exchange*** via the ***temp*** object we have created:



***temp = customerObject[i];***
***customerObject[i] = customerObject[i+1];***
***customerObject[i+1] = temp;***

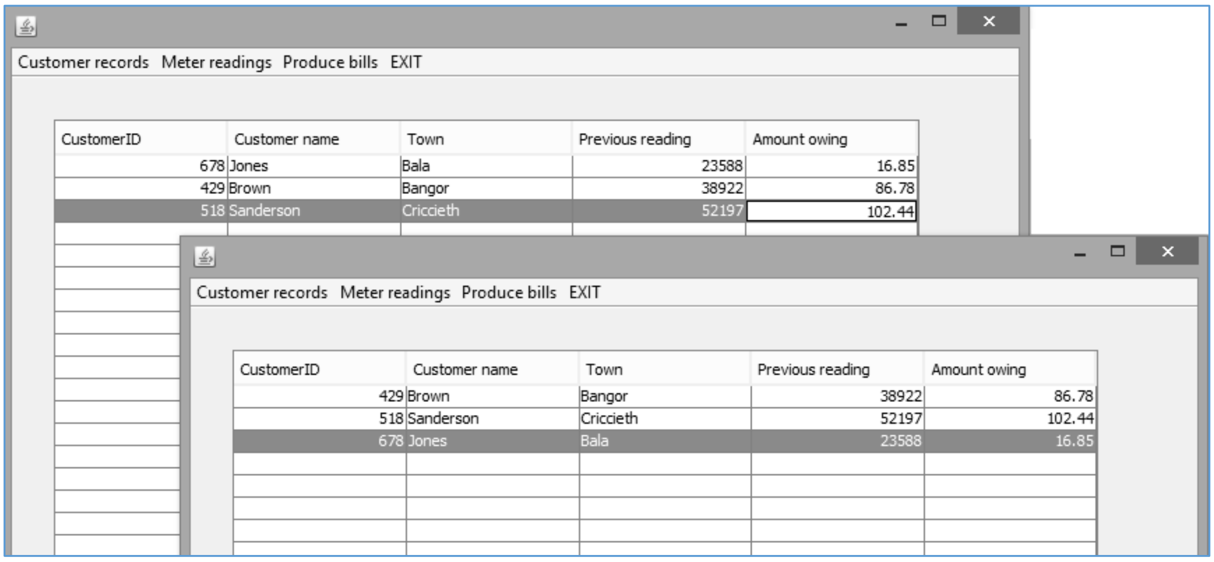
Add lines to the ***sort( )*** method which will obtain the ***customerID*** values for each pair of objects, compare the values, and carry out the triangular exchange if necessary. The Boolean variable '***swap***' will indicate when the sorting has been completed.

```
public static void sort()
{
    Customer temp=new Customer(0,"","",0,0);
    Boolean swap=true;
    while (swap==true)
    {
        swap=false;
        for (int i=0; i<customerCount-1;i++)
        {
            int customerID1 = customerObject[i].customerID;
            int customerID2 = customerObject[i+1].customerID;
            if (customerID1>customerID2)
            {
                swap=true;
                temp=customerObject[i];
                customerObject[i]=customerObject[i+1];
                customerObject[i+1]=temp;
            }
        }
    }
}
```

Return to the ***electricity.java*** page. Use the ***Design*** tab to move to the form layout view. Double click the '***Save***' button to open the button click method. This currently contains the line of code to create Customer objects from the table data. Add commands to sort the Customer objects, and redisplay the sorted data in the table.

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

     Customer.sort();
     displayCustomers();

 }
```

Run the program. Enter several customer records, then click the '***Save***' button. The records should be sorted into order of ***customerID***, then redisplayed in the table.



| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 678 | Jones | Bala | 23588 | 16.85 |
| 429 | Brown | Bangor | 38922 | 86.78 |
| 518 | Sanderson | Criccieth | 52197 | 102.44 |

| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 429 | Brown | Bangor | 38922 | 86.78 |
| 518 | Sanderson | Criccieth | 52197 | 102.44 |
| 678 | Jones | Bala | 23588 | 16.85 |

Close the program using the ***Exit*** menu option. Return to the ***Customer*** class.

We will add a method to save the set of ***Customer*** objects to a disc file. Begin by adding Java modules at the start of the class, which will be needed for the file operations.

```
package electricityPackage;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

public class Customer {
           private int customerID;
           private String customerName;
```

Insert the ***saveCustomers( )*** method below the ***sort( )*** method. We will aAdd definitions for variables which will be required. We will then delete any existing ***master.dat*** file, ready to replace it a new file.

```
public static void saveCustomers()
{
    String fCustomerID;
    String fCustomerName;
    String fTown;
    String fOldReading;
    String fOwing;
    File oldfile = new File(filename);
    oldfile.delete();
}
```

We will use ***fixed length records*** for the master file. Set up a ***TRY … CATCH*** structure to handle any file errors, then add a loop to save each of the customer records.

```
    File oldfile = new File(filename);
    oldfile.delete();

    try (RandomAccessFile file = new RandomAccessFile(filename, "rw"))
    {
        for (int i=0; i<customerCount; i++)
        {


        }
        file.close();
    }
    catch(IOException e)
    {

    }
}
```

It is possible to store the number fields of each record directly in ***integer***, ***long*** or ***double*** format. Using binary number formats can save a small amount of file space, but has the disadvantage that the file is no longer readable if displayed in a text editing application such as Notepad:

```
CNGgm?/kvd34   kYVkyorN1h /]uTXRhesQT  ^&gSJ|\\o4JubxF  nihO6v0E8x
```

Scientists, engineers and other users of large computing applications generally prefer numbers to be converted to text format for storing in data files. The numbers are then displayed in a normal readable form:

```
3589          276922          38.45              178.50
```

This makes it easier to directly check the contents of the data file and identify any errors. We will use this approach in the current program.

Each customer record has five fields. We can set the lengths in bytes for each field, and add an ***'end of record'*** marker.

- ***CustomerID*** is a three digit number, so can be represented as three text characters.
- ***Previous reading*** is a five digit number, so can be represented as five text characters.
- ***Amount owing*** is a number with two decimal places, representing pounds and pence. This can be stored within an eight character text field.
- The lengths of the ***Customer name*** and ***Town*** fields are each set as 24 characters.

| CustomerID | Customer Name | Town | Previous reading | Amount owing | *** |
|---|---|---|---|---|---|
| 3 bytes | 24 bytes | 24 bytes | 5 bytes | 8 bytes | 3 bytes |

This gives a total record length of 67 bytes.

Add lines of code to the loop which will:

- Obtain the data for each field from the current Customer object.
- Where necessary, convert number fields into text format.
- Set the correct field lengths, adding blank spaces if necessary.
- Save the data into the file on disc.

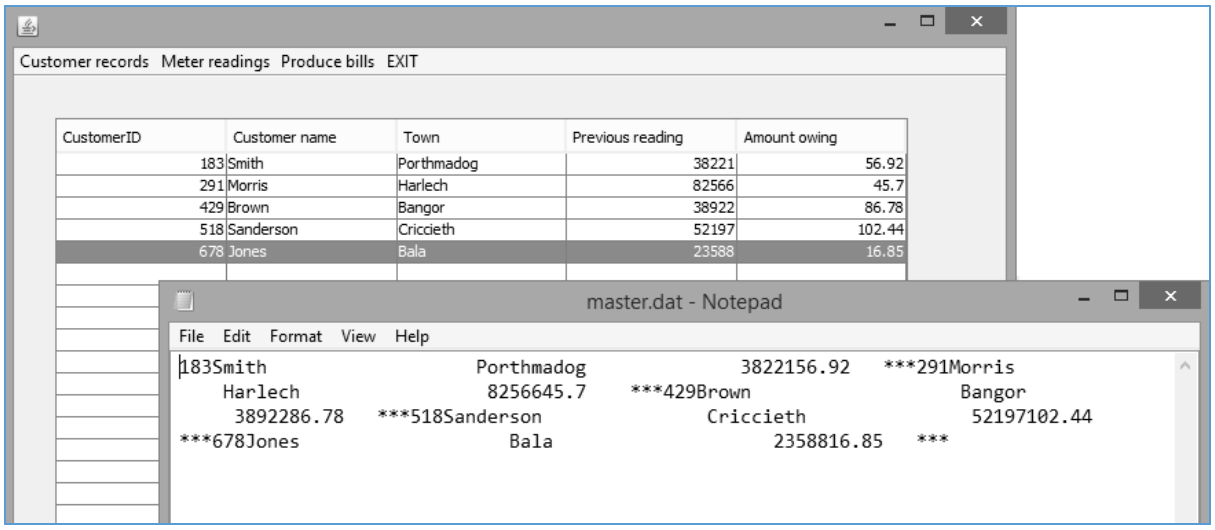
```
        try (RandomAccessFile file = new RandomAccessFile(filename, "rw"))
        {
            for (int i=0; i<customerCount; i++)
            {
               fCustomerID=String.valueOf(customerObject[i].customerID);
               fCustomerID=String.format("%-3s",fCustomerID);
               file.write(fCustomerID.getBytes());

               fCustomerName=customerObject[i].customerName;
               fCustomerName=String.format("%-24s",fCustomerName);
               file.write(fCustomerName.getBytes());

               fTown=customerObject[i].town;
               fTown=String.format("%-24s",fTown);
               file.write(fTown.getBytes());

               fOldReading=String.valueOf(customerObject[i].oldReading);
               fOldReading=String.format("%-5s",fOldReading);
               file.write(fOldReading.getBytes());

               fOwing=String.valueOf(customerObject[i].owing);
               fOwing=String.format("%-8s",fOwing);
               file.write(fOwing.getBytes());

               file.write("***".getBytes());
            }
            file.close();
        }
        catch(IOException e)
        {

        }
    }
```

Move to the ***electricity.java*** page and open the Source code view. Locate the ***btnSave( )*** method, and add a line of code to call ***saveCustomers( )***.

```
    private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
            makeCustomerObjects();
            Customer.sort();
            displayCustomers();

            Customer.saveCustomers();

        }
```

Run the program. Enter some test data, ensuring that ***CustomerID*** values are ***3 digit numbers*** and ***Previous reading*** values are ***5 digit numbers***.

Use ***Windows Explorer*** to locate the ***master.dat*** file in the ***electricity*** program folder. Open ***master.dat*** in a text editing application such as ***Notepad***, and check that the test data has been stored correctly.



| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 183 | Smith | Porthmadog | 38221 | 56.92 |
| 291 | Morris | Harlech | 82566 | 45.7 |
| 429 | Brown | Bangor | 38922 | 86.78 |
| 518 | Sanderson | Criccieth | 52197 | 102.44 |
| 678 | Jones | Bala | 23588 | 16.85 |

```
183Smith                    Porthmadog                  3822156.92    ***291Morris
    Harlech                   8256645.7    ***429Brown                    Bangor
     3892286.78    ***518Sanderson            Criccieth                  52197102.44
***678Jones                      Bala                        2358816.85    ***
```

Close the program by selecting the ***Exit*** menu option. Click the tab at the top of the NetBeans screen to return to the ***Customer.java*** class.

We will now add a method to re-load the customer records from the ***master.dat*** file. Create a ***loadCustomers( )*** method below ***saveCustomers( )***. Add definitions for the variables which will be required.

```
public static void loadCustomers()
{
    int position;
    String fCustomerID;
    String fCustomerName;
    String fTown;
    String fOldReading;
    String fOwing;
}
```

Add a ***TRY … CATCH*** structure to handle any file errors, then begin a loop to load each customer record. Each record has a fixed length of 67 bytes, the number of records can be calculated by dividing the total file size by 67.

