Chapter 3 - Project LabClass

Chapter 3

Book exercises:

Ex 3.1

Think again about the **lab-classes** project that we discussed in Chapter 1 and Chapter 2. Imagine we create a **LabClass** object and three **Student** objects. We can then enroll all three students in that lab. Try to draw a **class diagram** and an **object diagram** for that situation. Identify and explain the differences between them.

Class Diagram

Remember that a **class diagram** represents a **static** view - It depicts what we have at the time of writing the program.

The diagram would show **1 Student class definition** and **1 LabClass class definition**.

Object Diagram

The object diagram, on the other hand, represents a **dynamic** view - It depicts the situation at runtime.

The diagram would show **3 instances of Student objects** and **1 instance of a LabClass**.

Chapter 3 - Project clock-display

Ex 3.2 Perform action as indicated in the text.

Ex 3.3 What happens when the **setValue** method is called with an illegal value? Is this a good solution? Can you think of a better solution?

The code for the **setValue** method is shown below:

```
public void setValue(int replacementValue)
{
    if((replacementValue >= 0) && (replacementValue < limit))
        value = replacementValue;
}
```

If you invoke the **setValue** method with an illegal value then **nothing at all happens**. The check that is performed for a sensible value - i.e. one that is **greater than or equal to 0** and **less than the limit** - ensures that **no changes are made** if an illegal value is passed.

Is this a good solution? Well it certainly stops erroneous data. However, it could be cited as being **inadequate** in one respect - **if** an illegal value is passed then **nothing at all happens**… **no warning**, no **default value** is set, **nothing at all**!

Thus, there is room for improvement with the provision of an **else** within the code to cope with illegal values.

Ex 3.4 What would happen if you replaced the '**>=**' operator in the test with '**>**', so that it read:

if((replacementValue > 0) && (replacementValue < limit))

This would mean that **0** would **not be deemed as acceptable**. 1 would be the first acceptable value.

Ex 3.5 What would happen if you replaced the '**&&**' operator in the test with '**||**', so that it read:

if((replacementValue >= 0) || (replacementValue < limit))

Remember that the **&&** operator checks to see that **both conditions are true** i.e. in this case, that the **replacementValue** was greater than or equal to 0 **and** that the **replacementValue** was less than the **limit**.

If we change the operator to **|| or**, then **only one condition need be true** for the consequent to be performed. Obviously, this would not be desirable for the operation of our clock, as it would 'break' the sensible behaviour that the current code enforces.

The result would be that the **condition would always evaluate to true** if the **limit is larger than or equal to 0**.

Ex 3.6

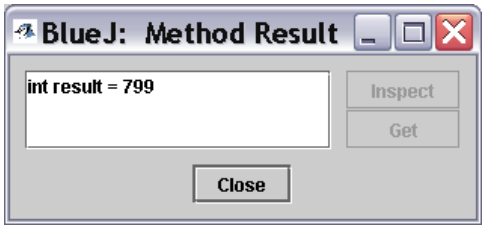
Does the **getDisplayValue** method work correctly in all circumstances? What assumptions are made within it? What happens if you create a **NumberDisplay** with limit 800, for instance?

The code for the **getDisplayValue** method is shown below:

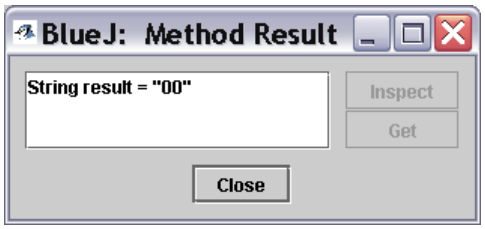
```
public String getDisplayValue()
{
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
```

The assumption with the **getDisplayValue** method is that the value will always be a **two-digit value** and it is therefore coded in such a way as to only ever **add one leading zero** if and when required.

