13 Recursion

In computer programs, it is frequently necessary for sections of program code to be repeated. We have seen how this can be done using ***loop*** structures:

- If a section needs to be repeated a ***definite number of time***s, for example: to input a temperature for each day of the week, then a ***FOR … TO … DO*** loop can be used:

```
for (int day = 1; day <= 7; day++)
 {
      //lines of code to be repeated
 }
```

- If a section needs to be repeated ***until some condition is met***, for example: to enter purchases until the user has completed their shopping, then a ***WHILE … DO*** loop is used:

```
while (finishedShopping == false)
{
     //lines of code to be repeated
}
```

In this chapter we will examine a different approach to repeating sections of a program, using a technique called ***recursion***.

Recursion implies that another version of something is happening ***within itself***. Imagine that you are looking at a computer. On the screen is a picture of the same computer, which has a picture of the same computer on its screen, and so on … all the way down!

If a computer problem involves another version of something happening ***within itself***, then ***recursion*** can be used to find a solution.

To illustrate how recursion works in a program, consider a simple example:

> In mathematics, a ***factorial*** is a total produced by multiplying together all the integers from 1 up to some specified value. For example:
>
> factorial 4 = 1 * 2 * 3 * 4
> = 24
>
> A computer program is required which will calculate factorials.

Let us consider the problem of finding factorials in a bit more detail:

factorial 4 = 1 * 2 * 3 * 4
factorial 3 = 1 * 2 * 3
factorial 2 = 1 * 2
factorial 1 = 1

We can therefore consider the problem of finding ***factorial 4*** as the sequence:

factorial 4 = (***factorial 3***) * 4
↳ ***factorial 3*** = (***factorial 2***) * 3
↳ ***factorial 2*** = (***factorial 1***) * 2
↳ 1

We can spot a pattern here. Suppose that we need to find the factorial of a number N:

- If ***N = 1***, then the answer is ***1***
- If ***N is bigger than 1***, then the answer for factorial N can be found by finding the factorial of the number one less than N, then multiplying:

  factorial N = factorial (N - 1) * N

This could be written in Java as a method with the general structure:

```
getFactorial ( N )
{
        if ( N == 1 )
        {
              result = 1;
        }
        else
        {
              result = N * getFactorial( N - 1 );
        }
}
```

If the required factorial N is 1, then we have an immediate result of 1, and the method can end.

If the required factorial N is a number greater than 1, then the method will run ***getFactorial( )*** again inside itself to find the ***factorial of N-1***. When this answer comes back, it can be multiplied by N to give the final result for the problem.

We can now produce the program to calculate factorials.

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

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

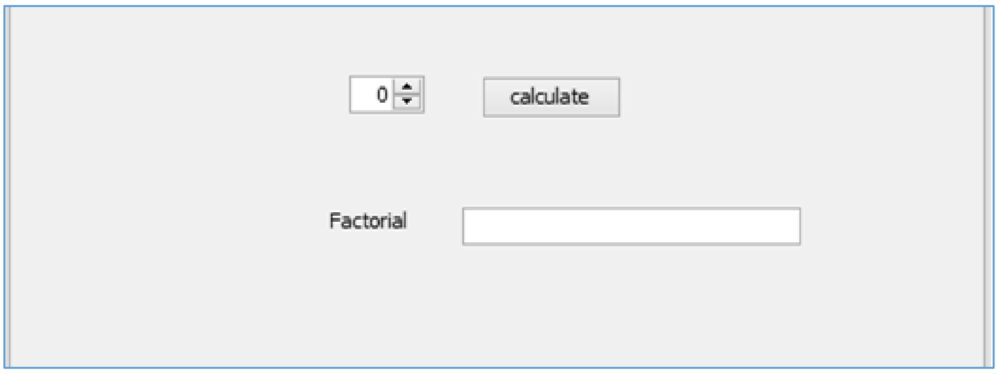
Return to the NetBeans editing screen.

- 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:

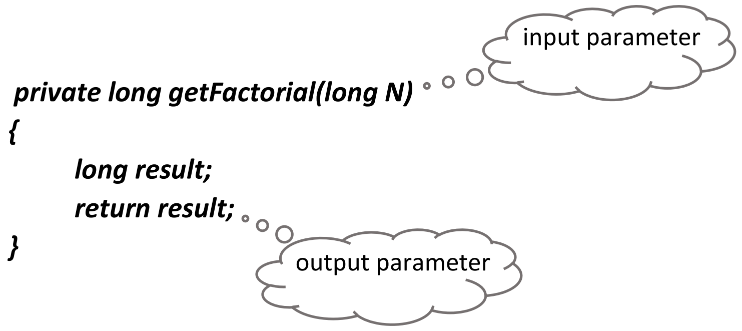
- A ***spinner***. Give this the name ***spinNumber***.
- A ***button*** with the caption'***calculate***'. Rename the button as ***btnCalculate***.
- A ***label*** '***Factorial***', with a ***text field*** alongside. Rename the text field as ***txtOutput***.



Use the ***Source*** tab to move to the program code page. Set up the overall structure for a ***getFactorial( )*** method below the ***factorial( )*** method

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

    private long getFactorial(long N)
    {
        long result;

        return result;
    }
```

The method has some important features. It uses an ***input parameter N***, which specifies the number whose factorial is to be found. It also returns an ***output parameter*** which is the result of the calculation.



```
private long getFactorial(long N)
{
        long result;
        return result;
}
```

When we call the ***getFactorial( )*** method, we will give it ***N*** and it will give us back ***result***. Factorials can be very large numbers, so we will use the ***long*** number format, rather than smaller ***int*** variables.

Add lines of code to the method. Notice the way in which ***getFactorial( )*** calls another version of ***getFactorial( )*** inside itself, but with a different input parameter ***N-1*** instead of ***N***.

```
private long getFactorial(long N)
{
    long result;

    if (N==1)
    {
        result=1;
    }
    else
    {
        result=N*getFactorial(N-1);
    }

    return result;
}
```

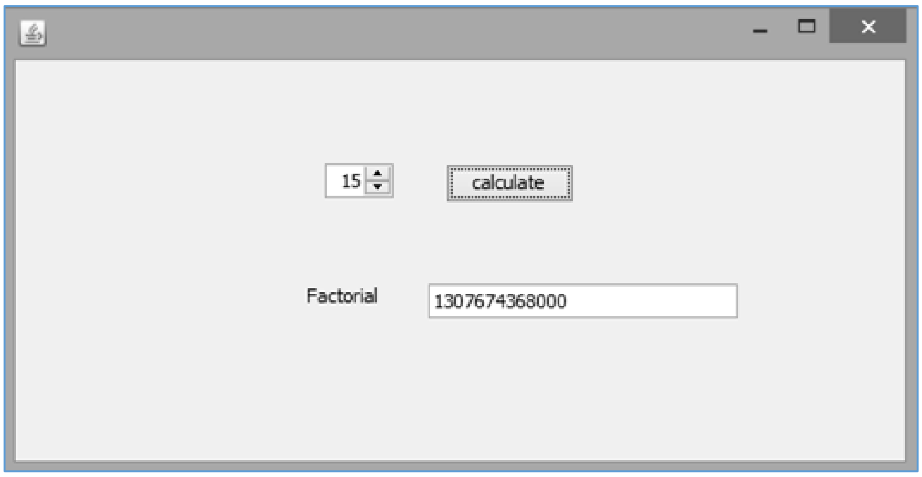
Use the Design tab to move back to the form layout view. Double click the '***calculate***' button to create a method. We will add lines of code which will:

- Collect the required factorial number ***N*** from the spin component.
- Call the ***getFactorial( )*** method, which in turn will call ***getFactorial( )*** recursively as many times as are necessary. The method returns ***result*** as a number in ***long*** format.
- ***Result*** is converted to text and displayed in the text field.

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

    int N=(int) spinNumber.getValue();
    long result=getFactorial(N);
    String answer;
    answer=String.valueOf(result);
    txtOutput.setText(answer);

}
```

Run the program. Choose a small input number and click the '***calculate***' button. Check that the correct ***factorial*** is displayed.

Try larger numbers. You will find that factorials increase in size very rapidly due to the multiplication sequence.



```
1307674368000
```

Although this was a simple program, we have discovered all the main characteristics of recursive methods:

For a method to be recursive, it must ***call itself*** from inside the method.

A recursive method must have one or more ***input parameters***, which can change each time the method is called from inside itself.

If the recursive method is carrying out a calculation, then it should ***return a result***. This will be passed back to the previous call of the method, and eventually back to the main program.

A recursive method must have a ***non-recursive stopping condition***. In the case of the factorial program, this was the condition that:

if N = 1, then result = 1

Without a non-recursive stopping condition, the method will continue to call itself until all the available memory space in the computer has been used up, and the program crashes with a system error.

A feature of recursive algorithms is that complex tasks can be carried out with very small amounts of program code. This can be particularly important, for example, if programs have to be stored on a small chip inside an electronic device. Many of the functions on an electronic calculator make use of recursive algorithms in the program code.

For our next project, we will look at another simple problem where recursion can be used.

A program is required which can convert positive integer (base-10) numbers into unsigned binary numbers up to 32 bits in length. For example:

349856728 = 0001 0100 1101 1010 0110 0011 1101 1000

Unsigned binary numbers are made up from powers of two, for example:

1	0	1	1	0	1	0	1
128	64	32	16	8	4	2	1

Adding the values represented by the binary-1 digits gives:

128 + 32 + 16 + 4 + 1 = 181

We can analyse the programming task by choosing a simple number to convert to binary: ***26***.

We will divide the number by 2 as many times as possible until zero is reached, making a note of the remainder for each division.

```
26  =  2 * 13  with remainder 0
13  =  2 *  6  with remainder 1
 6  =  2 *  3  with remainder 0
 3  =  2 *  1  with remainder 1
 1  =  2 *  0  with remainder 1
```

Dividing the original number 26 by two gives a remainder of zero. This means that 26 is an even number. A value of 1 should not be include in the binary total:

?	?	?	?	0
16	8	4	2	1

We now want to know whether there are an even or odd number of twos in the original number. We halved the number and obtained a result of 13, which tells us that there were 13 twos included in the original number 26. This is an odd number, so a value of 2 will need to be included in the binary total.

?	?	?	1	0
16	8	4	2	1

Continuing in the same way, we now want to know whether there are an even or odd number of fours in the original number. When we halved the number a second time, we obtained a result of 6, which tells us that there were 6 fours included in the number 26. This is an even number, so a value of 4 will not be needed in the binary total.

?	?	0	1	0
16	8	4	2	1

We now move on to consider the number of eights in the original number. Halving for a third time gives 3, which tells us that there were 3 eights included in the number 26. This is an odd number, so a value of 8 will need to be included in the binary total.

?	1	0	1	0
16	8	4	2	1

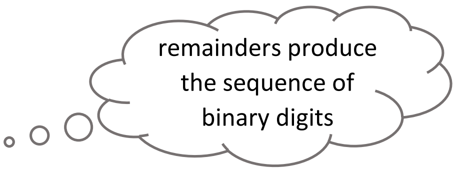
Finally we consider the number of sixteens in the original number. Halving for a fourth time gives 1, which tells us that there is 1 sixteen included in the number 26. This is an odd number, so a value of 16 will will need to be included in the binary total.