```
        String fOldReading;
        String fOwing;

        try
        {
            RandomAccessFile file = new RandomAccessFile(filename, "r");
            customerCount=(int) file.length()/67;
            for (int i=0; i<customerCount; i++)
            {

            }
            file.close();
        }
        catch(IOException e)
        {

        }
    }
```

We calculate the file position for each record using:

***[file location] = [record sequence number] * [record length]***

A block of 67 bytes is loaded at this point, split into the separate fields, then the data is used to create a ***Customer object***. Add the lines of code below, remembering that the command:

***customerObject[i] = new Customer( … )***

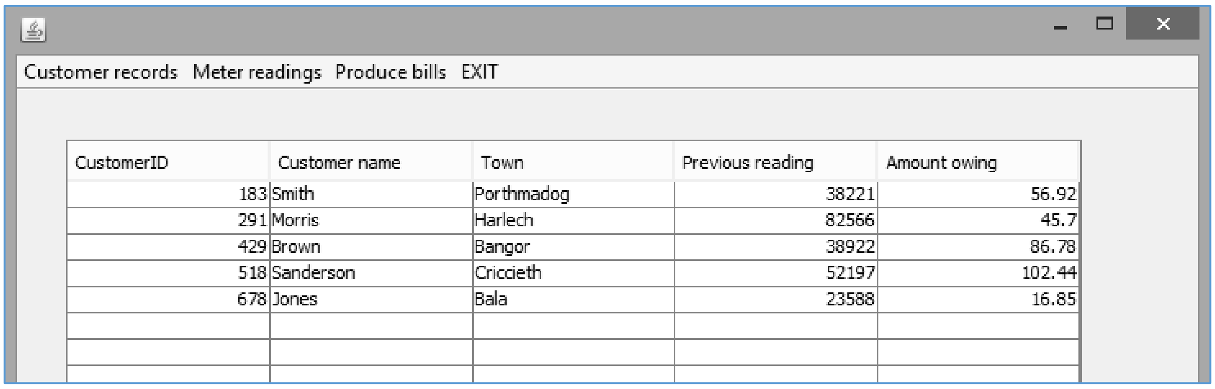
should be entered as a single line of code without line breaks.

```
RandomAccessFile file = new RandomAccessFile(filename, "r");
customerCount=(int) file.length()/67;
 for (int i=0; i<customerCount; i++)
 {
     position=i*67;
     file.seek(position);
     byte[] bytes = new byte[67];
     file.read(bytes);
     String s=new String(bytes);
     fCustomerID=s.substring(0,3); s=s.substring(3);
     fCustomerName=s.substring(0,24).trim(); s=s.substring(24);
     fTown=s.substring(0,24).trim(); s=s.substring(24);
     fOldReading=s.substring(0,5); s=s.substring(5);
     fOwing=s.substring(0,8);
     customerObject[i] = new Customer(Integer.parseInt(fCustomerID), fCustomerName,
                   fTown, Long.parseLong(fOldReading), Double.parseDouble(fOwing));
 }
 file.close();
```

Return to the ***electricity.java*** Source code page. Locate the ***electricity( )*** method and a line to call the ***loadCustomers( )*** method.

```
    public electricity() {
        initComponents();

        Customer.loadCustomers();

        displayCustomers();
    }
```

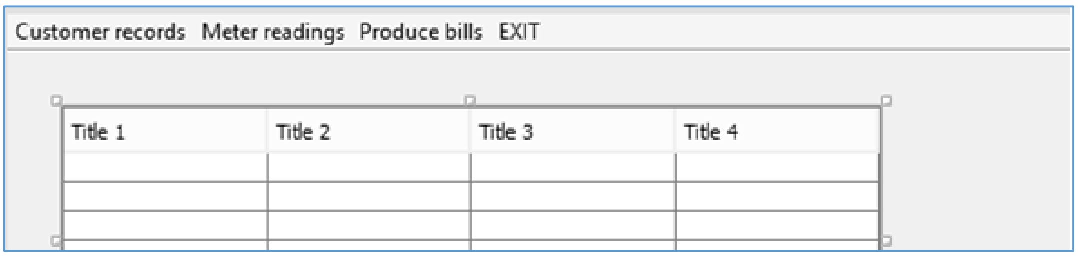
Run the program. Customer records should now be immediately displayed in the table.



| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 183 | Smith | Porthmadog | 38221 | 56.92 |
| 291 | Morris | Harlech | 82566 | 45.7 |
| 429 | Brown | Bangor | 38922 | 86.78 |
| 518 | Sanderson | Criccieth | 52197 | 102.44 |
| 678 | Jones | Bala | 23588 | 16.85 |

Close the program with the ***Exit*** menu option. Select the ***meterReadings.java*** page and click the ***Design*** tab if necessary to open the form layout view.

Add a table to the form and rename this as ***tblReadings***.



Go to the Properties window and locate the model property. Click in the right column to open the table editing window. Set ***Rows*** to **0** and ***Columns*** to **2**. Give the Titles and Data Types as shown below:

| | |
|---|---|
| ***Customer*** | ***String*** |
| ***New meter reading*** | ***Long*** |

Remove the ticks from the ***Editable*** column as shown, then click the ***OK*** button.

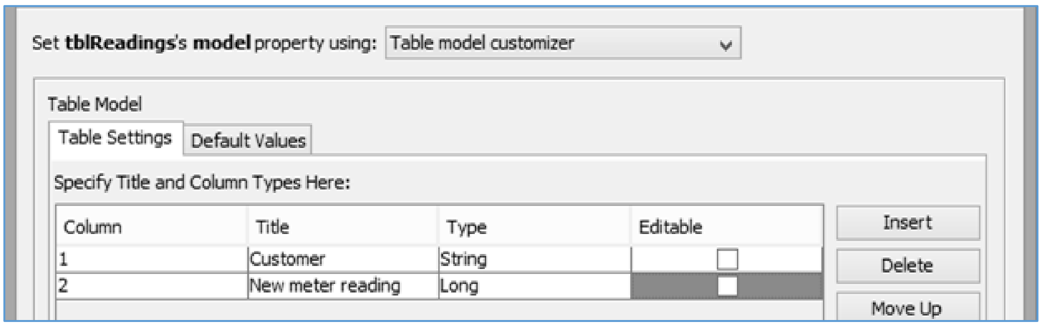

Set **tblReadings's model** property using: Table model customizer

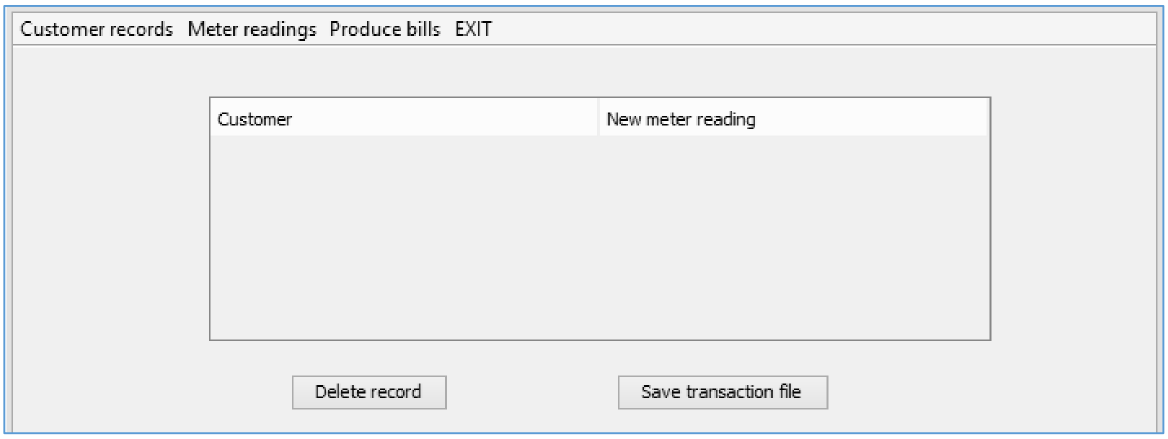
Table Model

Specify Title and Column Types Here:

| Column | Title | Type | Editable |
|---|---|---|---|
| 1 | Customer | String | ☐ |
| 2 | New meter reading | Long | ☐ |

Run the program and select the ***Meter readings*** menu option. Check that an empty table is displayed with the correct headings.

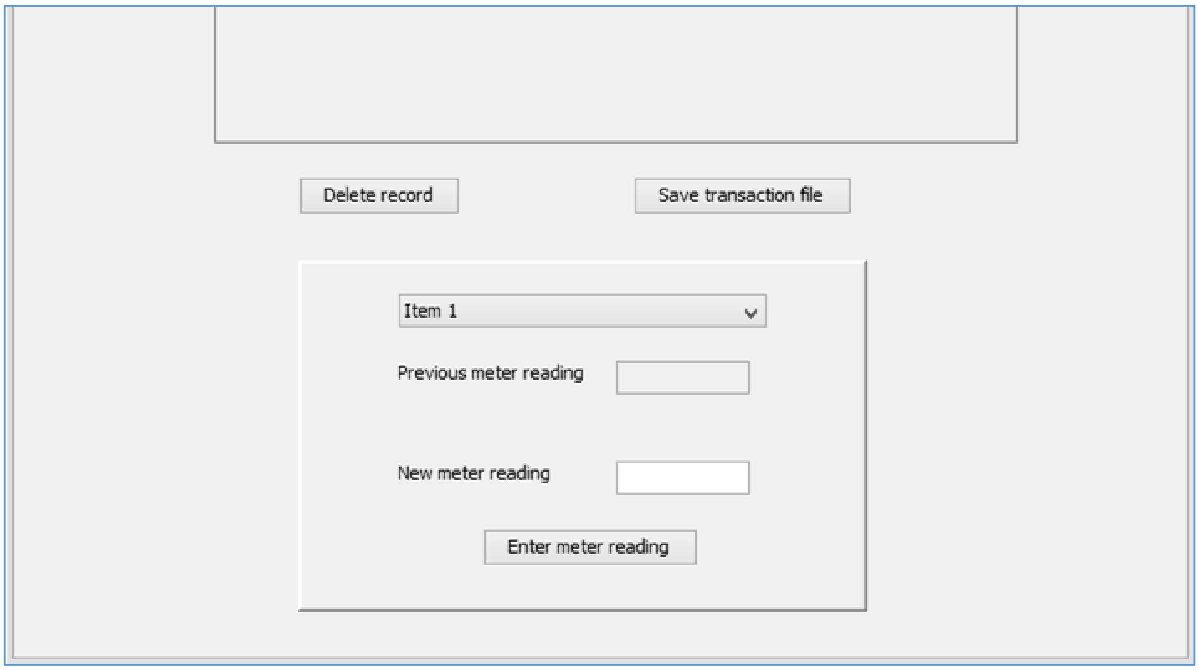
Close the program window to return to the form layout screen. Add buttons below the table with the captions '***Delete record***' and '***Save transaction file***'. Give the buttons the names ***btnDelete*** and ***btnSave***.



Add a ***Panel*** to the form. Set the border property to ***BevelBorder***. Right-click the panel and select ***Set Layout*** / ***Absolute Layout***.

Add components to the panel:

- A ***Combo Box***. Rename this as ***cmbCustomers***.
- A label with the caption '***Previous meter reading***'. Add a text field alongside, with the name ***txtOldReading***. Delete the text which is shown in the box. Locate the ***enabled*** property for the text box and remove the tick.
- A label with the caption '***New meter reading***'. Add a text field alongside, with the name ***txtNewReading***. Delete the text which is shown in the box.
- A button with the caption '***Enter meter reading***'. Rename the button as ***btnEnter***.



When meter readings are collected from customers' homes, the data will be entered into the table. It will be convenient to identify the customer by their ID number, followed by their name, for example:

**183 Smith**

The new meter reading will be stored as a long integer to allow for large five digit numbers, for example:

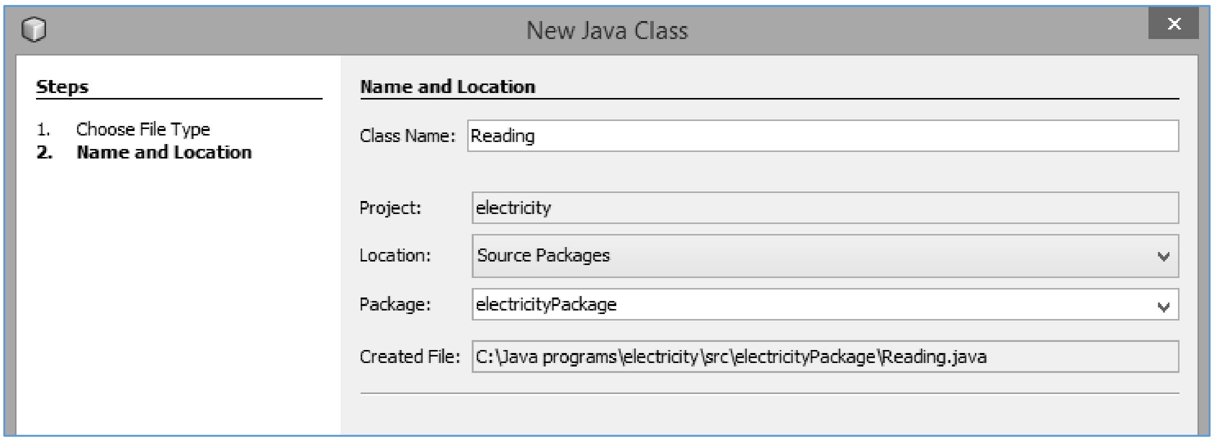
**38221**

Using an object oriented approach, we will create Reading objects to handle this data. The required properties and methods can be shown in a class diagram.

| Reading | |
|---|---|
| - customer | string |
| - newReading | long |
| + readingObject | array of Reading |
| + readingCount | integer |
| + Reading( ) | |
| + saveReadings( ) | |
| + loadReadings( ) | |
| + sortReadings( ) | |
| + getReadingDetails( ) | |

Before carrying out further work on the input form, we will create the Reading class.

Locate ***electricityPackage*** in the Project window at the top left of the screen. Right-click on ***electricityPackage*** and select ***New*** / ***Java Class***. Set the ***Class Name*** as ***Reading***. Leave the ***Package*** name as ***electricityPackage***.


New Java Class

Steps
1. Choose File Type
2. **Name and Location**

Name and Location

Class Name: Reading

Project: electricity

Location: Source Packages

Package: electricityPackage

Created File: C:\Java programs\electricity\src\electricityPackage\Reading.java

The ***Reading.java*** class file will open.

We will begin by adding Java modules which will be required for file handling.

We will then define the properties specified in the ***class diagram***. Notice that ***readingCount*** and ***Reading[ ]*** are marked as ***static***, as only ***one copy*** of these variables will exist for the whole class. This is in contrast to the properties of individual objects, ***customer*** and ***newReading***, which are created multiple times as each new ***Reading object*** is added.

The file name '***transaction.dat***' will be given to the file which will store the meter readings. Finally, add the ***Reading( )*** constructor method which will the create objects when the program runs.

