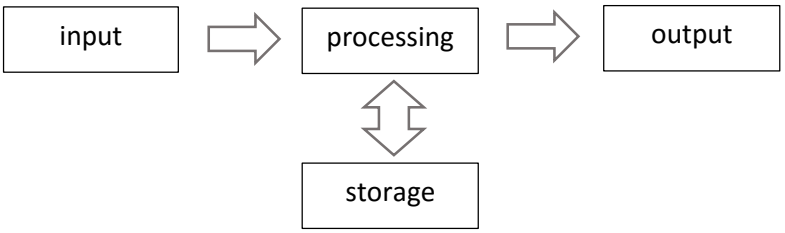
11 Stacks and queues

Computer software applications involve four components: input, processing, storage and output of data.



An important aspect of program design is to select suitable ways for representing the data which will be handled by the system.

Java provides a variety of ***simple data types*** for representing single data items. We have made use of some of these in previous programs:

integer	whole numbers, such as 89
double	decimal (real) numbers, such as 36.25
String	text, such as "Dafydd Jones"
Boolean	true or false

Sometimes it is convenient to organise data items into a group, so that they can be processed together or stored together more conveniently. We then use a ***data structure*** made up from the simple data types. Examples of data structures which we have met in previous programs are ***arrays*** and ***records***:

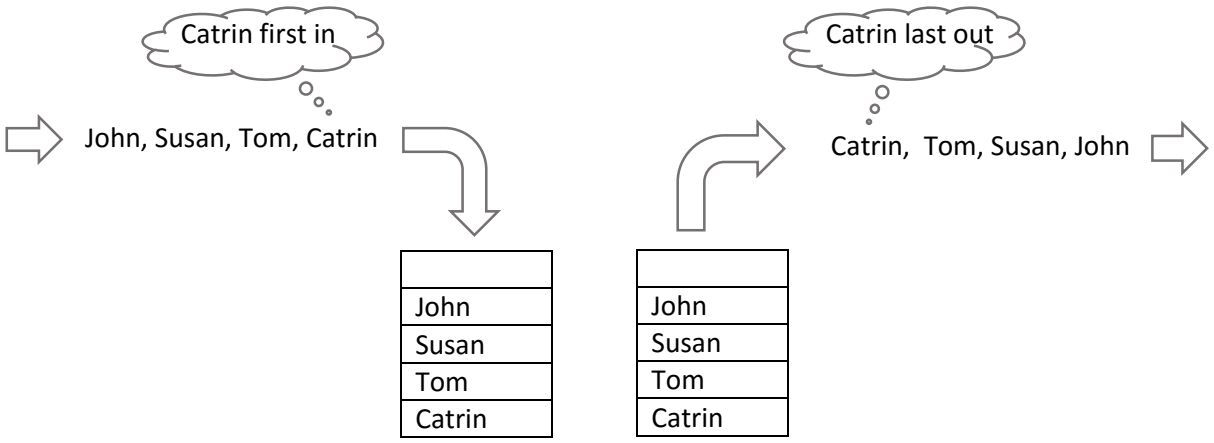
- Data items are often stored in an ***array*** when sorting or searching is required.
- Data items relating to a particular entity, such as an employee or shop stock item, are often combined into a ***record*** for storage on disc.

We can go one step further in programs by using a special arrangement of data known as an '***abstract data structure***'. As we shall see, an abstract data structure has specific rules for the ways in which data items are added and retrieved.

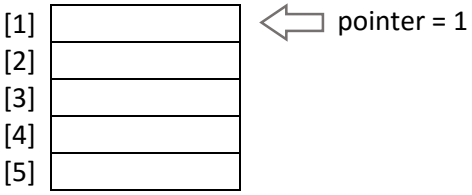
There are four common types of abstract data structure: ***stack***, ***queue***, ***linked list*** and ***binary tree***. We will examine the first two of these in the current chapter.

Stack

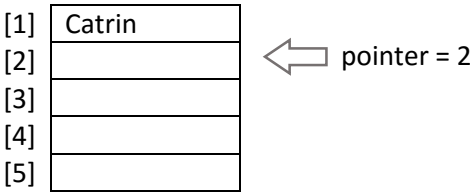
A stack is a structure in which data items are stored in a particular order, then retrieved in the reverse order. It can be described as a '***first in, last out***' structure:



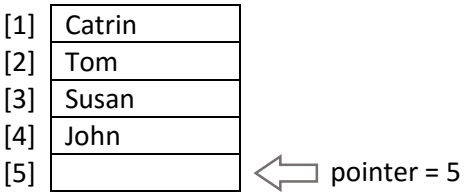
A stack can be operated with an ***array*** and an ***integer variable*** for use as a pointer. Initially the array is empty, and the pointer is set to the first array element.



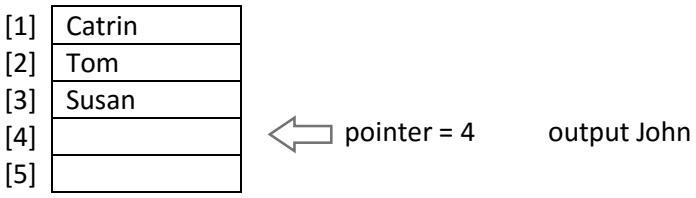
The first data item is added at the position of the pointer, then the pointer value is increased by one:



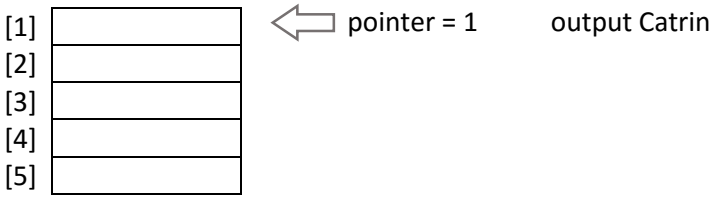
Further data items can be added, with the pointer moving downwards each time. The pointer always indicates the next empty array element where a further data item can be added.



To retrieve the data, the pointer is moved upwards by one position and the data item at that location is output.



The process can continue, with the output of each data item and the pointer moving upwards until the array is again empty. The ***first data item input*** will be the last ***data item output*** from the array.



For our first project in this chapter, we will produce a program to demonstrate the operation of a stack. Begin the project in the standard way. Close all previous projects, then set up a ***New Project***. Give this the name ***stack***, and ensure that the ***Create Main Class*** option is not selected.

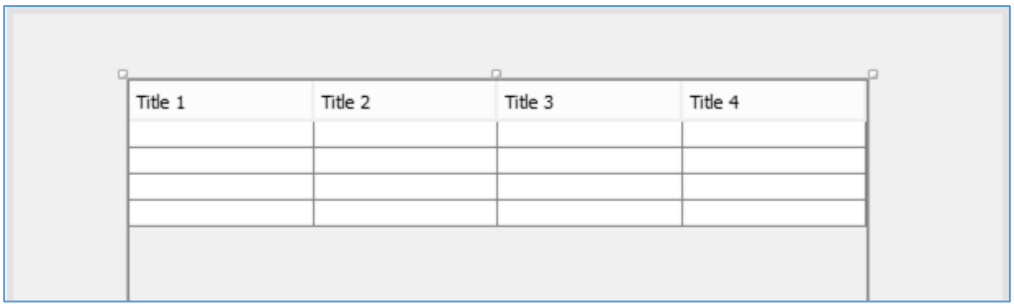
Return to the NetBeans editing page. Right-click on the ***stack*** project, and select ***New*** / ***JFrame Form***. Give the ***Class Name*** as ***stack***, and the ***Package*** as ***stackPackage***:

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.

Add a Table component to the form. Rename this as ***tblStack***.

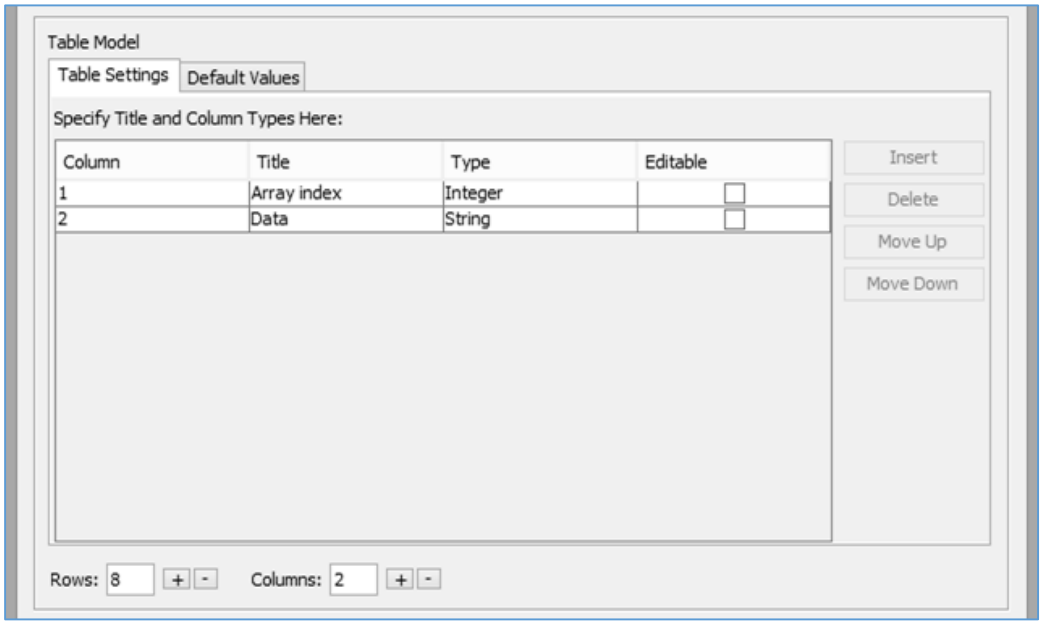


Go to the ***Properties*** window for the table and locate the ***model*** property. Click in the right column to open the editing window. Set the number of ***Rows*** to 8, and the number of ***Columns*** to 2.

Give *titles* and ***data types*** for the columns:

Array index	***Integer***
Data	***String***

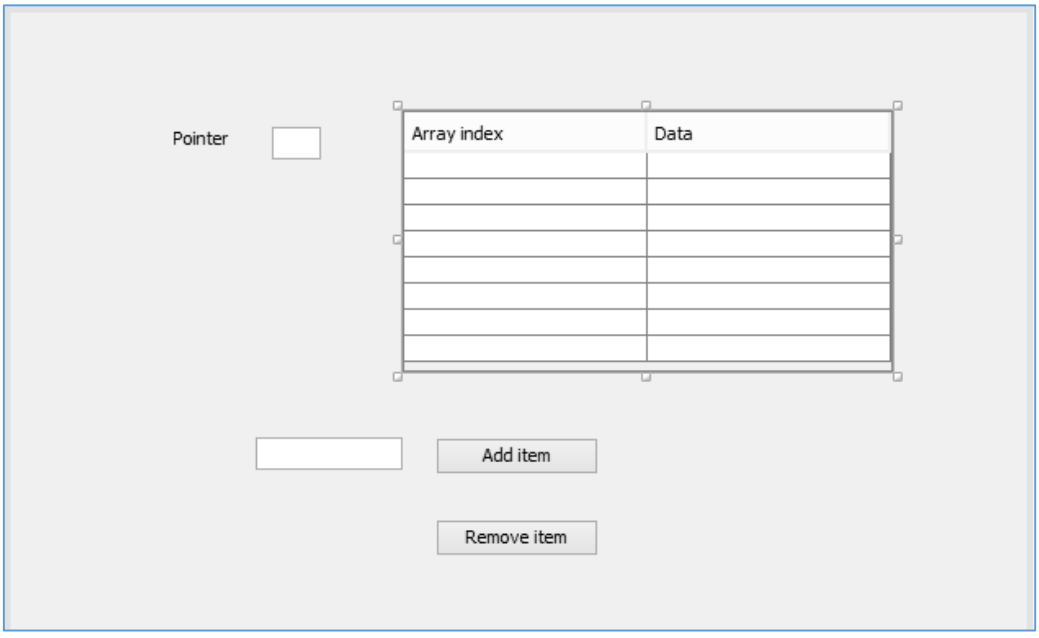
Remove the ticks from the ***Editable*** column.