1	1	0	1	0
16	8	4	2	1

Checking the result: 16 + 8 + 2 = 26 as required.

Consider again the sequence of divisions we carried out:

$$\begin{tikzcd}
26 = 2 * 13 \text{ remainder } 0 \arrow[dr] & & & & \\
& 13 = 2 * 6 \text{ remainder } 1 \arrow[dr] & & & \\
& & 6 = 2 * 3 \text{ remainder } 0 \arrow[dr] & & \\
& & & 3 = 2 * 1 \text{ remainder } 1 \arrow[dr] & \\
& & & & 1 = 2 * 0 \text{ remainder } 1
\end{tikzcd}$$



The requirement for a ***recursive algorithm*** has been met, as the problem involves further instances of a process occurring ***within the process itself***. At each level, we are dividing a number by two, finding the remainder, and passing on the divided number to the next call of the process.

Two programming functions which will be useful in developing the program are:

- DIV, which is the number of times one number divides into another, ignoring any remainder.
  For example: 43 DIV 5 = 8
- MOD, which is the remainder when one number is divided by another.
  For example: 43 MOD 5 = 3

If the original number to be converted to binary is N, then a design for the method is:

```
convertToBinary ( N)
{
    if ( N > 1 )
    {
        binaryNumber = convertToBinary( N DIV 2 )
    }
    add ( N MOD 2 ) as the next bit of binaryNumber.
    return binaryNumber
}
```

We can summarise the sequence of recursive calls which would be made for N = 26. A sequence of calls of the ***convertToBinary( )*** method are opened by recursion until the ***input parameter N*** is reduced to 1. Each binary digit is then added to ***binaryNumber*** as the calls close in the reverse order.

call	remainder N MOD 2	binaryNumber
convertToBinary(26)	0	11010
convertToBinary(13)	1	1101
convertToBinary(6)	0	110
convertToBinary(3)	1	11
convertToBinary(1)	1	1

We are now ready to produce the program.

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

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

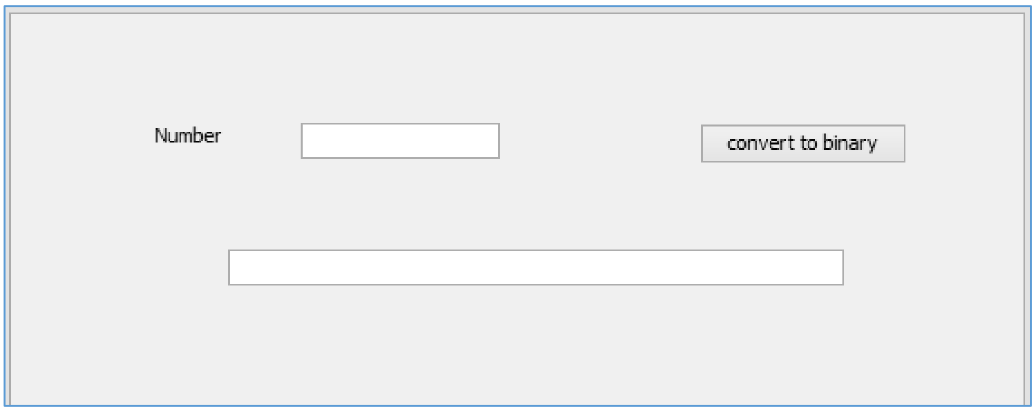
Return to the NetBeans editing screen.

- 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:

- A ***label*** '***Number***'.
- A ***text field*** with the name ***txtNumber***.
- A ***button*** alongside with the caption '***convert to binary***'. Rename the button as ***btnBinary***.
- A ***text field*** at the bottom of the form with the name ***txtBinary***.



Use the Source tab to move to the program code screen. We will add the recursive method ***getBinary( )*** below ***binary( )***. The method has two input parameters:

- ***N***, which is the base-10 number which is to be converted to binary. N is divided by 2 during each recursion, then passed on the next call of the ***getBinary( )*** method.
- ***Output***, which is the binary string that is built up as each recursive call closes. The value of ***output*** is passed back through each previous call of ***getBinary( )***, with each method adding another binary digit until the output string is completed.

The symbol in Java for the ***MOD*** function is '**%**', and the ***DIV*** function is '**/**'.

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

private String getBinary(long N, String output)
{
    long result;
    int digit= (int) (N%2);
    if (N>1)
    {
       long next= (long) (N/2);
       output=getBinary(next,output);
    }
    output+=digit;
    return output;
}
```

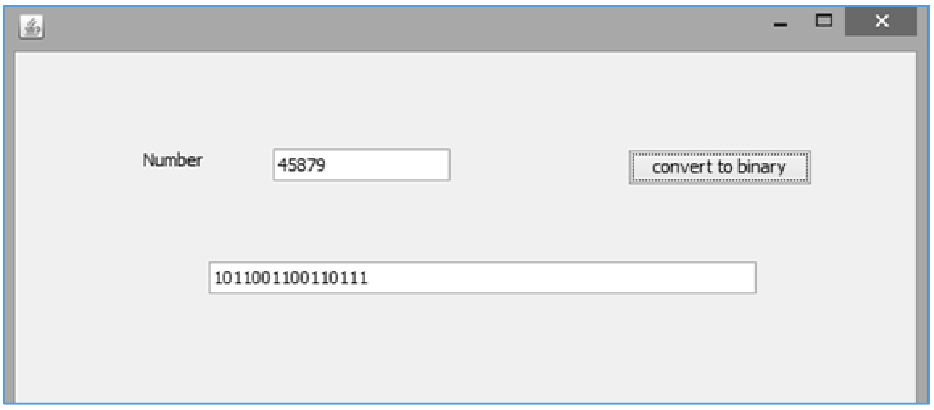
Use the ***Design*** tab to return to the form layout view, then double click the ***'convert to binary***' button to create a method. Add lines of code which will:

- collect the required number ***N*** from the text field for conversion to binary,
- call the ***getBinary*** method using ***N*** as an input parameter, then
- display the ***output*** string in the ***txtBinary*** text field.

```
private void btnBinaryActionPerformed(java.awt.event.ActionEvent evt) {
    long N=Integer.parseInt(txtNumber.getText());
    String output = getBinary( N, "");
    txtBinary.setText(output);
}
```

Run the program. Enter some small numbers, such as 26, and check that the binary patterns are correct.

Enter some larger numbers.



```
45879
1011001100110111
```

The binary patterns produced are correct, but are not well formatted. Binary numbers are usually displayed with groups of four digits separated by a space, for example:

0101 1101 0111 1010

This makes it easier to read the number, and reduces the chance of a programmer making an error when copying the binary values.

Close the program window and return to the program code page. We will add a method to improve the formatting of the binary output. Call this from the button click method.

```
    private void btnBinaryActionPerformed(java.awt.event.ActionEvent evt) {
        long N=Integer.parseInt(txtNumber.getText());
        String output = getBinary( N, "");

        output = formatBinaryNumber(output);

