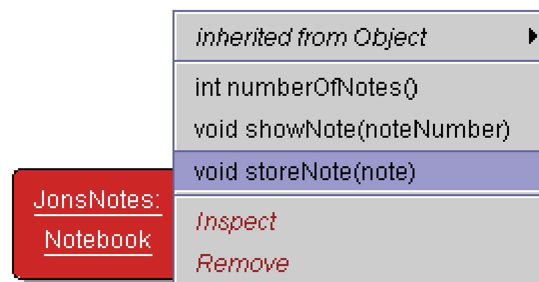
Chapter 4 - notebook1 Project

Chapter 4 Book exercises:

Ex 4.1 Open the **notebook1** project in BlueJ and create a **Notebook** object. Store a few notes into it - they are simple strings - then check the number of notes returned by **numberOfNotes** matches the number that you stored.

When you use the **showNote** method, you will need to use a parameter value of 0 (zero) to print the first note, 1 (one) to print the second note, and so on. We shall explain the reason for this numbering in due course.



Concept "**Collections**: Collection objects are objects that can store an arbitrary number of other objects."[1]

Note The main point to note is that the **notebook1** project makes use of **Java library classes**…

```
import java.util.ArrayList;
```

…and specifically, notebook1 makes use of the **ArrayList** class that is defined in the `java.util` package.

Note that the **import** statement makes the **ArrayList** class from the `java.util` package available to us for use in our **notebook1** project.

Constructor method Note that the **notebook1** constructor method creates an **instance** of an **ArrayList** object, with the object reference **notes**.

```
public Notebook()
{
    notes = new ArrayList();
}
```

The methods in the **notebook1** project make use of two of the methods that have been **predefined** for all **ArrayList** objects. These are…

```
public void storeNote(String note)
{
    notes.add(note);
}
```

…**add**, providing us with the ability to **add a note**, and…

[1] Barnes & Kolling, Objects First With Java, p78

```
public int numberOfNotes()
{
    return notes.size();
}
```

…**size**, providing us with the size of the current **notes ArrayList** object.

Remember that an **ArrayList** object keeps its own private count of the number of items that it is storing and that the **size** method returns this number.

The provision of this facility means that we **do not have to** consider maintaining our **own records** of how many notes are stored.

The remaining method in the **notebook1** project, **showNote**, simply checks to see if the note that the user has opted to display is a **valid note number** and, if it is, then proceeds to **display** the note in the **terminal window**.

Note that the numbering of the notes **starts at 0**, thus the initial check is to ensure that the user has not opted to display a note that is less than 0.

The next check is to ensure that the user has not opted to display a note that is above the actual number of notes i.e. one that does not yet exist.

```
public void showNote(int noteNumber)
{
    if(noteNumber < 0) {
        // This is not a valid note number, so do nothing.
    }
    else if(noteNumber < numberOfNotes()) {
        // This is a valid note number, so we can print it.
        System.out.println(notes.get(noteNumber));
    }
    else {
        // This is not a valid note number, so do nothing.
    }
}
```

Note

At this point we should appreciate the ability to use predefined classes without necessarily knowing the ins and outs of exactly how those classes work. Suffice to say that, we should be happy enough knowing that we can use **ArrayList** objects **to provide storage for arbitrary number of objects in the classes that we define**.

Positioning

Remember that items stored in collections are numbered from zero upwards. The position of an object within a collection is referred to as its index. The line of code shown below makes use of this index position to display a specific note, as requested by the user.

```
System.out.println(notes.get(noteNumber));
```

Range

Note that the code of the showNote method ensures that the note the user is opting to display is 'in range'. The range can be defined as follows:

```
[0 … (size-1)]
```

Why **size - 1**? If the size is **10**, for example then there are **10** notes in total. These will be numbered as shown below:

```
1st  note    0
2nd  note    1
3rd  note    2
4th  note    3
5th  note    4
6th  note    5
7th  note    6
8th  note    7
9th  note    8
10th  note   9
```

Ex 4.2 Implement a **removeNote** method in your notebook.

Add the code shown on page 83 of the text.

```
public void removeNote(int noteNumber)
{
    if(noteNumber < 0){
        // this is not a vaid note number, so do nothing
    }
    else if(noteNumber < numberOfNotes()){
        // this is a valid note number, so we can remove it
        notes.remove(noteNumber);
    }
    else{
        // this is not a valid note number, so do nothing
    }
}
```

Add the code, compile and then create an instance of a **Notebook** object. **Enter 3 or 4 similar notes** so that it is easy to keep track of them.

I would suggest that you then invoke the **removeNote** method a number of times, removing the note with the **zero index** each time. Then invoke the **showNote** method, again opting for the note with the **zero index**.

If you do this, you will see how the removal of a note in the zero index position results in the **movement of the following notes** - the previous note at **index 1 moves** to **index position zero** and so on.

Note It should be obvious to you from the code that you have just added, that **remove** is another example of a **predefined method** that is available to **objects** of the **ArrayList** class.

Ex 4.3 What might the header of a **listAllNotes** method look like? What sort of return type should it have? Does it need to take any parameters?

```
public void listAllNotes()
```

You could write all of the notes to a String and return this String. However, I would suggest that it would be easiest to **just display the notes to the terminal window**. Thus, I have opted for a return type of **void** in the above line of code to indicate that there is **nothing returned**.

Because you are defining a method that will list **all of the notes** then there is **no need to pass any parameters**. If, for example, you wanted to display a **specific note** then this would, of course, be different - you would want to specify **which note** to print. However, as you wish to display **all** notes, there is no need to pass any parameters.

Ex 4.4

We know that the first note is stored at index zero in the **ArrayList** so could we write the body of **listAllNotes** along the following lines?

```
System.out.println(notes.get(0));
System.out.println(notes.get(1));
System.out.println(notes.get(2));
```

etc.

How many `System.out.println(notes.get(x));` would you need? When would you stop? What if you attempted to get a note that no longer existed? What if you added new notes? Basically, the suggested solution above is not really feasible. Thus, we need to look at how else to complete this task - **loops**.

Ex 4.5

Implement the **listNotes** method in your version of the notebook project. (A solution with this method implemented is provided in the **notebook2** version of this project, but to improve your understanding of the subject we recommend that you write this method yourself.)

```
public void listNotes()
{
    int index = 0;
    while(index < notes.size()){
        System.out.println(notes.get(index));
        index++;
    }
}
```

The **listNotes** method makes use of a local variable of type int called **index**. This is initialised to **0** an it is then used as the **counter** that **controls the repetition of the while loop** - **index** must be less than the size returned by `notes.size()` and, as long as this condition is **true**, there must **still be a note to print**. Thus, the action is performed - the note is printed. The value of **index** is then **incremented** so that, next time round the **while** loop, **index** refers to the **next note**.

Ex 4.6

Create a **Notebook** and store a few notes in it. Use the **listNotes** method to print them out to check that the method works as it should.

The method does work!

Ex 4.7

If you wish, you could use the debugger to help yourself understand how the statements in the body of the while loop are repeated. Set a breakpoint just before the loop, and step through the method until the loops condition evaluates to **false**.

Watch as the value of **index** is incremented. Note the effect that this has in determining whether the **while** loop runs again or not.

Ex 4.8

Modify **showNote** and **removeNote** to print out an error message if the note number entered was not valid.