Click ***OK*** to return to the form layout screen. Check that the table headings are displayed correctly.

Add components to the form:

- A label '***Pointer***', and a text field alongside with the name ***txtPointer***.
- A text field with the name ***txtNewItem***, and a button alongside with the name ***btnAdd*** and the caption '***Add item***'
- A button with the name ***btnRemove*** and the caption '***Remove item***'.



Use the ***Source*** tab to move to the program code screen. We will begin by setting up the ***array*** and ***pointer*** variable needed for the stack structure.

```
package stackPackage;

public class stack extends javax.swing.JFrame {
    String[] data = new String[9];
    int pointer;

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

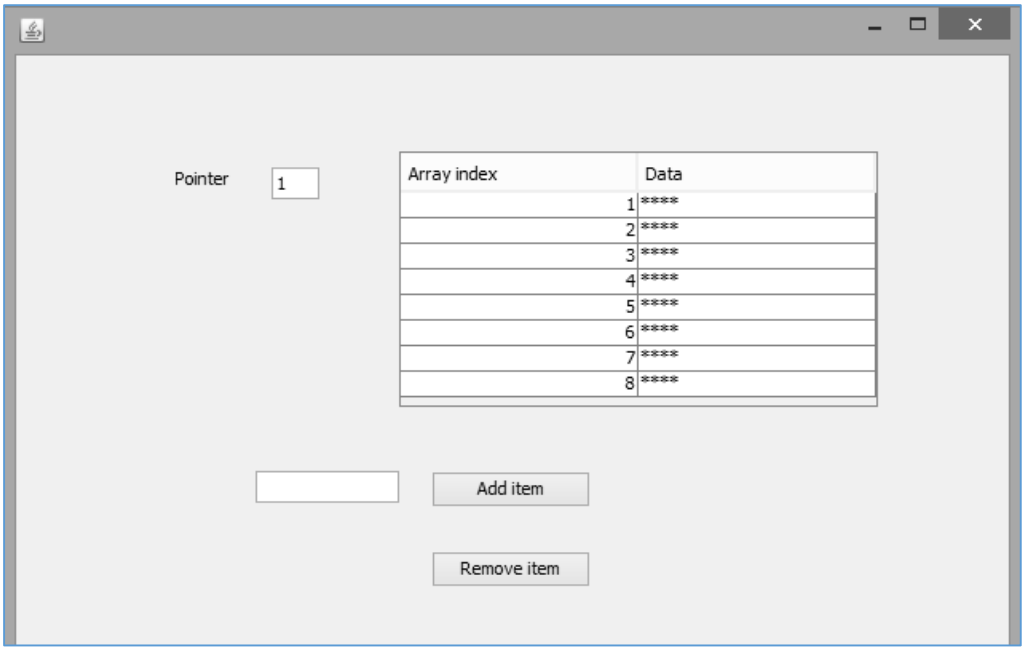
Go now to the stack( ) method and use a loop to initialise each element in the data array. We will set the pointer value to position 1.

```
    public stack() {
        initComponents();
        for (int i=1; i<=8; i++)
        {
            data[i]="****";
        }
        pointer=1;
    }
```

We will now create a ***display( )*** method to display the array data in the table, and the pointer value in the text field. Call this method from ***stack( ).***

```
    public stack() {
        initComponents();
        for (int i=1; i<=8; i++)
        {
            data[i]="****";
        }
        pointer=1;
        display();
    }
    private void display()
    {
        for (int i=1; i<=8; i++)
        {
            tblStack.getModel().setValueAt(i,i-1,0);
            tblStack.getModel().setValueAt(data[i],i-1,1);
        }
        txtPointer.setText(String.valueOf(pointer));
    }
```

Run the program. Check that the pointer is shown with a value of 1, and the table contains '****' entries to indicate that no data has been entered yet.



Close the program window and return to the NetBeans editing screen. Use the ***Design*** tab to move to the form layout view, then double click the '***Add item***' button to create a method.

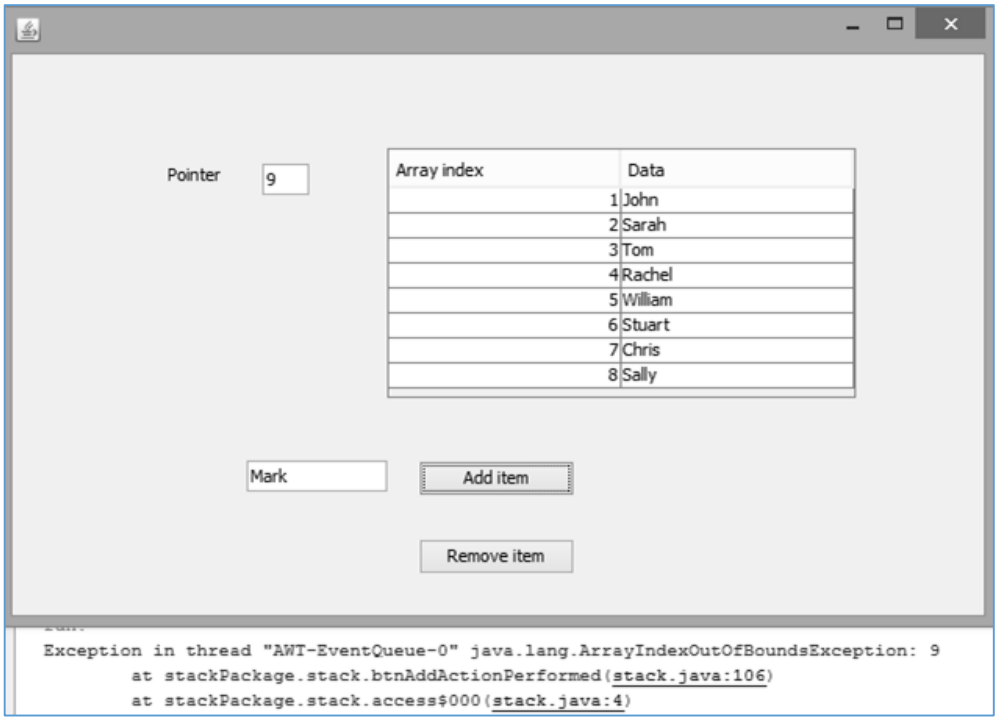
We will begin by collecting the new data item from the text field. A ***presence check*** is carried out to ensure that a data item has been entered, before continuing.

The data item is added to the array at the position indicated by the pointer variable, then the pointer is increased by one. This moves the pointer to the next array position.

Finally, we will refresh the table by calling the ***display( )*** method, then blank out the text field ready for another data entry.

```
private void btnAddActionPerformed(java.awt.event.ActionEvent evt) {
    String newItem = txtNewItem.getText();
    if (newItem.length()>0)
    {
        data[pointer]=newItem;
        pointer++;
        display();
        txtNewItem.setText("");
    }
}
```

Run the program and enter a series of data items. It is possible to add data to array elements 1 to 8 correctly, but an error occurs if we try to add a ninth data item as shown below.



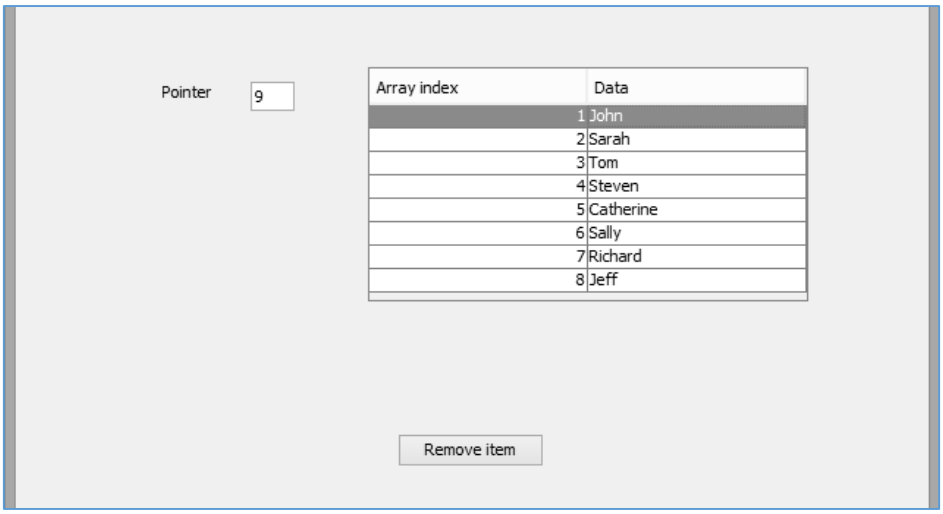
The error has occurred because we have attempted to store a data item in the array element ***data[9]*** which does not exist.

Close the program window and return to the code editing screen. We will add error trapping to the ***'Add item***' button click method.

The ***IF…*** conditional block will check for the pointer being set to a position beyond the end of the array. If this occurs, the '***Add item***' button and text field will be hidden so that no further data can be input. At he same time, however, we will make sure that the '***Remove item***' button is visible, so that existing items can be retrieved from the stack.

```
    private void btnAddActionPerformed(java.awt.event.ActionEvent evt) {
        String newItem = txtNewItem.getText();
        if (newItem.length()>0)
        {
            data[pointer]=newItem;
            pointer++;

            if (pointer>8)
            {
                btnAdd.setVisible(false);
                txtNewItem.setVisible(false);
            }

            display();
            txtNewItem.setText("");

            btnRemove.setVisible(true);

        }
    }
```

Run the program. Check that the '***Add item***' button now disappears when the stack is full, preventing an array error from occurring.



Close the program window and return to the NetBeans editing screen. Use the ***Design*** tab to move to the form layout view, then double click the 'Remove item' button to create a method.

We will add lines of program code to the method which will:

- Move the pointer back by one position.
- Reset the data value at the pointer position to '****', to indicate that the data item at this position has now been removed from the stack.
- If the pointer has reached position 1, the stack is empty. Any further attempts to remove data items could cause errors. The '***Remove item***' button is therefore hidden.
- The table is re-displayed, and we ensure that the '***Add item***' button is visible.