        txtBinary.setText(output);
    }

    private String formatBinaryNumber(String input)
    {
        int L=input.length();
        int zerosNeeded=4-(L % 4);
        if (zerosNeeded<4)
        {
            for (int i=1; i<=zerosNeeded;i++)
            {
               input = "0"+ input;
            }
        }
        String output="";
        for (int i=0; i<input.length(); i++ )
        {
            output = output + input.substring(i,i+1);
            if ((i%4)==3)
            {
                output += " ";
            }
        }
        return output;
     }
```

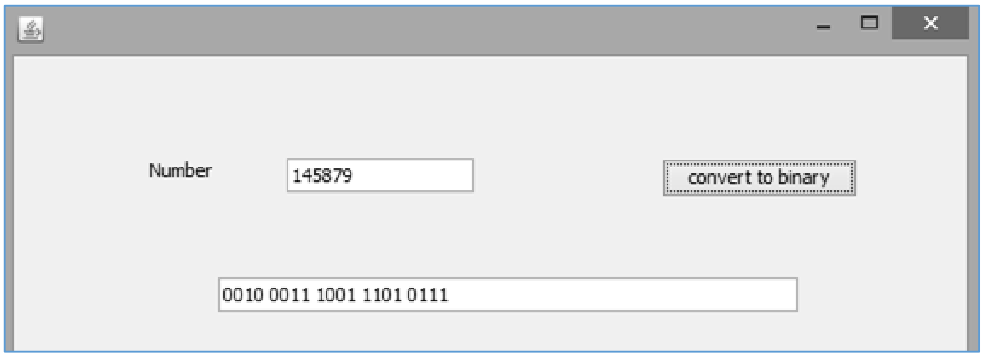
The ***formatBinaryNumber( )*** method begins by counting the number of binary digits in the number, then adds any extra zeros necessary at the beginning to complete a group of four. For example:

1010000 becomes ***01010000***

The program then works through the binary string, copying each digit in turn to produce the final output. A space character is inserted after each fourth digit:

01010000 now becomes ***0101 0000***

Run the program. Enter different numbers and check that the binary output is now in the correct format.

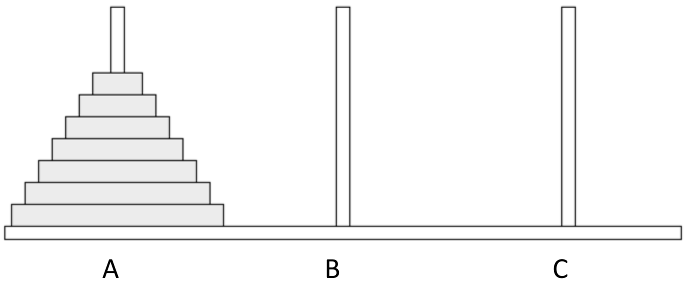

Number 145879

convert to binary

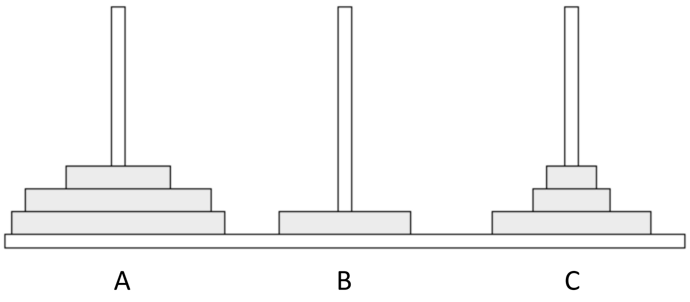
0010 0011 1001 1101 0111

For the next project, we will examine an interesting recursive problem where graphics can be used.

The ***Towers of Hanoi*** game involves a series of discs of reducing diameters, stacked on the vertical rod ***A***. The objective of the game is to move the discs to rod ***C***.

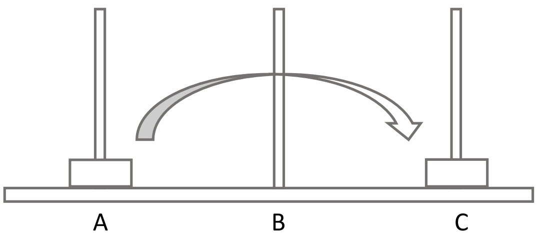


Rod ***B*** can be used as a temporary storage area, but at no time must a larger disc be placed on top of a smaller disc.

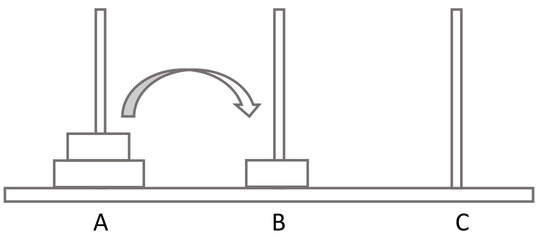


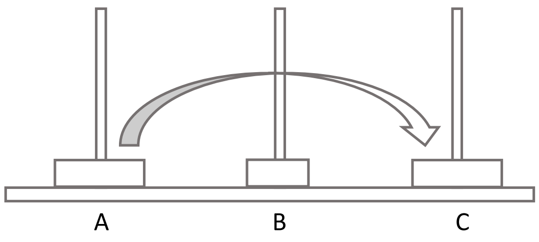
Develop a program to demonstrate solutions to the game for up to seven discs.

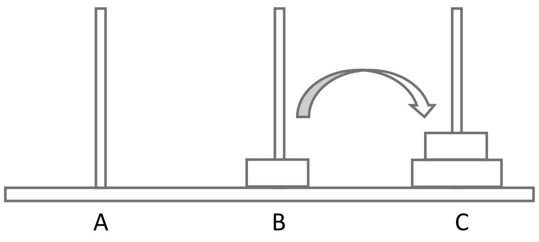
We will begin by analysing the problem for simple cases. If only one disc is used, the disc is simply moved from rod A to rod C.



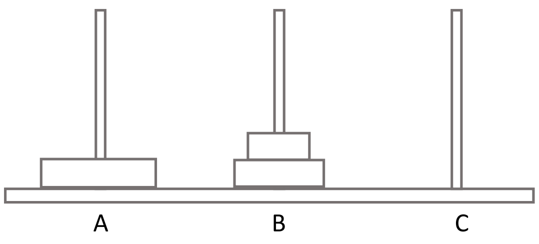
For two discs, rod B is used as a temporary store for the smaller disc. The larger disc can then be transferred to rod C, and the small disc moved on top of it.

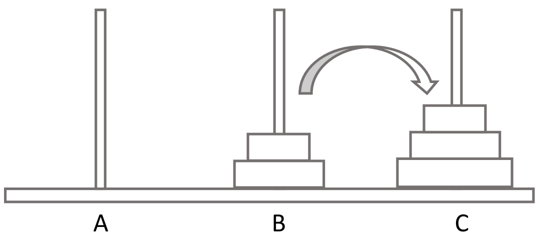




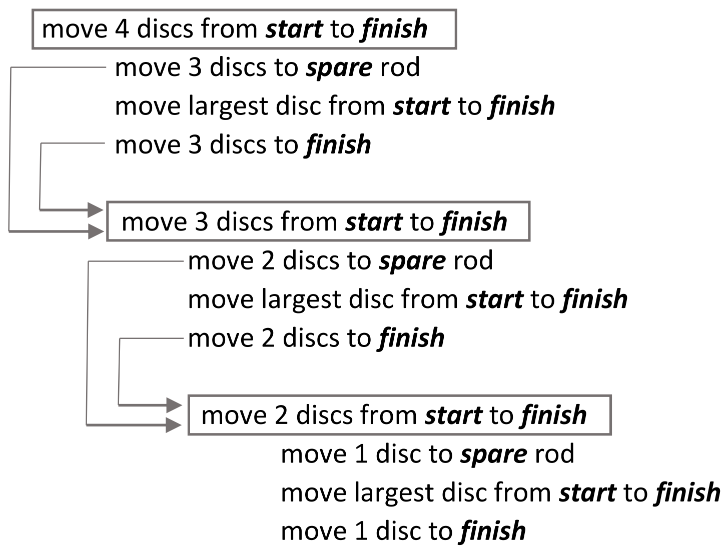


For three discs, the strategy is to move two discs to rod B, transfer the largest disc to C, then move the two discs from B to C to complete the tower.





Considering the case of four discs:


move 4 discs from ***start*** to ***finish***
move 3 discs to ***spare*** rod
move largest disc from ***start*** to ***finish***
move 3 discs to ***finish***
move 3 discs from ***start*** to ***finish***
move 2 discs to ***spare*** rod
move largest disc from ***start*** to ***finish***
move 2 discs to ***finish***
move 2 discs from ***start*** to ***finish***
move 1 disc to ***spare*** rod
move largest disc from ***start*** to ***finish***
move 1 disc to ***finish***

Larger numbers of discs can be transferred using a similar strategy. We can see that versions of the method to move a pyramid of discs from one rod to another are occurring ***within themselves***, so the problem can be solved by recursion.

We can now design a ***moveRings( )*** recursive method. For each call of the method, we will need to provide input parameters to indicate:

- the number of discs to be transferred
- the start position of the discs
- the finish position of the discs
- the position of the rod which can be used as a temporary store.

For example, the call:

moveRings(3, 'A', 'B', 'C')

would move three rings from rod A to rod B, using rod C as a temporary store. A possible design for the method is:

```
moveRings( N, start, finish, spare )
{
        if ( N > 1 )
        {
            moveRings( N-1, start, spare)
        }
        move largest ring from start to finish
        if ( N > 1 )
        {
            moveRings( N-1, spare, finish)
        }
}
```

We have stopping condition when N reaches 1, as no recursive calls will be made in that case.

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

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

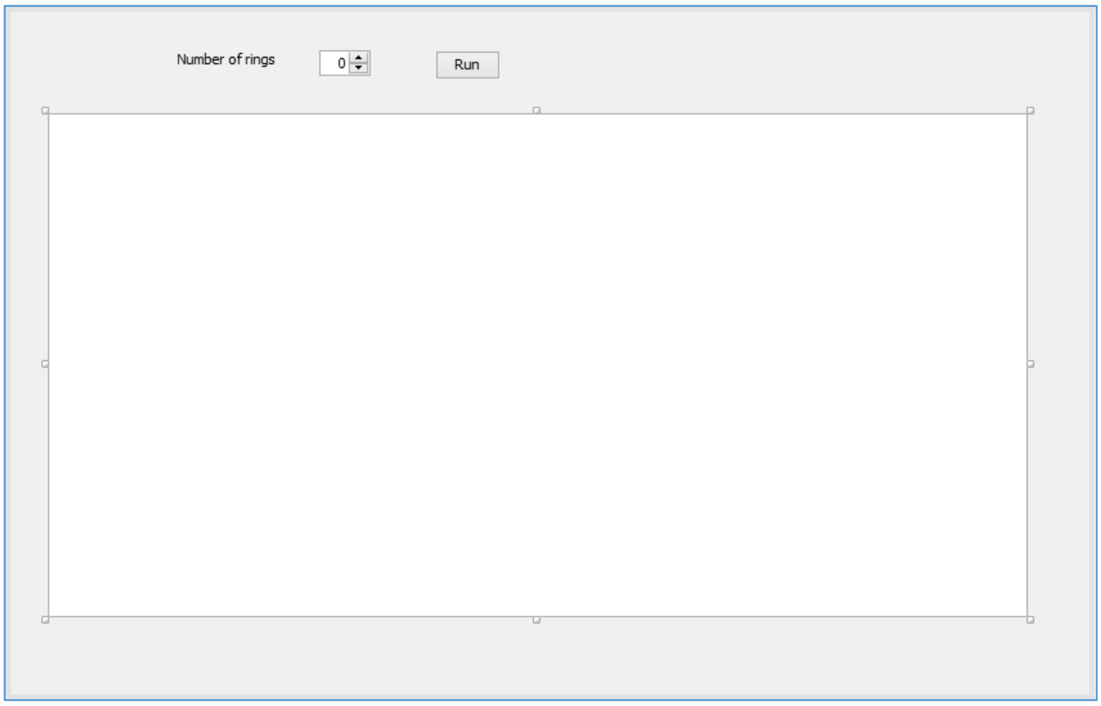
Return to the NetBeans editing screen.

- 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.

Use the Design tab to move to the form layout view. Add components to the form:

- A label '***Number of rings***'. Place a Spinner alongside and rename this as ***spinDiscs***.
- A button with the caption '***Run***' and the name ***btnRun***.
- A panel. Rename this as ***pnlGame***.



Select the panel and go to the Properties window.

- Set the ***background*** property by selecting ***White*** from the colour options.
- Set the size of the panel by setting both the ***maximumSize*** and ***minimumSize*** properties to:

[700, 400]

Use the ***Source*** tab to move to the program code screen. Add Java modules which will be needed to produce graphics for the project.

We can use three ***stack*** data structures to record the positions of the rings on each of the rods A, B and C as the game is being played. Add integer ***arrays*** and integer ***pointers*** for the stacks.

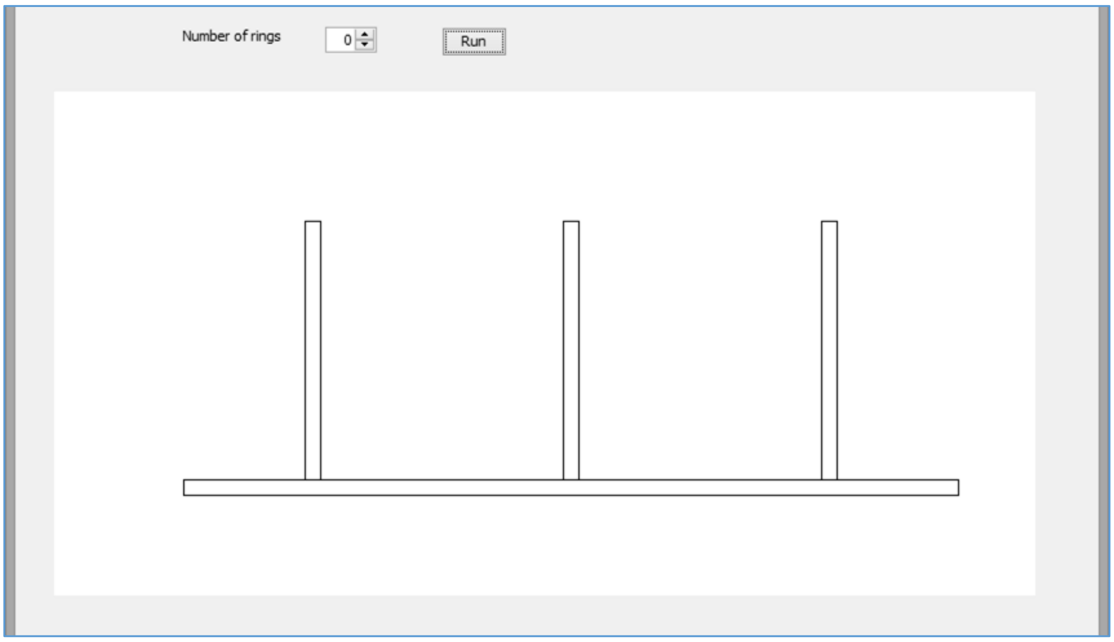
```
package hanoiPackage;

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

public class hanoi extends javax.swing.JFrame {

    int[] stack0 = new int[9];
    int[] stack1 = new int[9];
    int[] stack2 = new int[9];
    int pointer0;
    int pointer1;
    int pointer2;

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

Use the ***Design*** tab to move back to the form layout view. Double click the '***Run***' button to create a method. We will begin by drawing the three rods which will hold the rings.

