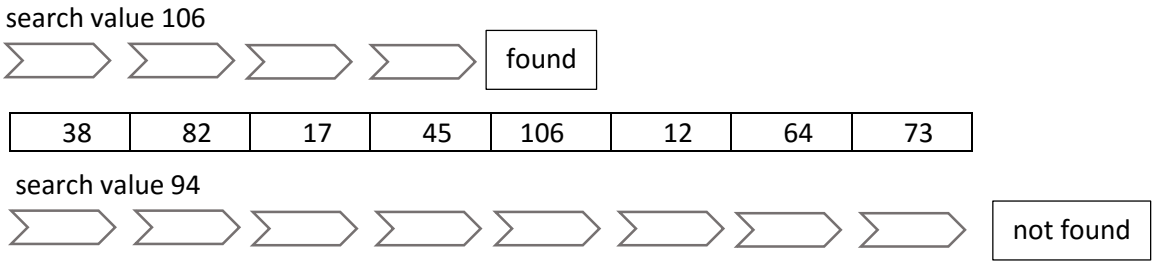
9 Searching

In the second part of this chapter, we will turn our attention to methods for finding a particular record within a set of data. The method that is used will depend on the way in which the data is organised. For data that is in random order, a ***linear search*** is used. However, if the data has been sorted into numerical or alphabetical order then a faster ***binary search*** method can be used.

Linear search

Linear searches are used with unsorted data. The procedure is simple:

- begin at the first data item,
- check each item in turn,
- stop when either the required data is found, or when the end of the data set is reached,
- give a message to the user if the required data was not present.



To explore how a linear search operates, we will set up a program which allows the user to search a set of unsorted data:

> A shop identifies twelve products by stock codes composed of a pair of upper case letters followed by a pair of digits, for example:
>
> ***FG78, RT12***
>
> The stock codes are stored in a table, and are not sorted into order.
>
> A program is required which will search the data:
>
> - If the required stock code is found, its position in the table should be displayed.
> - If the stock code is not found, a warning message should be given.

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

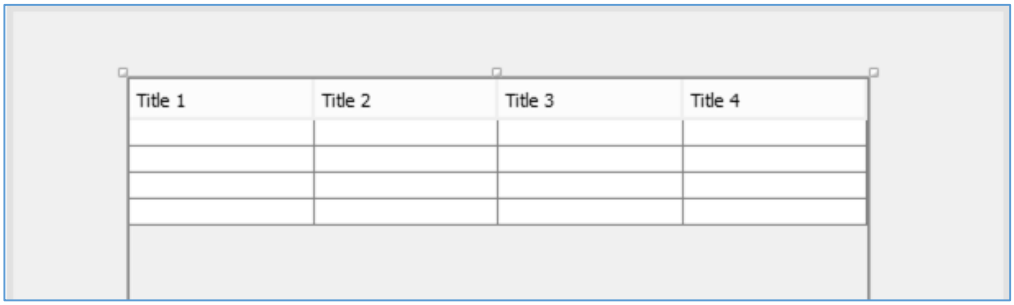
Return to the NetBeans editing page. Right-click on the ***linearSearch*** project, and select ***New*** / ***JFrame Form***. Give the ***Class Name*** as ***linearSearch***, and the ***Package*** as ***linearSearchPackage***:

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 on the ***Design*** tab to move to the form layout view.

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

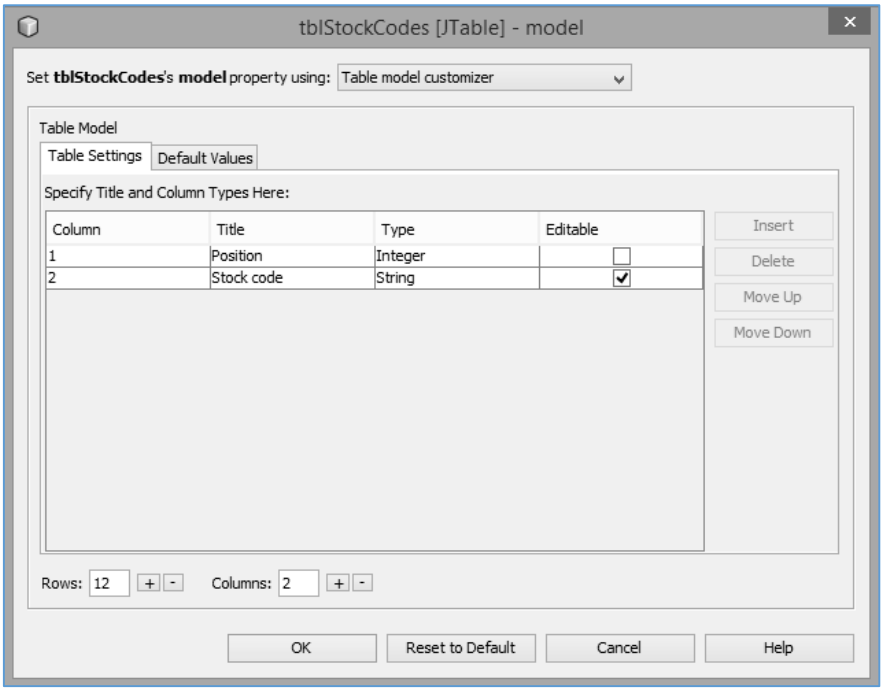


Go to the ***Properties*** window for the table and locate the ***model*** property. Click in the right column to open the editing window. Set the number of ***Rows*** to 12, and the number of ***Columns*** to 2.

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

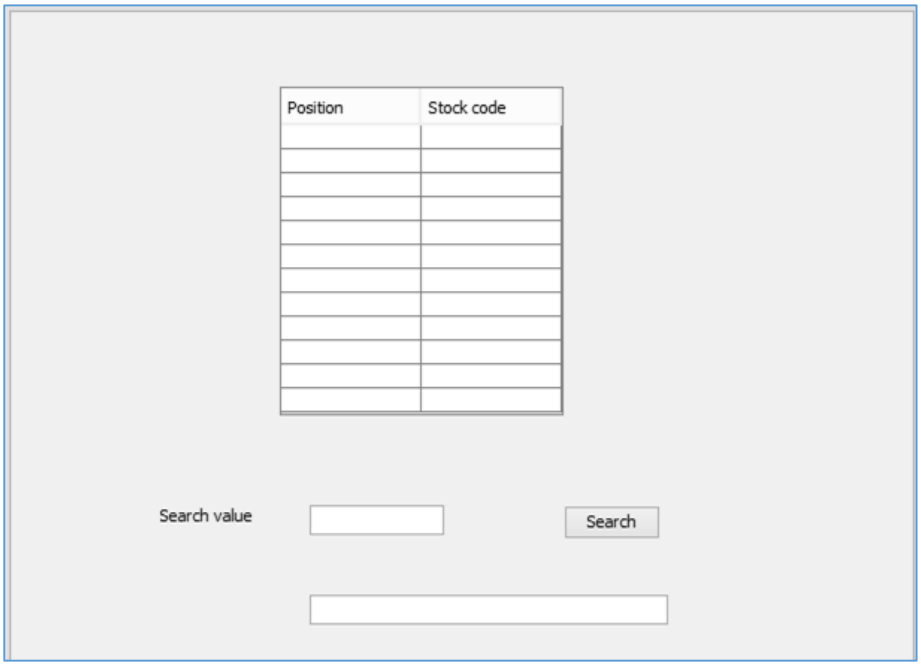
Position	***Integer***
Stock code	***String***

Set only the Stock code column to be editable.