```
private void btnRemoveActionPerformed(java.awt.event.ActionEvent evt) {
    pointer--;
    data[pointer]="****";
    if (pointer==1)
    {
        btnRemove.setVisible(false);
    }
    btnAdd.setVisible(true);
    txtNewItem.setVisible(true);
    display();
}
```

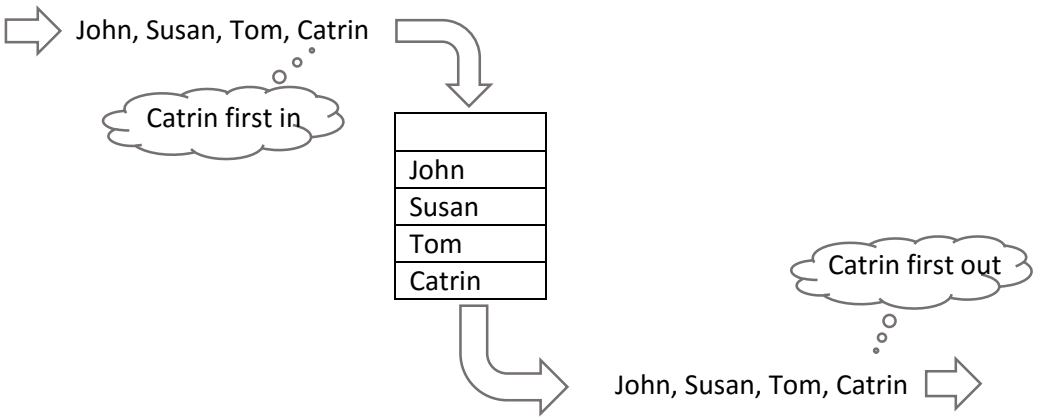
Run the completed program.

Add items to the stack, then remove them. Notice how the items are removed in the reverse of the order in which they were added.

Check that the ***'Remove item***' button disappears when the stack is empty, but reappears when further data items are added.

Queue

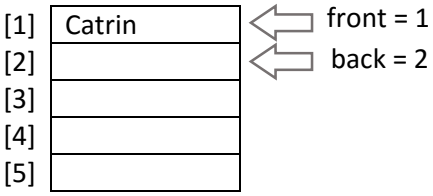
The second abstract data structure we will examine is a ***queue***. We are familiar with queues in everyday life, when waiting for a bus or waiting to be served in a shop. A queue is a ***first in – first out*** structure:



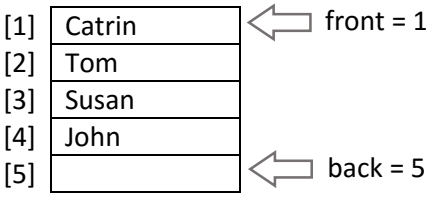
This contrasts with a ***stack***, which is a ***first in – last out*** data structure.

We can again use an array to operate a ***queue***, but two pointer variables are now needed. These are called 'front' and 'back'.

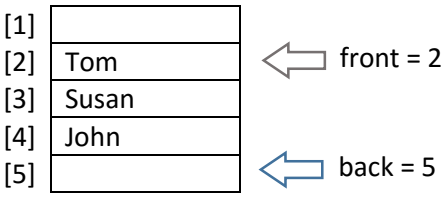
The '***back***' pointer operates in a similar way to the stack pointer by indicating the array position where the next data item will be added.



As data items are added, the back pointer value increases:



The '***front***' pointer is set to the position of the data item which has been in the queue longest. When this item is removed, the front point is moved to the next array element.



We will now produce a program to demonstrate the operation of a ***queue***. Begin the project in the standard way. Close all previous projects, then set up a ***New Project***. Give this the name ***queue***, and ensure that the ***Create Main Class*** option is not selected.

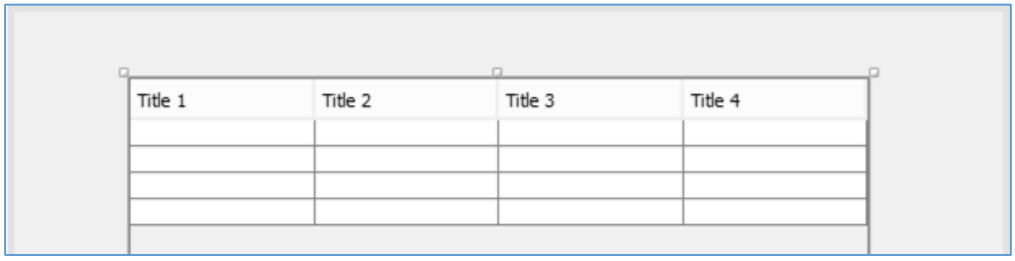
Return to the NetBeans editing page. Right-click on the ***queue*** project, and select ***New*** / ***JFrame Form***. Give the ***Class Name*** as ***queue***, and the ***Package*** as ***queuePackage***:

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.

Add a Table component to the form. Rename this as ***tblQueue***.

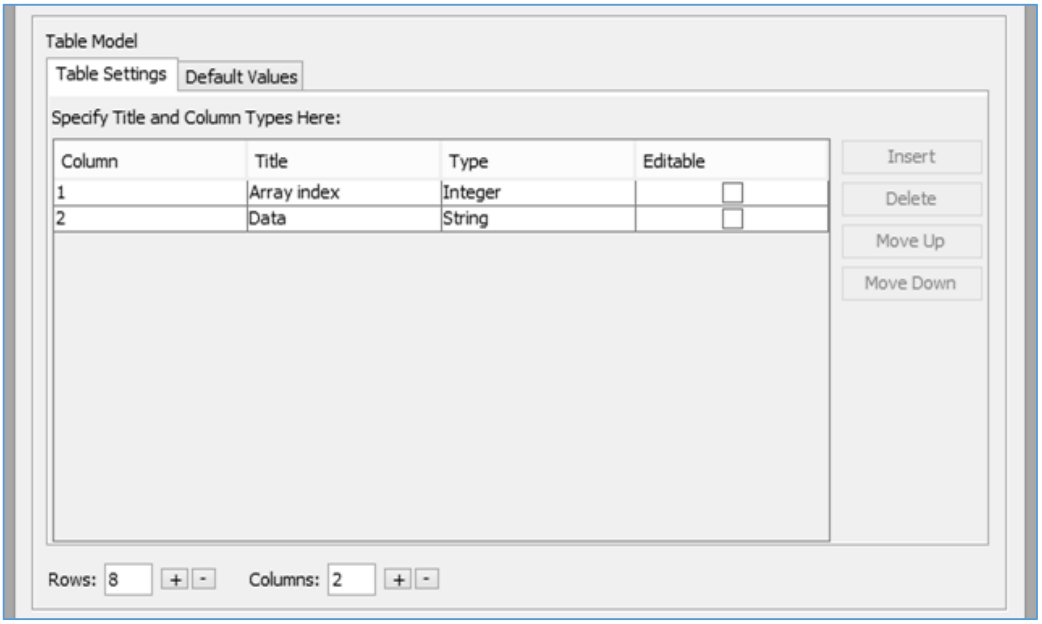


Go to the ***Properties*** window for the table and locate the ***model*** property. Click in the right column to open the editing window. Set the number of ***Rows*** to 8, and the number of ***Columns*** to 2.

Give *titles* and ***data types*** for the columns:

Array index	***Integer***
Data	***String***

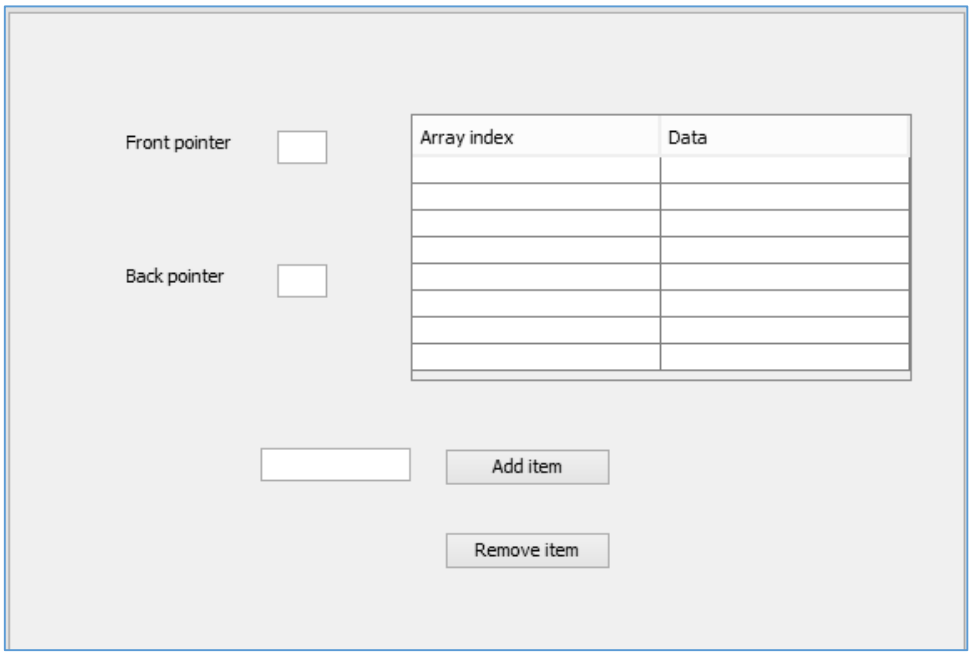
Remove the ticks from the ***Editable*** column.



Click ***OK*** to return to the form layout screen. Check that the table headings are displayed correctly.

Add components to the form:

- A label '***Front pointer***', and a text field alongside with the name ***txtFront***.
- A label '***Back pointer***', and a text field alongside with the name ***txtBack***.
- A text field with the name ***txtNewItem***, and a button alongside with the name ***btnAdd*** and the caption '***Add item***'
- A button with the name ***btnRemove*** and the caption '***Remove item***'.



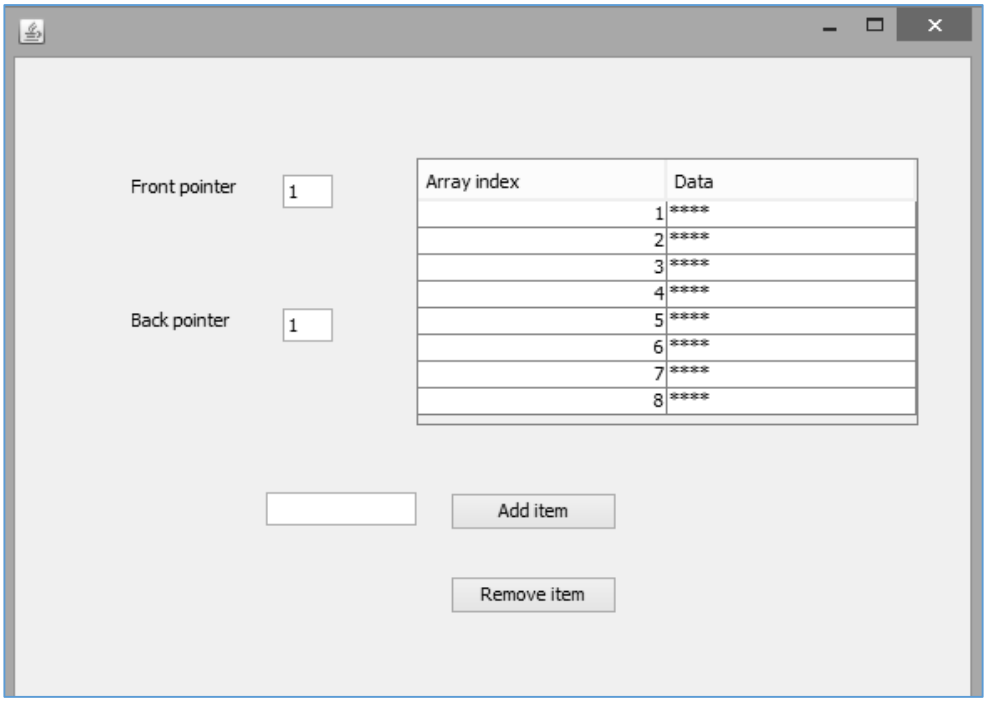
Use the ***Source*** tab to move to the program code screen. Begin by setting up the ***array*** and the ***front*** and ***back*** pointer variables needed for the queue structure. Set the data array values to '****', and initialise set both of the pointers to position 1 to represent an empty queue.

```
package queuePackage;

public class queue extends javax.swing.JFrame {
    String[] data = new String[9];
    int front;
    int back;

    public queue() {
        initComponents();

        for (int i=1; i<=8; i++)
        {
            data[i]="****";
        }
        front=1;
        back=1;

    }
```

We will now create a ***display( )*** method to display the array data in the table, and the pointer value in the text field. Call this method from ***stack( )***.

```
public queue() {
    initComponents();

    for (int i=1; i<=8; i++)
    {
        data[i]="****";
    }
    front=1;
    back=1;

    display();
}

private void display()
{
    for (int i=1; i<=8; i++)
    {
        tblQueue.getModel().setValueAt(i,i-1,0);
        tblQueue.getModel().setValueAt(data[i],i-1,1);
    }
    txtFront.setText(String.valueOf(front));
    txtBack.setText(String.valueOf(back));
}
```

Run the program. Check that the pointer is shown with a value of 1, and the table contains '****' entries to indicate that no data has been entered yet.



Close the program window and return to the NetBeans editing screen. Use the ***Design*** tab to move to the form layout view, then double click the '***Add item***' button to create a method.

We will now add lines of code which will carry out a series of tasks:

- Collect the new data item and carry out a presence check.
- If valid data has been entered, this will be stored in the array at the position of the ***back*** pointer.
- The back pointer will be moved to the next array position, ready for the next data item to be entered.
- The table data is re-displayed and the data field cleared.