Add a ***setup( )*** method and call this from the button click method.

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

    setup();

}

private void setup()
{
    Graphics2D g = (Graphics2D) pnlGame.getGraphics();
    g.setColor(Color.white);
    g.fillRect(0, 0, 900, 400);
    g.setColor(Color.black);
    g.drawRect(100, 300, 600, 12);
    g.drawRect(194, 100, 12, 200);
    g.drawRect(394, 100, 12, 200);
    g.drawRect(594, 100, 12, 200);
}
```

Run the program. Click the '***Run***' button and check that the graphics are displayed correctly.



Close the program and return to the program code page.

We will now create a method to draw discs. This will require three parameters:

- A letter, A, B or C, identifying the rod which holds the disc.
- The position of the disc above the base of the rod. For example, in a pyramid of three discs, the lowest disc will be at position 0, the middle disc at position 1, and the top disc at position 2.
- The diameter of the disc. The smallest disc has size 1, the next disc is size 2, with the size increasing by one for each larger size of disc.

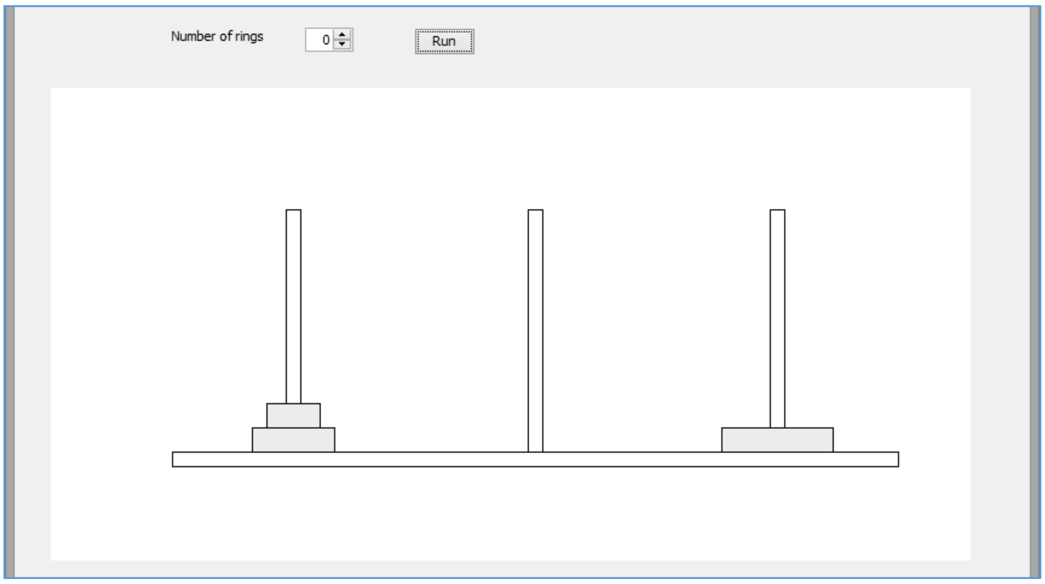
```
private void btnRunActionPerformed(java.awt.event.ActionEvent evt) {
   setup();
}

private void drawDisc(char column,int height,int discSize)
{
    Graphics2D g = (Graphics2D) pnlGame.getGraphics();
    int c=0;
    int x, y, w;
    switch (column)
    {
        case 'A': c=0; break;
        case 'B': c=1; break;
        case 'C': c=2; break;
    }
    x = 200 + 200 * c;
    y = 280 - 20 * height;
    w = 10 + 12 * discSize;
    g.setColor(Color.yellow);
    g.fillRect(x-w, y, w*2, 20);
    g.setColor(Color.black);
    g.drawRect(x - w, y, w * 2, 20);
}
```

The ***column*** letter is converted to a number, which is then used to calculate the position ***x*** of the column across the screen. The ***height*** parameter is used to calculate the vertical screen coordinate ***y*** for a rectangle representing the disc, and the ***discSize*** parameter is used to calculate the width ***w*** of the rectangle.

Test the ***drawDisc( )*** method by adding some calls to the ***Run*** button click method.

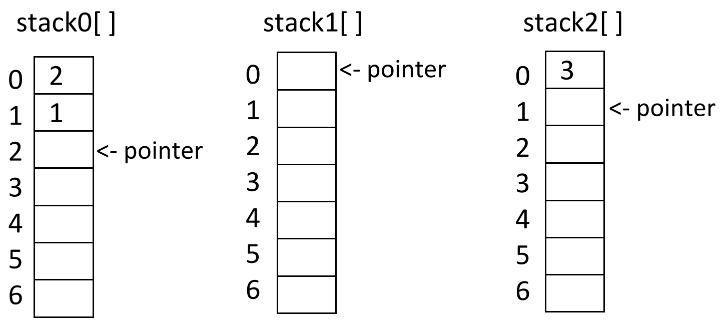
```
private void btnRunActionPerformed(java.awt.event.ActionEvent evt) {
    setup();

    drawDisc('A',0,2);
    drawDisc('A',1,1);
    drawDisc('C',0,3);

}
```

Run the program, then click the '***Run***' button. A stack of two discs should be shown on rod A at the left, and a larger disc on rod C at the right.

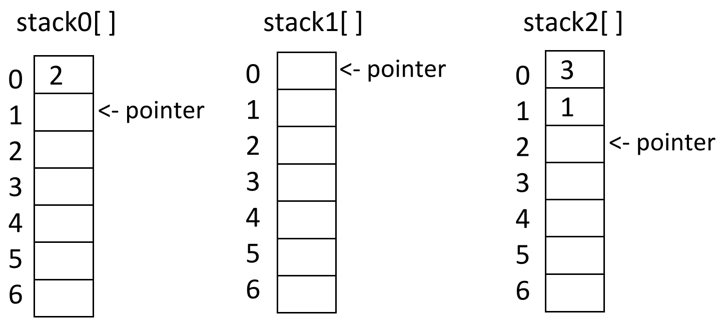


Close the program window and return to the program code screen. We now have the necessary graphics components available, and can begin the algorithm for playing the game.

We will represent the collections of discs on the rods using arrays. ***Stack0[ ]*** represents rod A, ***stack1[ ]*** represents rod B, and ***stack2[ ]*** represents rod C. The stack elements will contain the sizes of the rings at each level. For example, the situation shown above, with two rings on rod A and one ring on rod C would be represented as:



We can record a disc being moved from one stack to another by ***removing an entry*** from the stack at the start location, then ***adding the entry*** to the finish stack. For example, moving the size 1 disc from rod A to rod C would change the ***stack data*** and ***pointers*** to give:



The program can then check the stacks after each move in the game, to determine the positions and sizes of the discs to be drawn on the screen.

At the start of the game, the program should set up the required number of discs on rod A. Return to the '***Run***' button click method and remove the ***drawDisc( )*** test lines. Replace these with lines of code which will obtain the required number of discs ***N*** from the spin component, then initialise the ***stack*** arrays. A loop sets up the correct number of discs of decreasing size to form a pyramid on rod A. Pointers for the other two stacks are set to zero, to indicate that rods B and C are currently empty.

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

   int N = (int) spinDiscs.getValue();
   for (int j = 0; j < N; j++)
   {
        stack0[j] = N - j;
   }
   pointer0 = N;
   pointer1 = 0;
   pointer2 = 0;

}
```

We now require a method to read the arrays, and use the data to display the discs in their starting position on rod A. We will add a ***drawColumns( )*** method below the '***Run***' button click method.

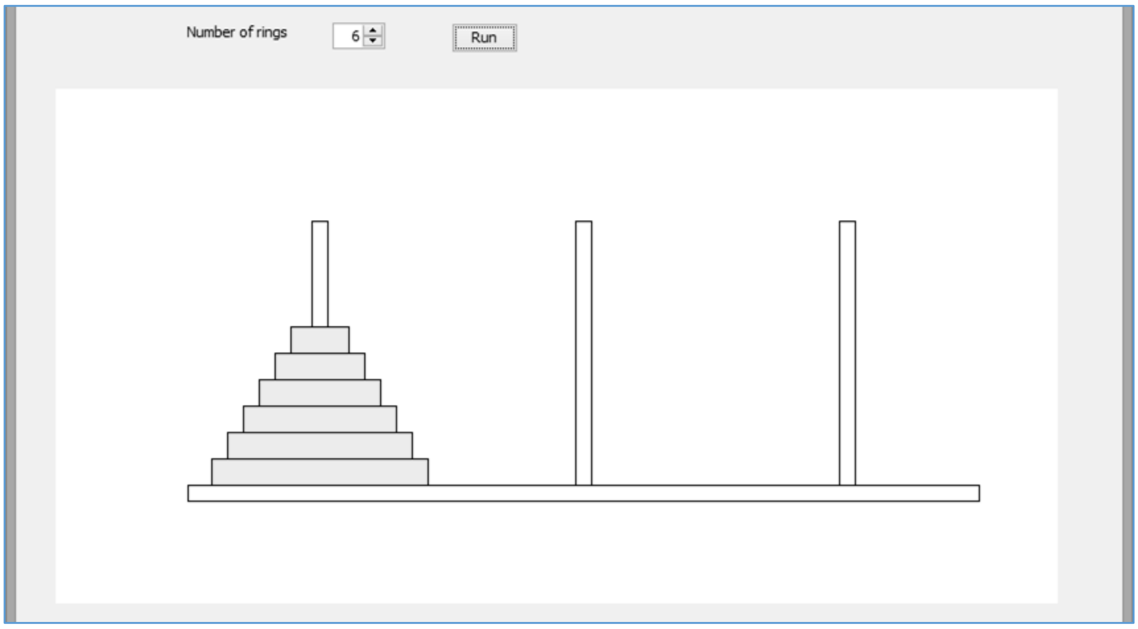
- This method uses three loops to check each of the ***stack*** arrays in turn.
- Each entry in the stack is collected. This will be an integer number indicating the size of disc at a particular height in the column corresponding to the array index. For example: ***array[0]*** would contain the size of disc at the bottom level in the column, with ***array[1]*** containing the size of disc lying on top of it at the next level up.
- The ***drawDisc( )*** method is called, with parameters indicating the ***identification letter*** for the rod, the ***height*** position up the column, and the ***size of disc*** at that position.

```
    private void drawColumns()
    {
        for (int j = 0; j < pointer0; j++)
        {
            drawDisc('A', j, stack0[j]);
        }
        for (int j = 0; j < pointer1; j++)
        {
            drawDisc('B', j, stack1[j]);
        }
        for (int j = 0; j < pointer2; j++)
        {
            drawDisc('C', j, stack2[j]);
        }
    }
```

Add a line of code to the '***Run***' button click method to call ***drawColumns( )***.

```
private void btnRunActionPerformed(java.awt.event.ActionEvent evt) {
     setup();
     int N = (int) spinDiscs.getValue();
     for (int j = 0; j < N; j++)
     {
         stack0[j] = N - j;
     }
     pointer0 = N;
     pointer1 = 0;
     pointer2 = 0;

     drawColumns();

}
```

Run the program. Select different numbers of rings up to a maximum of 9, and check that these are displayed on rod A when the '***Run***' button is clicked.