Click the ***OK*** button to return to the form design screen. Check that the table headings are displayed correctly. Add:

- a label "***Search value***", with a text field alongside called ***txtSearchValue***.
- a button with the caption "***Search***". Rename this as ***btnSearch***.
- a text field for output of the result of the search. Rename this as ***txtResult***.



Click the Source tab to move to the program code screen.

The program should search a set of twelve stock codes, each composed of a pair of upper case letters followed by a pair of digits, such as ***FG78*** or ***RT12***. Rather than having to create a set of test data, we can make the program generate the stock codes using random numbers.

Add the Java ***random number*** module at the start of the program. Define an array called ***stockCode*** to hold the test data, then insert a loop in the ***linearSearch( )*** method:

```
package linearSearchPackage;

import java.util.Random;

public class linearSearch extends javax.swing.JFrame {

   String[] stockCode=new String[13];

   public linearSearch() {
      initComponents();

      String stockCode;
      for (int i=0; i<12;i++)
      {

      }

   }
```

To generate the stock codes, we need to know how random numbers are produced in Java. To start the random number generator, we use:

Random r = new Random( );

We obtain random numbers with lines such as:

int n = r.nextInt(10);

The number in brackets after ***nextInt*** specifies the range for the random number. The command ***nextInt(10)*** will produce a number between **0** and **9**.

To produce the letters at the start of the stock code, we can make use of ***ASCII values***. These are numbers allocated to each of the characters of the keyboard, and are standardised for all computer systems. ***Upper case letters*** are numbered in sequence:

A has ASCII value 65,

B has ASCII value 66,

and so on up to ***Z*** which has ***ASCII code value 90***. If we generate a random number in the range **65-90**, this can be converted to the corresponding upper case letter.

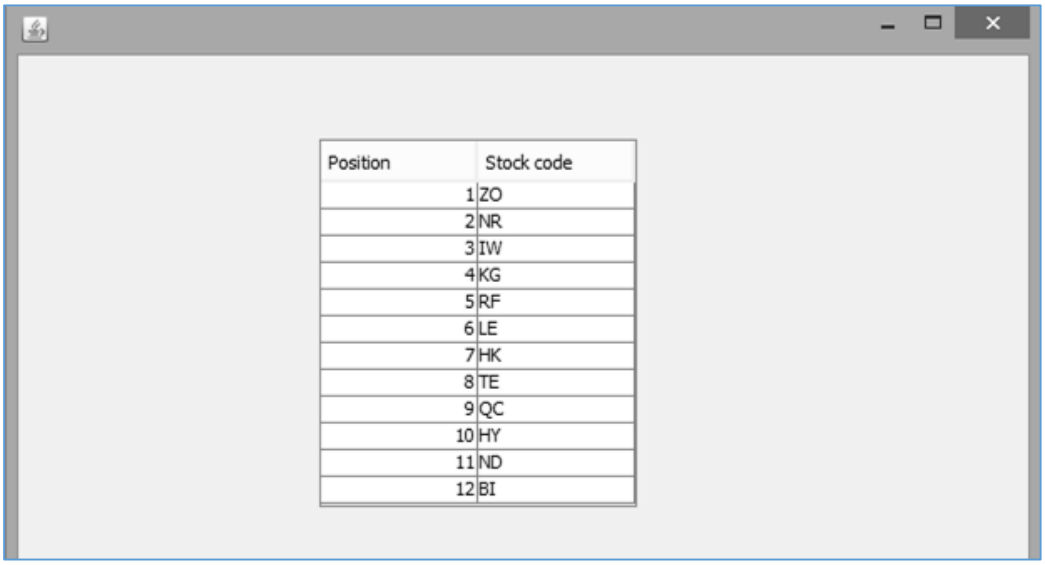
Add lines of code to the loop in the ***linearSearch( )*** method. We begin by inserting a sequence number into the left column of the table. The random number generator is started, and a loop generates two ASCII values in the range 65-90 using the formula:

65 + r.nextInt(26)

The ASCII values are converted to upper case letters, which are then added to create the stock code. The stock code is finally displayed in the right column of the table.

```
public linearSearch() {
   initComponents();
   String stockCode;
   for (int i=0; i<12;i++)
   {
      tblStockCodes.getModel().setValueAt(i+1,i,0);
      Random r = new Random();
      stockCode="";
      for (int j=0;j<2;j++)
      {
           char randomChar = (char) (65 + r.nextInt(26));
           stockCode += String.valueOf(randomChar);
      }
      tblStockCodes.getModel().setValueAt(stockCode,i,1);
   }
}
```

Run the program. Check that a ***Position*** number and random two letter ***Stock code*** are displayed on each row of the table, similar to the illustration below.



Position	Stock code
1	ZO
2	NR
3	IW
4	KG
5	RF
6	LE
7	HK
8	TE
9	QC
10	HY
11	ND
12	BI

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

We need to add two digits, each in the range ***0-9***, to complete the stock code. Numbers have been allocated ASCII values in a similar way to letters:

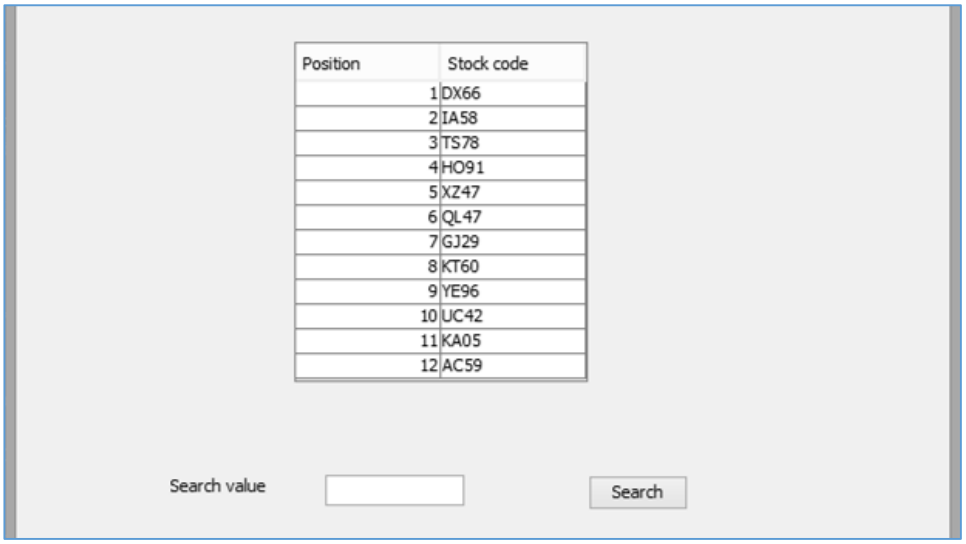
0 has ASCII value 48,

1 has ASCII value 49,

and so on up to ***9*** which has ***ASCII code value 57***. Return to the loop in the ***linearSearch( )*** method. Add lines of code to generate ***ASCII values*** between ***48*** and ***57***, then convert these into characters to add to the stock code:

```
for (int i=0; i<12;i++)
{
    tblStockCodes.getModel().setValueAt(i+1,i,0);
    Random r = new Random();
    stockCode="";
    for (int j=0;j<2;j++)
    {
         char randomChar = (char) (65 + r.nextInt(26));
         stockCode +=String.valueOf(randomChar);
    }
    for (int j=0;j<2;j++)
    {
         char randomChar = (char) (48 + r.nextInt(10));
         stockCode +=String.valueOf(randomChar);
    }
    tblStockCodes.getModel().setValueAt(stockCode,i,1);
}
```

Run the program. Check that full stock codes consisting of two letters and two digits are generated, as shown below.