If you create a **NumberDisplay** object with a limit of **800** you will find that it will enable you to set **value** to as high as **799**. You can invoke the **getValue** method and the following will be returned:



However, as soon as value 'ticks over', hitting the **800** limit, it will then go on to **00** as shown below:



Ex 3.7

Is there any difference in the result of writing…

return value + " ";

…rather than…

return " " + value;

…in the **getDisplayValue** method?

There is **no difference**. Had such a change been made where the leading 0 is added then this would have made an obvious difference. However, regardless of whether **""** is added to the **front** or the **back** of the **value**, **nothing is actually added**.

Ex 3.8 Explain the **modulo** operator. You may need to consult more resources (online Java language resources, other Java books, etc) to find out the details.

An exact definition can be found online at the following URL:

http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#24956

Ex 3.9 What is the result of the expression **(8 % 3)** ?

The result is: **2**

3 * 2 = 6 remainder 2.

Ex 3.10 What are all possible results of the expression **(n % 5)**, where **n** is an integer variable?

-4, -3, -2, -1, 0, 1, 2, 3, 4

Ex 3.11 What are all possible results of the expression **(n % m)**, where **n** and **m** are integer variables?

]-m,m[(the range from -(m-1) to (m-1))

Ex 3.12 Explain in detail how the **increment** method works.

The code for the **increment** method is shown below:

```
public void increment()
{
    value = (value + 1) % limit;
}
```

As long as the **value is smaller than the limit**, it gets **incremented by 1** (the modulo can be ignored).

When value reaches the limit, the modulo operation will result in the value being set to **0**.

So, the increment method increments value by one, until the **limit is reached**, at which point it will **start over at 0**.

The values alter as shown below. Note that the middle, bracketed number should indicate to you whether it is **minutes** or **hours** that are being incremented. Note also that I started the clock at 11:58 for the sake of demonstrating the **increment** method.

min	**58**	**(60)**	**59**
min	**59**	**(60)**	**0**
hours	**11**	**(24)**	**12**
min	**0**	**(60)**	**1**
min	**1**	**(60)**	**2**
	etc	**etc**	**etc**

Ex 3.13 Rewrite the **increment** method **without the modulo operator**, using an **if statement**. Which solution is better?

```
public void increment()
{
    value = value + 1;
    if(value >= limit)
        value = 0;
}
```

Which solution is better? You may find the second of the suggested solutions easier to follow. The first solution is certainly more concise.

Ex 3.14 Using the **clock-display** project in BlueJ, test the **numberDisplay** class by creating a few **numberDisplay** objects and calling their methods.

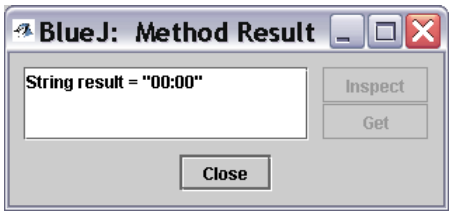
Perform actions as requested.

Ex 3.15 Create a **ClockDisplay** object by selecting the following constructor:

new ClockDisplay( )

Call its **getTime** method to find out the initial time the clock has been set to. Can you work out why it starts at that particular time?

The initial time is displayed as shown below. As can be seen the time is initialized to **00:00**.



Creating a **ClockDisplay** object using the **no parameter constructor** invokes the code shown below.

As can be seen, two new **NumberDisplay** objects are created.

The **NumberDisplay constructor method** is responsible for setting both of these **NumberDisplay** objects to values of **0**.

```
public ClockDisplay()
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    updateDisplay();
}
```

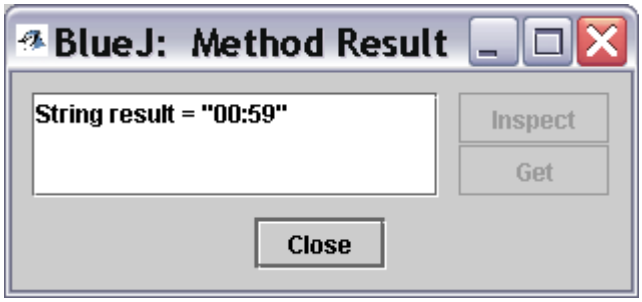
Compare the above constructor method with the following, parameter-based constructor method. Note that the parameter-based constructor invokes the **setTime** method, passing the provided parameters to the method. These parameters are then used to set values for **hour** and **minute**.

```
public ClockDisplay(int hour, int minute)
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}
```