```
package electricityPackage;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

public class Reading {

    private String customer;
    private long newReading;
    public static int readingCount=0;
    public static Reading[] readingObject=new Reading[50];
    static String filename = "transaction.dat";

    public Reading(String tCustomer, long tNewReading)
    {
        customer = tCustomer;
        newReading=tNewReading;
    }

}
```

Return to the ***meterReadings.java*** page and select the ***Source*** program code view.

Go to the start of the program listing and add Java modules which will be needed by the Combo box and Table components.

```
package electricityPackage;

import javax.swing.DefaultComboBoxModel;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableCellEditor;

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

When the ***meterReadings*** form opens, the program can produce a drop down list of all ***customer ID numbers*** and ***customer names*** in the ***Combo Box***.

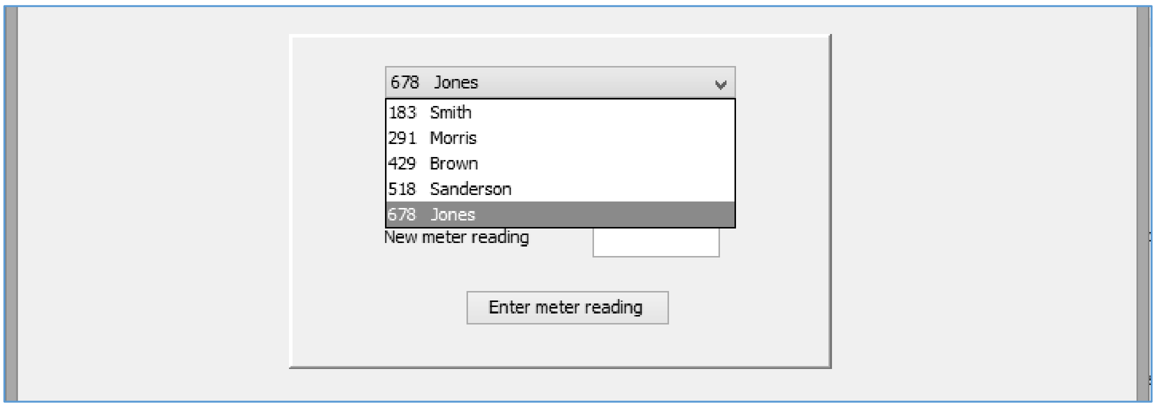
Locate the ***meterReadings( )*** method and add a line to call a new method called ***loadCustomers( )***.

```
public meterReadings() {
    initComponents();

    loadCustomers();

}
```

Add the ***loadCustomers( )*** method immediately below. This begins by clearing the Combo Box, then uses a loop to add an entry for each customer.

```
    public meterReadings() {
        initComponents();

        loadCustomers();

    }

    private void loadCustomers()
    {
        String s;
        int customerID;
        DefaultComboBoxModel model = (DefaultComboBoxModel)cmbCustomers.getModel();
        model.removeAllElements();
        for (int i=0;i<Customer.customerCount; i++)
        {
            customerID=Customer.customerObject[i].getCustomerID();
            s= Integer.toString(customerID);
            s+="   "+Customer.customerObject[i].getCustomerName();
            model.addElement(s.trim());
        }
    }
```

Run the program. Select the ***Meter readings*** menu option, then check that the customer ID numbers and customer names are shown correctly in the drop down list.


```
678  Jones
183  Smith
291  Morris
429  Brown
518  Sanderson
678  Jones
```

Close the program and return to the ***meterReadings.java*** page. Use the Design tab to move to the form layout view.

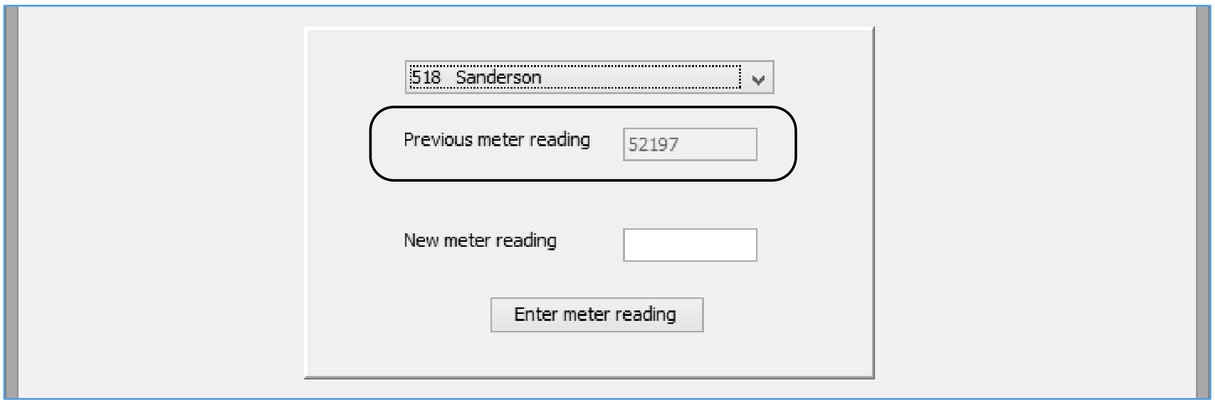
When the user selects a customer from the drop down list, the program should display the ***previous meter reading*** in a text field.

Double click on the ***Combo Box*** to produce a method which will operate when a list item is selected.

Add code which checks the ***index position*** of the ***selected item*** in the drop down list, then obtains the meter reading from the corresponding ***Customer object***.

```
private void cmbCustomersActionPerformed(java.awt.event.ActionEvent evt) {
    int n=cmbCustomers.getSelectedIndex();
    if (n>=0)
    {
       String s=Long.toString(Customer.customerObject[n].getOldReading());
       txtOldReading.setText(s);
       txtNewReading.setText("");
    }
}
```

Run the program and go to the Meter readings page. Check that previous meter readings are displayed correctly when customers are selected from the drop down list.



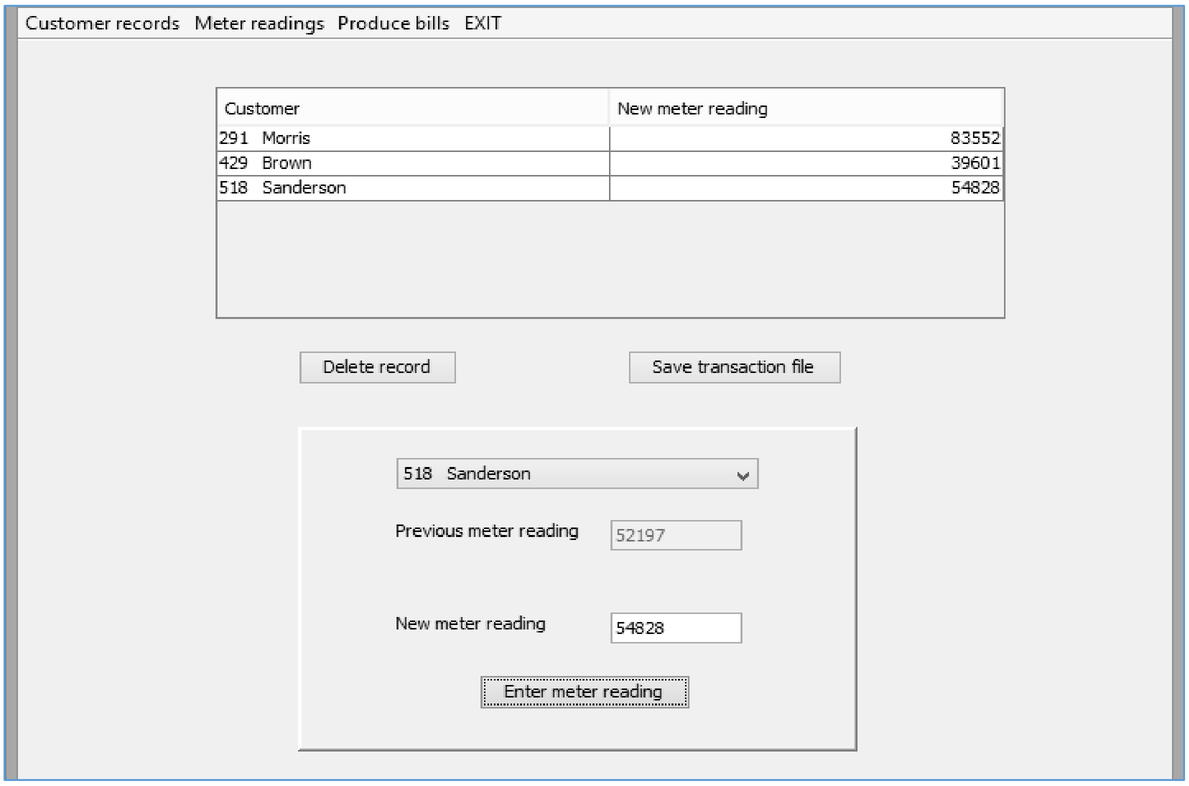
Close the program and return to the ***meterReadings.java*** screen. Use the ***Design*** tab to change back to the form layout view.