Close the program window and return to the NetBeans editing screen. Use the ***Design*** tab to move to the form layout page.

We are now ready to set up the search procedure. A flow chart for the ***linear search*** algorithm is given on the next page.

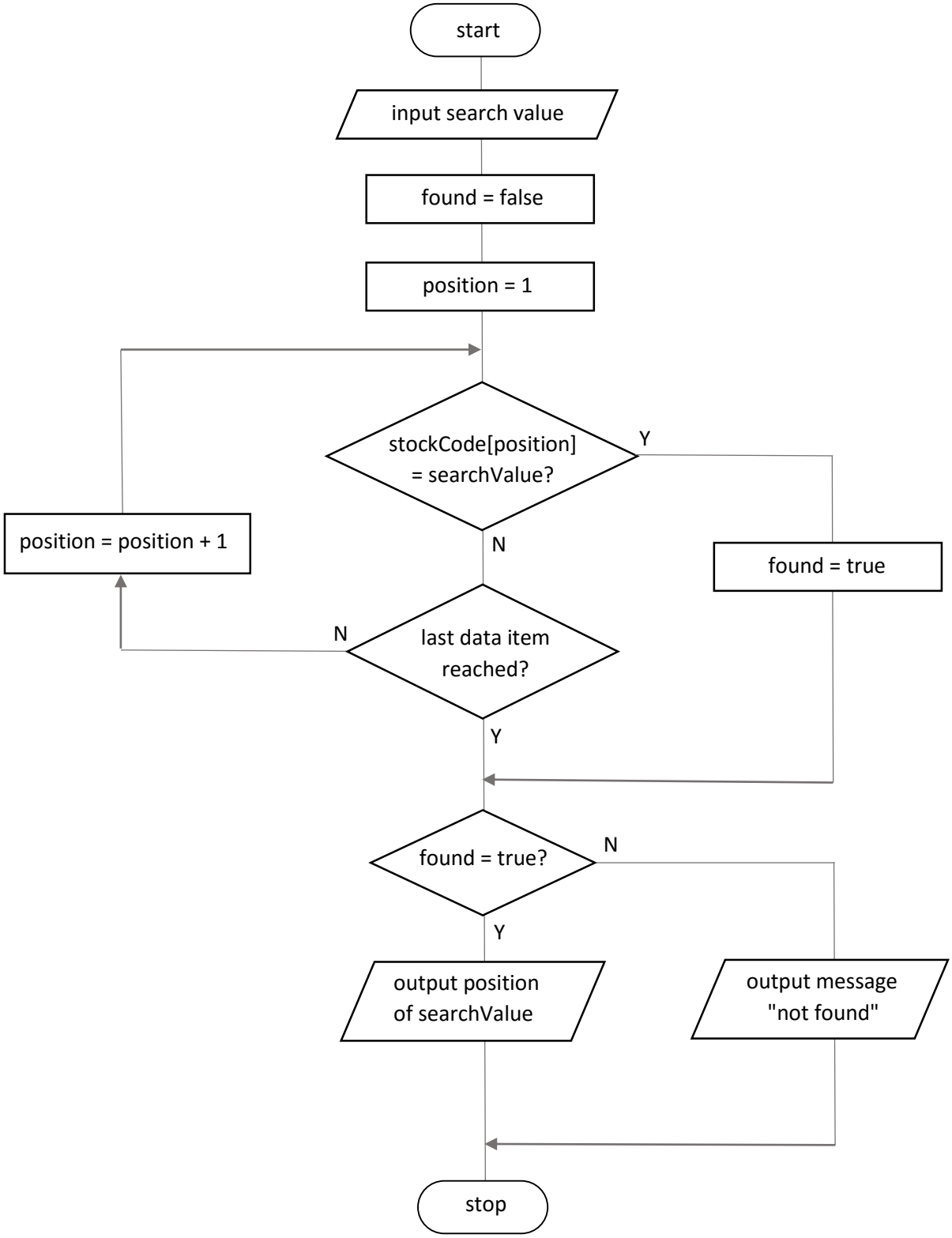
Double click the '***Search***' button to create a method. Begin by adding a loop to transfer the stock codes from the table into the ***stockCode*** array. We will also collect the required search value from the ***txtSearchValue*** text field.

```
private void btnSearchActionPerformed(java.awt.event.ActionEvent evt) {
    for (int i=1; i<=12;i++)
    {
        stockCode[i]=(String) tblStockCodes.getModel().getValueAt(i-1,1);
    }
    String searchValue=txtSearchValue.getText();
}
```

We will now add the loop to check each of the data items until either the required stock code is found, or the end of the array is reached.

```
    String searchValue=txtSearchValue.getText();
    int position=1;
    Boolean found=false;
    while (found==false && position<=12)
    {
        if(stockCode[position].equals(searchValue))
        {
            found=true;
        }
        else
        {
            position++;
        }
    }
}
```

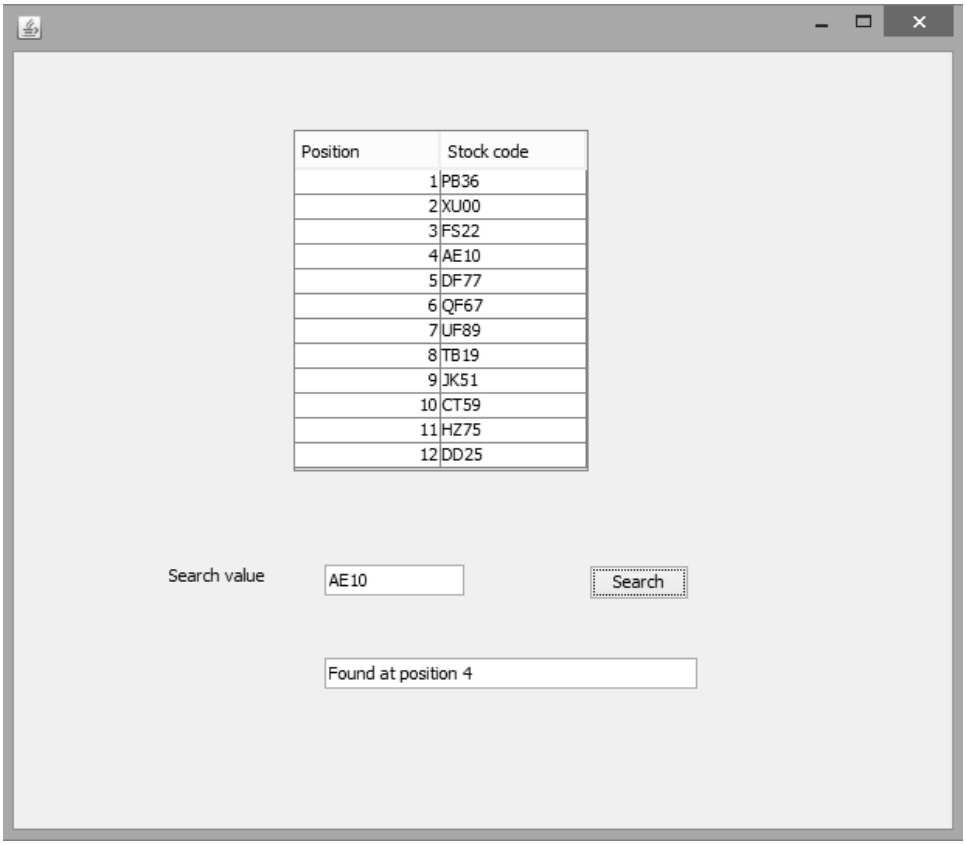
Linear search algorithm:



The final step is to output the position in the table where the required stock code was found, or give a message that the stock code was not found.

```
        while (found==false && position<=12)
        {
            if(stockCode[position].equals(searchValue))
            {
                found=true;
            }
            else
            {
                position++;
            }
        }
        if (found==true)
        {
            txtResult.setText("Found at position "+String.valueOf(position));
        }
        else
        {
            txtResult.setText("Search value not found");
        }
}
```

Run the program. Check that correct messages are given when searches are carried out for stock codes in the list, and also codes not present in the table.