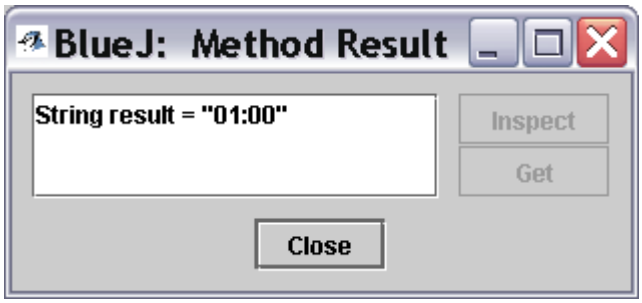
Ex 3.16

How many times would you need to call the **tick** method on a newly created **ClockDisplay** object to make its time reach **01:00**? How else could you make it display that time?

Invoking the **tick** method **59 times** produces the following:



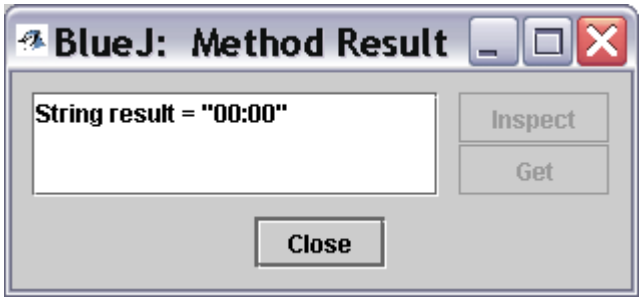
Invoking the **tick** method **60 times** produces the following:



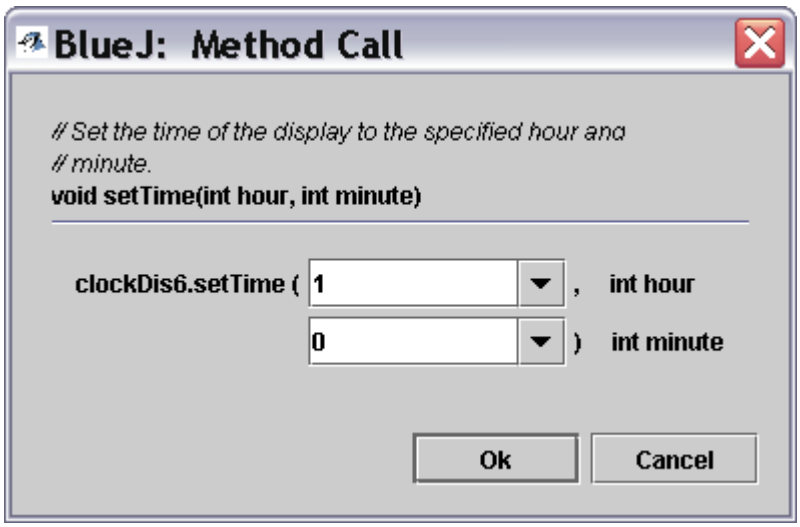
Thus, it takes **60 clicks** to get the desired reading of **01:00**.

You could get to this time a lot easier using **methods available to you**:

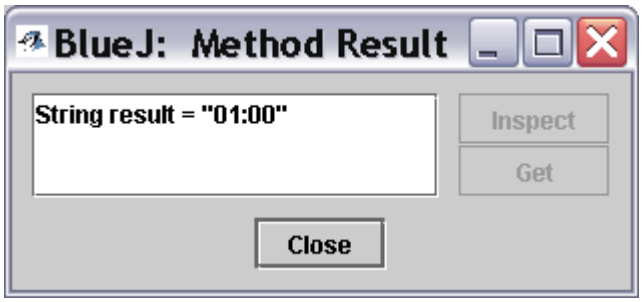
Create a new **ClockDisplay** object and then invoke the **getTime** method - The time should display as shown below:



Invoke the **setTime** method and provide the parameters as shown below - **1** for **hour** and **0** for **minute**:



Invoke the **getTime** method to ensure that the desired time has been set:



Ex 3.17

Look at the second constructor in **ClockDisplays** source code. Explain what it does and how it does it.