The suggested code for this is shown below:

The **showNote** method:

```
public void showNote(int noteNumber)
{
    if(noteNumber < 0) {
        // This is not a valid note number, so do nothing.
    }
    else if(noteNumber < numberOfNotes()) {
        // This is a valid note number, so we can print it.
        System.out.println(notes.get(noteNumber));
    }
    else {
        System.out.println("Invalid note number");
    }
}
```

The **removeNote** method:

```
public void removeNote(int noteNumber)
{
    if(noteNumber < 0){
        // this is not a vaid note number, so do nothing
    }
    else if(noteNumber < numberOfNotes()){
        // this is a valid note number, so we can remove it
        notes.remove(noteNumber);
    }
    else{
        System.out.println("Invalid note number");
    }
}
```

Note that the provision of an error message is easily achieved via the addition of an appropriate `System.out.println` line in place of the `// this is not a valid note number, so do nothing` comments that were there previously.

Ex 4.9

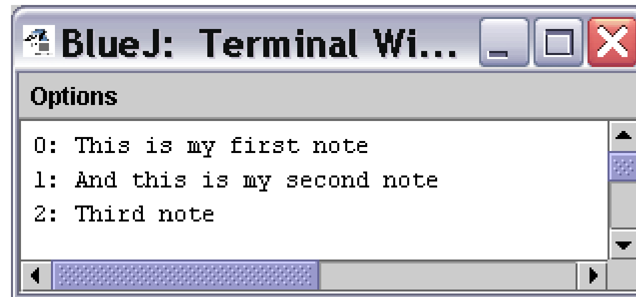
Modify the **listNotes** method so that it prints the value of the **index** local variable in fron of each note. For instance:

0: Buy some bread.
1: Recharge phone.
2: 11.30. Meeting with John.

This makes it much easier to provide the correct index when removing a note. The suggested code for this is shown below. Note that this involves a minor change to the `System.out.println` line, to include an actual reference to the **index** variable.

```
public void listNotes()
{
    int index = 0;
    while(index < notes.size()){
        System.out.println(index + ": " + notes.get(index));
        index++;
    }
}
```

The Result:



Ex 4.10

Within a single execution of the **listNotes** method, the **notes** collection is asked repeatedly how many notes it is currently storing. This is done every time the loop condition is checked. Does the value returned by **size** vary from one check to the next? If you think the answer is 'No', then rewrite the **listNotes** method so that the size of the notes collection is determined only once and stored in a local variable prior to execution of the loop. Then use the local variable in the loops condition rather than the call to **size**. Check that this version gives the same results. If you have problems completing this exercise, try using the debugger to see where things are going wrong.

There is no possibility that the **notes** collection will change in size at the same time as you are invoking the **listNotes** method and, thus, for the duration of this method, there will only ever be **one value for the size of the collection**. Thus, to continually request this value is inefficient.

A better option is provided below. Note that the size of the collection is obtained **only once** and it is **this value** that determines the number of times that the **while** loop will run.

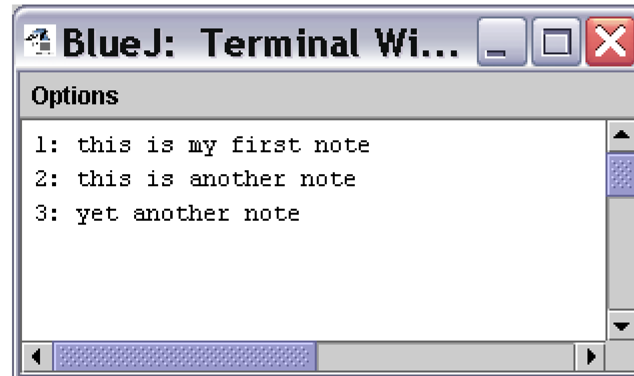
A local variable **size** of type **int** is assigned the value that is returned from `notes.size();` and it is this **size** variable that is then used to determine the number of times that the **while** loop runs.

```
public void listNotes()
{
    int size = notes.size();
    int index = 0;
    while(index < size){
        System.out.println(index + ": " + notes.get(index));
        index++;
    }
}
```

Ex 4.11

Change your notebook so that notes are numbered starting from 1, rather than zero. Remember that the **ArrayList** object will **still be using indices starting from zero**, but you can present the notes numbered from 1 in your listing. Make sure you modify **showNote** and **removeNote** appropriately.

The resulting display would appear as follows:



The complete code for the **Notebook** class is shown below. Note that the user will be able to think of the notes as being note 1, note 2 and so on but that these notes will actually be note 0, note 1 etc because of the way that the numbering of an **ArrayList** works.

Because of this, when the user opts to display a note or remove a note, we must **subtract 1** from the number that they enter and this gives us the true index position of the note in question.

Create a notebook with a number of notes and try this for yourself.

```
import java.util.ArrayList;

public class Notebook
{
   // Storage for an arbitrary number of notes.
   private ArrayList notes;

   public Notebook()
   {
       notes = new ArrayList();
   }

   public void storeNote(String note)
   {
       notes.add(note);
   }
```

```
    public int numberOfNotes()
    {
        return notes.size();
    }

    public void showNote(int noteNumber)
    {
        if(noteNumber < 1) {
            // This is not a valid note number, so do nothing.
        }
        else if(noteNumber <= numberOfNotes()) {
            // This is a valid note number, so we can print it.
            System.out.println(notes.get(noteNumber-1));
        }
        else {
            System.out.println("Invalid note number");
        }
    }


    public void removeNote(int noteNumber)
    {
        if(noteNumber < 1){
            // this is not a vaid note number, so do nothing
        }
        else if(noteNumber <= numberOfNotes()){
            // this is a valid note number, so we can remove it
            notes.remove(noteNumber-1);
        }
        else{
            System.out.println("Invalid note number");
        }
    }

    public void listNotes()
    {
        int size = notes.size();
        int index = 0;
        while(index < size){
            System.out.println((index + 1) + ": "
                                   + notes.get(index));
            index++;
        }
    }
}
```

Note: Iterator

"An **iterator** is an object that provides functionality to iterate over all elements of a collection."[2]

[2] Barnes & Kolling, Objects First With BlueJ, p87

Note: null

"The Java keyword **null** is used to mean **'no object'** when an object variable is not currently referring to a particular object. A field that has not explicitly been initialized will contain the value null by default."[3]

Ex 4.12

Find a further example of casting in the **getLot** method of the **Auction** class. The **getLot** method contains the line below that shows the use of casting:

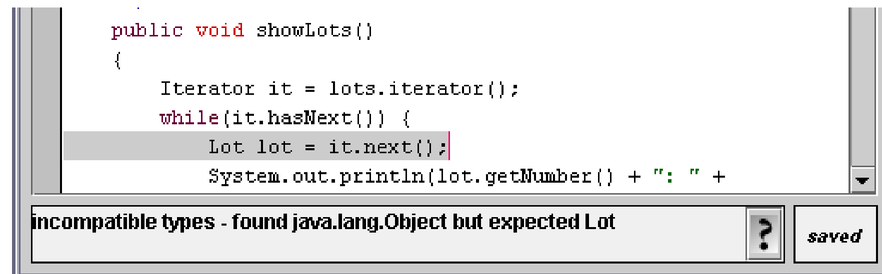
```
Lot selectedLot = (Lot) lots.get(number-1);
```

Ex 4.13

What happens if you try to compile the **Auction** class without one of the casts? For instance, edit the **showLots** method so that the first statement in the body of the while loop reads:

```
Lot lot = it.next( );
```

The effect of removing the cast is as shown below, an **Incompatible types** error message **found java.lang.Object but expected Lot**:



Ex 4.14

Add a **close** method to the **Auction** class. This should iterate over the collection of lots and print out details of all the lots. For lots that have been sold, the details should include the **name of the successful bidder**, and **the value of the winning bid**. For lots that have not been sold, print a message that indicates this fact.