Binary search

Linear searches are used with unsorted data, but we can user the faster and more efficient ***Binary Search*** method if the data is first sorted into alphabetical or numerical order. Speed can be important when searching large quantities of data.

The strategy used in a ***Binary Search*** is to first examine the ***middle*** item of the data set. There will be three possible outcomes:

- The required item is ***found*** and the search can end.
- The middle item is ***after*** the required item in the sequence.
- The middle item is ***before*** the required item.

For example, let us search for the item ***64*** in the data set:

12	17	38	45	64	73	82	106
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

The position of the middle item is found by averaging the index numbers of the first and last array elements, ignoring any decimal fraction:

$$\frac{(0+7)}{2} = 3.5$$ which is truncated to 3

Element 3 will be checked first:

12	17	38	45	64	73	82	106
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
left			middle				right

The required search item is 64, so this must lie after position 3 which contains a smaller number. All the array elements up to position 3 can be eliminated from the search, and the left pointer for the remaining group of search items is moved to position 4. The binary search is now repeated, focussing only on the remaining group of data items. A new middle position is calculated:

$$\frac{(4+7)}{2} = 5.5$$ which is truncated to 5

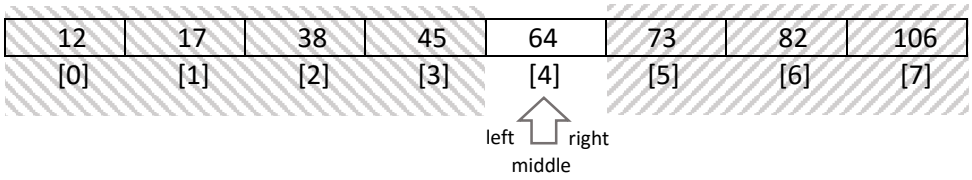
Element 5 will be checked next:

12	17	38	45	64	73	82	106
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
				left	middle		right

The required search item 64 is now less than the value of the middle item examined. All the array elements from position 5 upwards can be eliminated from the search, and the right pointer for the remaining group of items moved to position 4. A new middle position is calculated:

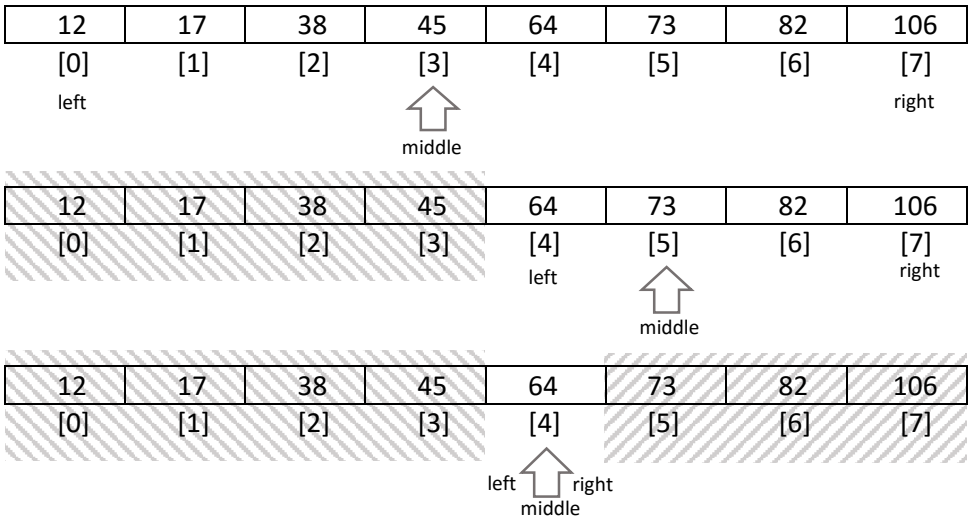
$$\frac{(4+4)}{2} = 4$$

Element 4 will be checked next. The required item has been found:

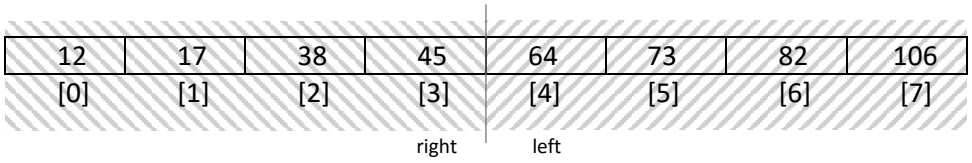


We have seen how this method will find items which are present in the data set, but what would happen if we tried to search for an item that was not present, such as ***58***?

The search would proceed in the same way as before, up to the point that position 4 was checked:



At this point we know that the required data item ***58*** should lie to left of position 4, so the right pointer is moved back to position 3.