```
private void btnAddActionPerformed(java.awt.event.ActionEvent evt) {
    String newItem = txtNewItem.getText();
    if (newItem.length()>0)
    {
        data[back]=newItem;
        back++;
        display();
        txtNewItem.setText("");
    }
}
```

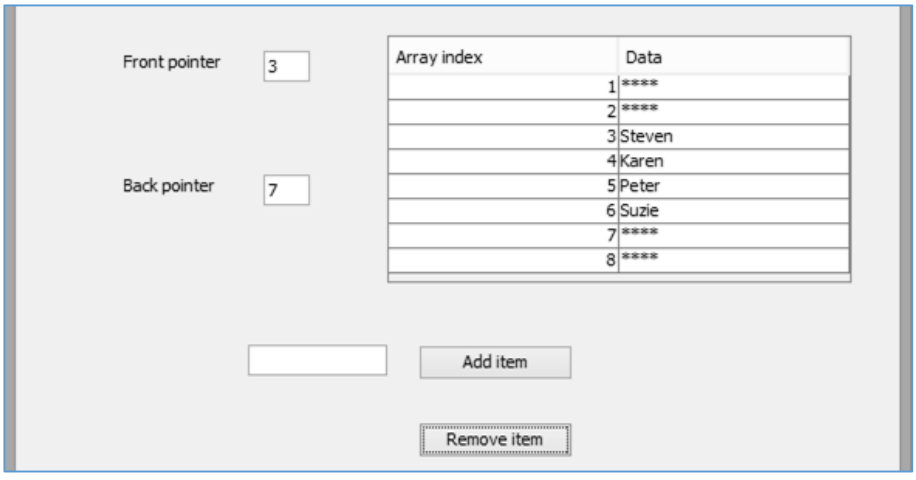
Use the ***Design*** tab to move back to the form layout view, then double click the '***Remove item***' button to create a method.

We will add lines of code which will carry out several tasks:

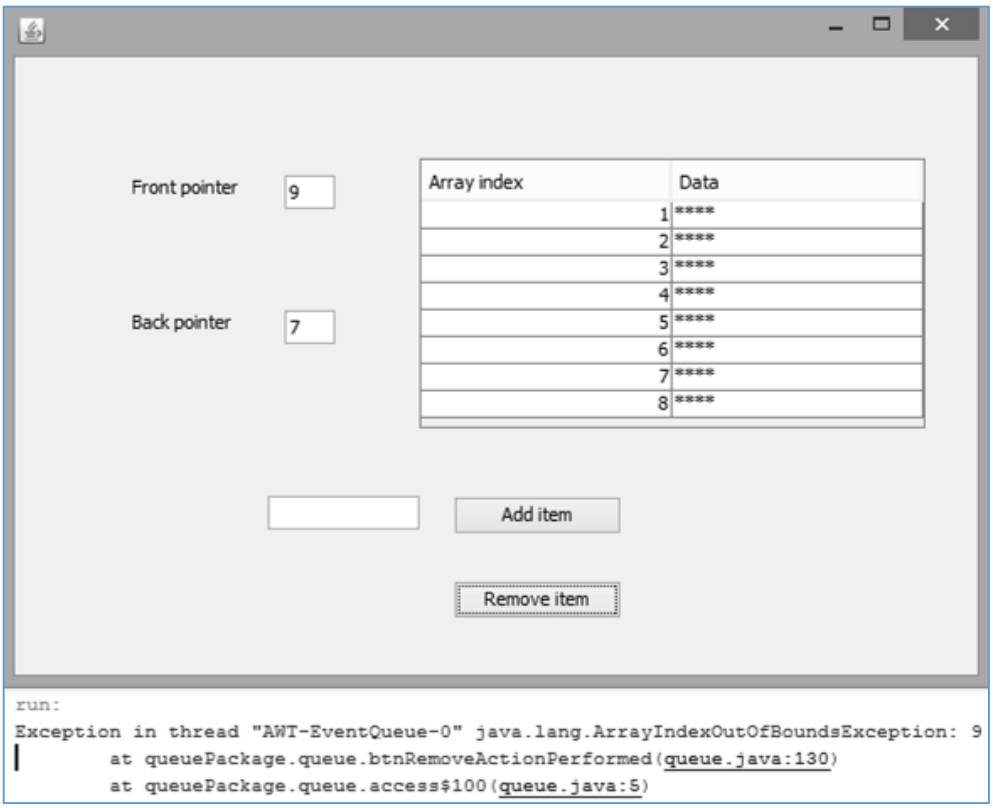
- Reset the data at the position of the ***front*** pointer.
- Move the front pointer to the next array position, ready for the next data item to be deleted.
- The updated table is re-displayed.

```
private void btnRemoveActionPerformed(java.awt.event.ActionEvent evt) {
    data[front]="****";
    front++;
    display();
}
```

Run the program. Check that data items can be added at the position of the ***back*** pointer, then removed at the position of the ***front*** pointer.

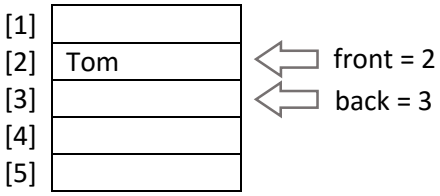


The queue is working correctly, but it is not yet error trapped. If you continue to click the '***Remove item***' button then the ***front*** pointer value will continue to increase after all the data items have been removed, until an array error occurs.

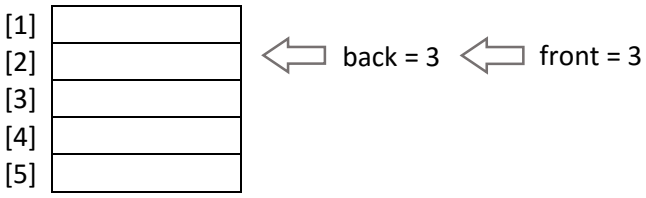


Close the program window and return to the code editing screen.

Consider what happens when the queue becomes empty.



Removing '***Tom***' moves the ***front*** pointer forward to position 3. The ***front*** pointer is now in the same position as the ***back*** pointer.



We can detect this pointer condition in the program and prevent further data items from being removed.

Add lines to the '***Remove item***' button click method to avoid the array error by hiding the 'Remove item' button when the queue is empty.

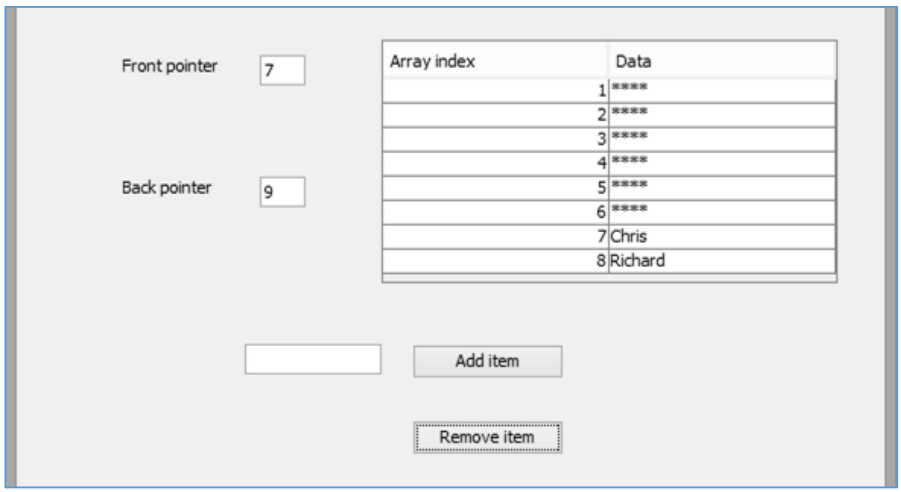
```
private void btnRemoveActionPerformed(java.awt.event.ActionEvent evt) {
    data[front]="****";
    front++;

    if (front==back)
    {
        btnRemove.setVisible(false);
    }
    btnAdd.setVisible(true);
    txtNewItem.setVisible(true);

    display();
}
```

Return to the '***Add item***' button click method. Add a line to make the '***Remove item***' button visible again when further data is entered.

```
private void btnAddActionPerformed(java.awt.event.ActionEvent evt) {
    String newItem = txtNewItem.getText();
    if (newItem.length()>0)
    {
        data[back]=newItem;
        back++;
        display();
        txtNewItem.setText("");

        btnRemove.setVisible(true);

    }
}
```

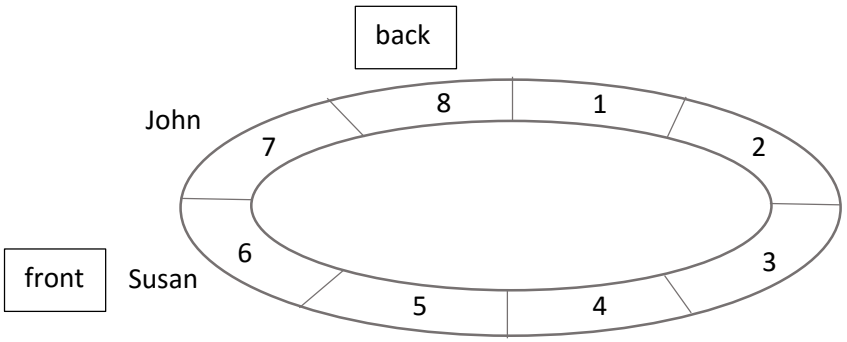
Run the program. Enter some items, then delete them. Check that the '***Delete item***' button disappears and ***front*** pointer cannot be changed if the queue is empty. The '***Delete item***' button should re-appear if further data items are entered.

At this point you might realise that there is a serious problem with the program! As data items are added and removed, the queue moves downwards through the array and very quickly reaches the limit of the array. After that, no further data items can be entered.

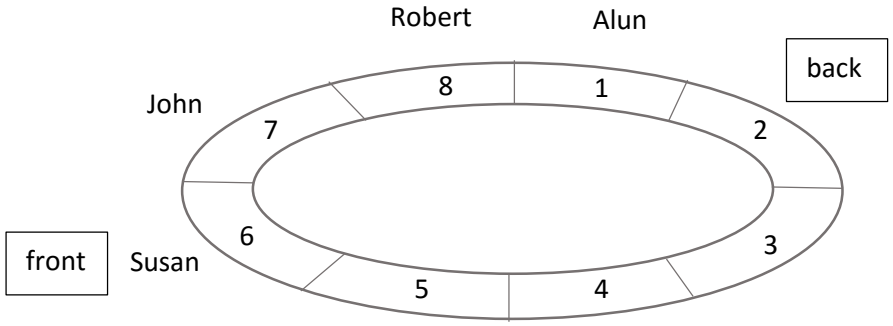


Close the program window and return to the NetBeans editing screen.

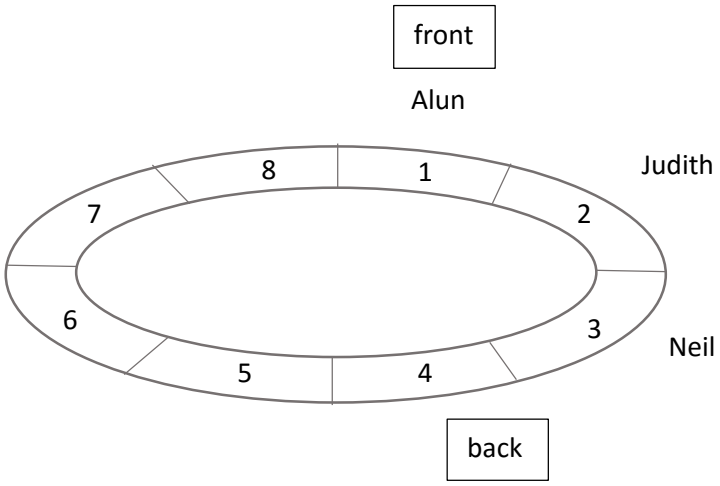
We must re-think our program strategy. The solution is to produce a circular queue. Consider the situation where two data items are present in the queue. '***Susan***' is at the front, and will be the next to leave. The ***back*** pointer is at position 8 ready for the next data item to be added.



When another data item '***Robert***' is added, the ***back*** pointer moves to position 1, ready for the next data item to be entered. The back pointer can then move forward in the normal way as each item is added.



After adding and then removing several more data items, the front pointer will also move from location 8 to location 1.



The queue will be able to continue to run indefinately, as long as no more than eight items are present at any time.

Add lines of code to the '***Delete item***' button click method to implement the circular queue. If the front pointer moves beyond position 8, it will be reset to position 1.