We can now transfer data from the panel to the table. Double click the '***Enter meter reading***' button to create a method. We will add lines of code which will:

- Collect the customer ID and name from the Combo Box.
- Collect the new meter reading from the text field.
- Create a row of data from these items.
- Insert the row of data into the table.

```
private void btnEnterActionPerformed(java.awt.event.ActionEvent evt) {
    String customer = (String) cmbCustomers.getSelectedItem();
    long newReading= Long.valueOf(txtNewReading.getText());
    Object[] row = { customer, newReading};
    DefaultTableModel model = (DefaultTableModel)tblReadings.getModel();
    model.addRow(row);
}
```

Run the program and go to the ***Meter readings*** page. Select a series of customers, entering a new meter reading for each customer and then clicking the '***Enter meter reading***' button. Check that the data is transferred to the table correctly.



| Customer | New meter reading |
|---|---|
| 291 Morris | 83552 |
| 429 Brown | 39601 |
| 518 Sanderson | 54828 |

Previous meter reading 52197

New meter reading 54828

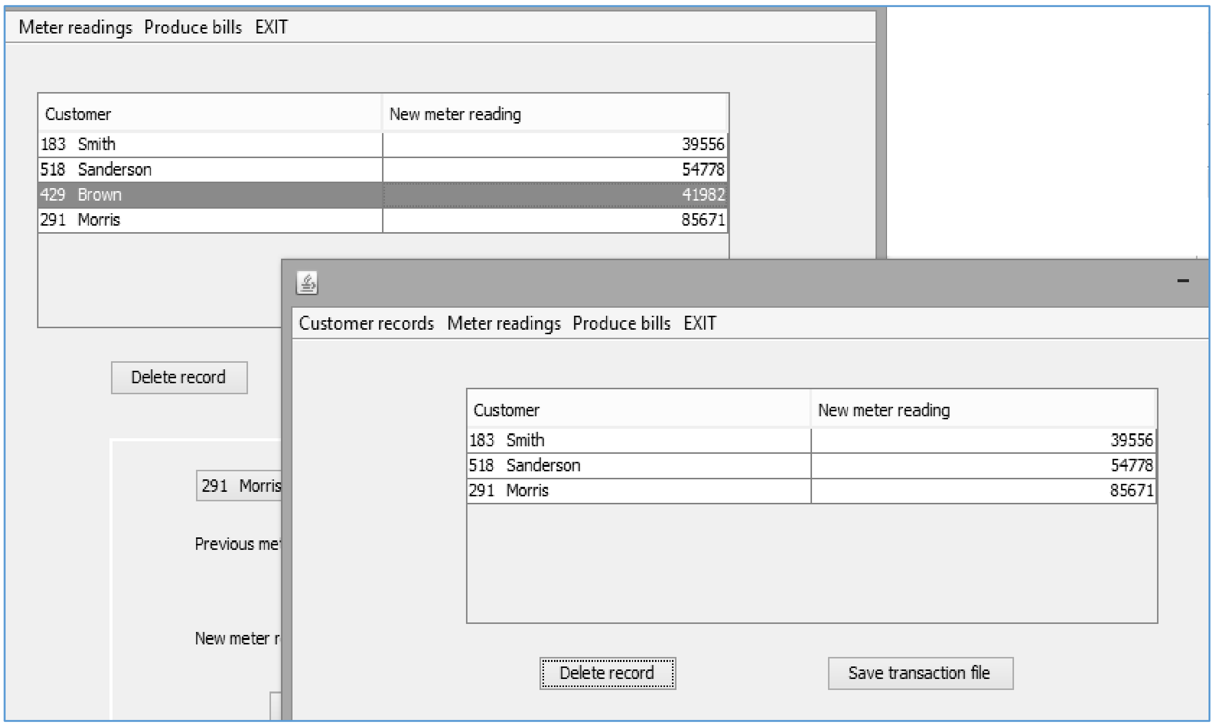
Close the program and return to the ***meterReadings.java*** screen. Use the ***Design*** tab to change back to the form layout view.

The table has been set to read-only, with all data entry taking place on the panel. We should, therefore, provide a 'Delete record' facility, in case the user spots an error after data has already been transferred the table. Double click the '***Delete record***' button to create a method.

Add code to the method to detect which row of the table is currently selected, then delete that row.

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

    int row=tblReadings.getSelectedRow();
    DefaultTableModel model = (DefaultTableModel)tblReadings.getModel();
    model.removeRow(row);

}
```

Run the program and go to the ***Meter readings*** page. Enter several rows of data into the table. Click the mouse on a row of the table, then click the ''***Delete record***' button. Check that the row is deleted correctly.


Meter readings Produce bills EXIT
Delete record
291 Morris
Previous me
New meter r
Customer records Meter readings Produce bills EXIT
Delete record
Save transaction file

| Customer | New meter reading |
|---|---|
| 183 Smith | 39556 |
| 518 Sanderson | 54778 |
| 429 Brown | 41982 |
| 291 Morris | 85671 |

| Customer | New meter reading |
|---|---|
| 183 Smith | 39556 |
| 518 Sanderson | 54778 |
| 291 Morris | 85671 |

Close the program and return to the ***meterReadings.java*** screen. We can now work on a method required to create ***Reading objects*** from the table data.

Add ***makeReadingObjects( )*** below the ***meterReadings( )*** method. Include definitions for the variables which will be required.

```
    public meterReadings() {
        initComponents();
        loadCustomers();
    }

    private void makeReadingObjects()

    {
        String customer;
        Long newReading;
    }
```

The structure of the ***makeReadingObjects( )*** method will be very similar to the method we used to create Customer objects. Add the lines of code below which will:

- Stop the table editor, so that all the data values are available for processing.
- Find the number of rows of data entered in the table, then loop for each row.
- Collect the ***customer*** and ***meter reading*** data, then use these to create a ***Reading object***.
- Update the ***readingCount*** variable to indicate how many objects have been created.

```
private void makeReadingObjects()
{
    String customer;
    Long newReading;

    TableCellEditor editor = tblReadings.getCellEditor();
    if (editor != null)
    {
        editor.stopCellEditing();
    }
    int n=tblReadings.getRowCount();
    for (int i=0; i<n; i++)
    {
        customer=(String) tblReadings.getModel().getValueAt(i,0);
        newReading=(Long) tblReadings.getModel().getValueAt(i,1);
        Reading.readingObject[i] = new Reading(customer, newReading.longValue());
    }
    Reading.readingCount=n;

}
```

Once the ***Reading objects*** have been created, they should be ***sorted*** into order of ***customerID***, ready to produce the electricity bills by a ***batch process***. We will carry out the sorting in the ***Reading class***.

Move to the ***Reading.java*** page. Insert a ***sort( )*** method immediately below the ***Reading( )*** constructor method. Create an empty ***temp*** object for use in the ***triangular exchange***, and start the loops for the ***bubble sort*** procedure.

```
    public Reading(String tCustomer, long tNewReading)
    {
        customer = tCustomer;
        newReading=tNewReading;
    }

    public static void sort()
    {
        Reading temp=new Reading("",0);
        Boolean swap=true;
        while (swap==true)
        {
            swap=false;
            for (int i=0; i<readingCount-1;i++)
            {

            }
        }
    }
```

We can now add code to complete the sorting:

- The sort procedure collects the customer entries from the two ***Reading objects*** which are being compared, then extracts the ***customerID numbers*** using the ***substring( )*** command.
- If the ***customerID numbers*** are in the wrong order, the objects are swapped in the ***readingObject[ ]*** array by means of a ***traingular exchange*** using the ***temp*** object.

```
while (swap==true)
{
    swap=false;
    for (int i=0; i<readingCount-1;i++)
    {
        String customer1 =readingObject[i].customer.toString();
        int customerID1=Integer.valueOf(customer1.substring(0,3));
        String customer2 =readingObject[i+1].customer.toString();
        int customerID2=Integer.valueOf(customer2.substring(0,3));

        if (customerID1>customerID2)
        {
            swap=true;
            temp=readingObject[i];
            readingObject[i]=readingObject[i+1];
            readingObject[i+1]=temp;
        }

    }
}
```

Now that we have sorted the objects, the list of readings can be redisplayed in the table in the correct order of customerID numbers. Return to the ***meterReadings.java*** screen to produce a method to do this.

Add a ***displayReadings( )*** method below the ***makeReadingObjects( )*** method. This begins by defining the variables which will be required, then clears the table by setting the number of rows to zero. A loop then accesses each of the ***Reading*** objects, makes a row of data from the ***customer*** and ***newReading*** properties, then adds this to the table.

```
private void displayReadings()
{
    String customer;
    long newReading;
    DefaultTableModel model = (DefaultTableModel)tblReadings.getModel();
    model.setRowCount(0);
    if (Reading.readingCount>0)
    {
        for (int i=0; i<Reading.readingCount; i++)
        {
            customer = Reading.readingObject[i].getCustomer();
            newReading= Reading.readingObject[i].getNewReading();
            Object[] row = { customer, newReading};
            model.addRow(row);
        }
    }
}
```

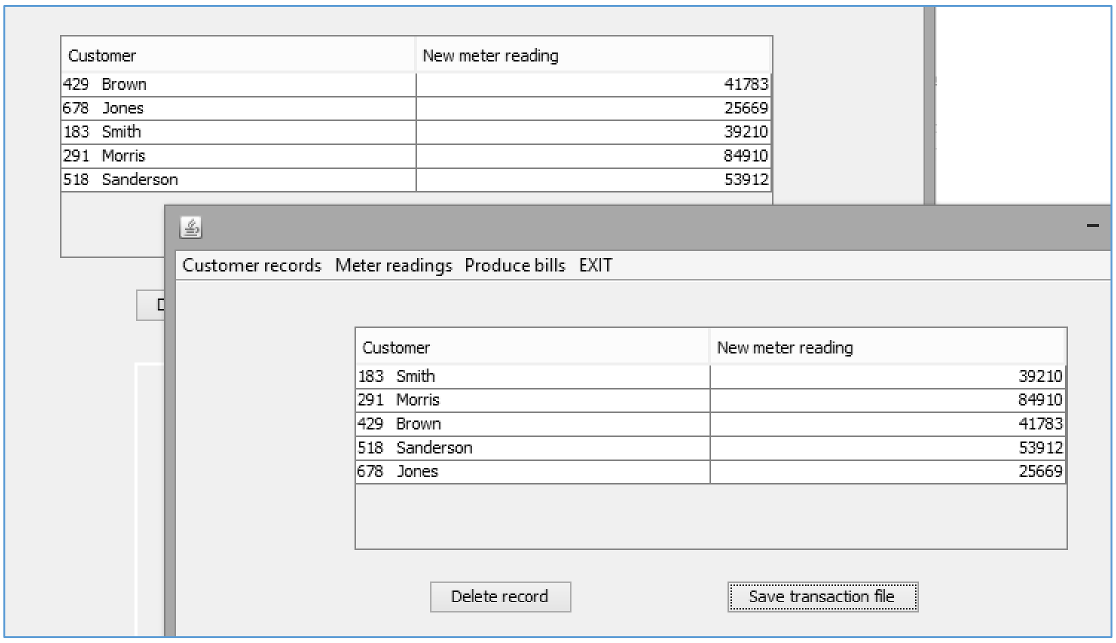
We will need to add methods to the ***Reading*** class file to make the ***customer*** and ***newReading*** data values available. Use the tab above the editing screen to move to ***Reading.java***, then add methods below the ***Reading( )*** constructor method.