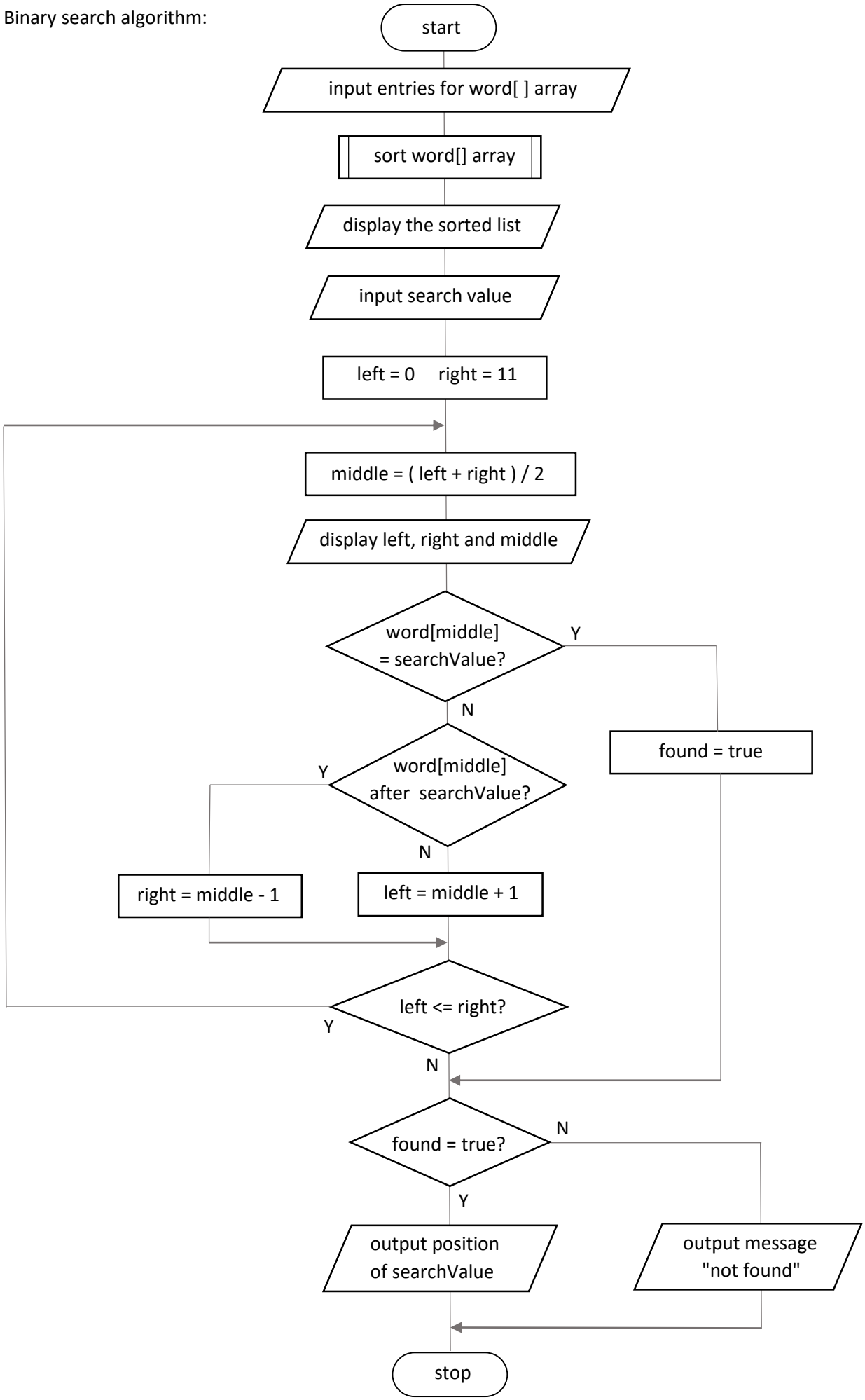


The left and right pointers, used to mark the limits of the remaining unsearched data, cross over one another. This indicates that there is no further data to search and required item cannot be present. The search can end, and a warning message can be displayed for the user.

> For the next project, we will produce a program to carry out a binary search of twelve words, sorted beforehand into alphabetical order. To demonstrate the operation of the binary search, the program will indicate the ***left***, ***right*** and ***middle*** positions of the remaining group of unsearched data at each stage during the search procedure.

A flowchart for the binary search algorithm is given on the next page.

Binary search algorithm:



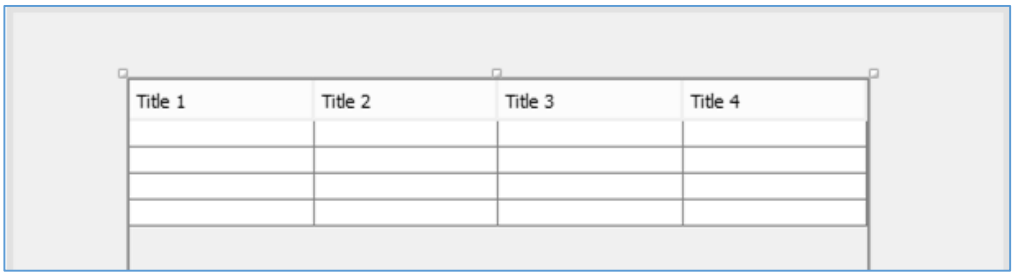
Begin the project in the standard way. Close all previous projects, then set up a ***New Project***. Give this the name ***binarySearch***, and ensure that the ***Create Main Class*** option is not selected.

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

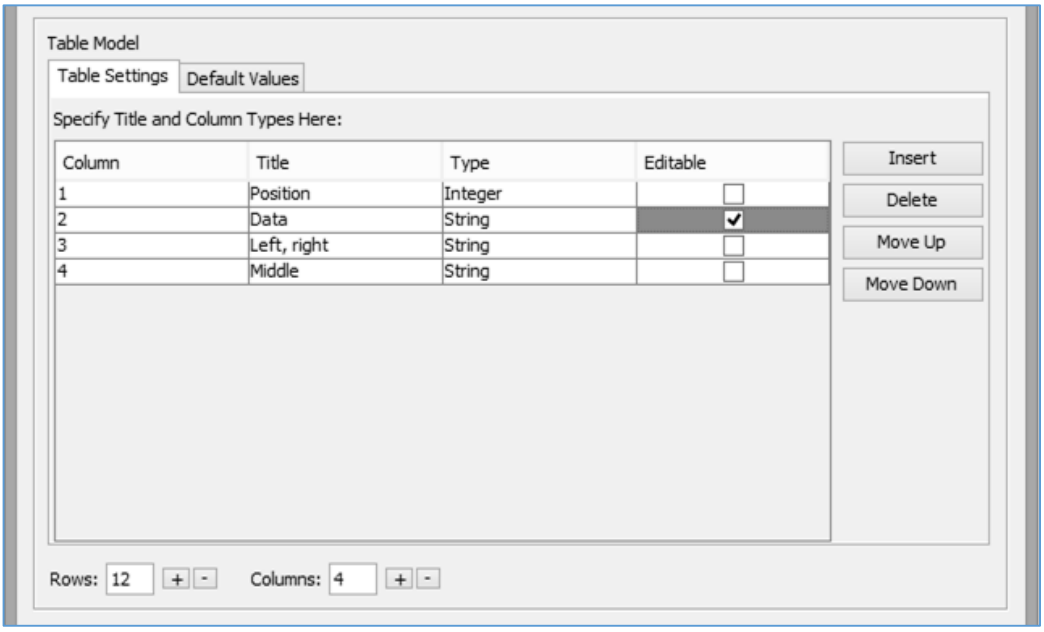


Go to the ***Properties*** window for the table and locate the ***model*** property. Click in the right column to open the editing window. Set the number of ***Rows*** to 12, and the number of ***Columns*** to 4.

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

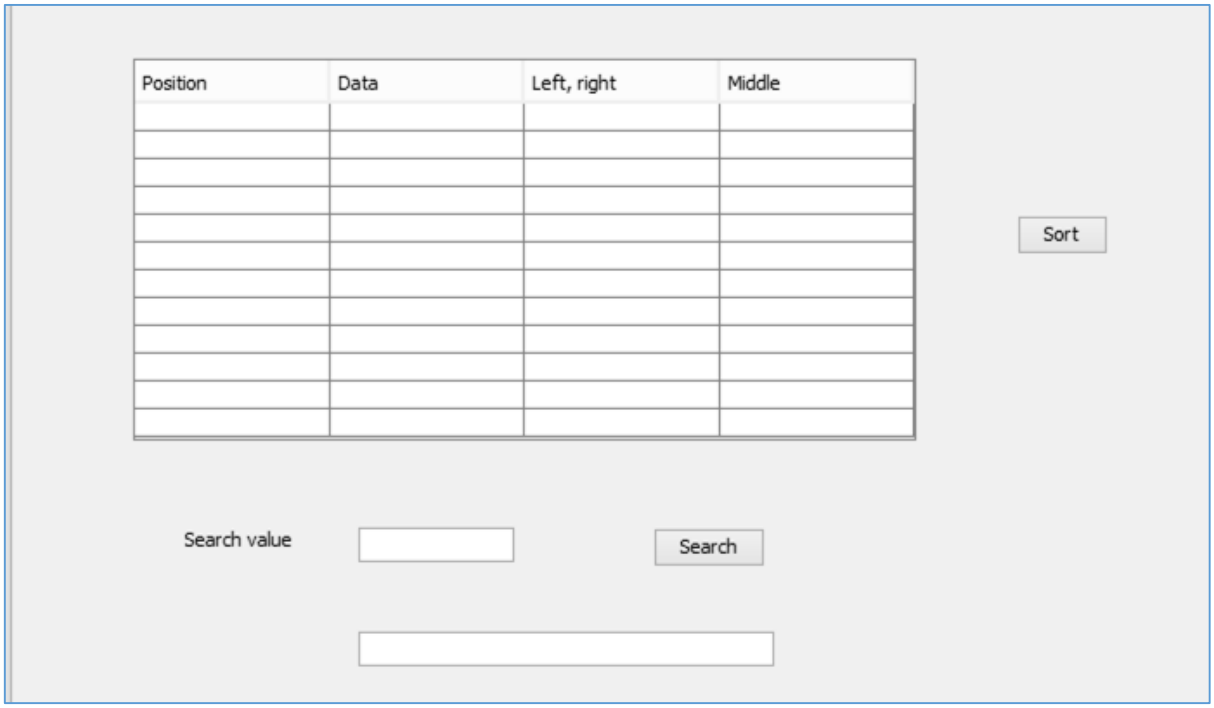
Position	***Integer***
Data	***String***
Left, right	***String***
Middle	***String***