Close the program and return to the program code screen. We will now add a method to move the top ring from one rod to another. Set up the method ***moveStack( )*** below the '***Run***' button-click method.

This method takes the letters of the start and finish rods as input parameters. The first half of the method removes the top ring from the start stack, then replaces its size number on the destination stack. We finally call the ***drawColumns( )*** method to redisplay the rings in their new positions.

```
private void moveStack(char start, char finish)
{
    int ring=0;
    if (start == 'A')
    {
        pointer0--;
        ring = stack0[pointer0];
    }
    if (start == 'B')
    {
        pointer1--;
        ring = stack1[pointer1];
    }
    if (start == 'C')
    {
        pointer2--;
        ring = stack2[pointer2];
    }
    if (finish == 'A')
    {
        stack0[pointer0]=ring;
        pointer0++;
    }
    if (finish == 'B')
    {
        stack1[pointer1]=ring;
        pointer1++;
    }
    if (finish == 'C')
    {
        stack2[pointer2]=ring;
        pointer2++;
    }
    setup();
    drawColumns();
}
```

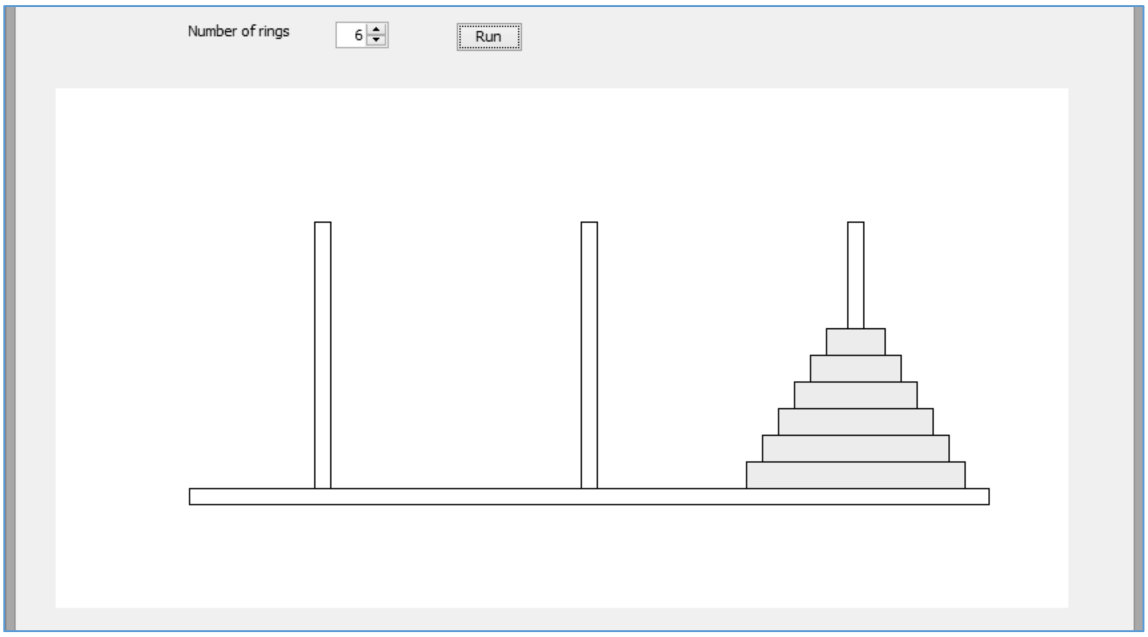
We will now ready to add the recursive method which will run the game. Add a ***moveRings( )*** method after ***moveStack( )***, as shown below. As we discussed earlier, this takes four input parameters: the ***number of rings*** to be moved, the letter identifying the ***starting location***, the letter identifying the ***destination***, and the rod which can be used for ***temporary storage***.

```
private void moveRings(int N, char start, char finish, char spare)
{
    if (N > 1)
    {
        moveRings(N - 1, start, spare, finish);
    }
    moveStack(start, finish);
    if (N > 1)
    {
        moveRings(N - 1, spare, finish, start);
    }
}
```

Return to the '***Run***' button click method and add a line which will make an initial call of the recursive method ***moveRings***. We request that the required number of rings ***N*** are moved from rod ***A*** to rod ***C***, with rod ***B*** available for temporary use.

```
        pointer0 = N;
        pointer1 = 0;
        pointer2 = 0;
        drawColumns();

        moveRings(N, 'A','C','B');

    }
```

Run the program. Select a number of rings, then click the '***Run***' button. The rings will probably appear immediately on ***rod C*** at the right of the diagram.

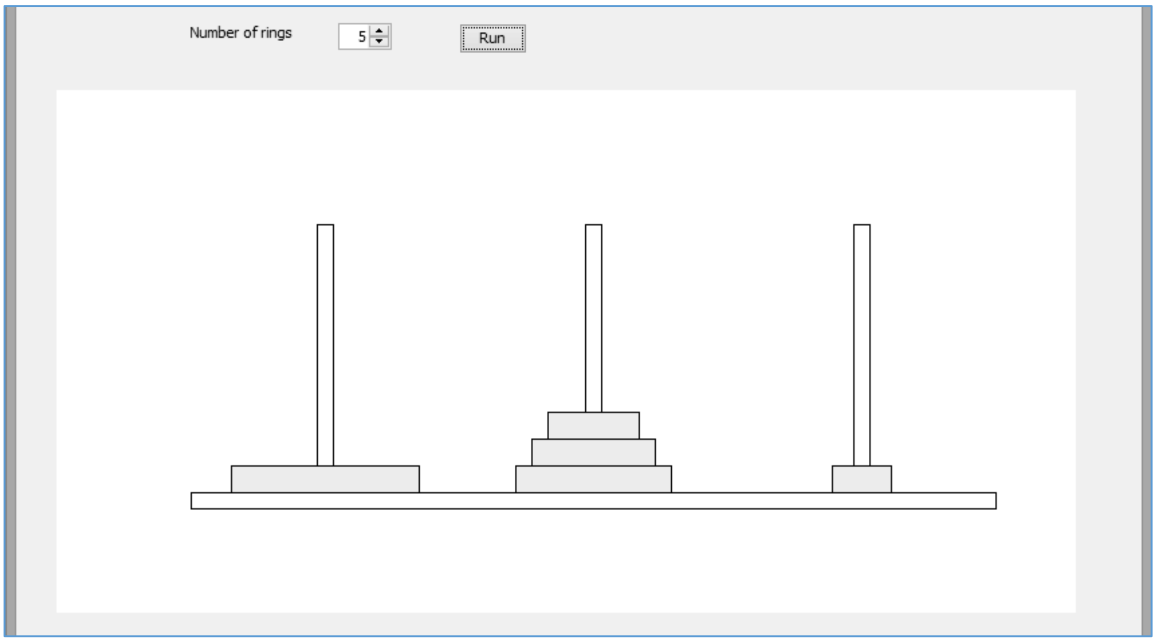


The program has worked correctly, but on most computers the processing takes place so fast that the individual moves cannot be seen. We need to introduce a time delay between moves.

Close the program window and return to the program code screen. An easy way to provide a delay is to make the computer waste time in carrying out some loop operation many times. Go to the end of the ***moveStack( )*** method and add a line of code to call a ***delay( )*** method. Write the method immediately underneath.

```
            if (finish == 'C')
            {
                stack2[pointer2]=ring;
                pointer2++;
            }
            setup();
            drawColumns();

            delay();

        }

        void delay()
        {
            double q;
            for (long i=0; i<20000;i++)
            {
                for (long z = 0; z <= 100000; z++)
                {
                     q = 0;
                }
            }
        }
```

Run the program again. Select a number of rings, then click the '***Run***' button. The sequence of moves should now be visible as an animation. You may wish to change the speed of the moves to suit your particular computer, by increasing or decreasing the number of repetitions which occur in the delay loops.



For the final program in this chapter, we will return to the topic of ***sorting*** data. You have already used the ***Bubble Sort*** algorithm in several projects. We will now look at another sorting method, called Quicksort, which operates by ***recursion***.

Investigate the speed and efficiency of the Quicksort algorithm by creating a program which will:

- input text documents of different lengths,
- in each case, split the document into individual words and sort the words alphabetically by two different sorting methods:
  Quicksort
  Bubble sort
- plot a graph to compare the sorting times achieved by the two methods when processing different lengths of text document.

To see how the Quicksort algorithm works, consider a series of words held in an array. These words need to be sorted into alphabetical order.

one	two	three	four	five	six	seven	eight	nine	ten

Quicksort takes one of the words as a comparison value, known as a ***pivot***. We will select the word '***one***' at the start of the array. The program then decides whether each of the other words would come before (←) or after (→) the comparison value in alphabetical order.

one	two	three	four	five	six	seven	eight	nine	ten
pivot	→	→	←	←	→	→	←	←	→

The program now makes a new copy of the data, putting all the items before the pivot value on the left, followed by the pivot itself, then all the items after the pivot on the right.

four	five	eight	nine	one	two	three	six	seven	ten
←	←	←	←	***pivot***	→	→	→	→	→

Although the groups of data to the left and right of the pivot are not yet sorted, the pivot itself must be in the correct position in the sequence. Any further sorting cannot alter its position.

The process is now repeated for the groups of unsorted data to the left and right of the pivot. The new comparison values will be '***four***' and '***two***'.

four	five	eight	nine	one	two	three	six	seven	ten
pivot	←	←	→	✓	***pivot***	←	←	←	←

Within each unsorted group, the program decides whether each word comes before or after the pivot value.

The words in each unsorted group are again rearranged, so that words before the pivot are listed first, then the pivot itself, then the words which come after the pivot value in alphabetical order.

five	eight	four	nine	one	three	six	seven	ten	two
←	←	***pivot***	→	✓	←	←	←	←	***pivot***

The pivot values '***four***' and '***two***' are now in correct positions in the sequence. However, '***nine***' must also be in a correct position as it is a single item which cannot be exchanged with any other word.

The sort continues by creating pivot comparison values for the remaining unsorted groups.

five	eight	four	nine	one	three	six	seven	ten	two
pivot	←	✓	✓	✓	***pivot***	←	←	←	✓

Rearranging the remaining data items gives:

eight	five	four	nine	one	six	seven	ten	three	two
←	***pivot***	✓	✓	✓	←	←	←	***pivot***	✓

The pivot items '***five***' and '***three***' are now in correct positions. The single item '***eight***' must also be correct. Sorting will continue with one remaining group of items.

eight	five	four	nine	one	six	seven	ten	three	two
✓	✓	✓	✓	✓	***pivot***	←	→	✓	✓

The final rearrangement completes the sorting.

eight	five	four	nine	one	seven	six	ten	three	two
✓	✓	✓	✓	✓	←	***pivot***	→	✓	✓

We can see that versions of the sorting method are taking place ***within itself***, so the problem can be solved by recursion. If a list of items ***L*** needs to be sorted, we will require three functions:

First(L) will find the ***first*** data item in the list, which will be used as the pivot comparison value

Before(L) will make a new list containing all the items which come ***before*** the pivot value in alphabetical order.