The idea is that the **close** method will run through the list of lots and will regard any that have at **least one bid** against them as being **sold**.

[3] Barnes & Kolling, Objects First With BlueJ, p90

close Method

The suggested code for the **close** method is shown below:

```
public void close()
{
  Iterator it = lots.iterator();
  while(it.hasNext()) {
    Lot lot = (Lot) it.next();
    if(lot.getHighestBid() != null)
    {
      System.out.println(lot.getNumber() + ": " +
                         lot.getDescription());
      System.out.println("Name of bidder: " +
                         lot.getHighestBid().getBidder().getName());
      System.out.println("Value of bid: " +
                         lot.getHighestBid().getValue());
     }
  }
}
```

Ex 4.15

Add a **getUnsold** method to the **Auction** class with the following signature:

public ArrayList getUnsold( )

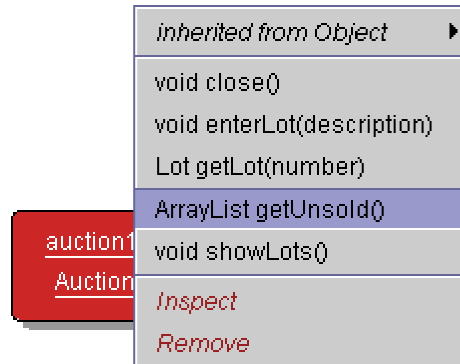
The idea is that the **getUnsold** method will run through the list of lots and will regard any that have **no bids** against them as being **unsold**.

This method should iterate over the **lots** field, storing unsold lots in a new **ArrayList** local variable. At the end of the method, return the list of unsold lots.

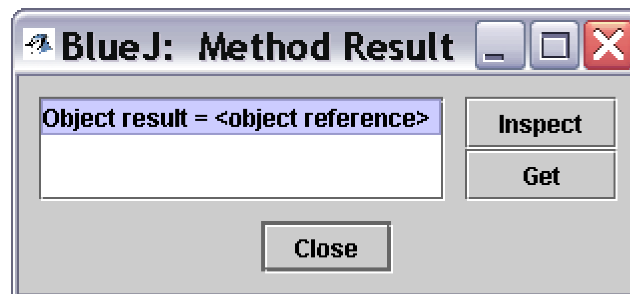
```
public ArrayList getUnsold()
{
  ArrayList unsoldLots = new ArrayList();
  int nextLotNumber = 1;

  Iterator it = lots.iterator();
  while(it.hasNext()) {
    Lot lot = (Lot) it.next();
    if(lot.getHighestBid() == null)
    {
      unsoldLots.add(new Lot(nextLotNumber,lot.getDescription()));
      nextLotNumber++;
    }
  }
  return unsoldLots;
}
```

Note that the text specifically requests that you store unsold lots in a new **ArrayList** local variable and that the method should **return the ArrayList**.

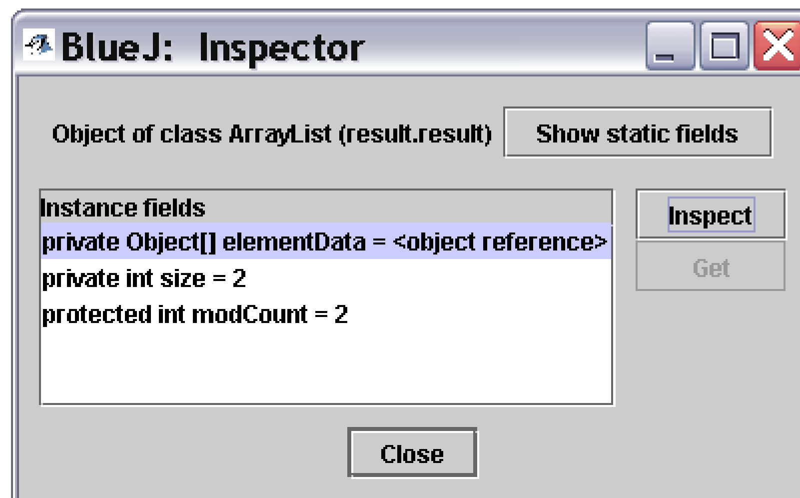
The use of an ArrayList in this way can be seen when you invoke the **getUnsold** method…



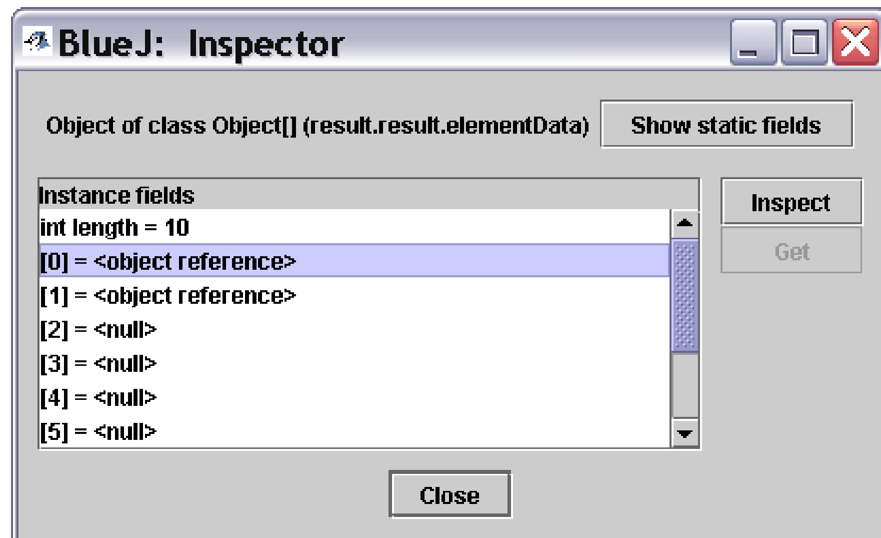
The following dialog is displayed…



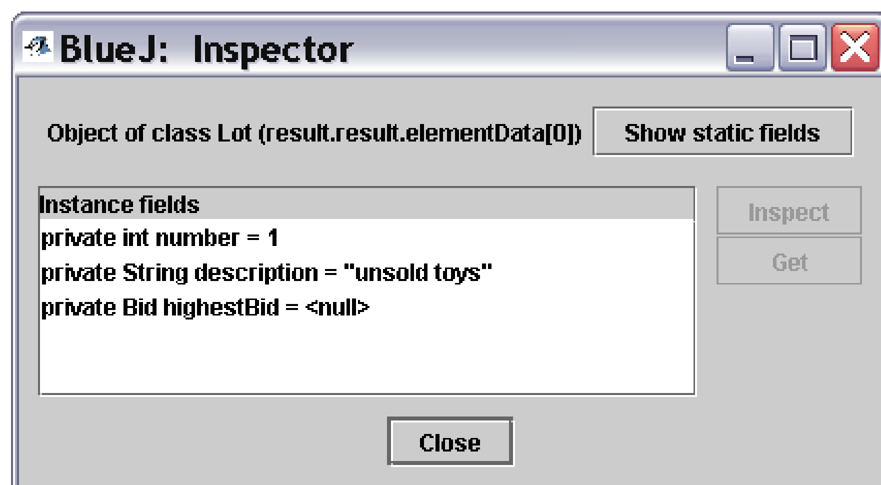
Click once to select the **<object reference>** and then click **Inspect**. You will see the following dialog. Click once on the **<object reference>** line shown below and again click **Inspect**.