As mentioned previously, the second **constructor** in the **ClockDisplay** class expects to be passed information pertaining to the required **hour** and **minute** settings.

The constructor initialises the time to the values passed to the method. It uses the method **setTime** to set the time to the initial value.

Note that the original, no parameter constructor, simply called the **updateDisplay** method.

In the second, parameter-based constructor, there is no need to call **updateDisplay** because this will be done in the **setTime** method.

```
public ClockDisplay(int hour, int minute)
{
    hours = new NumberDisplay(24);
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}
```

Ex 3.18 Identify the similarities and differences between the two constructors. Why is there no call to **updateDisplay** in the second constructor, for instance?

Similarities… The similarities are as follows:

Both constructors create two new **NumberDisplays**.

The differences between the two constructors have been detailed aready. There is no call to the **updateDisplay** method in the second constructor.

Invoking the **setTime** method (code show below) results in the provided parameters being set as the **actual values** for the **hours** and **minutes**. At **this** point, having made the appropriate changes, the **updateDisplay** method is **then** invoked.

The first constructor calls **updateDisplay** and the second calls **setTime(hour, minute)**. In the second constructor, there is no call to **updateDisplay** because this will be done in the method **setTime**.

```
public void setTime(int hour, int minute)
{
    hours.setValue(hour);
    minutes.setValue(minute);
    updateDisplay();
}
```

Ex 3.19 **Challenge Exercise**: Change the clock from a 24-hour clock to a 12-hour clock. Be careful, this is not as easy as it might at first seem. In a 12-hour clock the hours after midnight and after noon are **not shown as 00:30**, but as **12:30**. Thus **the minute display shows values from 0 to 59**, while the **hour display shows values from 1 to 12**!

Ex 3.20 There are (at least) two ways in which you can make a 12-hour clock. One possibility is to just store hour values from 1 to 12. On the other hand, you can just leave the clock to work internally as a 24-hour clock, but change the display string of the clock to show 4:23 or 4.23pm when the internal value is 16:23. Implement both versions. Which option is easier? Which is better? Why?

```
public void updateDisplay()
{
    int hour = hours.getValue();
    String suffix = "am";

    if(hour >= 12){
        hour = hour - 12;
        suffix = "pm";
    }
    if(hour == 0){
        hour = 12;
    }
    displayString = hour + "." + minutes.getDisplayValue() + suffix;
}
```

Note the use of 2 **local variables**, **hour** and **suffix**.

The **hour** variable, of type **int**, is assigned the value of whatever is returned by the **getValue** method when invoked on the **hours** object.

The **suffix** variable, of type **String**, is assigned the value **am** which will only be changed if we have a time that requires the suffix to be **pm**.

The **hour** variable is then tested - If it is **greater than 12** we **subtract 12** from it to give us a **12-hour time reading** instead of a **24-hour** one i.e. If the value of hour is **16** we subtract **12** and this correctly gives us the **12-hour clock** time reading of **4**. Because the time was greater than 12 we need to alter the value of **suffix** to **pm**.

If the value of **hour** is **0**, then this indicates that, in 24-hour clock time, the time is around **midnight**. Thus, we again alter **hour** so that it abides by conventions of the **12-hour clock** i.e. displays **12** instead of **0**.

Finally, we update the **ClockDisplay** class instance variable **displayString**. We make use of our **hour** and then invoke the **getDisplayValue** on the minutes **NumberDisplay** object to give us a correct minutes reading.