```
private void btnRemoveActionPerformed(java.awt.event.ActionEvent evt) {
    data[front]="****";
    front++;
    if (front>8)
    {
        front=1;
    }
    if (front==back)
    {
        btnRemove.setVisible(false);
    }
    btnAdd.setVisible(true);
    txtNewItem.setVisible(true);
    display();
}
```

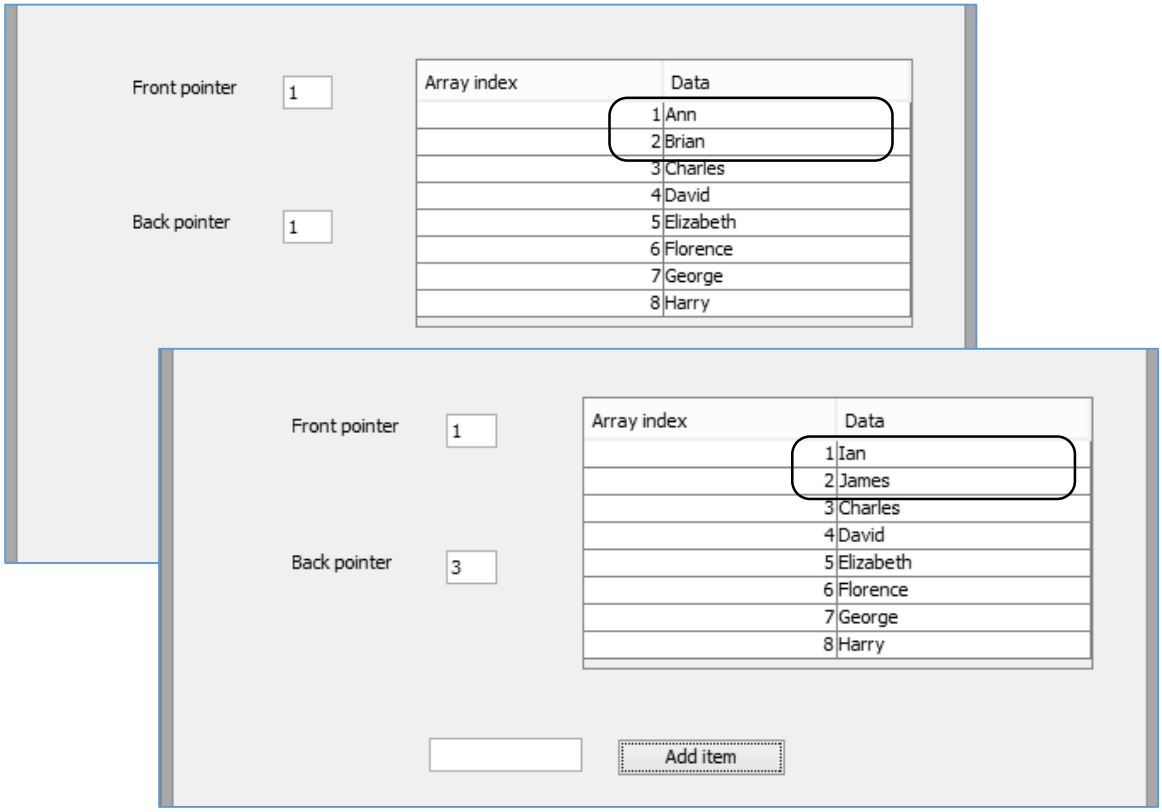
Move now to the '***Add item***' button click method and add a similar series of lines to impliment the circular queue. We will also make sure that the '***Remove item***' button is visible if there are items present in the queue.

```
private void btnAddActionPerformed(java.awt.event.ActionEvent evt) {
    String newItem = txtNewItem.getText();
    if (newItem.length()>0)
    {
        data[back]=newItem;
        back++;
        if (back>8)
        {
            back=1;
        }
        btnRemove.setVisible(true);
        display();
        txtNewItem.setText("");
    }
}
```

Run the program. Carry out a series of tests:

- Check that data items can be entered, then removed correctly in the same order that they were entered.
- The '***Remove item***' button should disappear when the queue becomes empty, and reappear when further data is entered.

You might discover one further problem! If the queue becomes full, data items will be overwritten when further data is entered, as in the example below. We must stop this happening.



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

When the queue becomes full, the back pointer moves forward to the same position as the front pointer. We can detect this condition and make the '***Add item***' button invisible, to prevent further data entries. Go to the '***Add item***' button click method and add the extra lines of code.

```
private void btnAddActionPerformed(java.awt.event.ActionEvent evt) {
    String newItem = txtNewItem.getText();
    if (newItem.length()>0)
    {
        data[back]=newItem;
        back++;
        if (back>8)
        {
            back=1;
        }
        display();
        txtNewItem.setText("");
        btnRemove.setVisible(true);

        if (front==back)
        {
            btnAdd.setVisible(false);
            txtNewItem.setVisible(false);
        }

    }
}
```

Run the completed queue program and check that this now fully working correctly.

Now that we have investigated how stacks and queues can be operated using arrays, we will look at a couple of example applications which make use of these abstract data structures.

> A simple drawing program is required. The user should be able to enter lines by clicking and dragging the mouse. The program should provide an '***undo***' function.

The user will draw a series of lines to build up an image. If the drawing becomes unsatisfactory at any time, it should be possible to progressively remove lines to return to a previous stage and try a different design. It will be necessary to remove lines in the reverse of the order in which they were added, so a ***stack*** data structure can be used.

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

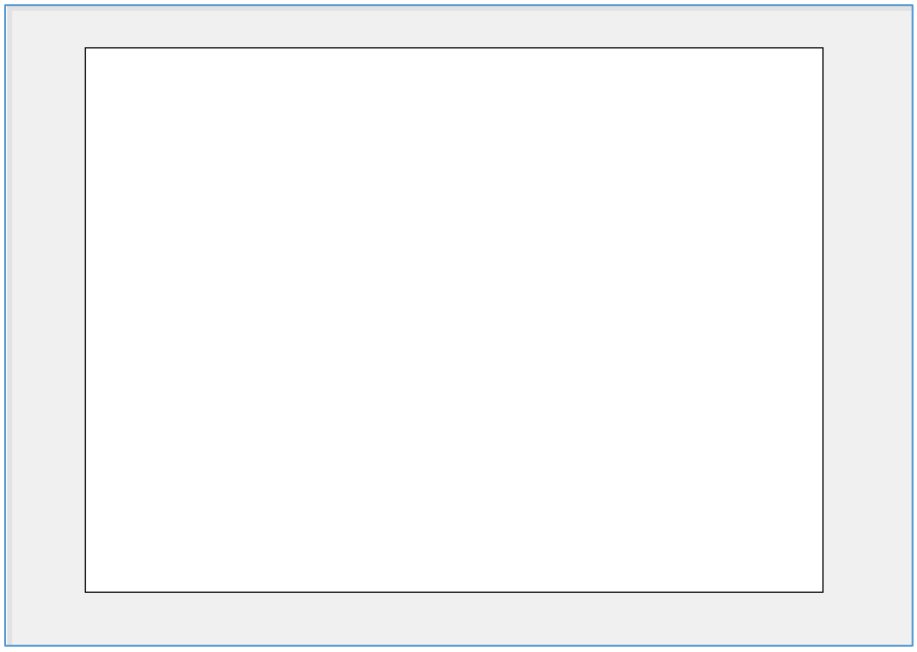
Return to the NetBeans editing page. Right-click on the ***draw*** project, and select ***New*** / ***JFrame Form***. Give the ***Class Name*** as ***draw***, and the ***Package*** as ***drawPackage***:

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***".

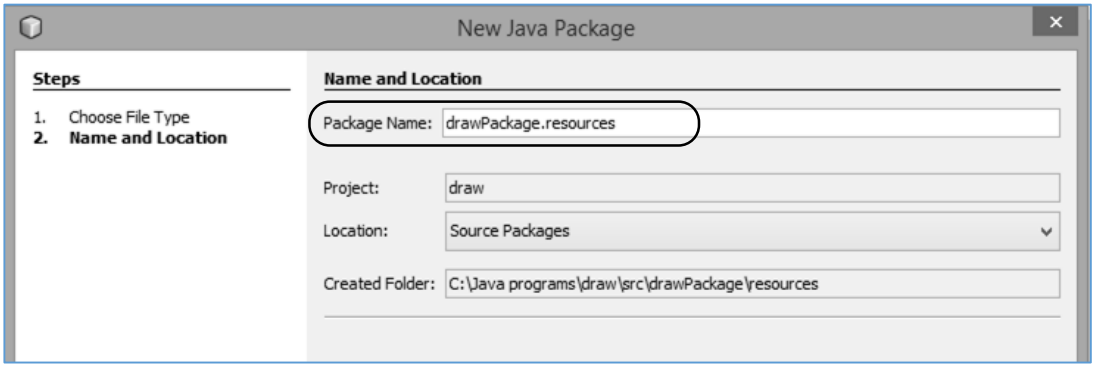
Run the program and accept the ***main*** class which is offered. Check that a blank window appears and has the correct size and colour scheme. Close the program and return to the editing screen. Click the ***Design*** tab to move to the form layout view.

Add a panel to the form, renaming this as ***pnlDraw***. Go to the ***Properties*** window and set the ***background*** property to ***White*** by means of the dropdown list. Set the ***preferredSize*** property to ***[600, 480].***



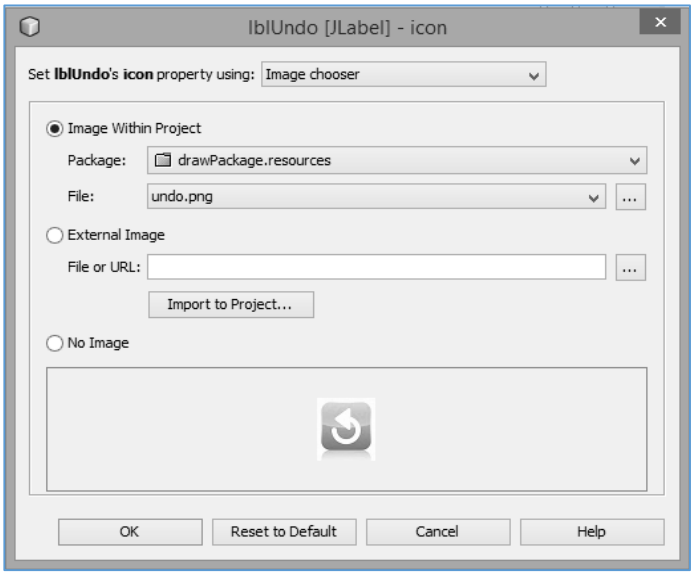
The panel will provide the drawing area for the application. We will now add an '***undo***' icon.

Go to the ***Projects*** window in the top left of the NetBeans page. Open the ***draw*** project until ***drawPackage*** is reached. Right-click on the drawPackage icon and select ***New*** / ***Java Package***. Name the package as '***resources***', then return to the NetBeans editing screen.

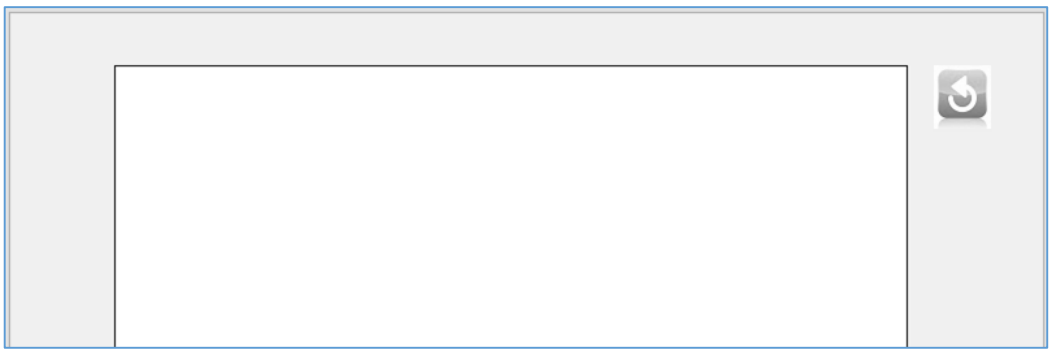


Before continuing, obtain or create a suitable icon for the 'undo' function. This should be in .JPG or .PNG format. Adjust the size of the icon to approximately 50 pixels square.

Add a label component to the form, to the right of the panel area. Rename the label as ***lblUndo***. Go to the Properties window and locate the ***icon*** property. Click the ellipsis (…) symbol to open a selection window. Use the '***Import to Project***' button to load the undo icon image.



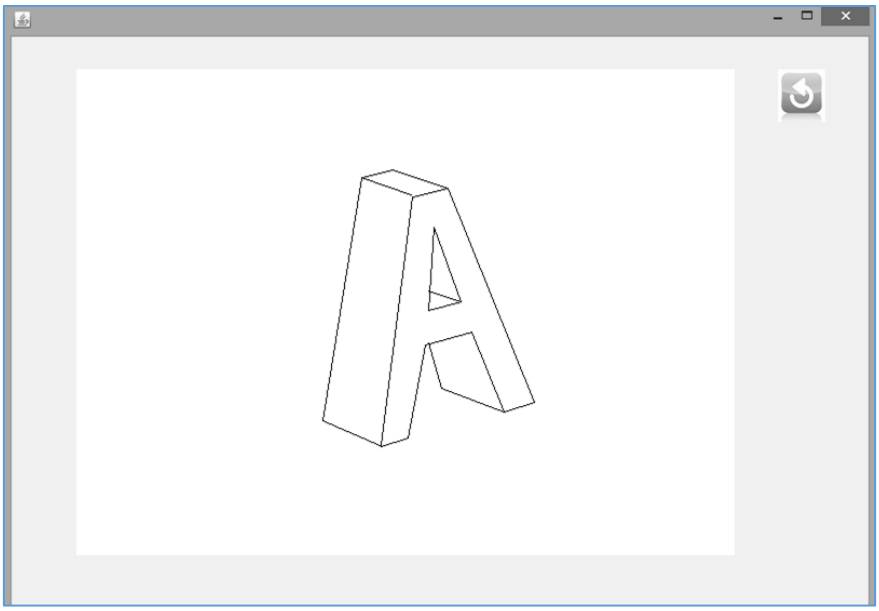
Return to the form layout page. Check that the icon image is displayed correctly. Right-click the icon and use the ***Edit Text*** option to remove the label caption.



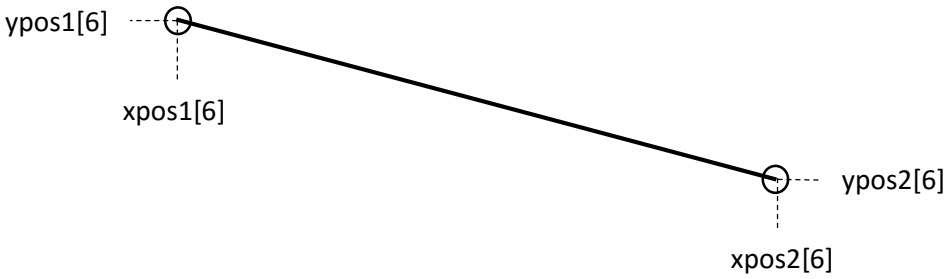
Use the Source tab to move to the program code screen. Add two Java modules which will be needed for producing the graphics, along with arrays to hold the x- and y-coordinates of the lines which we will be drawing.