The following dialog is displayed…



Click once on the **<object reference>** line shown below and again click **Inspect**. You should finally see a String description that relates to one of your unsold lots. Note also that the **highestBid** Bid object reference **does not point to an object** and is, instead, a **null** reference.



Note

You were specifically asked to add a **getUnsold** method to the **Auction** class with the following signature:

public ArrayList getUnsold( )

In other words, you were specifically asked to create a **getUnsold** method with **ArrayList** as the return type**.**

However, as you can see from above, our use of an **ArrayList** does complicate things somewhat in terms of accessing the result.

The methodology is not without merit - it does demonstrate that we can create a method with an **ArrayList** for a return type.

However, the code could be simplified as shown below in a way that makes it very similar to the **close** method - simply **printing out the details of the unsold lots**.

```
public void getUnsold()
{
    Iterator it = lots.iterator();
    while(it.hasNext()){
        Lot lot = (Lot) it.next();
        if(lot.getHighestBid() == null){
            System.out.println("Lot: " + lot.getNumber()
                               + ": " + lot.getDescription()
                               + " is unsold.");
        }
    }
}
```

Ex 4.16

Suppose that the **Auction** class includes a method that makes it possible to remove a lot from the auction. Assuming that the remaining lots do not have their **lotNumber** fields changed when a lot is removed, what impact would the ability to remove lots have on the **getLot** method?

The **Auction** class keeps a **'running tally'** of the number of lots by incrementing an instance variable called **nextLotNumber**.

The constructor method for the **Auction** class initialises the **nextLotNumber** variable to **1** and this is used in the actual **creation of the first Lot** - it is passed to the **constructor** method for the **Lot**. It is, at this point, assigned as the value of the **Lot's number** instance variable.

The **lots ArrayList**, into which **Lots** are entered, will start at **0**.

If the **Auction** class includes a method that makes it possible to remove a lot from the auction, and assuming that the remaining lots do not have their **lotNumber** fields changed when a lot is removed, the effect is that the linkage between the **nextLotNumber** instance variable, the **Lot's number** instance variable, and the **actual numbering in the ArrayList** is **lost**.

The **getLot** method is shown below:

```
public Lot getLot(int number)
{
    if((number >= 1) && (number < nextLotNumber)) {
        // The number seems to be reasonable.
        Lot selectedLot = (Lot) lots.get(number-1);
        // Include a confidence check to be sure we have the
        // right lot.
        if(selectedLot.getNumber() != number) {
            System.out.println("Internal error: " +
                               "Wrong lot returned. " +
                               "Number: " + number);
        }
        return selectedLot;
    }
    else {
        System.out.println("Lot number: " + number +
                           " does not exist.");
        return null;
    }
}
```

As can be seen from the above code, the **getLot** method expects an **integer** to be passed as a parameter. This integer - **number** - is then checked to ensure that it is greater than or equal to one and also that it is less than the value held in **nextLotNumber** - which is always going to be one more than the number of lots.

The **number** is then used to retrieve the actual **Lot** from the **Lots ArrayList**. Note that this involves a **cast**. To obtain the **actual** lot involves **deducting 1 from number** i.e. Lot number **1** will be in **ArrayList** index position **0**, Lot number **2** will be in **ArrayList** index position **1** and so on.

Note that there is a check whereby the **lot number** of the **selectedLot** object reference is **checked**, to ensure that the **correct lot** has been returned.

Note that a **null** reference is returned if the **number** passed to the method does not meet the criterion of the **if** statement i.e. the **number** is **1 or above <u>and</u> less than** the value of the **nextLotNumber** variable.

If we remove lots from the **ArrayList**, because of the **automatic contraction of the ArrayList**, the **number - 1** concept will fail to work, as the **lot's number** will **no longer be an accurate representation of its position in the ArrayList**.

Ex 4.17

Rewrite **getLot** so that it does not rely on a lot with a particular number being stored at index (**number-1**) in the collection. You may assume that lots are always stored in increasing order of their lot number.

If we consider that, when lots are created, they will be assigned a **number** from **1** to the incremented **nextlotNumber**, it is still sensible to check that the number passed to the method **falls between these two limits**.

If the number does not fall between these limits, then we are immediately told that this is "**Not a valid lot number**" and a **null** reference is returned.

If the number does fall between these limits then the **ArrayList** is checked for **Lot** objects where the **number** of the lot **matches the number entered**. If a match **is** found then that **lot** is **returned**. If **no match** is found, then the user is informed that "**No lot number x exists.**" and a **null** reference is returned.

```
public Lot getLot(int number)
{
    if((number >= 1) && (number < nextLotNumber)){
        Iterator it = lots.iterator();
        while(it.hasNext()) {
            Lot selectedLot = (Lot) it.next();
            if(selectedLot.getNumber() == number)
                return selectedLot;
        }// end of while loop
        System.out.println("No lot number " + number + " exists.");
        return null;
    }
    else{
        System.out.println("Not a valid lot number.");
        return null;
    }
}
```

Ex 4.18

Add a **removeLot** method to the **Auction** class, having the following signature:

```
/**
 * Remove the lot with the given lot number.
 * @param number The number of the lot to be removed
 * @return The lot with the given number, or null if
 * there is no such lot.
 */

public Lot removeLot(int number)
```

This method should not assume that a lot with a given number is stored at any particular location within the collection.

This follows on from the previous example - we should not assume that **number - 1** represents the **actual** position of the lot in the **ArrayList**. We therefore take a **similar approach** to that used immediately above.

```
public Lot removeLot(int number)
{
 if((number >= 1) && (number < nextLotNumber)){
     Iterator it = lots.iterator();
     int counter = 0;
     while(it.hasNext()) {
         Lot selectedLot = (Lot) it.next();
         if(selectedLot.getNumber() == number){
             lots.remove(counter);
             return selectedLot;
         }
         counter++;
     }// end of while loop
     System.out.println("No lot number " + number +
                        " exists, therefore unable to remove lot.");
     return null;
 }
 else{
     System.out.println("Not a valid lot number.");
     return null;
 }
}
```

Chapter 4 - weblog-analyzer Project

Ex 4.19

Explore the **weblog-analyzer** project by creating a **LogAnalyzer** object and calling its **analyzeHourlyData** method. Follow that with a call to its **printHourlyCounts** method, which will print the results of the analysis. Which are the busiest times of day?

The busiest times of day are as follows:

```
10: 227
14: 227
18: 237
```

Remember that the hours are numbered from **0** to **23** so can actually be thought of as follows:

Array Index	24hr time	'Real' time
10	11:00	11am
14	15:00	3pm
18	19:00	7pm

Ex 4.20

Write a declaration for an array variable **people** that could be used to refer to an array of **Person** objects.

```
private Person[] people;

private Person[] people = new Person[integer-expression];
```

Ex 4.21

Write a declaration for an array variable **vacant** that could be used to refer to an array of **boolean** values.

```
private boolean[] vacant;

private boolean[] vacant = new boolean[integer-expression];
```

Ex 4.22

Read through the **LogAnalyser** class and identify all the places where the **hourCounts** variable is used. At this stage, do not worry about what all the uses mean as they will be explained in the following sections. Note how often a pair of square brackets is used with the variable.

hourCounts is used in the following lines...