```
    public Reading(String tCustomer, long tNewReading)
    {
        customer = tCustomer;
        newReading=tNewReading;
    }

    public String getCustomer()
    {
        return customer;
    }

    public long getNewReading()
    {
        return newReading;
    }
```

Return to the meterReadings.java page and use the ***Design*** tab to change to the form layout view. Double click the '***Save transaction file***' button to create a method. Add lines of code to call the sequence of methods to create Reading objects, sort the objects into order of ***customerID numbers***, then redisplay the sorted records in the table.

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

      makeReadingObjects();
      Reading.sort();
      displayReadings();

  }
```

Run the program. Go to the ***Meter readings*** page, then enter a series of records in random order of ***customerID numbers***. Click the '***Save transaction file***' button and check that the records are redisplayed correctly in sorted order.



| Customer | New meter reading |
|---|---|
| 429 Brown | 41783 |
| 678 Jones | 25669 |
| 183 Smith | 39210 |
| 291 Morris | 84910 |
| 518 Sanderson | 53912 |

| Customer | New meter reading |
|---|---|
| 183 Smith | 39210 |
| 291 Morris | 84910 |
| 429 Brown | 41783 |
| 518 Sanderson | 53912 |
| 678 Jones | 25669 |

Close the program and return to the NetBeans editing screen. Use the tab to open the ***Reading.java*** class. We will complete the class by adding methods to save the ***Reading objects*** on disc as ***fixed length records***, and reload the records.

Insert a ***saveReadings( )*** method below the ***sort( )*** method. Add lines of code to define the variables which will be needed, and delete any previous ***transaction.dat*** file, ready to save the new file.

```
public static void saveReadings()
{
    String fCustomer;
    String fNewReading;
    File oldfile = new File(filename);
    oldfile.delete();
}
```

Set up a ***TRY … CATCH*** structure to handle file errors, then begin a loop to save each of the ***Reading*** objects to disc.

```
        File oldfile = new File(filename);
        oldfile.delete();
        try (RandomAccessFile file = new RandomAccessFile(filename, "rw"))
        {
            for (int i=0; i<readingCount; i++)
            {

            }
            file.close();
        }
        catch(IOException e)
        {

        }
    }
```

Each ***Reading*** record has two fields. We can set the lengths in bytes for each field, and add an ***'end of record'*** marker.

- ***Customer*** is a three digit number, followed by a ***Customer name*** field which was set to 24 characters. The complete field can therefore be stored in 27 characters.
- ***New reading*** is a five digit number, so can be represented as five text characters.

| Customer | New reading | *** |
|---|---|---|
| 24 bytes | 5 bytes | 3 bytes |

This gives a total record length of 35 bytes.

We will add lines of code to the loop which will:

- Obtain the data for each field from the current ***Reading*** object.
- Convert the ***newReading*** number field into text format.
- Set the correct field lengths, adding blank spaces if necessary.
- Save the data into the file on disc.

```
for (int i=0; i<readingCount; i++)
{
    fCustomer=readingObject[i].customer;
    fCustomer=String.format("%-27s",fCustomer);
    file.write(fCustomer.getBytes());

    fNewReading=String.valueOf(readingObject[i].newReading);
    fNewReading=String.format("%-5s",fNewReading);
    file.write(fNewReading.getBytes());

    file.write("***".getBytes());
}
file.close();
```

This completes the ***saveReadings( )*** method. We will finally add a method to reload the records from disc.

Below the ***saveReadings( )*** method, insert a ***loadReadings( )*** method. Add the program code below to create a ***TRY…CATCH*** structure to handle file errors, load the records and create ***Reading objects***.

```
public static void loadReadings()
{
    int position;
    String tCustomer;
    String tNewReading;
    try
    {
        RandomAccessFile file = new RandomAccessFile(filename, "r");
        readingCount=(int) file.length()/35;
        for (int i=0; i<readingCount; i++)
        {
            position=i*35;
            file.seek(position);
            byte[] bytes = new byte[35];
            file.read(bytes);
            String s=new String(bytes);
            tCustomer=s.substring(0,27); s=s.substring(27);
            tNewReading=s.substring(0,5).trim();
            readingObject[i]=new Reading(tCustomer,Long.parseLong(tNewReading));
        }
        file.close();
    }
    catch(IOException e)
    {
    }
}
```

The program carries out a series of tasks in this method:

- It opens the ***transaction.dat*** file.
- It determine the number of records in the file by dividing the file length by the record size of 35 byte.
- It runs a loop, moving the file pointer to the correct position to input each record.
- It divides the fixed length record into the ***customer*** and ***newReading*** fields, then uses these to create a ***Reading object***.
- It finally closes the file when all the records have been loaded.

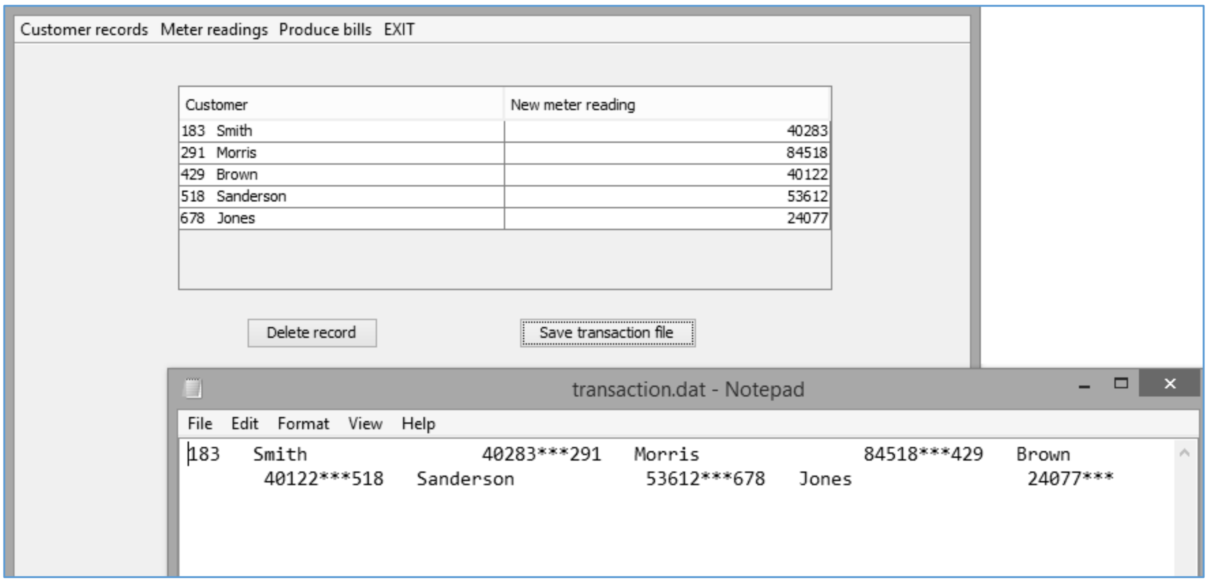
Return to the ***meterReadings.java*** page. Locate the ***btnSave( )*** method and add a line of code to call the ***saveReadings( )*** method.

```
private void btnSaveActionPerformed(java.awt.event.ActionEvent evt) {
    makeReadingObjects();
    Reading.sort();
    displayReadings();

    Reading.saveReadings();

}
```

Run the program and go to the ***Meter readings*** page and enter a series of records. Electricity usage is measured in units of ***kilowatt hours***. A typical usage for a household during the 3-month quarter for which a bill is produced would be ***1,000 kWh***.

Click the '***Save transaction file***' button. Check that the records now appear in the table in sorted order of ***customerID numbers***.

Use ***Windows Explorer*** to locate the ***transaction.dat*** file in the ***electricity*** project folder. Open transaction.dat in a text editor application such as Notepad. Check that the records have been stored correctly.


Customer records Meter readings Produce bills EXIT
Delete record
Save transaction file
transaction.dat - Notepad
File Edit Format View Help

| Customer | New meter reading |
|---|---|
| 183 Smith | 40283 |
| 291 Morris | 84518 |
| 429 Brown | 40122 |
| 518 Sanderson | 53612 |
| 678 Jones | 24077 |

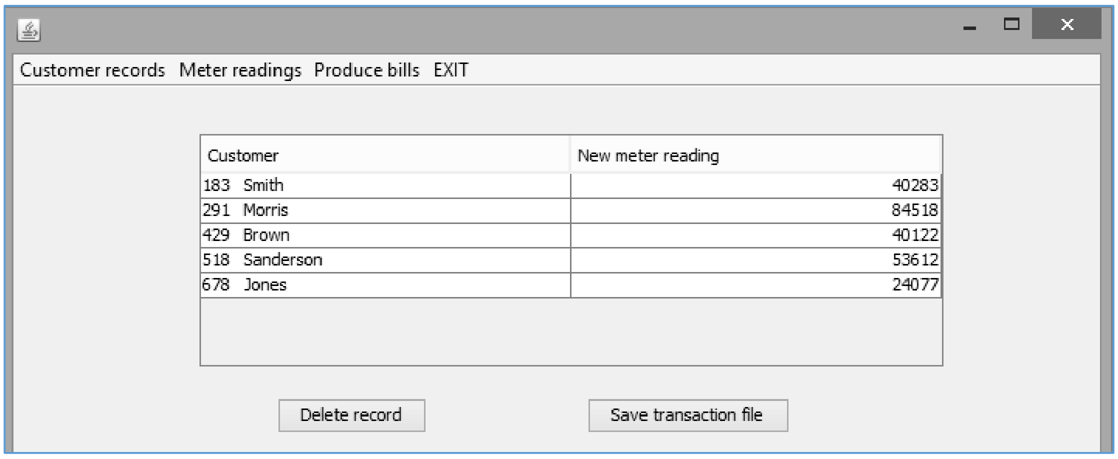
```
183    Smith              40283***291    Morris             84518***429    Brown
        40122***518    Sanderson           53612***678    Jones                24077***
```

Close the program and return to the ***meterReadings.java*** page. We will now test the method for reloading transaction data.

Locate the ***meterReadings( )*** method near the start of the program listing, then add lines of code to call the ***loadReadings( )*** and ***displayReadings( )*** methods.

```
    public meterReadings() {
        initComponents();
        loadCustomers();

        Reading.loadReadings();
        displayReadings();

    }
```

Run the program. The records should appear in the table immediately when the ***Meter readings*** option is selected.

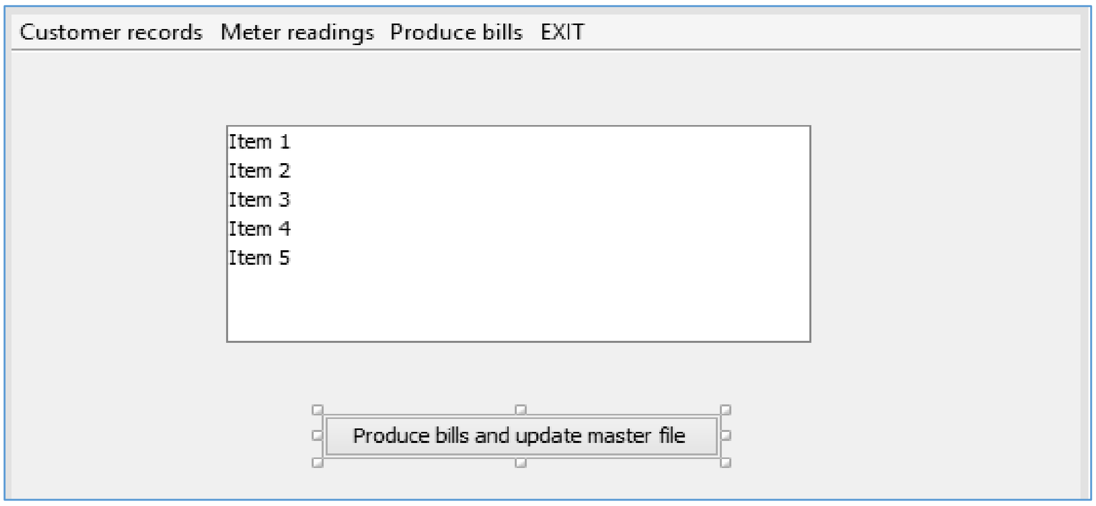


| Customer | New meter reading |
|---|---|
| 183 Smith | 40283 |
| 291 Morris | 84518 |
| 429 Brown | 40122 |
| 518 Sanderson | 53612 |
| 678 Jones | 24077 |

Close the program and return to the NetBeans editing screen. Select the ***produceBills.java*** page. We will now produce the ***batch processing*** procedures to calculate and display the bills for electricity usage, then update the master file with the new meter readings and amounts owed by customers.