After(L) will make a new list containing all the items which come ***after*** the pivot value in alphabetical order.

An outline algorithm for the sorting process can then be written as:

```
Sort (L)
{
    sorted list = Sort( Before(L) ) + First (L)  +  Sort( After(L) )
}
```

The first call of the method will split the original list into two unsorted lists, with the pivot value in between. The ***Sort( )*** method can then be called recursively to sort each of these smaller lists.

We can now begin the program. Close all previous projects, then set up a ***New Project***. Give this the name ***quicksort***, and ensure that the ***Create Main Class*** option is not selected.

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

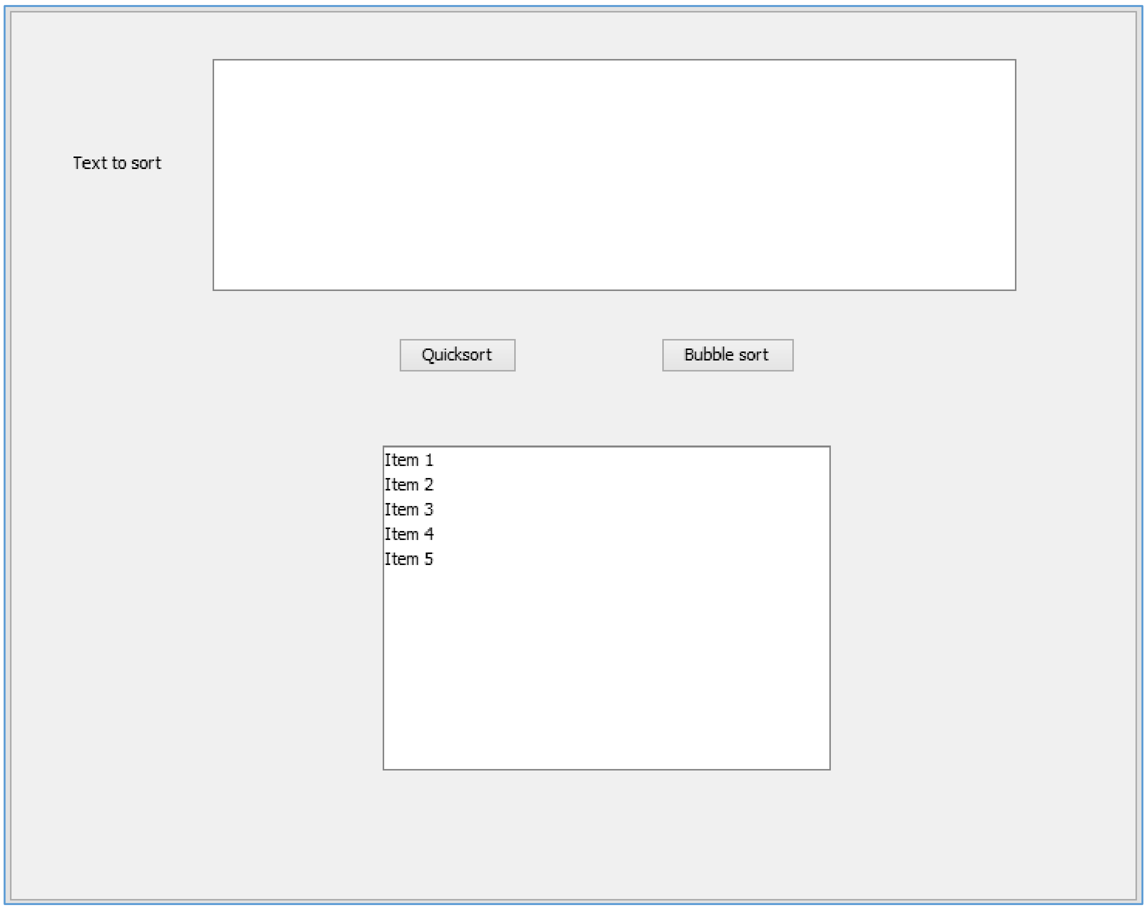
Return to the NetBeans editing screen.

- 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.

Use the Design tab to move to the form layout view. Add components to the form:

- A label '***Text to sort***'. Place a Text Area alongside and rename this as ***txtInput***.
- Buttons with the captions '***Quicksort***' and '***Bubble sort***'. Rename these as ***btnQuicksort*** and ***btnBubblesort***.
- A List component. Rename this as ***lstOutput***.



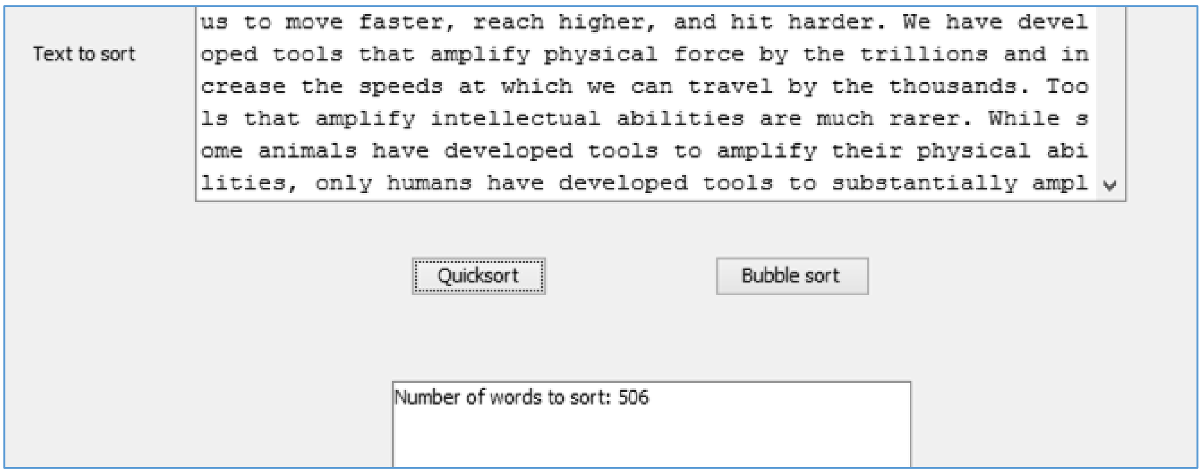
Use the ***Source*** tab to change to the program code screen. Add Java modules which will be needed for calculating the time taken by the sort operations, and to allow output to the list component. Set up the data model for the list.

```
package quicksortPackage;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.swing.DefaultListModel;

public class quicksort extends javax.swing.JFrame {

    DefaultListModel listModel;

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

Use the ***Design*** tab to return to the form layout view, then double click the '***Quicksort***' button to create a method. Add lines of code to the method. We begin by collecting the text document from the ***txtInput*** text area, then using the ***split( )*** function to separate this into an array of individual words. The number of words found will then be displayed in the ***lstOutput*** list box.

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

    listModel = new DefaultListModel();
    String s = txtInput.getText();
    s = s.toLowerCase();
    String[] wordArray = s.split(" ");
    lstOutput.setModel(listModel);
    int n = wordArray.length;
    listModel.addElement("Number of words to sort: " + n);
    lstOutput.setModel(listModel);

}
```

Run the program. Obtain a substantial amount of text data, perhaps from a Word document you have written or from an internet article, and paste this into the text area. Click the '***Quicksort***' button. The number of words in the text should be shown in the list box.



```
us to move faster, reach higher, and hit harder. We have devel
oped tools that amplify physical force by the trillions and in
crease the speeds at which we can travel by the thousands. Too
ls that amplify intellectual abilities are much rarer. While s
ome animals have developed tools to amplify their physical abi
lities, only humans have developed tools to substantially ampl
```

```
Number of words to sort: 506
```

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

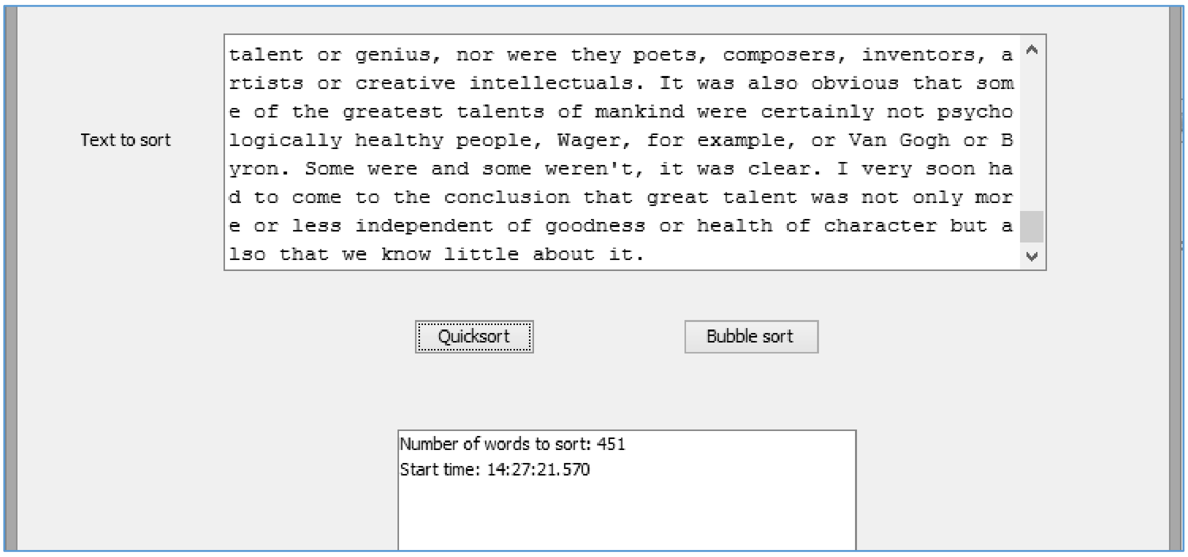
We wish to time the sorting operations, so that the speeds of the Quicksort and Bubble sort methods can be compared. To do this, the time can be printed out from the computer's clock just before the sorting begins, and again immediately that the sorting is completed.

Add lines of code to display the time at the start of the sort operation. This can be shown very accurately to a thousandth of a second!

```
        String[] wordArray = s.split(" ");
        lstOutput.setModel(listModel);
        int n = wordArray.length;
        listModel.addElement("Number of words to sort: " + n);

        DateFormat dateFormat = new SimpleDateFormat("HH:mm:ss.SSS");
        Date date = new Date();
        String startTime= dateFormat.format(date);
        listModel.addElement("Start time: " +startTime);

        lstOutput.setModel(listModel);
    }
```

Run the program. Input some text and click the '***Quicksort***' button. The ***time*** should be displayed. This is in the format:

hours : minutes : seconds . thousandths of a second



```
talent or genius, nor were they poets, composers, inventors, a
rtists or creative intellectuals. It was also obvious that som
e of the greatest talents of mankind were certainly not psycho
logically healthy people, Wager, for example, or Van Gogh or B
yron. Some were and some weren't, it was clear. I very soon ha
d to come to the conclusion that great talent was not only mor
e or less independent of goodness or health of character but a
lso that we know little about it.
```

```
Number of words to sort: 451
Start time: 14:27:21.570
```

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

Return to the '***Quicksort***' button click method. We will add a call to a ***quicksort( )*** method, which will carry out the actual sort. The array of unsorted words obtained from the text will be passed to the ***quicksort( )*** method as an input parameter.