Alternatively, we can obtain the same result with the following code:

```
public ClockDisplay()
{
    hours = new NumberDisplay(12); //changed
    minutes = new NumberDisplay(60);
    updateDisplay();
}

public ClockDisplay(int hour, int minute)
{
    hours = new NumberDisplay(12); //changed
    minutes = new NumberDisplay(60);
    setTime(hour, minute);
}


private void updateDisplay()
{
    int hour = hours.getValue();
    if(hour == 0)
        hour = 12;

    displayString = hour + "." + minutes.getDisplayValue();
}
```

Note that both of the **ClockDisplay constructor** methods must be changed to construct **NumberDisplay** objects with **limits** of only **12** hours instead of **24** hours.

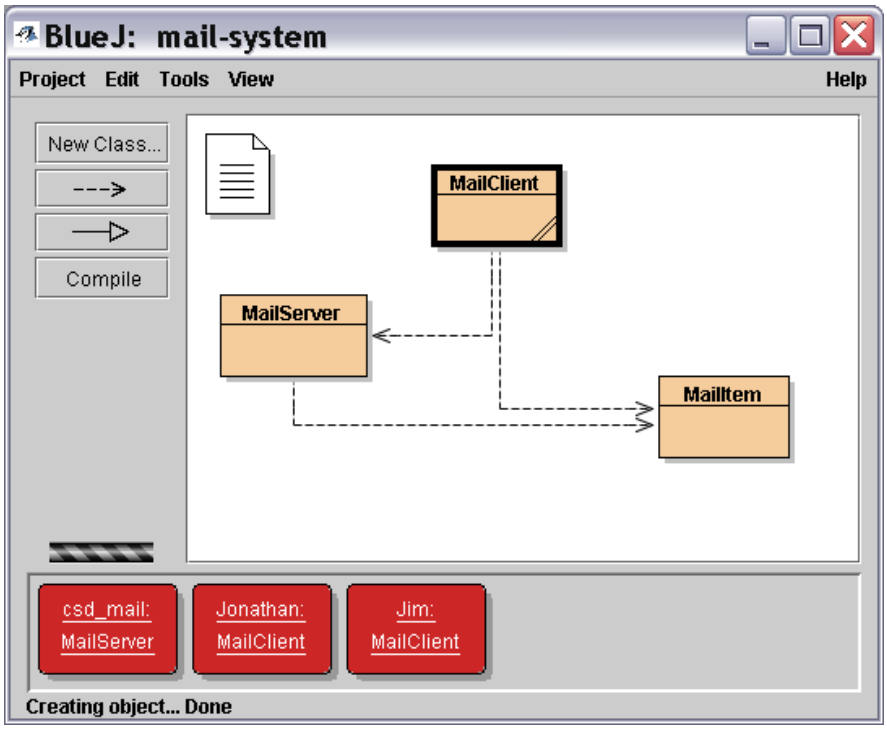
This accordingly simplifies the **updateDisplay** method as the only test that needs to be made is for an **hour** value of **0**, in which case, the value of **hour** is then changed to **12**.

Ex 3.21

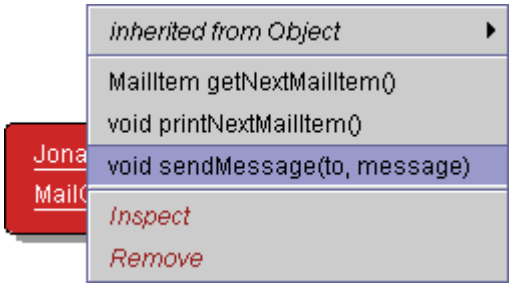
Open the **mail-system** project, which you can find in the books support material. Create a **MailServer** object. Create two **MailClient** objects. When doing this, you need to supply the **MailServer** instance, which you just created, as a parameter. You also need to specify a username for the mail client. (a mail client is a program to read and write email. Every instance you create represents an email program for a different user.)