```
package drawPackage;

import java.awt.Graphics2D;
import java.awt.Color;

public class draw extends javax.swing.JFrame {

    int[] xpos1=new int[100];
    int[] ypos1=new int[100];
    int[] xpos2=new int[100];
    int[] ypos2=new int[100];
    int count=0;
    Boolean drawing=false;

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

Images will be made up from a series of line segments.



For each line, the **x** and **y** pixel cooordinates for the end points will be stored in the corresponding elements of the four ***parallel arrays***. For example, for ***line segment 6***, the end coordinates will be stored as: ***xpos1[6]***, ***ypos1[6]***, ***xpos2[6]***, and ***ypos2[6]***



Add a ***drawImage( )*** method to plot the lines which are stored in the ***xpos*** and ***ypos*** arrays.

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

private void drawImage(int x, int y)
{
    Graphics2D g = (Graphics2D) pnlDraw.getGraphics();
    g.setColor(Color.white);
    g.fillRect(0,0, 620,500);
    g.setColor(Color.black);
    for (int i=1; i<count; i++)
    {
        g.drawLine(xpos1[i], ypos1[i], xpos2[i], ypos2[i]);
    }
    g.drawLine(xpos1[count], ypos1[count], x, y);
}
```

The ***drawImage( )*** method begins by clearing the drawing area using a white rectangle fill. A loop will then plot each of the completed lines of the drawing.

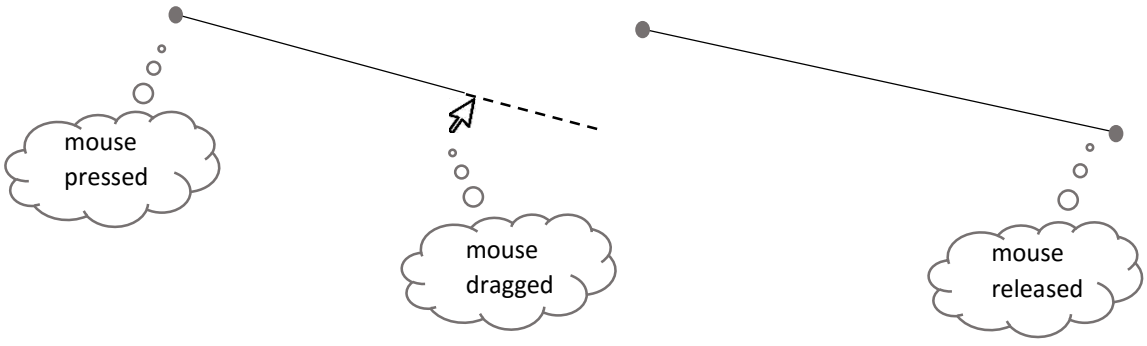
The last command of the method:

g.drawLine(xpos1[count], ypos1[count], x, y);

draws a line from the start of the final line segment to the current mouse position (***x***, ***y***). This will produce a ***'rubber band***' line which can expand as the mouse is dragged to the required finishing position of the new line segment.

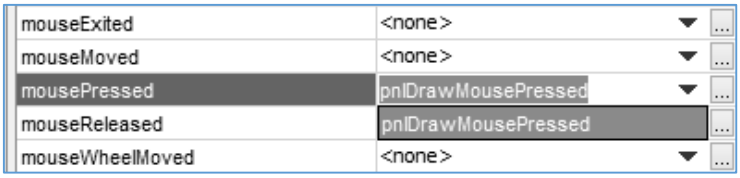
Java provides different methods to respond to: the mouse button being pressed, the mouse being dragged with the button held down, and the mouse button being released. We will make use of all three of these:

mousePressed	the ***start position*** of the line (***xpos1***, ***ypos1***) will be recorded
mouseDragged	we will draw a rubber band line from the ***start position*** of the line to the ***current mouse position***
mouseReleased	the ***end position*** of the line (***xpos2***, ***ypos2***) will be recorded



Use the ***Design*** tab to move to the form layout page, then select the white panel. Go to the Properties window and open the ***Events*** list.

Locate the ***mousePressed*** event, and select ***pnlDrawMousePressed*** from the drop down list.



Add program code to the ***pnlDrawMousePressed*** method which:

- gets the **x** and **y** cordinates for the current mouse position,
- adds one to the ***count*** of line segments,
- enters the mouse x and y cooordinates into the ***xpos1[ ]*** and ***ypos1[ ]*** arrays to record the start position for the line.

```
private void pnlDrawMousePressed(java.awt.event.MouseEvent evt) {
    int x=evt.getX();
    int y=evt.getY();
    count++;
    xpos1[count]=x;
    ypos1[count]=y;
    drawImage(x,y);
}
```

Click the ***Design*** tab to return to the form layout view. Select the white panel and go to the ***Events*** list in the Properties window. Locate the the ***mouseDragged*** event, and select ***pnlDrawMouseDragged*** from the drop down list.

For this method, we simply need to get the current mouse coordinates **x** and **y**, then pass these to the ***drawImage( )*** method so that a rubber band line can be drawn from the starting point of the line to the current mouse position.

```
private void pnlDrawMouseDragged(java.awt.event.MouseEvent evt) {
    int x=evt.getX();
    int y=evt.getY();
    drawImage(x,y);
}
```

Return once more to the form layout view. Select the panel, go to the Events list in the Properties window, and locate the the ***mouseReleased*** event. Select ***pnlDrawMouseReleased***.

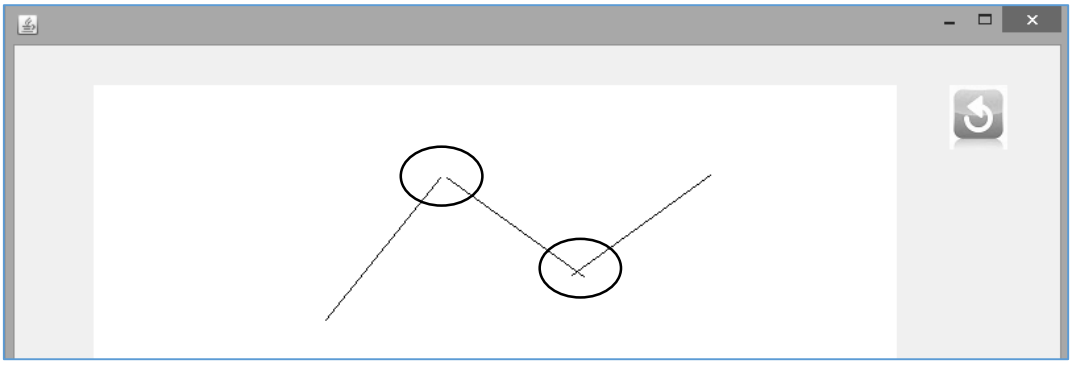
In this method we will get the mouse x and y cooordinates and store these in the *xpos2[ ]* and ***ypos2[ ]*** arrays to mark the end of the line segment.

```
private void pnlDrawMouseReleased(java.awt.event.MouseEvent evt) {
    int x=evt.getX();
    int y=evt.getY();
    xpos2[count]=x;
    ypos2[count]=y;
    drawImage(x,y);
}
```

Run the program. Draw a line by:

- pressing the mouse button down on the start point,
- dragging the mouse to the finish point, then
- releasing the mouse button.

Repeat this sequence to produce more lines.



You may notice that it is difficult to draw a continuous line made of from separate segments, as gaps may or appear or the ends of the lines may cross if the mouse is not positioned accurately at the start of the line. We can reduce this difficulty by making the start of each line segment snap to the end of the previous line if this is sufficiently close.

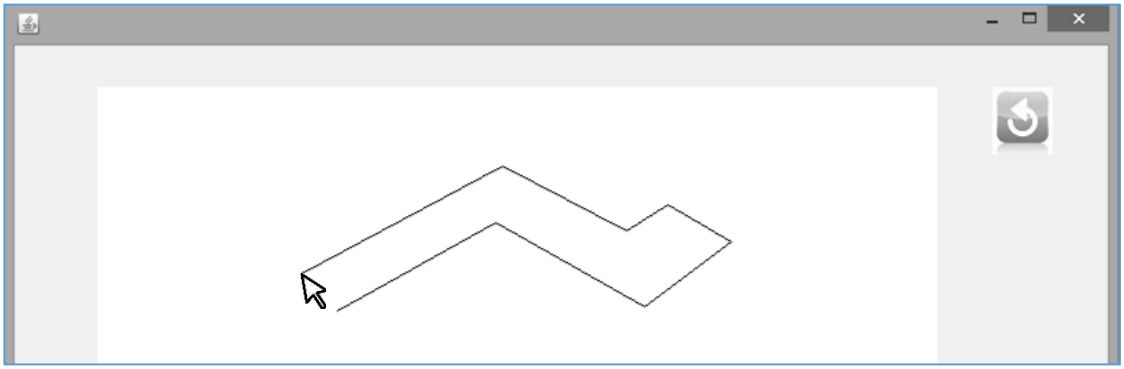
Close the program window and return to the program code view. Locate the ***pnlDrawMousePressed*** method. We will add lines of code to operate the ***snap*** procedure. These lines carry out a series of tasks when the user presses the mouse button down at the start of a new line segment:

- A loop checks each of the previous lines in turn.
- The difference in x position is calculated between the end of the previous line and the start of the current line.
- The difference in y position is similarly calculated.
- If both the x difference and y difference are less than 5 pixels, then the start position of the current line is snapped to the end position of this previous line segment.

Add lines of code:

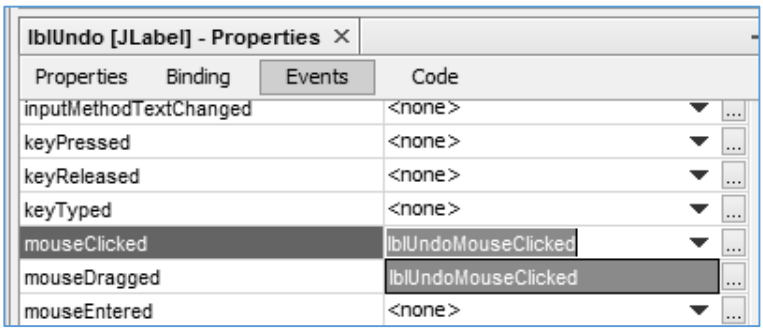
```
private void pnlDrawMousePressed(java.awt.event.MouseEvent evt) {
   int x=evt.getX();
   int y=evt.getY();
   count++;
   xpos1[count]=x;
   ypos1[count]=y;