When the sorting is completed, the time can again be read from the computer's clock and displayed. The difference in times will indicate the time taken by the sorting process.

```
        listModel.addElement("Number of words to sort: " + n);
        DateFormat dateFormat = new SimpleDateFormat("HH:mm:ss.SSS");
        Date date = new Date();
        String startTime= dateFormat.format(date);

        quicksort(wordArray);
        date = new Date();
        String finishTime= dateFormat.format(date);

        listModel.addElement("Start time: " +startTime);

        listModel.addElement("Finish time: " +finishTime);

        lstOutput.setModel(listModel);
    }

    private void quicksort(String[] wordArray)
    {

     }
```

The operation of the Quicksort method depends on having functions available to produce new lists of the data items which should occupy positions *before* or ***after*** the pivot item. We will create these functions, before moving on to complete the ***quicksort( )*** recursive method.

Set up a ***findBefore( )*** method below ***quicksort( )***.

```
     private void quicksort(String[] wordArray)
     {

      }

     public String[] findBefore(String[] words)
     {
         int n=words.length;
         String[] list = new String[n];
         int count=0;
         for (int i = 1; i < n; i++)
         {
             if (words[i].compareTo(words[0]) < 0)
             {
                 list[count]=words[i];
                 count++;
             }
         }
         String[] result = new String[count];
         for (int i = 0; i < count; i++)
         {
            result[i]=list[i];
         }
         return result;
     }
```

The operation of this method can be explained using our earlier example of sorting the names of the number from one to ten.

We pass the unsorted collection of data to the ***findBefore( )*** method as an array called ***wordArray***.

wordArray	one	two	three	four	five	six	seven	eight	nine	ten

The method then checks the number of elements in ***wordArray*** and creates another empty array called ***list*** with the same number of elements.

list										

The method reads the first element of ***wordArray*** and uses this as the ***pivot*** comparison value. A ***loop*** checks each remaining element of ***wordArray*** to see if it should come before the pivot value in alphabetical order. If so, the word is copied into the next empty element of the ***list*** array.

list	four	five	eight	nine						

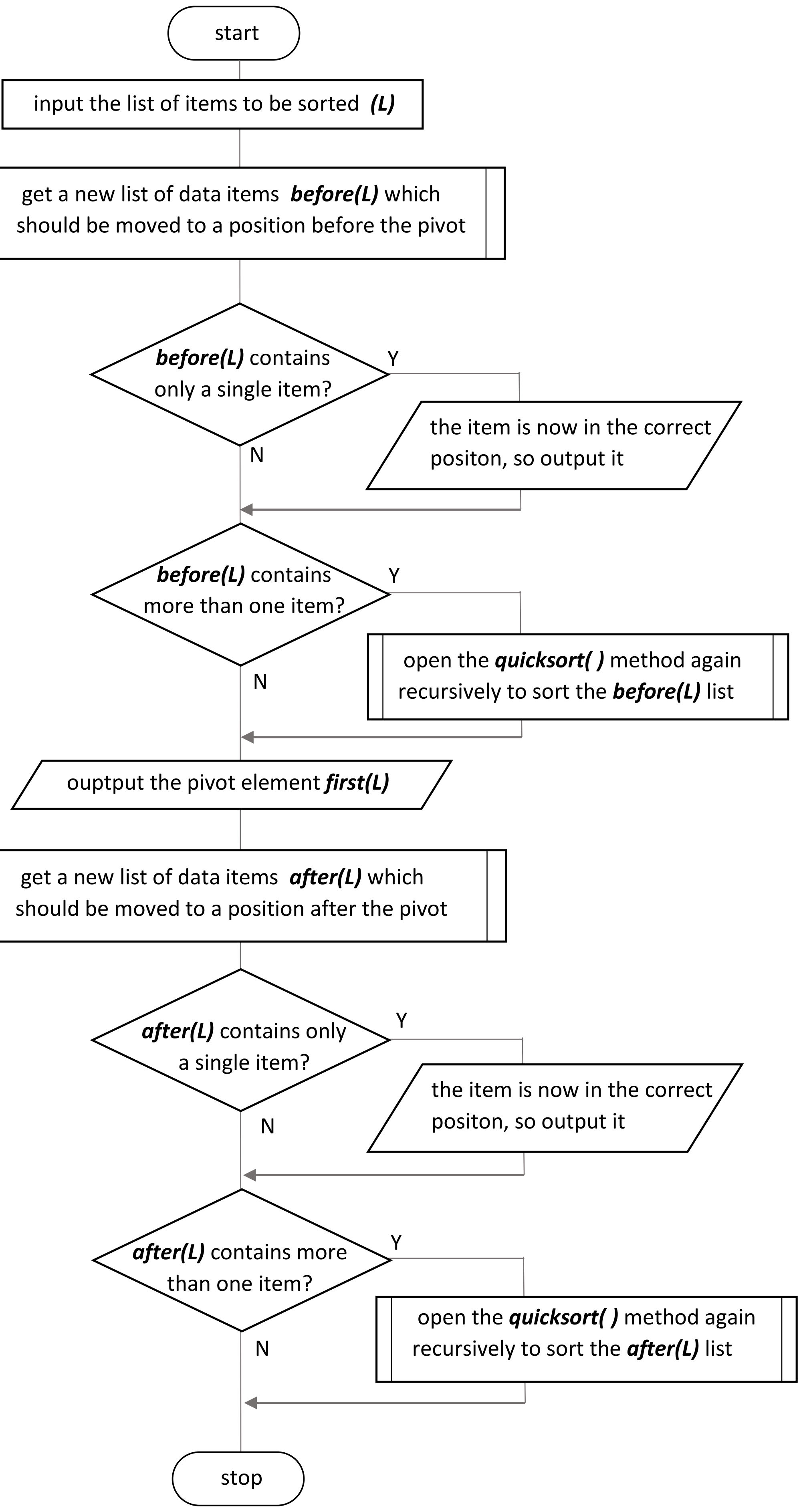
The final step is to set up a ***result*** array with the correct number of elements to hold the set of '***before***' words which were found, then return this to the ***quicksort( )*** method as the output parameter.

result	four	five	eight	nine

Add the ***findAfter( )*** method below ***findBefore( )***. This is almost identical, except that the '***less than***' sign in the ***compareTo( )*** function is replaced by a '***greater than***' symbol.

```
public String[] findAfter(String[] words)
{
    int n=words.length;
    String[] list = new String[n];
    int count=0;
    for (int i = 1; i < n; i++)
    {
        if (words[i].compareTo(words[0]) > 0)
        {

            list[count]=words[i];
            count++;
        }
    }
    String[] result = new String[count];
    for (int i = 0; i < count; i++)
    {
       result[i]=list[i];
    }
    return result;
}
```

We can now work on the ***quicksort( )*** method itself. A design is given in the flowchart.

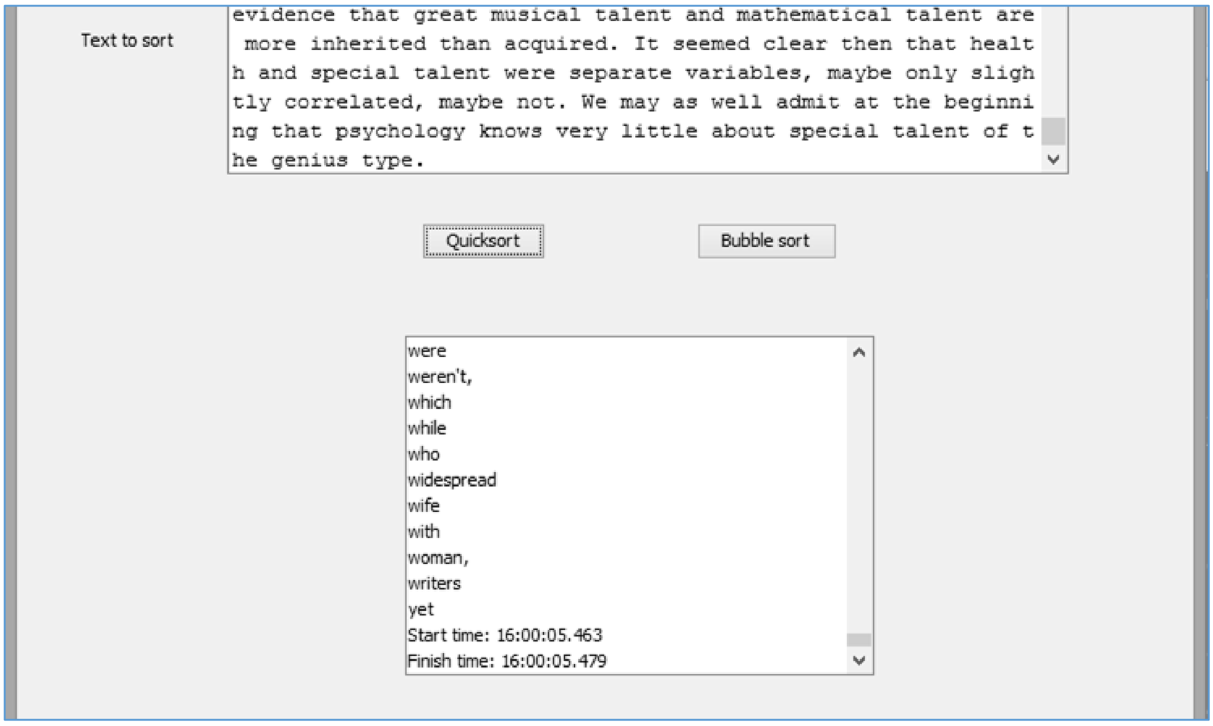

start
input the list of items to be sorted ***(L)***
get a new list of data items ***before(L)*** which should be moved to a position before the pivot
before(L) contains only a single item?
Y
N
the item is now in the correct positon, so output it
before(L) contains more than one item?
Y
N
open the ***quicksort( )*** method again recursively to sort the ***before(L)*** list
ouptput the pivot element ***first(L)***
get a new list of data items ***after(L)*** which should be moved to a position after the pivot
after(L) contains only a single item?
Y
N
the item is now in the correct positon, so output it
after(L) contains more than one item?
Y
N
open the ***quicksort( )*** method again recursively to sort the ***after(L)*** list
stop

Locate the empty ***quicksort( )*** method which we set up earlier. Add the lines of code to implement the recursive procedure as shown in the flowchart above.

```
private void quicksort(String[] wordArray)
{
    lstOutput.setModel(listModel);
    String[] before = findBefore(wordArray);
    if (before.length == 1)
    {
        listModel.addElement(before[0]);
    }
    if (before.length > 1)
    {
        quicksort(before);
    }

    listModel.addElement(wordArray[0]);

    String[] after = findAfter(wordArray);
    if (after.length == 1)
    {
        listModel.addElement(after[0]);
    }
    if (after.length > 1)
    {
        quicksort(after);
    }
}
```

Run the program, enter a block of text, then click the '***Quicksort***' button. Scroll through the output list to check that the words have been sorted into alphabetical order correctly.