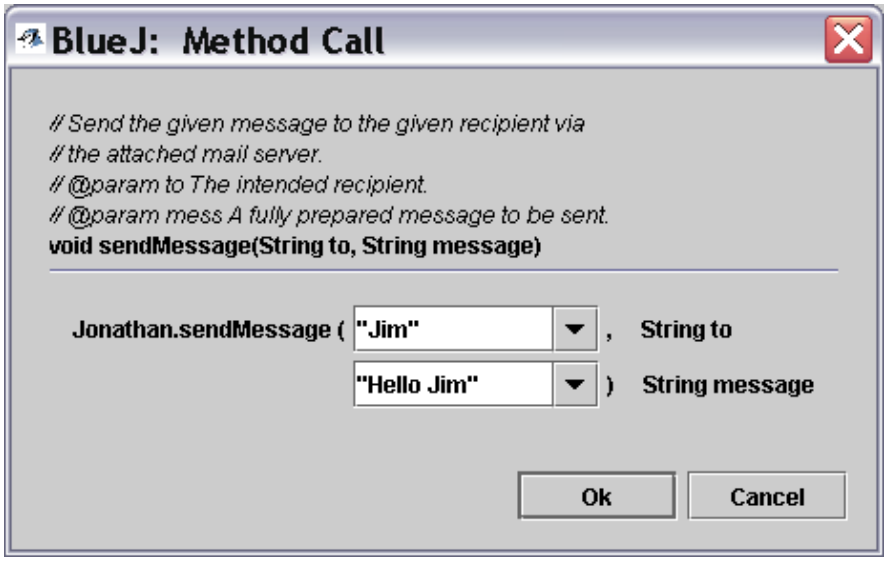
Experiment with the **MailClient** objects. They can be used to send messages from one mail client to another (using the **sendMessage** method) and to receive messages (using the **getNextMailItem** or **printNextMailItem** methods).

As can be seen from the following diagram, I created a single MailServer object, **csd_mail**, and two **MailClient** objects, **Jonathan** and **Jim**.

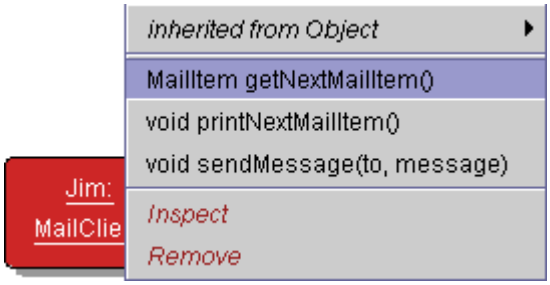


Sending an email from **Jonathan** to **Jim** - Invoke the **sendMessage** method and then specify **to** and the **message**.

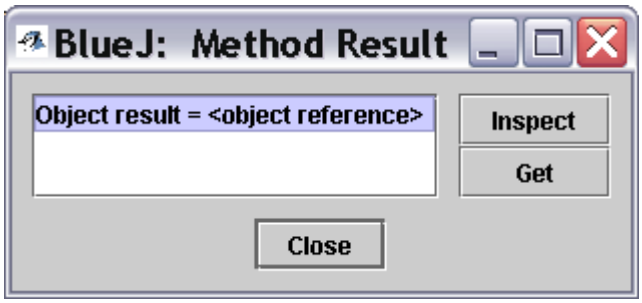




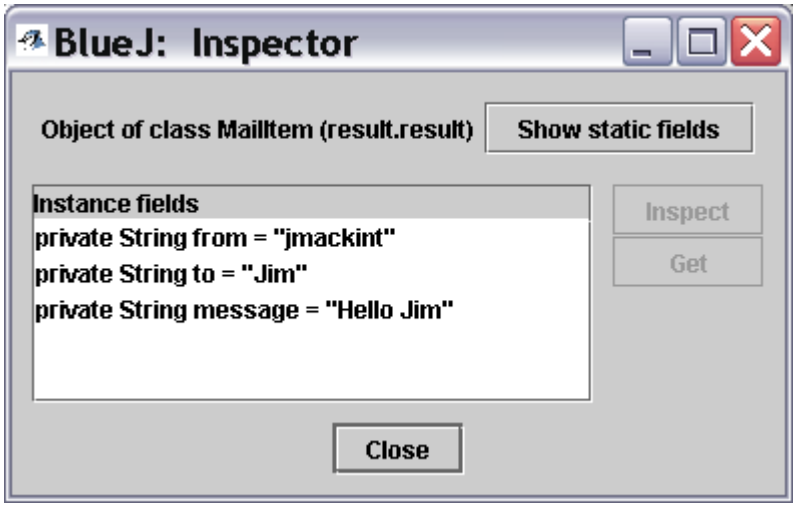
Jim checks his email:



…and sees that there is an **object reference** i.e. there is an actual **MailItem** waiting to be read.

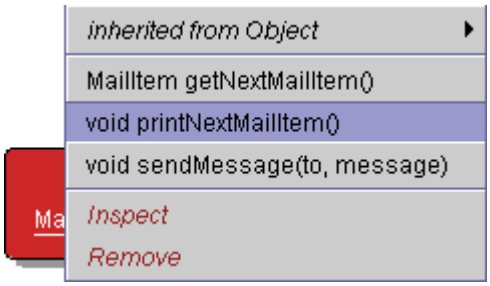


Highlight and then **Inspect**…

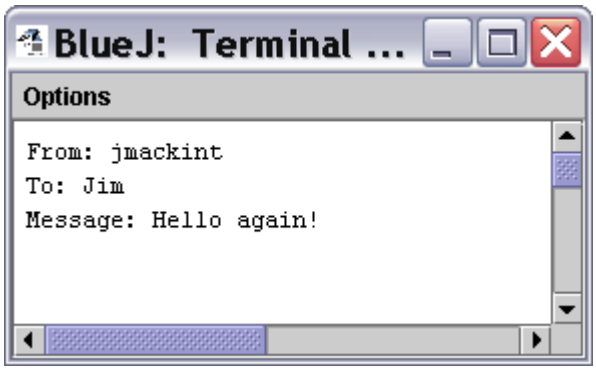


As can be seen - this illustrates the values of **from**, **to** and **message**,just as is the case of a real email.

Alternatively, invoke the **printNextMailItem** method:



The message is displayed in the **Terminal Window**:



Ex 3.22

Draw an **object diagram** of the situation you have after creating a **MailServer** and three **MailClients**. Object diagrams were discussed in section 3.6.

The object diagram should basically show that you have one **MailServer** object and three **MailClient** objects.

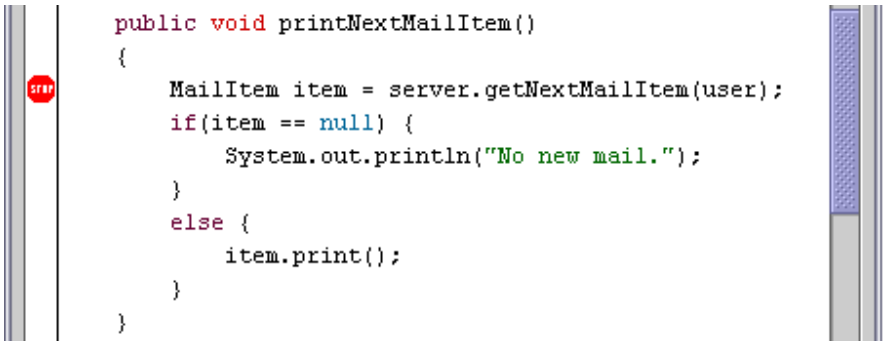
Ex 3.23

Set up a scenario for investigation: Create a mail server, and then create two mail clients for the users 'Sophie' and 'Juan' (You should name the instances Sophie and Juan as well, so that you can better distinguish them on the object bench).

Then use Sophie's **sendMessage** method to send a message to Juan. Do not yet read the message.