…when **declaring instance variables**...

```
private int[] hourCounts;
```

Within the **constructor** method for the **LogAnalyzer** class...

```
hourCounts = new int[24];
```

Within the **AnalyseHourlyData** method

```
hourCounts[hour]++;
```

Within the **printHourlyCounts** method...

Note that **hourCounts** is used in terms of the **length** of the array and also with the index **hourCounts[hour]**

```
public void printHourlyCounts()
{
    System.out.println("Hr: Count");
    for(int hour = 0; hour < hourCounts.length; hour++) {
        System.out.println(hour + ": " + hourCounts[hour]);
    }
}
```

Ex 4.23

Given the following variable declarations:

double[ ] readings;
String[ ] urls;
TicketMachine[ ] machines;

Write assignments that accomplish the following tasks:

(a) Make the **readings** variable refer to an array that is able to hold 60 **double** values;
(b) Make the **urls** variable refer to an array that is able to hold 90 **String** objects;
(c) Make the **machines** variable refer to an array that is able to hold five **TicketMachine** objects.

Results are as follows:

```
double[] readings;
readings = new double[60];

String[] urls;
urls = new String[90];

TicketMachine[] machines;
machines = new TicketMachine[5];
```

Ex 4.24

How many String objects are created by the following declaration?

String[ ] labels = new String[20];

Watch this **trick question**... **no actual String objects are created** by the above line. What the line does do is to create a **fixed-size collection** that is **able to have 20 Strings stored within it**.

Note: Array indices always start at zero.

Note: "Two very common errors are to think that the valid indices of an array start at 1, and to use the value of the length of the array as an index. Using indices outside the bounds of an array will lead to a runtime error called an ArrayOutOfBoundsException."[4]

Note: Expressions that select an element from an array can be used anywhere a value of the base type of the array could be used.

---

[4] Barnes & Kolling, Objects First With BlueJ, p99

Note: Using an array index on the left-hand side of an assignment is the array equivalent of a mutator (or set method) because the contents of the array will be changed. Using one anywhere else represents the equivalent of an accessor (or get method).

Note: `hour < hourCounts.length`

"All arrays contain a field length that contains the value of the fixed size of that array. The value of this field will always match the value of the integer expression used to create the array object."[5]

Ex 4.25 Check to see what happens if the for loops condition is incorrectly written using the '<=' operator in **printHourlyCounts**:

for(int hour = 0; hour <= hourCounts.length; hour ++)

The result of using **<=** is an **ArrayIndexOutOfBoundsException** error when you attempt to run the program.

Ex 4.26 Rewrite the body of **printHourlyCounts** so that the **for** loop is replaced by an equivalent **while** loop. Call the rewritten method to check that it prints the same results as before.

The code is shown below. Note that I have **left the for loop commented into the method** for purposes of **comparison**:

```
public void printHourlyCounts()
{
    /**
     * for(int hour = 0; hour < hourCounts.length; hour++) {
     *    System.out.println(hour + ": " + hourCounts[hour]);
     * }
     */

    System.out.println("Hr: Count");
    int hour = 0;
    while(hour < hourCounts.length){
        System.out.println(hour + ": " + hourCounts[hour]);
        hour++;
    }
}
```

[5] Barnes & Kolling, Objects First With BlueJ, p101

Ex 4.27

Rewrite the following method from the **Notebook** class in the **notebook2** project so that it uses a for loop rather than a while loop:

(As before, I have left the old loop commented into the body of the method)

```
public void listNotes()
{
    /** int index = 0;
     *  while(index < notes.size()) {
     *      System.out.println(notes.get(index));
     *      index++;
     *  }
     */

    for(int index = 0; index < notes.size(); index++){
        System.out.println(notes.get(index));
    }
}
```

Ex 4.28

Complete the **numberOfAccesses** method, below, to count the total number of accesses recorded in the log file. Complete it by using a loop to iterate over **hourCounts**:

(Note that I have made use of **System.out.println** so that the exact actions of the loop can be easily viewed.)

```
public int numberOfAccesses()
{
    int total = 0;
    // Add the value in each element of hourCounts to total.

    for(int index = 0; index < hourCounts.length; index++){
        total += hourCounts[index];
        System.out.println("(" + hourCounts[index]
                           + ") in hour: " + (index + 1)
                           + ". Total: " + total);
    }
    return total;
}
```

The printout in the Terminal Window is as follows:

(149) in hour: 1. Total: 149
(149) in hour: 2. Total: 298
(148) in hour: 3. Total: 446
(109) in hour: 4. Total: 555
(92) in hour: 5. Total: 647
(177) in hour: 6. Total: 824
(185) in hour: 7. Total: 1009
(142) in hour: 8. Total: 1151
(91) in hour: 9. Total: 1242
(85) in hour: 10. Total: 1327
(227) in hour: 11. Total: 1554
(142) in hour: 12. Total: 1696
(114) in hour: 13. Total: 1810
(164) in hour: 14. Total: 1974
(227) in hour: 15. Total: 2201
(185) in hour: 16. Total: 2386
(167) in hour: 17. Total: 2553
(198) in hour: 18. Total: 2751
(237) in hour: 19. Total: 2988
(172) in hour: 20. Total: 3160
(142) in hour: 21. Total: 3302
(113) in hour: 22. Total: 3415
(168) in hour: 23. Total: 3583
(166) in hour: 24. Total: 3749

(note that the hours run from 1 to 24 because of the addition of 1 to index)

Total accesses : 3749

Ex 4.29

Add your **numberOfAccesses** method to the **LogAnalyzer** class and check that it gives the correct result. Hint: You can simplify your checking by having the analyzer read log files containing just a few lines of data. That way you will find it easier to determine whether or not your method gives the correct answer. The **LogfileReader** class has a constructor with the following signature to read from a particular file:

```
/**
 * Create a LogfileReader that will supply data
 * from a particular log file.
 * @param filename The file of log data.
 */

 public LogfileReader(String filename)
```

Trust me when I tell you that the answer that you should get when using the original file is **3749**!

Ex 4.30

Add a method **busiestHour** to **LogAnalyzer** that returns the busiest hour. You can do this by looking through the **hourCounts** array to find the element with the biggest count. Hint: Do you need to check every element to see if you have found the busiest hour? If so, use a **for** loop.

The code for the **busiestHour** method is as shown below:

You need to **assume** that the **first hour is the busiest** hour until, as a result of a comparison, you find that **another hour is, in fact, busier**.

When you do **find a busier hour**, then **this becomes your new busiest hour and so on**.

This should be easy to test as **the busiest time is 237** in **hour 18** on the **0 to 23 index** (**19:00**, or **7pm**)

```
public int busiestHour()
{
    int busiestHour = 0;
    for(int index = 0; index < hourCounts.length; index++){
        if(hourCounts[index] > hourCounts[busiestHour]){
            System.out.println("Previous busiest hour: "
                               + hourCounts[busiestHour]
                               + ", new busiest hour: "
                               + hourCounts[index]);
            busiestHour = index;
        }
    }
    return busiestHour;
}
```

The local variable **busiestHour** is set to **0**, indicating that we are assuming the first hour, hour **0**, is the busiest.