Begin by adding components to the form:

- A ***List*** with the name ***lstOutput***.
- A button with the caption '***Produce bills and update master file***'. Rename the button as ***btnPrintBills***.



Double click the button to create a method which will produce the electricity bills.

We will begin by setting up ***constants*** for the cost of electricity. We will assume that customers pay:

- ***14.1 pence per kilowatt hour*** of electricity used, and
- a ***standing charge of £17.50*** per 3-month period.

The next group of variables:

***int customerID;***

***....***

***long newReading;***

will hold data from the records which are being processed. The variables:

***long unitsUsed;***

***double electricityCost;***

***double total;***

will be needed in the calculation of the bills.

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

    double unitCost=14.1;
    double standingCharge=17.50;

    int customerID;
    String customerName;
    String address;
    int readingID;
    long oldReading;
    long newReading;

    long unitsUsed;
    double electricityCost;
    double total;
    double owing;

    DefaultListModel listModel = new DefaultListModel();
    String s;
    int customerPos=0;
    int readingPos=0;
    Boolean finished=false;

}
```

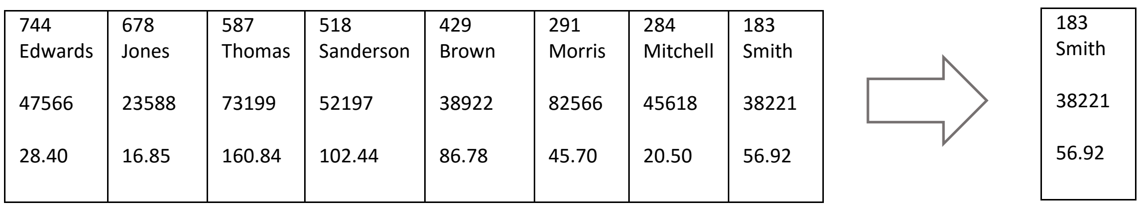
A red error symbol will appear next to the ***DefaultListModel*** line but ignore this; we will correct the problem shortly.

We can now plan a strategy for producing the electricity bills.

It was mentioned earlier that it was important to sort the ***master*** and ***transaction*** records into order of ***customerID number***. It is more efficient to carry out the batch processing using the sorted sets of objects, rather than by repeatedly accessing the disc files directly. Processing can be carried out in the fast RAM memory of the computer, rather than by many much slower disc access operations.

Let us consider the set of test data shown below. It is possible that we do not have transaction records for every customer listed in the master file. For example, a customer may have closed their account because they have moved away from the area or changed to a different supplier.

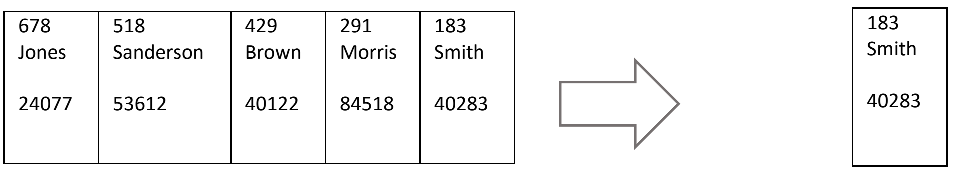
***Customer records*** in the ***master.dat*** file



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 744<br>Edwards<br><br>47566<br><br>28.40 | 678<br>Jones<br><br>23588<br><br>16.85 | 587<br>Thomas<br><br>73199<br><br>160.84 | 518<br>Sanderson<br><br>52197<br><br>102.44 | 429<br>Brown<br><br>38922<br><br>86.78 | 291<br>Morris<br><br>82566<br><br>45.70 | 284<br>Mitchell<br><br>45618<br><br>20.50 | 183<br>Smith<br><br>38221<br><br>56.92 |

| |
|---|
| 183<br>Smith<br><br>38221<br><br>56.92 |

***Reading*** records in the ***transaction.dat*** file



| | | | | |
|---|---|---|---|---|
| 678<br>Jones<br><br>24077 | 518<br>Sanderson<br><br>53612 | 429<br>Brown<br><br>40122 | 291<br>Morris<br><br>84518 | 183<br>Smith<br><br>40283 |

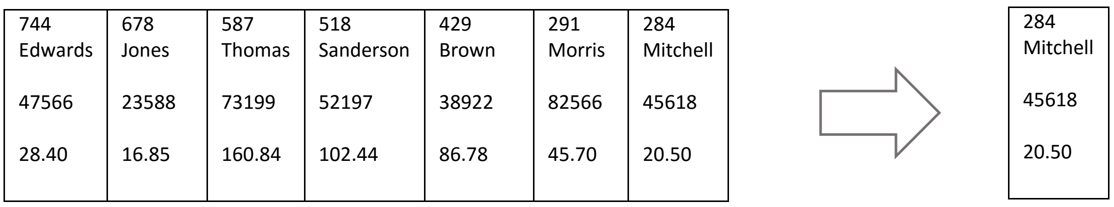
| |
|---|
| 183<br>Smith<br><br>40283 |

We will load the first records from the Customer and Reading data sets. Both records apply to the same customer, ***183 Smith***. The computer can then calculate the electricity bill and update the Customer record by carrying out a sequence of steps:

- The amount of electricity (kWh) used is found by subtracting the old reading from the new reading.
- The cost of the electricity is calculated by multiplying the amount used by the price per kilowatt hour.
- The standing charge is added to find the total amount to be paid.
- The electricity bill is printed.
- The Customer record is updated by replacing the old meter reading with the new meter reading value.
- The Customer record is updated by adding the new bill total to the previous amount owing.

The program then loads the next records from the Customer and Reading data sets.

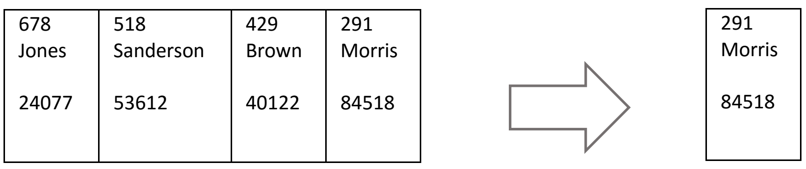
***Customer records*** in the ***master.dat*** file



| | | | | | | |
|---|---|---|---|---|---|---|
| 744<br>Edwards<br><br>47566<br><br>28.40 | 678<br>Jones<br><br>23588<br><br>16.85 | 587<br>Thomas<br><br>73199<br><br>160.84 | 518<br>Sanderson<br><br>52197<br><br>102.44 | 429<br>Brown<br><br>38922<br><br>86.78 | 291<br>Morris<br><br>82566<br><br>45.70 | 284<br>Mitchell<br><br>45618<br><br>20.50 |

| |
|---|
| 284<br>Mitchell<br><br>45618<br><br>20.50 |

***Reading*** records in the ***transaction.dat*** file

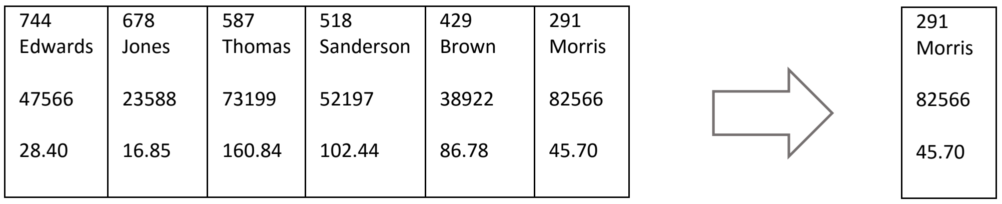


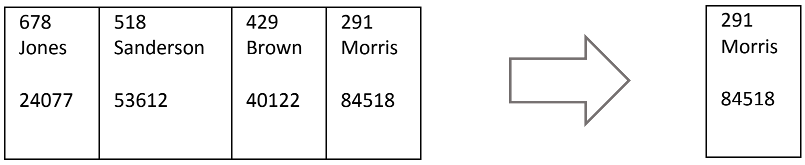
| | | | |
|---|---|---|---|
| 678<br>Jones<br><br>24077 | 518<br>Sanderson<br><br>53612 | 429<br>Brown<br><br>40122 | 291<br>Morris<br><br>84518 |

| |
|---|
| 291<br>Morris<br><br>84518 |

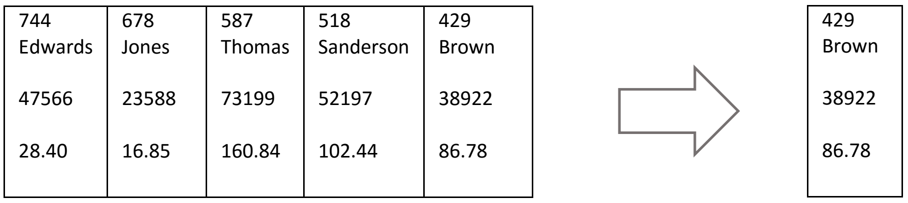
In this case the ***Customer*** and ***Reading*** records do not refer to the same account. We have no transaction record for customer ***284 Mitchell***.

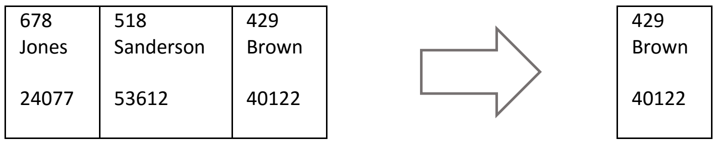
The program will continue to load ***Customer*** records until a match is found with the Reading record which is waiting to be processed.





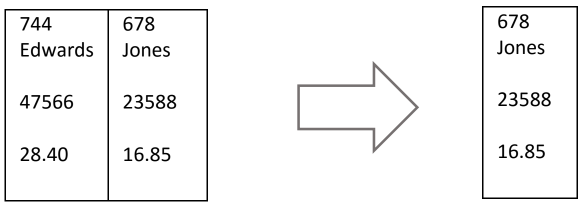
The ***Customer record*** for ***291 Morris*** has now been reached, so an electricity bill can be produced. The process continues, loading the next ***Customer*** and ***Reading*** records.

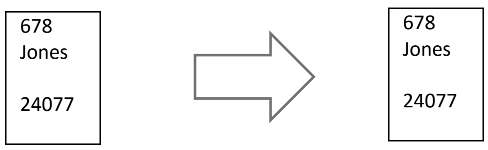




In this case, the ***Customer*** and ***Reading*** records are both for ***429 Brown***, so a bill can be produced.

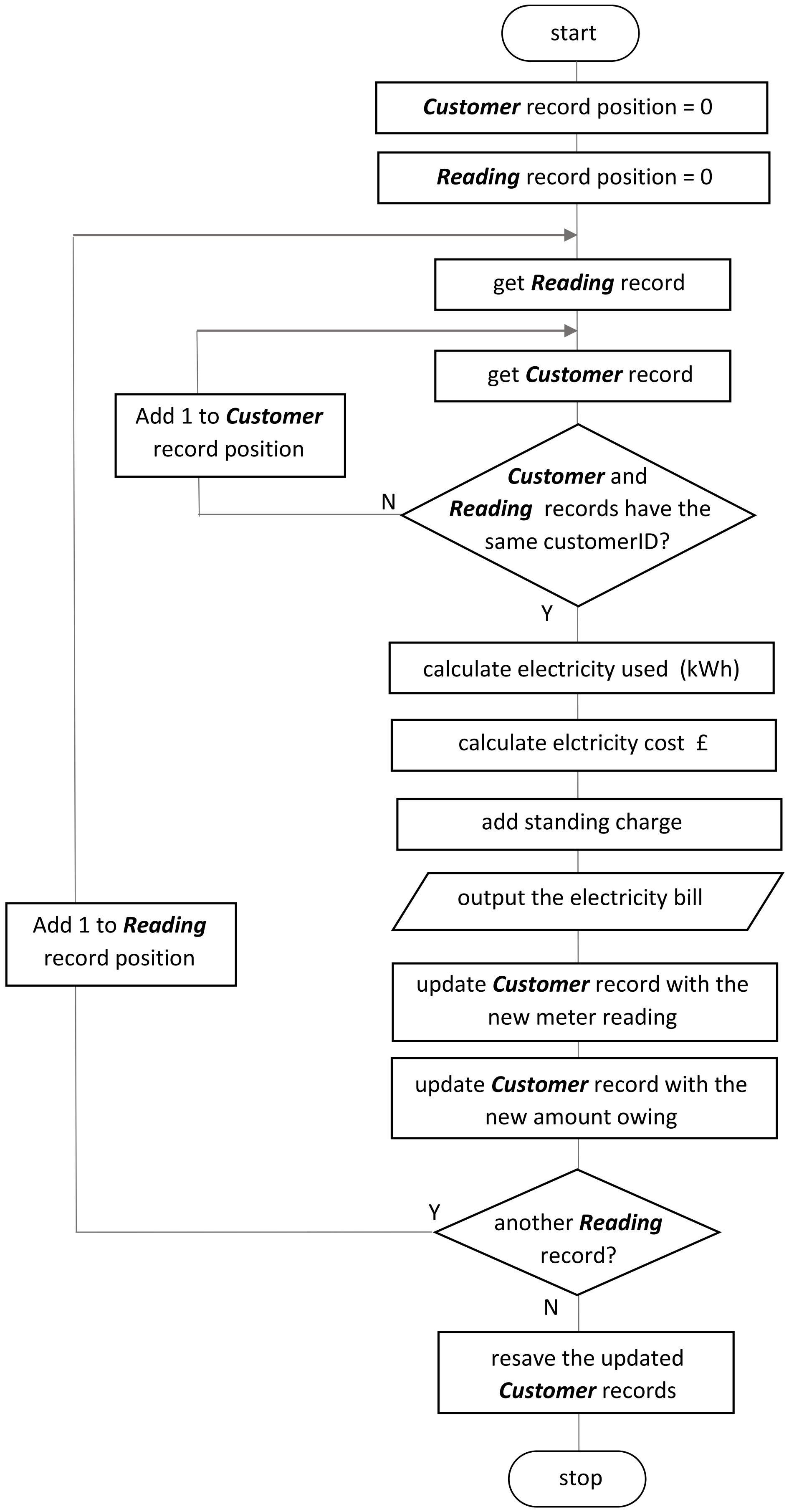
After further processing, the last Reading record is reached.





When the bill for ***678 Jones*** has been produced, the batch process is complete. The final ***Customer*** record, ***744 Edwards***, does not need to be updated.

The algorithm we have used is known as a ***sequential update***. An overall flowchart for the process can be shown as:


start
Customer record position = 0
Reading record position = 0
get Reading record
get Customer record
Add 1 to Customer record position
N
Customer and Reading records have the same customerID?
Y
calculate electricity used (kWh)
calculate elctricity cost £
add standing charge
output the electricity bill
Add 1 to Reading record position
update Customer record with the new meter reading
update Customer record with the new amount owing
Y
another Reading record?
N
resave the updated Customer records
stop

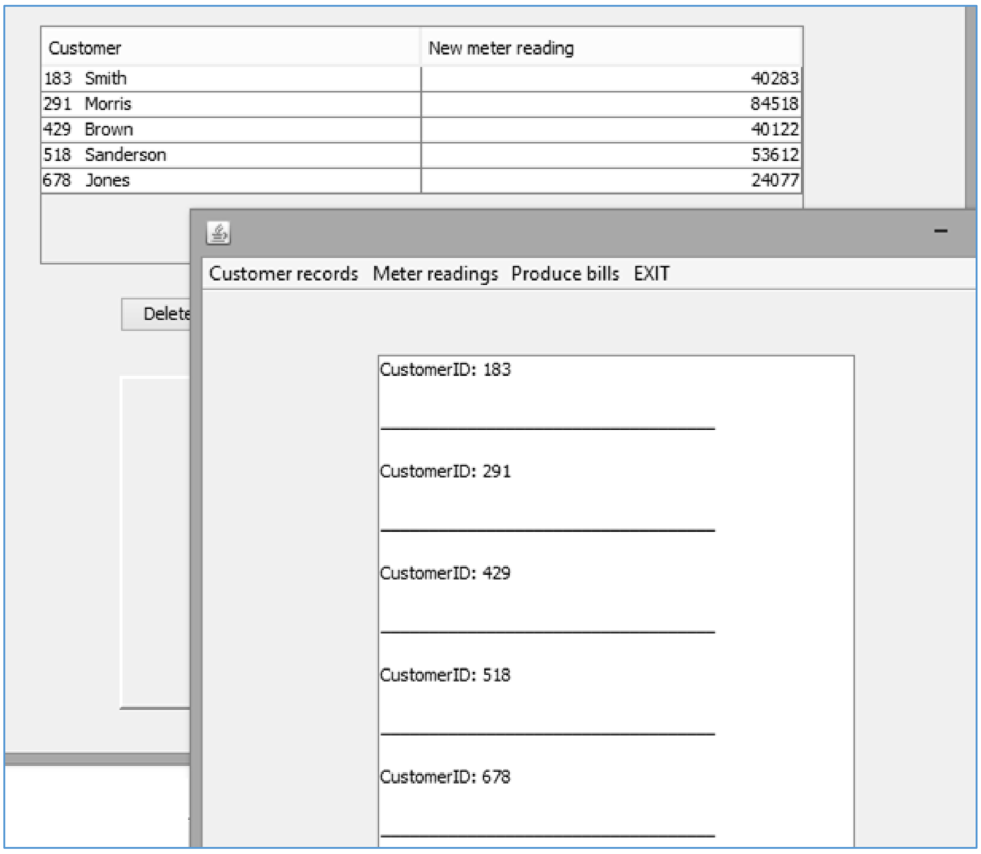
Move to the beginning of the ***Produce Bills*** program listing. Add Java modules needed to use the ***List*** component and for file handling. Locate the ***produceBills( )*** method and add lines of code to load the ***Customer*** and ***Reading*** records, ready to carry out the sequential update process.