Set only the ***Data*** column to be editable.



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

- a button with the caption "***Sort***". Rename this as ***btnSort***.
- a label "***Search value***", with a text field alongside called ***txtSearchValue***.
- a button with the caption "***Search***". Rename this as ***btnSearch***.
- a text field for output of the result of the search. Rename this as ***txtResult***.



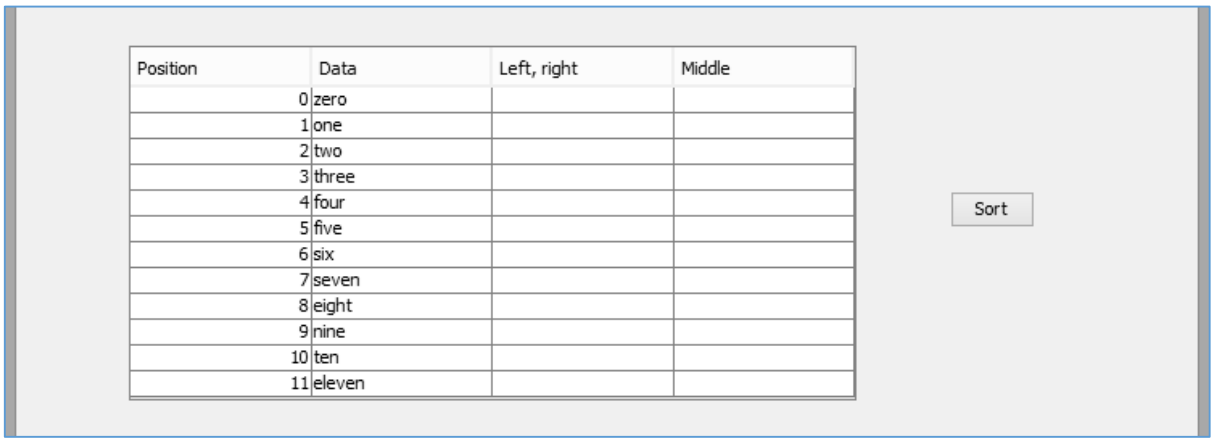
Use the ***Source*** tab to move to the program code view. We will begin by adding a Java module for editing the table and defining an array to hold the data values. Add a series of words to the array as test data. The user will be able to edit these entries later if they wish, when the program is running.

```
package binarySearchPackage;

import javax.swing.table.TableCellEditor;

public class binarySearch extends javax.swing.JFrame {

    String[] word=new String[12];

    public binarySearch() {
        initComponents();

        word[0]="zero";
        word[1]="one";
        word[2]="two";
        word[3]="three";
        word[4]="four";
        word[5]="five";
        word[6]="six";
        word[7]="seven";
        word[8]="eight";
        word[9]="nine";
        word[10]="ten";
        word[11]="eleven";

    }
```

Add a loop to number the rows of the table and copy the array values into the Data column.

```
        word[10]="ten";
        word[11]="eleven";

        for (int i=0; i<12; i++)
        {
            tblSearch.getModel().setValueAt(i,i,0);
            tblSearch.getModel().setValueAt(word[i],i,1);
        }

    }
```

Run the program and check that the row numbers and data items are displayed correctly.



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

The ***binary search*** will only work for sorted data, so we will use a ***bubble sort*** method to arrange the words in alphabetical order. Click the Design tab to move to the form layout view. Double click the "***Sort***" button to create the method, then add lines of code for the bubble sort.

We will begin by closing the table editor, to ensure that all data values are available for processing. We will then collect the data items from the table and transfer them back into the ***word[ ]*** array.

```
    private void btnSortActionPerformed(java.awt.event.ActionEvent evt) {
    {
        TableCellEditor editor = tblSearch.getCellEditor();
        if (editor != null)
        {
             editor.stopCellEditing();
        }
        for (int i=0; i<12; i++)
        {
            word[i]=(String) tblSearch.getModel().getValueAt(i,1);

        }
    }
```

The next step is to add the bubble sort loop.

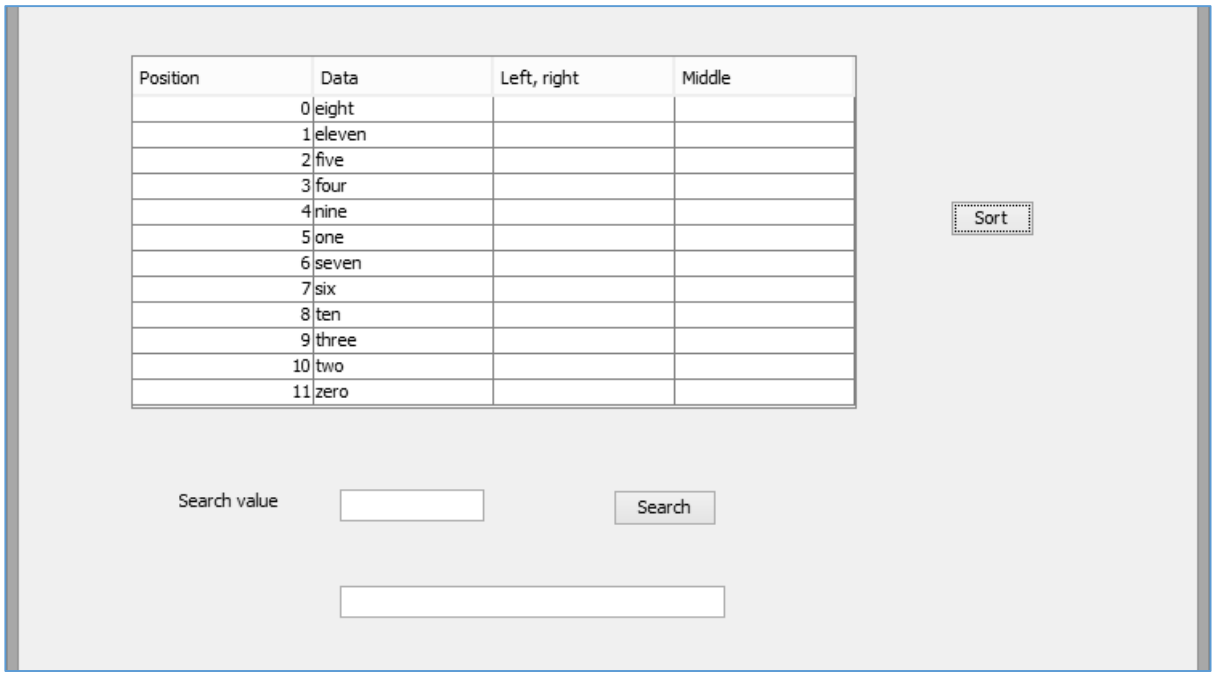
```
        for (int i=0; i<12; i++)
        {
            word[i]=(String) tblSearch.getModel().getValueAt(i,1);

        }
        Boolean swap=true;
        String tempWord;
        while (swap==true)
        {
            swap=false;
            for (int i=0; i<11;i++)
            {
                if (word[i].compareTo(word[i+1])>0)
                {
                    swap=true;
                    tempWord=word[i];
                    word[i]=word[i+1];
                    word[i+1]=tempWord;
                }
            }
        }
    }
```