   for(int j=1;j<count;j++)
   {
       int xDiff= Math.abs(xpos2[j]-xpos1[count]);
       int yDiff= Math.abs(ypos2[j]-ypos1[count]);
       if (xDiff<5 && yDiff<5)
       {
           xpos1[count]=xpos2[j];
           ypos1[count]=ypos2[j];
       }
   }

   drawImage(x,y);
}
```

Run the program and check that it is now easier to create a continuous series of line segments without gaps or lines crossing.



Close the program window and return to the NetBeans editing screen. Use the ***Design*** tab to move to the form layout view. We will now create a button click method for the '***Undo***' icon.

Select the '***Undo***' icon and go to the ***Properties*** window. Click the ***Events*** tab and locate the ***mouseClicked*** method. Select ***lblUndoMouseClicked*** from the drop down list.



The button click method will open on the program code page.

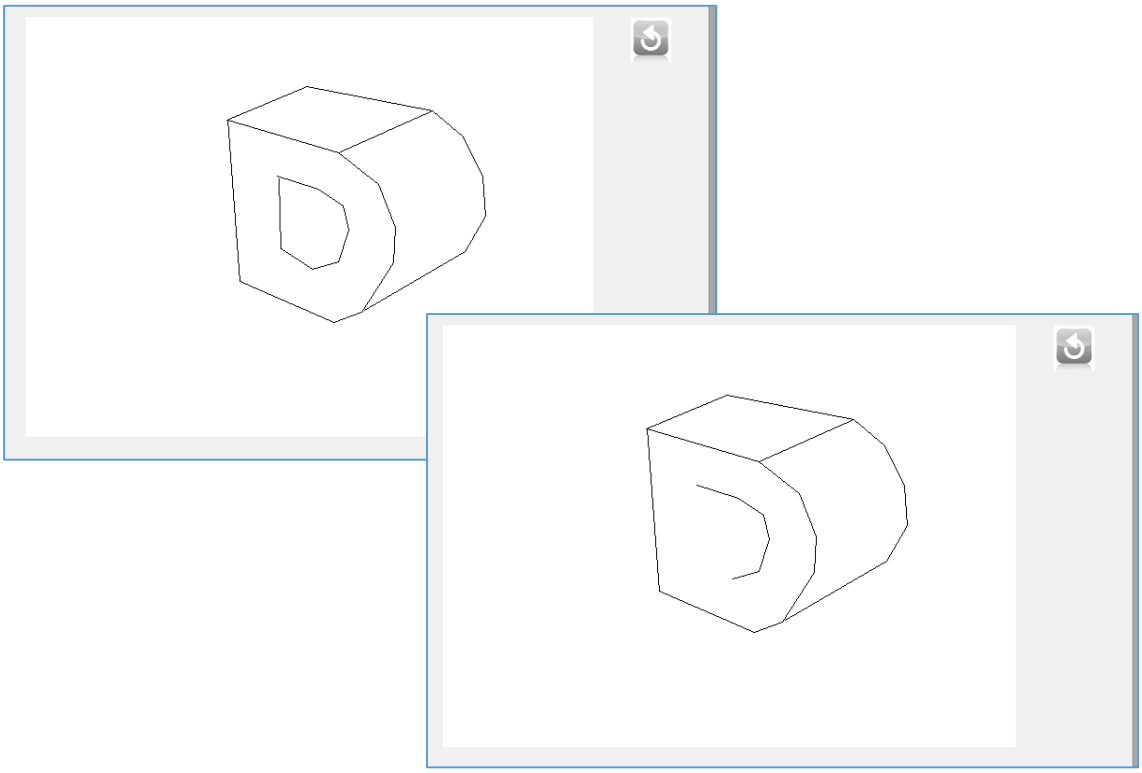
It is very easy to operate a ***stack*** using the **x-** and ***y-coordinate arrays*** and the ***count*** variable. ***Count*** is acting as a pointer to the array position where the most recent line has been entered. To remove a line, we only need to decrease the value of ***count*** by one, so that the last line segment is no longer included when the ***drawImage( )*** method is called. If the user enters a replacement line, ***count*** will be increased again and the new x- and y-coorodinates will overwrite the old values.

We will check that ***count*** is greater than zero before attempting to delete a line, to prevent an error occurring with an already empty array.

After deleting the most recent line, we call ***drawImage( )*** to update the screen display. A rubber band line should not be drawn to the current position of the mouse, which is clicking the ***Undo*** button! We will therefore set the final drawing position as the end of the previous line segment at ***xpos2[count]***, ***ypos2[count]***.

```
private void lblUndoMouseClicked(java.awt.event.MouseEvent evt) {
    if (count>0)
    {
       count--;
       drawImage(xpos2[count], ypos2[count]);
    }
}
```

Run the program. Enter a series of lines, then check that these can be deleted in the correct reverse order.



For the final project in this chapter, we will develop a program which uses a ***queue*** data structure.

> The County Highways Department needs to carry out major repairs to a bridge on a main road. The work will take a number of months, and will require the road across the bridge to be reduced to a single lane controlled by traffic lights.
>
> Before beginning the work, the project planners need to estimate the likely maximum length of traffic queues at the bridge when the traffic lights are in operation, and the likely maximum waiting time for road users.
>
> It is intended that the traffic lights will be alternately red and green for periods of three minutes each. During the time that the traffic lights are green, a maximum of eight vehicles per minute will be able to cross the bridge.
>
> A traffic survey has been carried out near the bridge during busier times of the day. It is found that the number of cars arriving in any minute is randomly distributed between a minimum of zero and a maximum of six.

We will set up a program to run a simulation of this scenario, and determine the maximum length of traffic queue and maximum vehicle waiting time.

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

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

Return to the NetBeans editing screen.

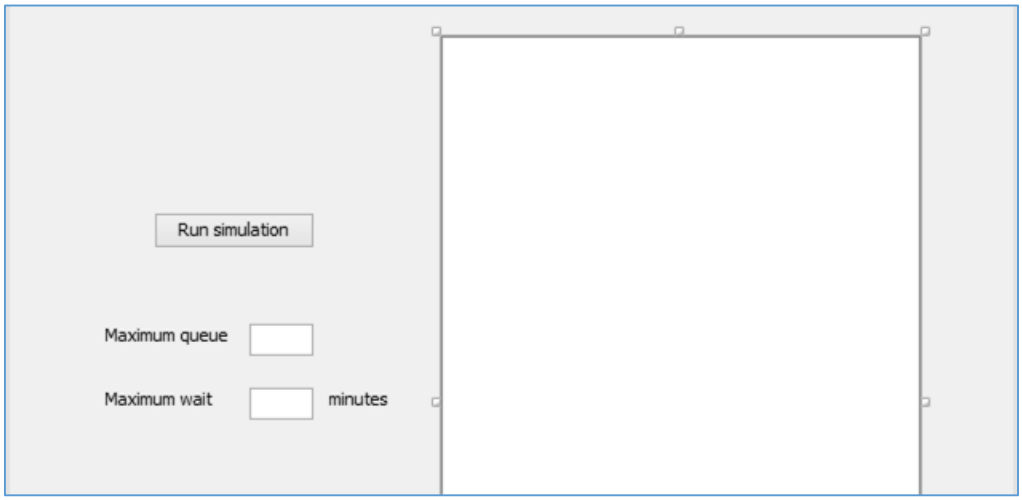
- Right-click on the ***form***, and select ***Set layout*** / ***Absolute layout***.
- Go to the ***Properties*** window on the bottom right of the screen and click the ***Code*** tab. Select the option: ***Form Size Policy*** / ***Generate pack()*** / ***Generate Resize code***.
- Click the Source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

Add components to the form as shown below:

- A ***List*** component. Give this the name ***lstOutput***.
- A ***button*** with the caption '***Run simulation***' and the name ***btnRun***.
- A label ***'Maximum queue***', followed by a ***text field*** with the name ***txtMaxQueue***.
- A label ***'Maximum wait***', followed by a ***text field*** with the name ***txtMaxWait***. Follow the text field with a further label '***minutes***'.

Select the ***List***, then go to the Properties window. Locate the ***model*** property and delete the entries '***Item 1, Item 2, Item 3, Item 4, Item 5***' from the right hand column.



Use the ***Source*** tab to move to the program code page. Add Java modules which will be needed to generate random numbers and output results to the List box during the simulation.

```
package trafficLightsPackage;

import java.util.Random;
import javax.swing.DefaultListModel;

public class trafficLights extends javax.swing.JFrame {

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

Use the ***Design*** tab to return to the form layout page. Double click the '***Run simulation***' button to create a method. Begin by adding variables which will be needed for the simulation.

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

    int[] car=new int[100];
    int front=1;
    int back=1;
    int minute;
    int queue=0;
    int waitingTime;
    int arrival;
    int passing;
    int maxQueue=0;
    int maxWaitingTime=0;
    String colour="red";
    DefaultListModel listModel = new DefaultListModel();

}
```

Car[ ] is an array with 100 elements, to represent the vehicles queueing at any time. We are making an assumption that the queue will never exceed 100 vehicles, but the array could be enlarged later if this turns out not to be the case.

The ***car[ ]*** array will record the number of the minute in which each vehicle ***arrives*** at the traffic lights and joins the queue. It will be possible to calculate the waiting time by comparing the arrival time with the time when the vehicle passes the lights and leaves the queue. For example:

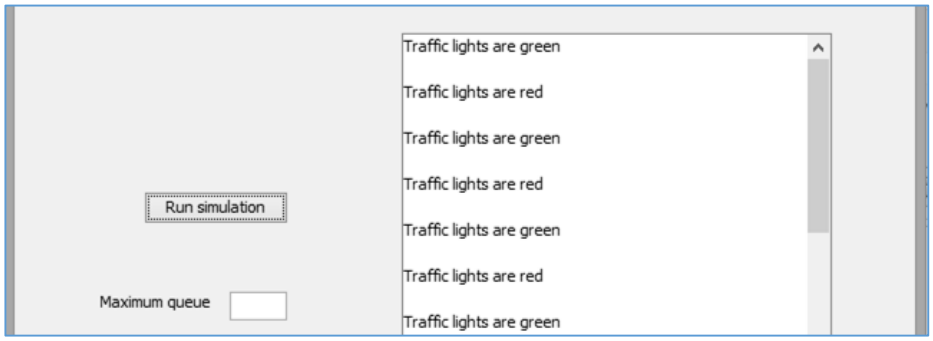
vehicle arrives in minute 17
vehicle passes the traffic lights in minute 20
waiting time 3 minutes

Front and ***back*** are pointers to allow us to operate a ***circular queue structure*** with the ***car[ ]*** array.

The next step is to produce a loop which operates for each traffic light colour change. It will be reasonable to run the simulation for 50 trafic light changes, representing two and a half hours of real time. Add code which will set the 'colour' variable alternately to red or green, then display the traffic light state in the list box.