The **for** loop goes through the entire array comparing the values held in each element of the array.

When a **larger value is found**, the **busiestHour** variable is changed to the **index position** of the **larger value**.

The result when you test this method is as follows:

```
Previous busiest hour: 149, new busiest hour: 177
Previous busiest hour: 177, new busiest hour: 185
Previous busiest hour: 185, new busiest hour: 227
Previous busiest hour: 227, new busiest hour: 237
```

The result that is returned... 18...
(would be hour 19 if it were 1 to 24)

Ex 4.31

Add a method **quietestHour** to **LogAnalyzer** that returns the number of the least busy hour. Note: This sounds almost identical to the previous exercise, but there is a small trap for the unwary here. Be sure to check your method with some data in which every hour has a non-zero count.

Note that I have again constructed this method in such a way as to print to the terminal window so that it is easy to see what is happening.

The 'trap' is that you must remember to test for **less than**, as opposed to **greater than** in the previous method.

< instead of >

```
public int quietestHour()
{
    int quietestHour = 0;
    for(int index = 0; index < hourCounts.length; index++){
        if(hourCounts[index] < hourCounts[quietestHour]){
            System.out.println("Previous quietest hour: "
                               + hourCounts[quietestHour]
                               + ", new quietest hour: "
                               + hourCounts[index]);
            quietestHour = index;
        }
    }
    return quietestHour;
}
```

The result when you test this method is as follows:

```
Previous quietest hour: 149, new quietest hour: 148
Previous quietest hour: 148, new quietest hour: 109
Previous quietest hour: 109, new quietest hour: 92
Previous quietest hour: 92, new quietest hour: 91
Previous quietest hour: 91, new quietest hour: 85

The result that is returned... 9...
```

(would be hour 10 if it were 1 to 24)

Ex 4.32

Which hour is returned by your **busiestHour** method if more than one hour has the biggest count?

With the suggested code, both the **busiestHour** and the **quietestHour** methods will return the **first hour** that they come across in the event that there are **two or more hours** with the **same (highest or lowest) value**.

Why?

Because the methods check to see if the value is...

greater than for the **busiest**

...or...

less than for the **quietest**

If we were to **amend the code** to check for **greater than <u>or equal to</u>**, or **less than <u>or equal to</u>**, then we would end up with the **last (highest or lowest) value** as the equal part would result in **that particular index value being remembered**.

Ex 4.33

Add a method to **LogAnalyzer** that finds which **two-hour period** is the **busiest**. Return **the value of the first hour of this period**.

The code is as follows:

```
public int busiestTwoHourPeriod()
{
  int busiestHours = 0;
  int totalHours = 0;
  int startingAt = 0;
  for(int index = 0; index < hourCounts.length - 1; index++){
      totalHours = hourCounts[index] + hourCounts[index+1];
      System.out.println("Hour: " + index + ", Hour: " + (index + 1)
                          + ", Total: " + totalHours);
      if(busiestHours < totalHours){
          busiestHours = totalHours;
          startingAt = index;
          System.out.println("New busiest total: " + busiestHours);
      }
  }
  return hourCounts[startingAt];
}
```

The method returns **198** from index position **17** in the array. Together with the value held in index position **18** of the array, **237**, the **combined total** for these two hours is **435**, the **largest combination of two concurrent hours possible**.

Note that the suggested code lets you follow the **'action'** in the method:

```
Hour: 0, Hour: 1, Total: 298 New busiest total: 298
Hour: 1, Hour: 2, Total: 297
Hour: 2, Hour: 3, Total: 257
Hour: 3, Hour: 4, Total: 201
Hour: 4, Hour: 5, Total: 269
Hour: 5, Hour: 6, Total: 362 New busiest total: 362
Hour: 6, Hour: 7, Total: 327
Hour: 7, Hour: 8, Total: 233
Hour: 8, Hour: 9, Total: 176
Hour: 9, Hour: 10,Total: 312
Hour: 10,Hour: 11,Total: 369 New busiest total: 369
Hour: 11,Hour: 12,Total: 256
Hour: 12,Hour: 13,Total: 278
Hour: 13,Hour: 14,Total: 391 New busiest total: 391
Hour: 14,Hour: 15,Total: 412 New busiest total: 412
Hour: 15,Hour: 16,Total: 352
Hour: 16,Hour: 17,Total: 365
Hour: 17,Hour: 18,Total: 435 New busiest total: 435
Hour: 18,Hour: 19,Total: 409
Hour: 19,Hour: 20,Total: 314
Hour: 20,Hour: 21,Total: 255
Hour: 21,Hour: 22,Total: 281
Hour: 22,Hour: 23,Total: 334
```

Ex 4.34

Challenge Exercise: Save the **weblog-analyzer** project under a different name, so that you can develop a new version that performs a more extensive analysis of the available data. For instance, it would be useful to know which days tend to be quieter than others - are there any seven-day cyclical patterns, for instance? In order to perform analysis of daily, monthly, or yearly data you will need to make some changes to the **LogEntry** class. This already stores all the values from a single log line, but only the hour and minute values are available via accessors. Add further methods that make the remaining fields available in a similar way. Then add a range of additional analysis methods to the analyzer.

Ex 4.35

Challenge Exercise: If you have completed the previous exercise, you could extend the log file format with additional numerical fields. For instance, servers commonly store a numerical code that indicates whether an access was successful or not: the value 200 stands for a successful access, 403 means that access to the document was forbidden, and 404 means that the document could not be found. Have the analyzer provide information on the number of successful and unsuccessful accesses. This exercise is likely to be very challenging, as it will require you to make changes to every class in this project.

Challenge Exercises **4.34** and **4.35** No Solutions.

Chapter 4 - Additional Exercises

Ex 4.36

In the **lab-classes** project that we have discussed in previous chapters, the **LabClass** includes a **students** field to maintain a collection of Student objects. Read through the **LabClass** class in order to reinforce some of the concepts we have discussed in this chapter.

The code for the **LabClass** is contained below:

```
import java.util.*;

public class LabClass
{
  private String instructor;
  private String room;
  private String timeAndDay;
  private List students;
  private int capacity;
```

The above code defines the **instance variables** for the **LabClass**. Note that, as well as an **int** and three **Strings**, there is also a **students List** defined. Note also the **import statement**, importing the `java.util` package.

```
 /**
   * Create a LabClass with a maximum number of enrolments.
   * All other details
   * are set to default values.
   */
  public LabClass(int maxNumberOfStudents)
  {
      instructor = "unknown";
      room = "unknown";
      timeAndDay = "unknown";
      students = new ArrayList();
      capacity = maxNumberOfStudents;
  }
```

The **constructor** method for **LabClass**. Note that the **students** reference is declared as being an **ArrayList**.

```
/**
 * Add a student to this LabClass.
 */
public void enrolStudent(Student newStudent)
{
    if(students.size() == capacity) {
        System.out.println("The class is full, you cannot enrol.");
    }
    else {
        students.add(newStudent);
    }
}
```

The **enrolStudent** method. Note that this method expects a reference to a Student object to be passed as a parameter.

Note that one of the instance variables defined relates to the **capacity** of the **LabClass**. A check is performed to see if a call to the size method of the students ArrayList returns a size that equals the capacity. If it does, then no new students are added.