```
package electricityPackage;

import javax.swing.DefaultListModel;
import java.io.File;

public class produceBills extends javax.swing.JFrame {

     public produceBills() {
        initComponents();

        DefaultListModel listModel = new DefaultListModel();
        lstOutput.setModel(listModel);
        Customer.loadCustomers();
        Reading.loadReadings();

     }
```

Return to the ***button-click method***. We will add the main loop which will load each ***Reading*** record and find the corresponding ***Customer*** record.

```
      String s;
      int customerPos=0;
      int readingPos=0;
      Boolean finished=false;

      customerID=Customer.customerObject[customerPos].getCustomerID();
      s=(Reading.readingObject[readingPos].getCustomer()).substring(0,3);
      readingID=Integer.parseInt(s);
      while(finished==false)
      {
         while(!(customerID==readingID))
         {
             customerPos++;
             customerID=Customer.customerObject[customerPos].getCustomerID();
         }
         listModel.addElement("CustomerID: "+customerID);
         readingPos++;
         if (readingPos<Reading.readingCount)
         {
            s=(Reading.readingObject[readingPos].getCustomer()).substring(0,3);
            readingID=Integer.parseInt(s);
         }
         else
         {
             finished=true;
         }
         listModel.addElement(" ");
         listModel.addElement("___________________________________ ");
         listModel.addElement(" ");
      }
      lstOutput.setModel(listModel);

  }
```

Run the program. Select the ***Meter readings*** option and check that a series of records have been entered and saved. Move now to the Produce bills page and click the '***Produce bills***' button. An entry should appear in the list box for each customer for whom we have saved a ***Reading*** record. Any customers in the ***master file*** who do not appear in a ***Reading*** transaction record should not be shown.



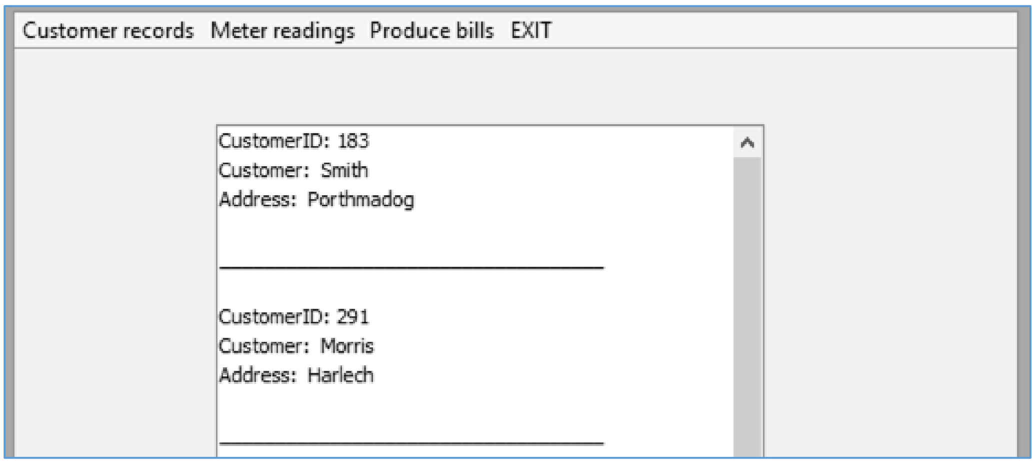
| Customer | New meter reading |
|---|---|
| 183 Smith | 40283 |
| 291 Morris | 84518 |
| 429 Brown | 40122 |
| 518 Sanderson | 53612 |
| 678 Jones | 24077 |

```
CustomerID: 183
CustomerID: 291
CustomerID: 429
CustomerID: 518
CustomerID: 678
```

Close the program and return to the program code screen. We will now work on the invoice calculations. Begin by adding lines of code to display the customer's name and town.