```
private void btnRunActionPerformed(java.awt.event.ActionEvent evt) {
   int[] car=new int[100];
   int front=1;
   int back=1;
   int minute;
   int queue=0;
   int waitingTime;
   int arrival;
   int passing;
   int maxQueue=0;
   int maxWaitingTime=0;
   String colour="red";
   DefaultListModel listModel = new DefaultListModel();
   for (int t=0; t<50;t++)
   {
       if (colour.equals("red"))
       {
           colour="green";
       }
       else
       {
           colour="red";
       }
       listModel.addElement("Traffic lights are "+colour);
       listModel.addElement(" ");
       lstOutput.setModel(listModel);
   }
}
```

Run the program. Click the '***Run simulation***' button and check that changes of traffic light colour are shown, as in the illustration below. Adjust the width of the ***List*** if necessary, so that all the text is visible.

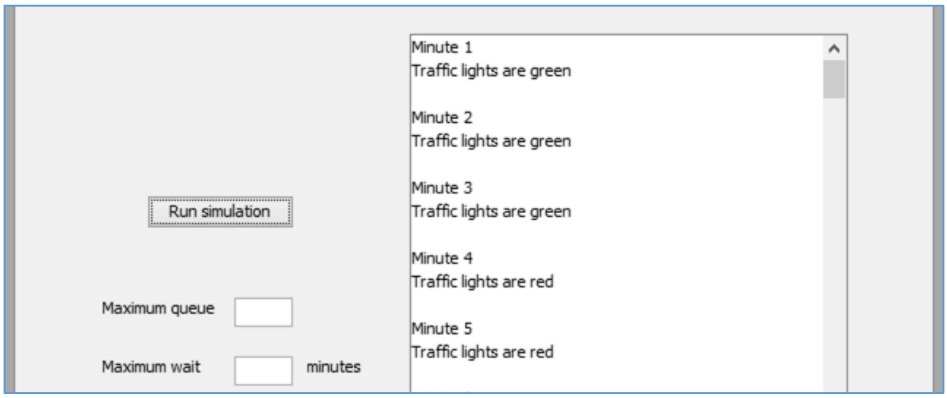


Close the program window and return to the ***Run simulation*** button click method on the program code page.

Each traffic light colour is displayed for ***three minutes***. We can add a loop which calculates each minute number from the start of the simulation. Three minutes have elapsed for each completed time period **t**, plus the number of minutes **m** since the start of the current time period.

```
    for (int t=0; t<50;t++)
    {
        if (colour.equals("red"))
        {
            colour="green";
        }
        else
        {
            colour="red";
        }
        for (int m=1; m<=3;m++)
        {
            minute=t*3 + m;
            listModel.addElement("Minute "+minute);
            listModel.addElement("Traffic lights are "+colour);
            listModel.addElement(" ");
            lstOutput.setModel(listModel);
        }
    }
}
```

Run the program. Click the '***Run simulation***' button and check that minutes are shown correctly.

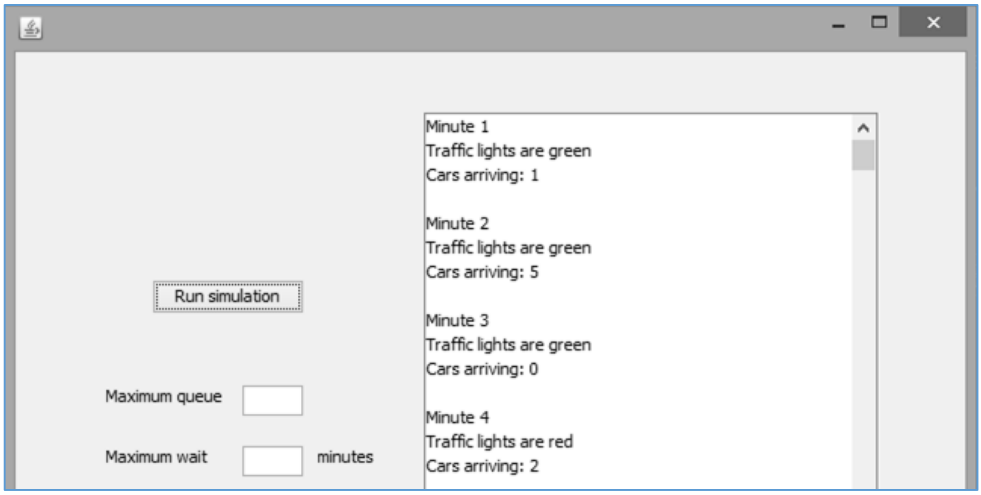


Close the program window and return to the code page.

We will now use the random number generator to create a random number between 0 and 6, to represent cars arriving at the traffic lights each minute.

```
for (int m=1; m<=3;m++)
{
    minute=t*3 + m;

    Random r = new Random();
    arrival= r.nextInt(7);

    listModel.addElement("Minute "+minute);
    listModel.addElement("Traffic lights are "+colour);

    listModel.addElement("Cars arriving: "+arrival);

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

Run the program. Check that the numbers of cars arriving each minute are within the correct range, and appear randomly distributed. Different sets of random numbers should be produced each time the 'Run simulation' button is clicked.



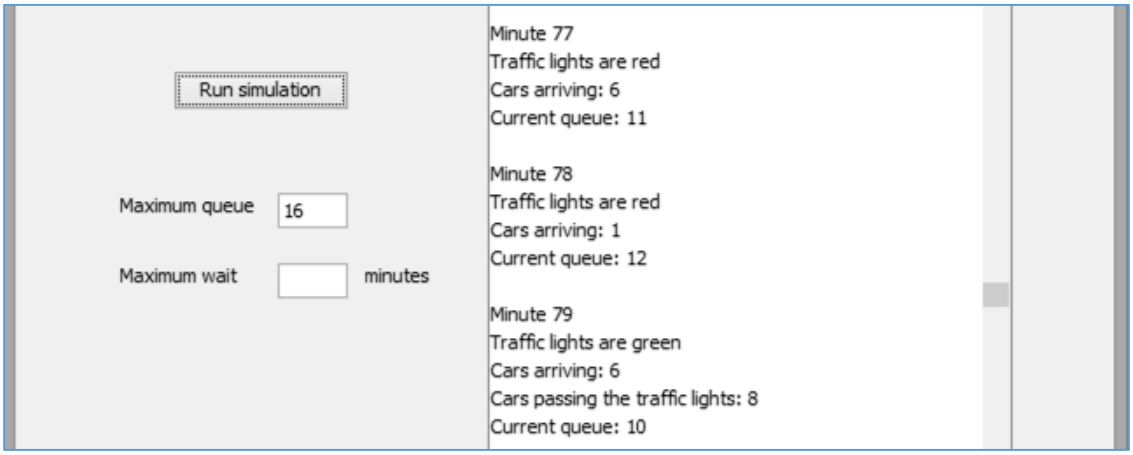
Close the program window and return to the code editing screen.

The next step in the simulation is to determine the length of any queue each minute, and determine the maximum queue length during the simulation period. We will add lines of code to the '***Run simulation***' method which will:

- Add vehicles arriving each minute to the current queue.
- If the traffic light is green, allow the waiting vehicles to pass the road works and leave the queue, up to a maximum of eight vehicles per minute.
- If the current queue is the longest so far in the simulation, make this the new maximum queue length.
- Output the current queue length at the end of each minute, and output the maximum queue length when the simulation ends.

```
        for (int m=1; m<=3;m++)
        {
            minute=t*3 + m;
            Random r = new Random();
            arrival= r.nextInt(7);

            queue=queue+arrival;
            passing=0;
            if (colour.equals("green"))
            {
                passing=8;
                if (queue<8)
                {
                    passing=queue;
                }
                queue=queue-passing;
            }
            if (queue>maxQueue)
            {
                maxQueue=queue;
            }

            listModel.addElement("Minute "+minute);
            listModel.addElement("Traffic lights are "+colour);
            listModel.addElement("Cars arriving: "+arrival);

            if (colour.equals("green"))
            {
                listModel.addElement("Cars passing the traffic lights: "+passing);
            }
            listModel.addElement("Current queue: "+queue);

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

    txtMaxQueue.setText(Integer.toString(maxQueue));
}
```

Start the program and run a simulation. Scroll the output list to a point in the middle of the simulation period. Check that the numbers of vehicles arriving, numbers of vehicles passing the traffic lights and the queue lengths are consistant over a period of several minutes.



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

We will try to calculate the waiting times for individual vehicles. The first step is to record the arrival time of each vehicle in a queue structure. We will add lines of code to do this:

- If any vehicles have arrived at the traffic lights during the current minute, then carry out a loop for each of the arriving vehicles.
- Add the number of the ***arrival minute*** to the ***car[ ]*** array at the position of the ***back pointer***, then move the back pointer to the next array element, ready for the next vehicle to join the queue.
- If the back pointer has passed the end of the array, then reset the back pointer to array element 1. This provides a circular queue structure which can continue indefinitely, provided that the number of vehicles in the queue never exceeds 100.

```
for (int m=1; m<=3;m++)
{
    minute=t*3 + m;
    Random r = new Random();
    arrival= r.nextInt(7);
    if (arrival>0)
    {
        for (int c=1; c<=arrival; c++)
        {
            car[back]=minute;
            back++;
            if (back>99)
            {
                back=1;
            }
        }
    }
    queue=queue+arrival;
    passing=0;
    if (colour.equals("green"))
    {
```

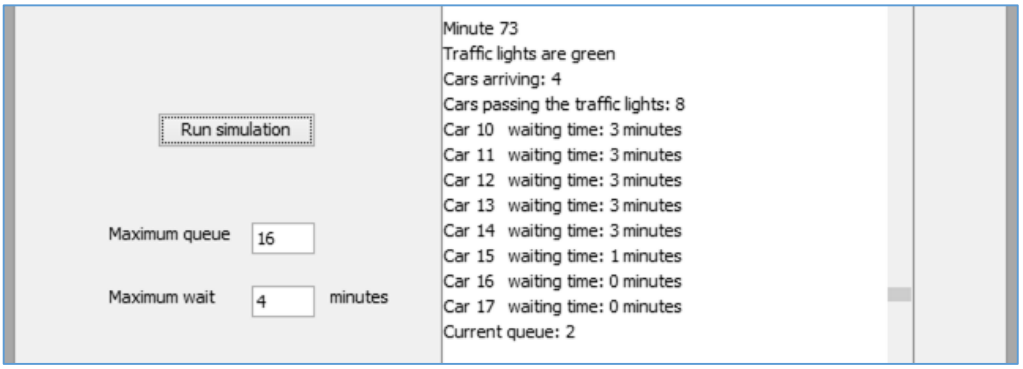
Move down to the section of the method which handles the periods when the traffic light is green. We will now process the vehicles which pass the traffic light and leave the queue.

We will add lines of code, as shown below, to carry out a series of tasks:

- If there are any vehicles in the queue which will pass the traffic lights, then loop for each of the vehicles passing.
- Obtain the time of arrival of the vehicle from the ***car[ ]*** array. Calculate the waiting time by subtracting the arrival time from the current minute.
- If this waiting time is the longest found so far in the simulation, then make this the ***maximum waiting time***.
- Display the waiting time for the current vehicle.
- Remove the vehicle from the queue by advancing the ***front*** pointer. If the pointer moves past the end of the array, then reset the front pointer position to array element 1, so that a circular queue is created.
- At the end of the simulation, output the maximum waiting time.

```
            listModel.addElement("Minute "+minute);
            listModel.addElement("Traffic lights are "+colour);
            listModel.addElement("Cars arriving: "+arrival);
            if (colour.equals("green"))
            {
                listModel.addElement("Cars passing the traffic lights: "+passing);
                if (passing>0)
                {
                    for (int c=1;c<=passing;c++)
                    {
                        waitingTime = minute-car[front];
                        if (waitingTime>maxWaitingTime)
                        {
                            maxWaitingTime=waitingTime;
                        }
                        listModel.addElement("Car "+front+
                                    "   waiting time: "+waitingTime+" minutes");
                        front++;
                        if (front>99)
                        {
                            front=1;
                        }
                    }
                }
            }
            listModel.addElement("Current queue: "+queue);
            listModel.addElement(" ");
            lstOutput.setModel(listModel);
        }
    }
    txtMaxQueue.setText(Integer.toString(maxQueue));
    txtMaxWait.setText(Integer.toString(maxWaitingTime));
}
```

Note that the line beginning '***listModel.addElement("…***' should be entered as a single line of code. Start the program and run a simulation. Check that the results are consistant and are displayed correctly.



Run the simulation a number of times. What advice would you give to the Highways Department concerning the likely maximum queue length and maximum waiting time for vehicles?