If, however, the **LabClass** is not yet at capacity then the add method is called passing the reference to the Student object as its parameter.

```
/**
* Return the number of students currently
* enrolled in this LabClass.
*/
public int numberOfStudents()
{
    return students.size();
}
```

The **numberOfStudents** method simply returns the number of students in the **students ArrayList**.

```
/**
 * Set the room number for this LabClass.
 */
public void setRoom(String roomNumber)
{
    room = roomNumber;
}
```

The **setRoom** method simply assigns whatever String **roomNumber** is to the instance variable **room**.

```
/**
 * Set the time for this LabClass.
 * The parameter should define the day
 * and the time of day, such as "Friday, 10am".
 */
public void setTime(String timeAndDayString)
{
    timeAndDay = timeAndDayString;
}
```

The **setTime** method simply assigns whatever String **timeAndDayString** is to the instance variable **timeAndDay**.

```
/**
 * Set the name of the instructor for this LabClass.
 */
public void setInstructor(String instructorName)
{
    instructor = instructorName;
}
```

The **setInstructor** method simply assigns whatever String **instructorName** is to the instance variable **instructor**.

```
/**
 * Print out a class list with other LabClass
 * details to the standard terminal.
 */
public void printList()
{
    System.out.println("Lab class " + timeAndDay);
    System.out.println("Instructor: " + instructor +
                       "   room: " + room);
    System.out.println("Class list:");
    Iterator i = students.iterator();
    while(i.hasNext()) {
        Student student = (Student)i.next();
        student.print();
    }
    System.out.println("Number of students: " +
                       numberOfStudents());
}
```

The **printList** method prints the values held by the instance variables of the **LabClass** to the terminal window. Note the use of an **iterator object** to work through the **students ArrayList**.

Note also the use of **casting** - to a **Student object**.

```
}
```

Finally, the closing **}** finishes the **LabClass** class definition.

Ex 4.37

The **LabClass** class enforces a limit to the **number of students** who may be **enrolled** in a particular tutorial group. In view of this, do you think it would be more appropriate to use a **fixed-size array** rather than a flexible-size collection for the **students** field? Give reasons both for and against the alternatives.

Use of a **fixed-size collection** would be appropriate **if the class size were to remain fixed**.

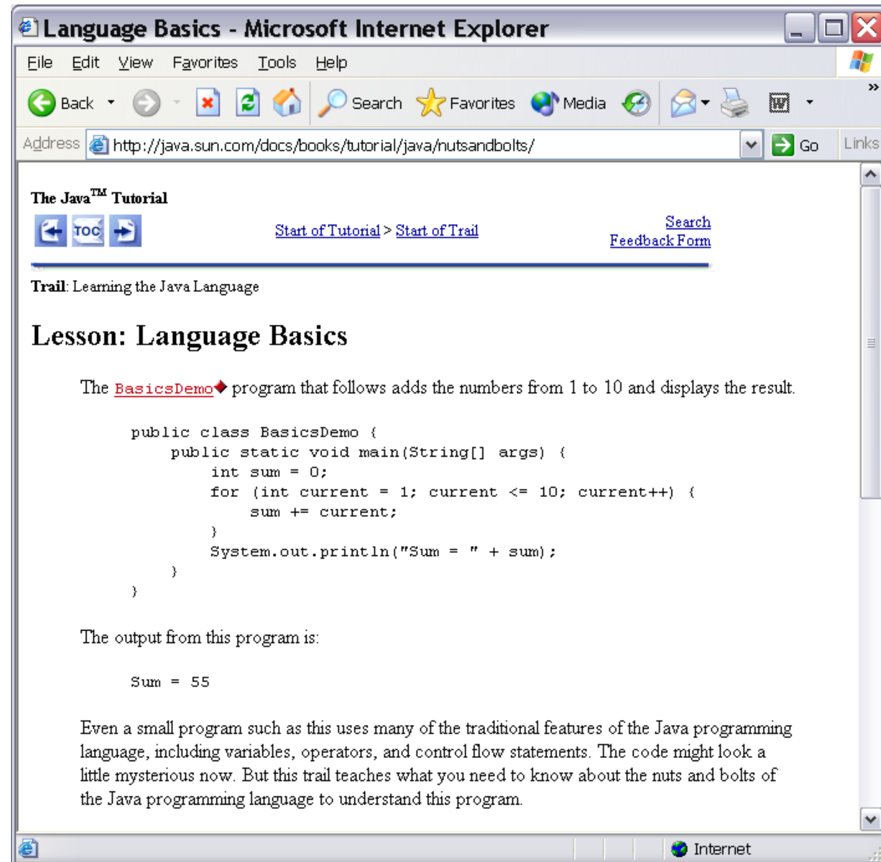
Ex 4.38

Java provides another type of loop: the **do-while** loop. Find out how this loop works and describe it. Write an example of a **do-while** loop that prints out the numbers from **1 to 10**. To find out about this loop, find a description of the Java language (for example at:

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/

in the section **Control Flow Statements**.)

The URL is for a **Java Language Basics** site as illustrated below:



The following information can be obtained about do - while loops:

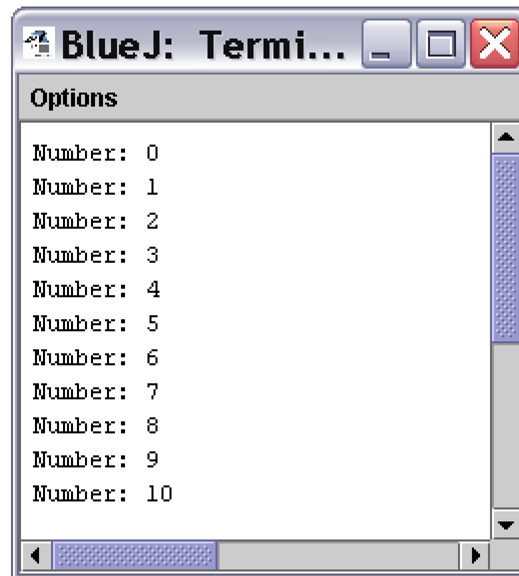
do

> **A Java(TM) programming language keyword used to declare a loop that will iterate a block of statements. The loop`s exit condition can be specified with the "while" keyword.**

An example of a **do-while** loop that **prints the numbers from 1 to 10** is shown below:

```
int number = 0;
  do{
      System.out.println("Number: " + number);
      number++;
  }while(number < 11);
```

The following result is obtained:



Note that a **do while** loop will always perform an action at least once. Once the initial action has been completed, it will then perform a test that determines whether the loop runs again.

Ex 4.39

Rewrite the notebooks **listNotes** method using a **do-while** loop.

The **listNotes** method can be rewritten with a **do-while** loop as follows:

```
public void ListNotes()
{
    int index = 0;
    do{
        System.out.println(notes.get(index));
        index++;
    }while(index < notes.size());
}
```

The problem arises when you have **no notes** stored as the **actions are performed once** before the test on `notes.size()` is performed.

When there are no notes, an **IndexOutOfBoundsException** error is obtained as shown below:

```
public void ListNotes()
{
    int index = 0;
    do{
        System.out.println(notes.get(index));
        index++;
    }while(index < notes.size());
}
```

IndexOutOfBoundsException:
Index: 0, Size: 0 (in java.util.ArrayList)

saved

Ex 4.40 Find out about Javas **switch-case** statement. What is its purpose? How is it used? Write an example. (This is also a control flow statement, so you find information in similar locations as for the do-while loop.)