```
while(finished==false)
{
   while(!(customerID==readingID))
   {
       customerPos++;
       customerID=Customer.customerObject[customerPos].getCustomerID();
   }
   listModel.addElement("CustomerID: "+customerID);

   customerName=Customer.customerObject[customerPos].getCustomerName();
   listModel.addElement("Customer:  "+customerName);
   address=Customer.customerObject[customerPos].getTown();
   listModel.addElement("Address:  "+address);

   readingPos++;
   if (readingPos<Reading.readingCount)
   {
      s=(Reading.readingObject[readingPos].getCustomer()).substring(0,3);
      readingID=Integer.parseInt(s);
   }
```

Run the program. Select the ***Produce bills*** option, then click the '***Produce bills***' button. Check that customer names and towns are displayed correctly.

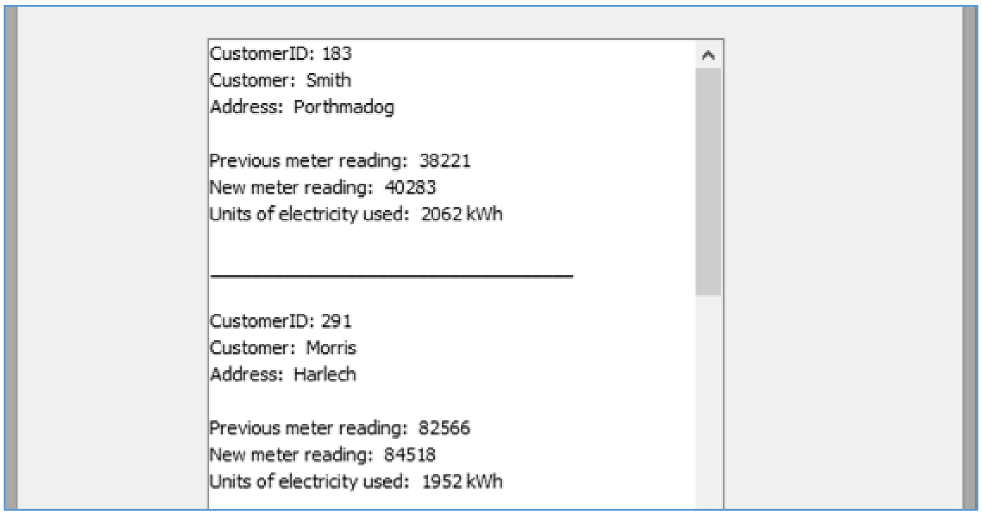


```
CustomerID: 183
Customer:  Smith
Address:  Porthmadog

________________________________

CustomerID: 291
Customer:  Morris
Address:  Harlech

________________________________
```

Close the program and return to the program code screen. Add lines of code to calculate and display the units of electricity used.

```
customerName=Customer.customerObject[customerPos].getCustomerName();
listModel.addElement("Customer:  "+customerName);
address=Customer.customerObject[customerPos].getTown();
listModel.addElement("Address:  "+address);

listModel.addElement(" ");
oldReading =Customer.customerObject[customerPos].getOldReading();
listModel.addElement("Previous meter reading:  "+oldReading);
newReading =Reading.readingObject[readingPos].getNewReading();
listModel.addElement("New meter reading:  "+newReading);
unitsUsed=newReading-oldReading;
listModel.addElement("Units of electricity used:  "+unitsUsed+" kWh");

readingPos++;
if (readingPos<Reading.readingCount)
{
   s=(Reading.readingObject[readingPos].getCustomer()).substring(0,3);
   readingID=Integer.parseInt(s);
}
```

Run the program. Go to the ***Produce bills*** page and check that correct electricity usage is calculated.

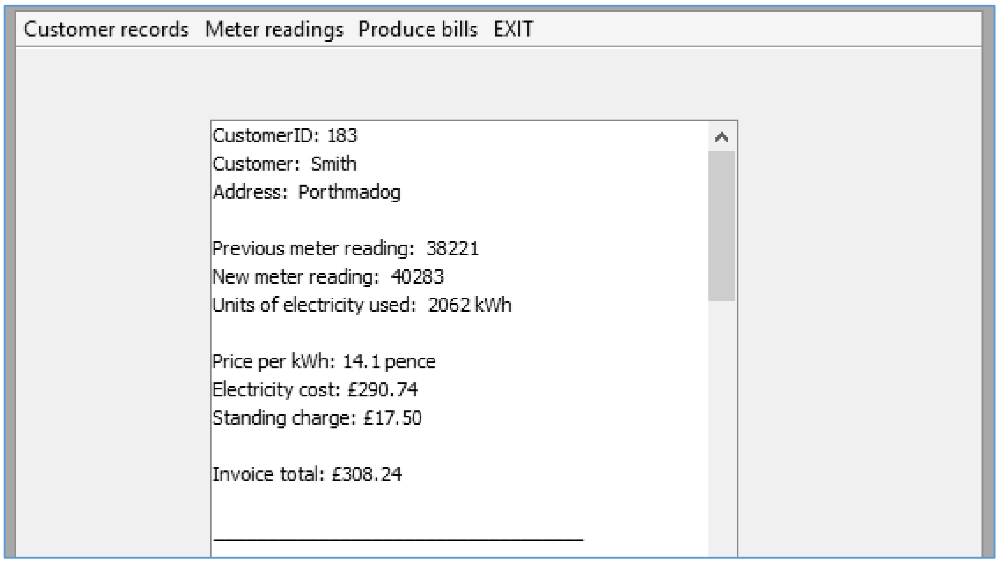


```
CustomerID: 183
Customer:  Smith
Address:  Porthmadog

Previous meter reading:  38221
New meter reading:  40283
Units of electricity used:  2062 kWh

________________________________

CustomerID: 291
Customer:  Morris
Address:  Harlech

Previous meter reading:  82566
New meter reading:  84518
Units of electricity used:  1952 kWh
```

Close the program and return to the program code screen. We will now add lines of code to calculate and display the electricity bills.

```
   unitsUsed=newReading-oldReading;
   listModel.addElement("Units of electricity used:  "+unitsUsed+" kWh");

   listModel.addElement(" ");
   listModel.addElement("Price per kWh: "+unitCost+" pence");
   electricityCost=(unitCost*unitsUsed)/100;
   listModel.addElement("Electricity cost: £"+String.format("%.2f", electricityCost));
   listModel.addElement("Standing charge: £"+String.format("%.2f", standingCharge));
   listModel.addElement(" ");
   total=electricityCost+standingCharge;
   listModel.addElement("Invoice total: £"+String.format("%.2f", total));

   readingPos++;
   if (readingPos<Reading.readingCount)
   {
      s=(Reading.readingObject[readingPos].getCustomer()).substring(0,3);
      readingID=Integer.parseInt(s);
   }
```

Run the program. Go to the ***Produce bills*** page and check that a correct invoice is displayed for each customer.



```
CustomerID: 183
Customer:  Smith
Address:  Porthmadog

Previous meter reading:  38221
New meter reading:  40283
Units of electricity used:  2062 kWh

Price per kWh: 14.1 pence
Electricity cost: £290.74
Standing charge: £17.50

Invoice total: £308.24
```

Close the program and return to the program code screen.

We will now update the Customer record with the new reading and amount owing. To do this, two small methods need to be added to the Customer class. Use the tab above the editing screen to move to the Customer.java page.

Locate the series of methods used to get the properties of Customer objects, such as:

***getOldReading( );***

Add the new methods after these, as shown below.

```
        public long getOldReading()
        {
           return oldReading;
        }

        public double getOwing()
        {
           return owing;
        }

        public void setOldReading(long fOldReading)
        {
           oldReading = fOldReading;
        }

        public void setOwing(double fOwing)
        {
           owing = fOwing;
        }
```

Return to the ***produceBills.java*** page, and locate the position in the button-click method where we have been adding lines of code. Add the program code needed to update the meter reading and amount owed by the customer.

```
      total=electricityCost+standingCharge;
      listModel.addElement("Invoice total: £"+String.format("%.2f", total));

      Customer.customerObject[customerPos].setOldReading(newReading);
      owing =Customer.customerObject[customerPos].getOwing();
      owing += total;
      String sOwing=String.format("%.2f", owing);
      Customer.customerObject[customerPos].setOwing(Double.parseDouble(sOwing));

      readingPos++;
      if (readingPos<Reading.readingCount)
      {
         s=(Reading.readingObject[readingPos].getCustomer()).substring(0,3);
         readingID=Integer.parseInt(s);
      }
```

Move now to the end of the button-click method. We need to carry out several tasks after all the records have been processed and the loop ends:

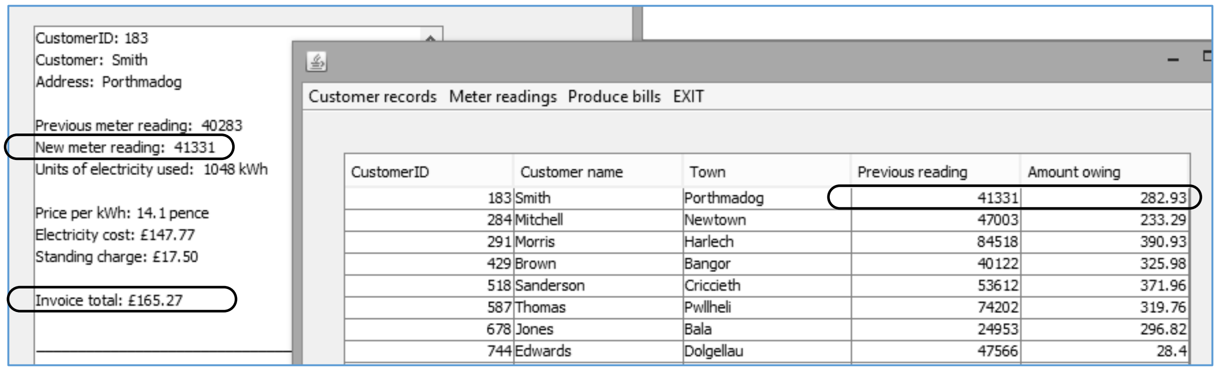
- The updated ***Customer objects*** must be saved back into the ***master.dat*** file.
- The ***transaction file*** must be cleared, to remove the ***Reading records*** which have now been processed. If this is not done, the customer could be charged more than once for the electricity used!
  Rather than completely delete the ***transaction.dat*** file, we will simply change its name to ***archive.dat***. This could then be kept by the electricity company in case of any enquiries at a later date.

Add the lines of code to the button-click method.

```
          listModel.addElement(" ");
          listModel.addElement("___________________________________ ");
          listModel.addElement(" ");
       }
       lstOutput.setModel(listModel);

       Customer.saveCustomers();
       File oldfile =new File("transaction.dat");
       File newfile =new File("archive.dat");
       oldfile.renameTo(newfile);
       Reading.readingCount=0;

 }
```

Run the program. Carry out a complete test of the system:

- Enter a series of ***customer records***. ***CustomerID*** numbers should be ***3 digits***, and ***Previous readings*** should be ***5 digits*** in length. Check that the ***Customer*** records are stored in the ***master.dat*** file and can be reloaded correctly.
- Go to the Meter readings page and enter new readings. Typical electricity usage for a customer would be around 1,000 kilowatt hours for the 3-month period. You might omit one or two customers, to test the algorithm when you carry out the sequential update. Check that the ***Reading*** records are stored in the ***transaction.dat*** file and can be reloaded correctly.
- Go to the ***Produce bills*** page and run the sequential update by clicking the '***Produce bills***' button. Check that the electricity usage and costs are calculated correctly.
- Return to the Customer records page. Check that the ***Previous reading*** field has been updated correctly, and the new bill total has been added to the ***Amount owing*** field.



```
CustomerID: 183
Customer:  Smith
Address:  Porthmadog

Previous meter reading:  40283
New meter reading:  41331
Units of electricity used:  1048 kWh

Price per kWh: 14.1 pence
Electricity cost: £147.77
Standing charge: £17.50

Invoice total: £165.27
```

| CustomerID | Customer name | Town | Previous reading | Amount owing |
|---|---|---|---|---|
| 183 | Smith | Porthmadog | 41331 | 282.93 |
| 284 | Mitchell | Newtown | 47003 | 233.29 |
| 291 | Morris | Harlech | 84518 | 390.93 |
| 429 | Brown | Bangor | 40122 | 325.98 |
| 518 | Sanderson | Criccieth | 53612 | 371.96 |
| 587 | Thomas | Pwllheli | 74202 | 319.76 |
| 678 | Jones | Bala | 24953 | 296.82 |
| 744 | Edwards | Dolgellau | 47566 | 28.4 |

- Use Windows Explorer to open the ***electricity*** project folder. The ***transaction.dat*** file should now have been renamed as ***archive.dat***.
- Return to the ***Meter readings*** page, and no transactions should now be shown.