The final step is to redisplay the sorted data in the table.

```
                if (word[i].compareTo(word[i+1])>0)
                {
                    swap=true;
                    tempWord=word[i];
                    word[i]=word[i+1];
                    word[i+1]=tempWord;
                }
            }
        }
        for(int i=0;i<12;i++)
        {
            tblSearch.getModel().setValueAt(word[i],i,1);
        }
    }
```

Run the program and check that the list of words in the table can be sorted into alphabetical order correctly, as shown below.

It should be possible to edit the data in the table, then re-sort the new data set.



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 set up the ***binary search*** procedure. Double click the "***Search***" button to create an empty method. We will add lines of code to carry out a series of tasks:

- Define an array called ***limit[ ]*** which will display information about the positions of the left and right pointers at different stages during the binary search. This will help us to understand how the program has carried out the search procedure.
- Use a loop to collect the data values from the table and transfer them to the ***word[ ]*** array, ready for searching.
- Within the loop we will also initialise the entries in the ***limit[ ]*** array to be blank, and clear the columns of the table which will display the pointer information.
- Finally, we will collect the ***search value*** entered in the text field by the user.

```
private void btnSearchActionPerformed(java.awt.event.ActionEvent evt) {
    String[] limit=new String[12];
    for (int i=0; i<12;i++)
    {
        word[i]=(String) tblSearch.getModel().getValueAt(i,1);
        limit[i]="";
        tblSearch.getModel().setValueAt("",i,2);
        tblSearch.getModel().setValueAt("",i,3);
    }
    String searchValue=txtSearchValue.getText();
}
```

We will first implement the binary search algorithm as shown in the flowchart above, but without displaying any pointer information. This will allow us to test that the search procedure works correctly.

The program begins by initialising the left and right pointers to the lower and upper limits of the array. We then enter a loop which will continue until either the required data item is found, or we know that it is not present.

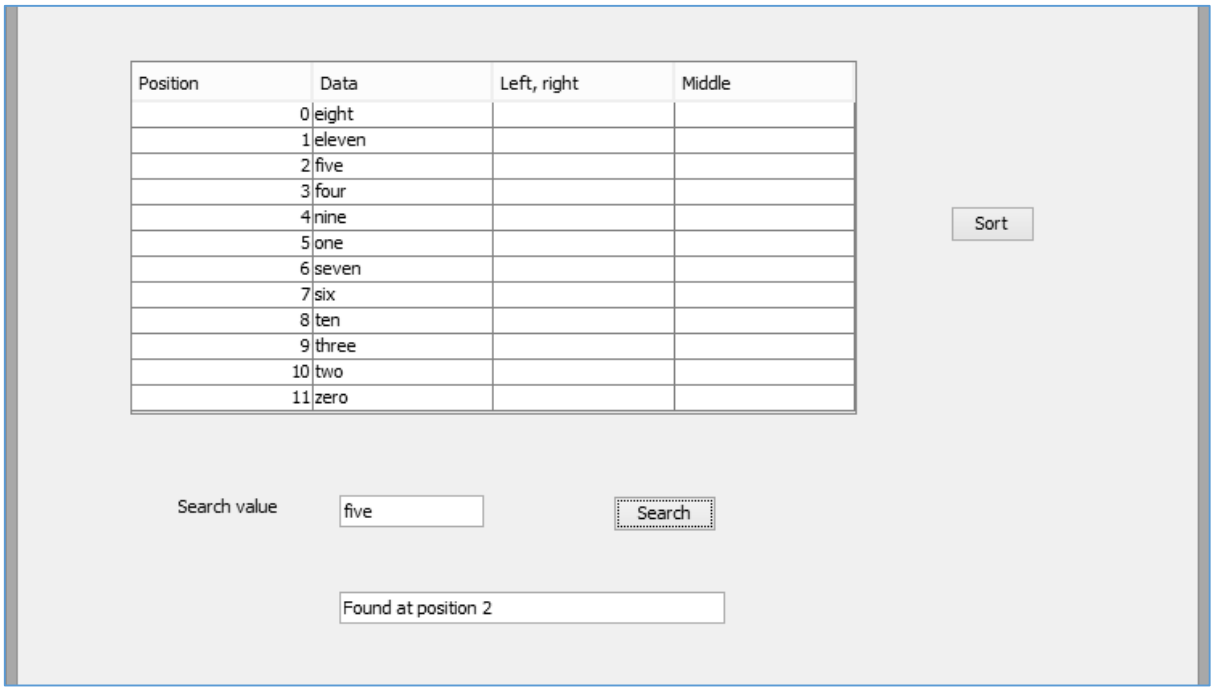
Each time around the loop, the program calculates the middle position for the current group of unsearched data items. The array element at the middle position is checked, and the left or right pointer moved if necessary to reduce the size of the unsearched data group.

When the loop ends, a message is displayed to show the result of the search.

```
    for (int i=0; i<12;i++)
    {
        word[i]=(String) tblSearch.getModel().getValueAt(i,1);
        tblSearch.getModel().setValueAt("",i,2);
        tblSearch.getModel().setValueAt("",i,3);
        limit[i]="";

    }
    String searchValue=txtSearchValue.getText();
    int left=0;
    int right=11;
    int middle=0;
    Boolean found=false;
    while (found==false && left<=right)
    {
        middle= Math.round( ((left+right)/2)- 0.5f);
        if(word[middle].equals(searchValue))
        {
            found=true;
        }
        else
        {
            if (word[middle].compareTo(searchValue)>0)
            {
                right=middle-1;
            }
            else
            {
                left=middle+1;
            }
        }
    }
    if (found==true)
    {
        txtResult.setText("Found at position "+String.valueOf(middle));
    }
    else
    {
        txtResult.setText("Search value not found");
    }
}
```

Run the program. Be careful to sort the data before carrying out a binary search.

Enter search values and check that the correct search results are displayed, when either the search value is present or not present in the table.