Examine the start and finish times for the sort. You may find that the sorting operation took only a few milliseconds.



```
evidence that great musical talent and mathematical talent are
 more inherited than acquired. It seemed clear then that healt
h and special talent were separate variables, maybe only sligh
tly correlated, maybe not. We may as well admit at the beginni
ng that psychology knows very little about special talent of t
he genius type.
```

```
were
weren't,
which
while
who
widespread
wife
with
woman,
writers
yet
Start time: 16:00:05.463
Finish time: 16:00:05.479
```

Close the program window and return to the NetBeans editing screen. We will now set up a Bubble Sort method to sort the same text, so that the speeds can be compared.

Use the Design tab to move to the form layout view, then double click the '***Bubble sort***' button to create a method. The code which we will add is almost identical to the '***Quicksort***' button click method:

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

    listModel = new DefaultListModel();
    String s = txtInput.getText();
    s = s.toLowerCase();
    String[] wordArray = s.split(" ");
    lstOutput.setModel(listModel);
    int n = wordArray.length;
    listModel.addElement("Number of words to sort: " + n);
    DateFormat dateFormat = new SimpleDateFormat("HH:mm:ss.SSS");
    Date date = new Date();
    String startTime= dateFormat.format(date);
    bubbleSort(wordArray);
    date = new Date();
    String finishTime= dateFormat.format(date);
    listModel.addElement("Start time: " +startTime);
    listModel.addElement("Finish time: " +finishTime);
    lstOutput.setModel(listModel);

}
```

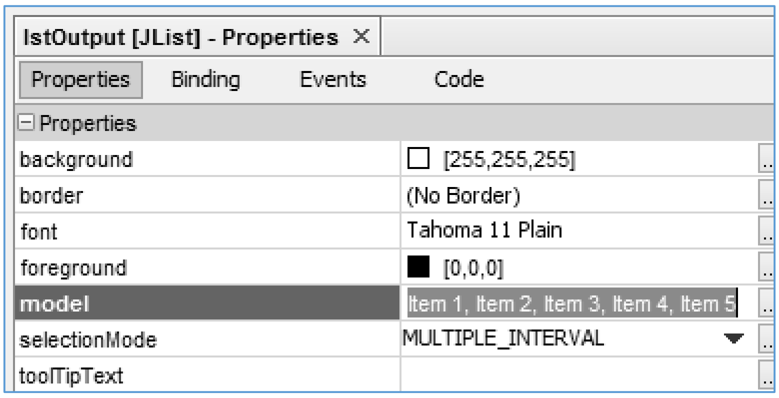
The button click method calls a ***bubbleSort( )*** method, which we will add immediately underneath. This uses the same algorithm as in several of our previous programs.

```
private void bubbleSort(String[] words)
{
    int n = words.length;
    Boolean swap=true;
    while (swap==true)
    {
        swap = false;
        String temp;
        for (int i = 0; i < n-1; i++)
        {
            if (words[i].compareTo(words[i + 1]) > 0)
            {
                temp=words[i];
                words[i] = words[i + 1];
                words[i + 1] = temp;
                swap = true;
            }
        }
    }
 }
```

Some common words, such as '***and***' and '***the***', are likely to occur many times in the text used as test data. To simplify the output, we will add a loop which avoids the same word being output more than once in the alphabetical list. Add lines of code near the end of the method to do this.

```
    for (int i = 0; i < n-1; i++)
    {
      if (words[i].compareTo(words[i + 1]) > 0)
      {
        temp=words[i];
        words[i] = words[i + 1];
        words[i + 1] = temp;
        swap = true;
      }
    }
  }
  lstOutput.setModel(listModel);
  listModel.addElement(words[0]);
  for (int i = 1; i < n; i++)
  {
    if (words[i].compareTo(words[i-1]) !=0)
     listModel.addElement(words[i]);
  }
}
```

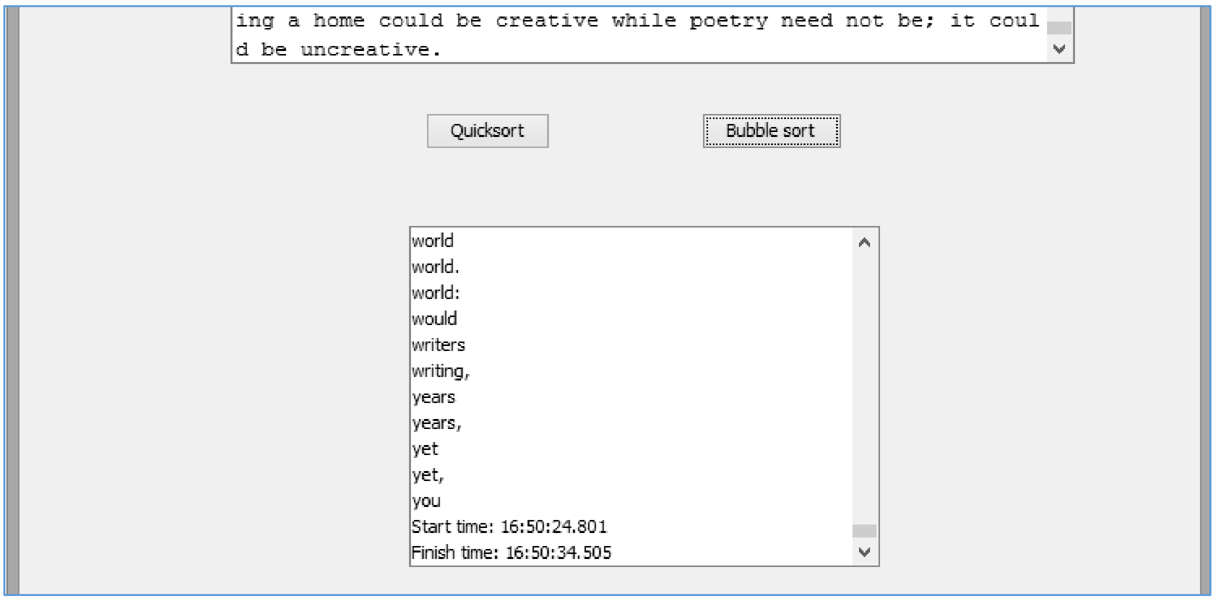
To complete the program, use the Design tab to return to the form layout page. Select the lstOutput list. Go to the Properties window and locate the model property. Delete the entries item1 … item 5 from the right hand column, so that the list appears empty when the program first runs.



background	☐ [255,255,255]
border	(No Border)
font	Tahoma 11 Plain
foreground	■ [0,0,0]
model	Item 1, Item 2, Item 3, Item 4, Item 5
selectionMode	MULTIPLE_INTERVAL
toolTipText	

Run the program. Enter test data and click the 'Bubble sort' button. Check that the words of the text are correctly sorted and displayed.

You may find that sorting times are much longer for the Bubble sort in comparison to the Quicksort, particularly for large text articles.



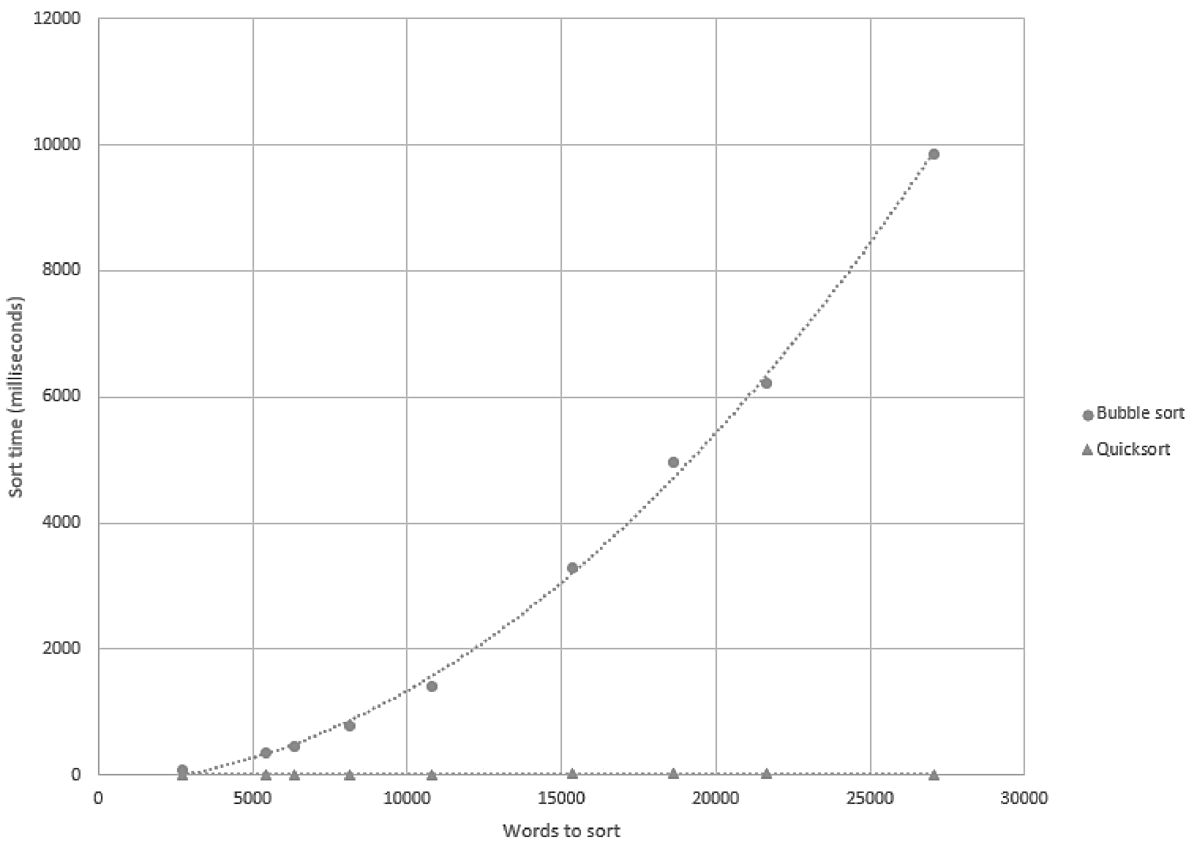
```
ing a home could be creative while poetry need not be; it coul
d be uncreative.
```

```
world
world.
world:
would
writers
writing,
years
years,
yet
yet,
you
Start time: 16:50:24.801
Finish time: 16:50:34.505
```

Carry out experiments with different lengths of text article, up to about 30,000 words in length, and plot the results as a graph using a spread sheet.

Sorting times for the ***Bubble sort*** should increase as a ***polynomial curve***, whereas the times for ***Quicksort*** are closer to ***linear***.

On a fast computer, the ***Quicksort*** method may show very little increase in sort time as the size of the data increases. The actual sorting method is so fast that other system processes, such as output of the screen display, are dominating the overall sort times that we are recording.


Sort time (milliseconds)
Words to sort
● Bubble sort
▲ Quicksort