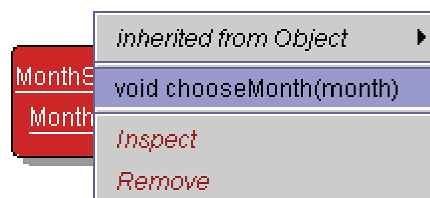
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/switch.html

Use the **switch** statement to conditionally perform statements based on an integer expression.

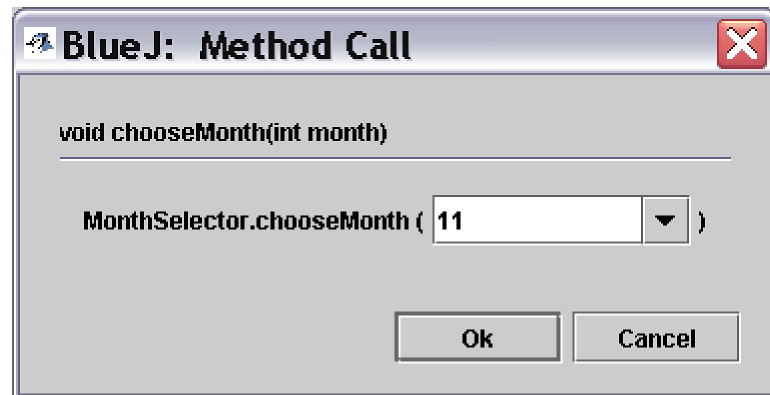
The following exemplifies the use of a switch statement:

```
public class MonthPicker
{
    public MonthPicker()
    {
        // nothing to do in constructor
    }

    public void chooseMonth(int month)
    {
        int monthSelected = month;
        String selectedMonth="";
        switch (month) {
            case 1:  selectedMonth = "January"; break;
            case 2:  selectedMonth = "February"; break;
            case 3:  selectedMonth = "March"; break;
            case 4:  selectedMonth = "April"; break;
            case 5:  selectedMonth = "May"; break;
            case 6:  selectedMonth = "June"; break;
            case 7:  selectedMonth = "July"; break;
            case 8:  selectedMonth = "August"; break;
            case 9:  selectedMonth = "September"; break;
            case 10: selectedMonth = "October"; break;
            case 11: selectedMonth = "November"; break;
            case 12: selectedMonth = "December"; break;
        }
        System.out.println("You selected " + selectedMonth);
    }
}
```

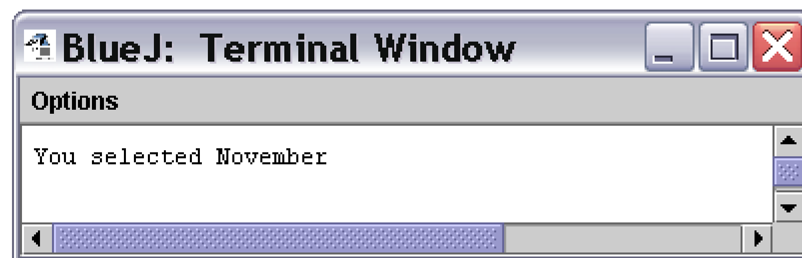
I created a test class called **monthPicker** and then created an instance of a **monthPicker** object. I then invoked the **chooseMonth** method:



Entering an appropiate value:

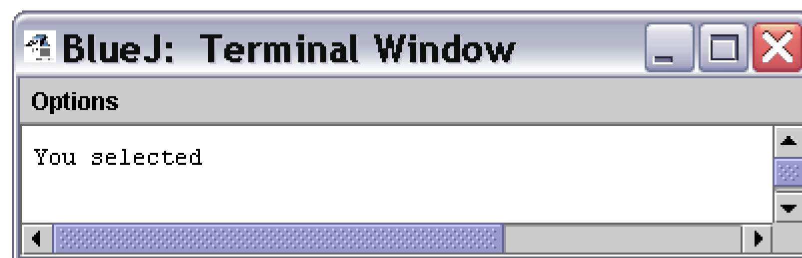


Produced the following result:



The switch statement executes the appropriate case statement for the integer value that was entered - on this occasion, 11 results in the **selectedMonth** variable being set to **November** and this is then used outwith the switch statement when we display the variable.

Invoking the **chooseMonth** method again, this time passing a different parameter such as 14 resulted in a totally different result:



No value has ever been assigned to the **selectedMonth** String and, thus, the original, blank String displays.

This oversight can be easily remedied with the provision of a catch-all clause to our switch statement.

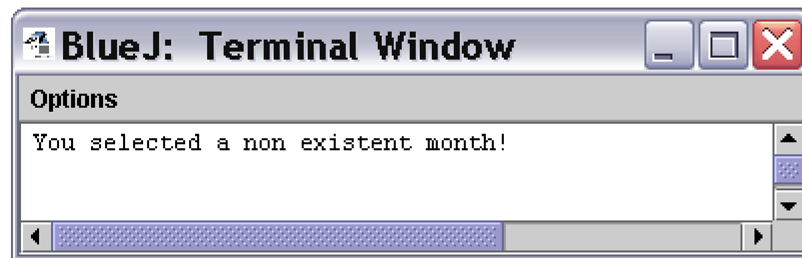
The addition of **one line of code** to the end of our switch statement provides our catch-all clause. This can be seen below:

```
            case 11: selectedMonth = "November"; break;
            case 12: selectedMonth = "December"; break;
            default: selectedMonth = "a non existent month!"; break;
        }
        System.out.println("You selected " + selectedMonth);
    }
```

saved

```
default: selectedMonth = "a non existent month!"; break;
```

Entering an inapropriate month number, such as 14, now results in an altogether different result:



if or switch?

if or **switch**?

You may find that use of switch aids readability of code. Note however, that switch is not without limitations:

"An `if` statement can be used to make decisions based on ranges of values or conditions, whereas a `switch` statement can make decisions based only on a single integer value. Also, the value provided to each `case` statement must be unique."[6]

break;

Note the use of the **break;** statements.

This is used so that, when the correct action has been performed, the **break** statement **terminates** the **switch** statement.

Without the break statements, control will flow sequentially through subsequent case statements. However, the following example does demonstrate that, even this may have its uses:

The following example makes use of case statements with the break purposefully missing. Why? Because, depending upon the option selected, you do want the **same outcome** - if the user selects any of the months with **31 days** then you want the same action to be performed - the value of the **numDays** variable should be set to **31**.

---

[6] http://java.sun.com/docs/books/tutorial/java/nutsandbolts/switch.html

```
public class MonthPicker
{
  public MonthPicker()
  {
      // nothing to do in constructor
  }

  public void chooseMonth(int monthSelected)
  {
     int month = monthSelected;
     int year = 2000;
     int numDays = 0;

     switch (month) {
       case 1:
       case 3:
       case 5:
       case 7:
       case 8:
       case 10:
       case 12:
         numDays = 31;
         break;
       case 4:
       case 6:
       case 9:
       case 11:
         numDays = 30;
         break;
       case 2:
         if(((year%4 == 0) && !(year%100 == 0)) || (year%400 == 0) )
             numDays = 29;
         else
             numDays = 28;
         break;
     }
     System.out.println("Number of Days in month "
                        + month + " is " + numDays);
    }
}
```