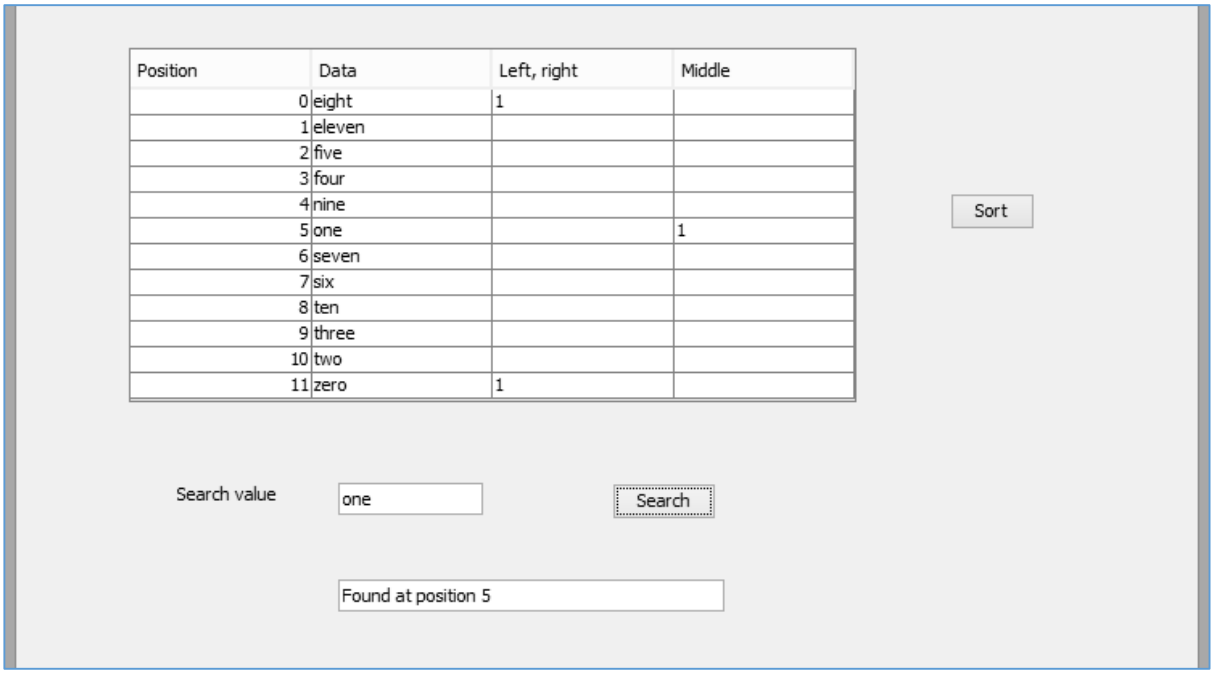


Close the program and return to the NetBeans editing screen.

We will now add lines of code to the program which will show the positions of the ***left***, ***right*** and ***middle pointers*** as each step of the search is carried out. The additional lines of code are indicated in the program listing on the next page.

When you have added the extra code, run the program and examine the output in the table.

The program begins with ***search 1*** of the data. At this stage, the ***left pointer*** is at ***position 0***, the ***right pointer*** is at ***position 11***, and ***middle*** will be calculated as ***position 5***. If the required data item happens to be at this position then the search can end:



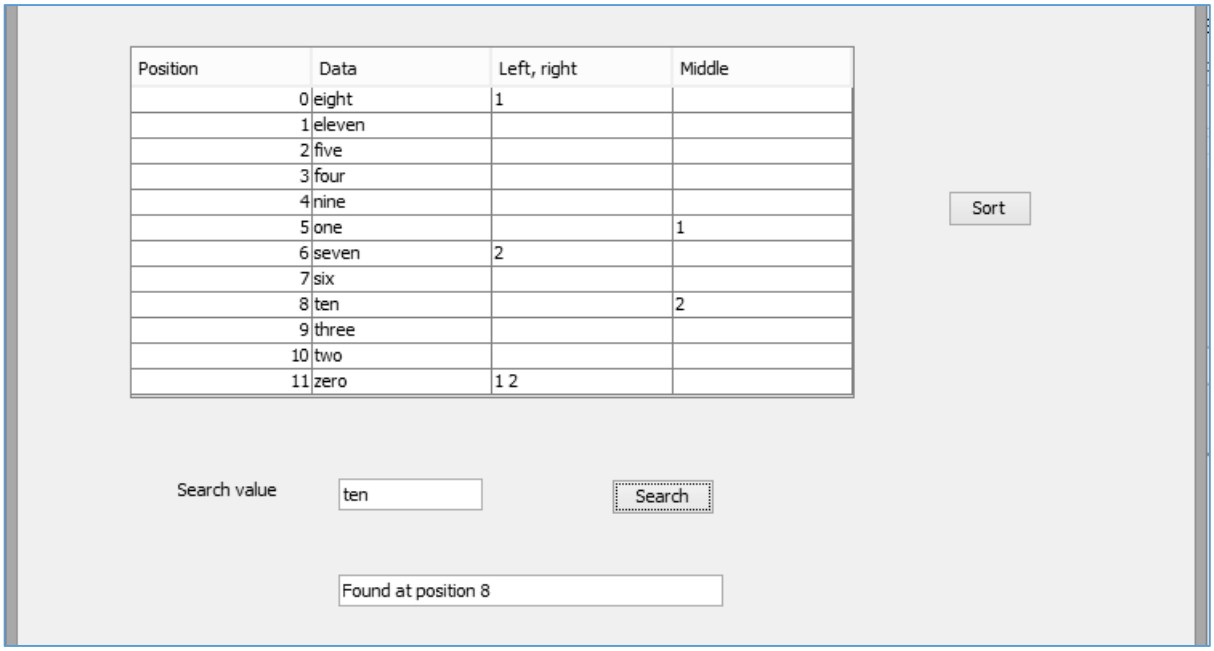
```
private void btnSearchActionPerformed(java.awt.event.ActionEvent evt) {
    String[] limit=new String[12];
    for (int i=0; i<12;i++)
    {
        word[i]=(String) tblSearch.getModel().getValueAt(i,1);
        limit[i]="";
        tblSearch.getModel().setValueAt("",i,2);
        tblSearch.getModel().setValueAt("",i,3);
    }
    String searchValue=txtSearchValue.getText();
    int left=0;
    int right=11;
    int middle=0;
    int count=1;
    Boolean found=false;
    while (found==false && left<=right)
    {
       middle= Math.round( ((left+right)/2)- 0.5f);
       limit[left]=limit[left]+String.valueOf(count)+" ";
       limit[right]=limit[right]+String.valueOf(count)+" ";
       for (int i=0; i<12; i++)
       {
           tblSearch.getModel().setValueAt(limit[i],i,2);
       }
       tblSearch.getModel().setValueAt(count,middle,3);
       if(word[middle].equals(searchValue))
       {
           found=true;
       }
       else
       {
           if (word[middle].compareTo(searchValue)>0)
           {
               right=middle-1;
           }
           else
           {
               left=middle+1;
           }
       }
       count++;
    }
    if (found==true)
    {
        txtResult.setText("Found at position "+String.valueOf(middle));
    }
    else
    {
        txtResult.setText("Search value not found");
    }
}
```

If the search value is not yet found, the ***left*** or ***right*** pointer will be moved to reduce the size of the remaining unsearched group of data items, and a new ***middle*** position will be calculated.

In this example we are searching for the word "***ten***".

After ***search 1***, the required item is not found. The ***right pointer*** remains at ***position 11*** but the ***left pointer*** is now moved to ***position 6***, ready for ***search 2***.

A new ***middle pointer*** value is calculated as ***position 8***, and the required data is found at this location.



In some cases, more than two searches may be needed to find the required data value, or to prove that it is not present. In this example, four searches were needed to find the word "***six***".