Ex 3.24

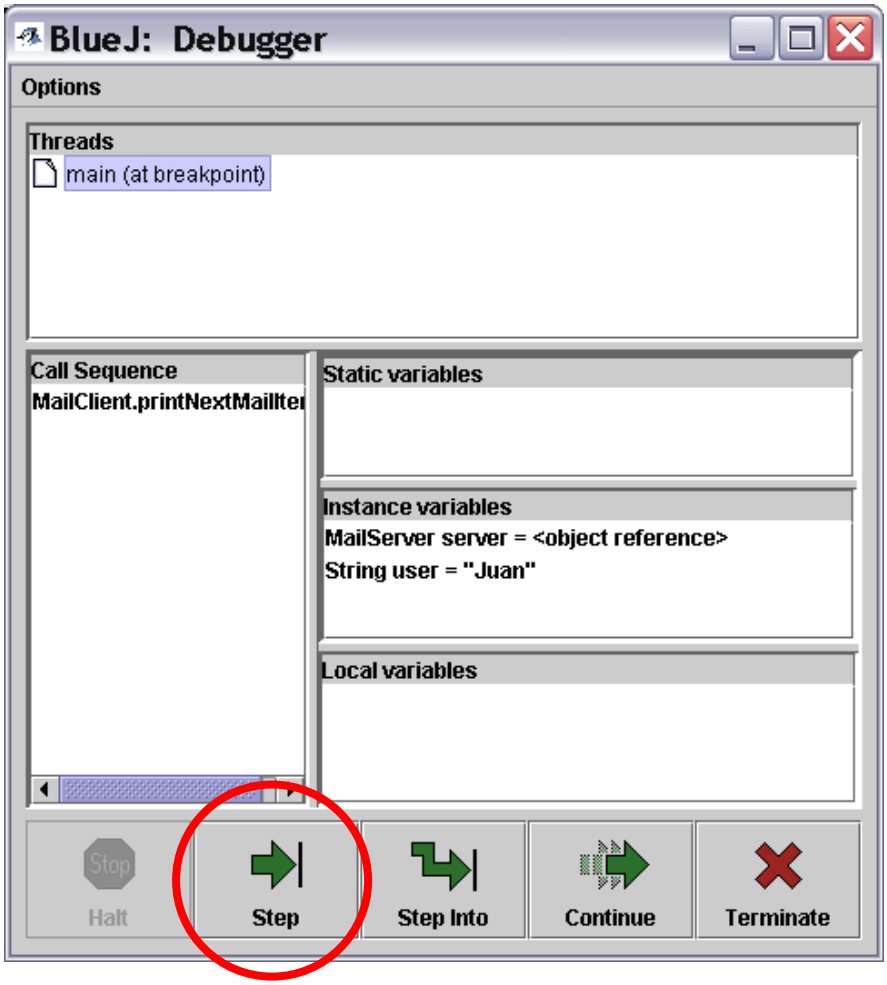
Open the editor for the **MailClient** class and set a breakpoint at the first line of the **printNextMailItem** method, as shown in Figure 3.5.



```
public void printNextMailItem()
{
    MailItem item = server.getNextMailItem(user);
    if(item == null) {
        System.out.println("No new mail.");
    }
    else {
        item.print();
    }
}
```

Ex 3.25

Step one line forward in the execution of the **printNextMailItem** method by clicking the **Step** button.



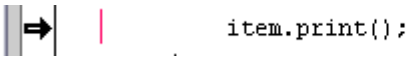
Ex 3.26

Predict which line will be marked as the next line to execute after the next step. Then execute another single step and check your prediction. Were you right or wrong? Explain what happened and why.

The code for the **printNextMailItem** method is show below:

```
public void printNextMailItem()
{
    MailItem item = server.getNextMailItem(user);
    if(item == null) {
        System.out.println("No new mail.");
    }
    else {
        item.print();
    }
}
```

You should have predicted that the next line of code to be performed is the line **`item.print();`** as shown below:



Why? Because you purposefully sent an email, you would expect that an email would exist. Therefore you would expect **item** to point to an actual **MailItem** object and not to **null**.

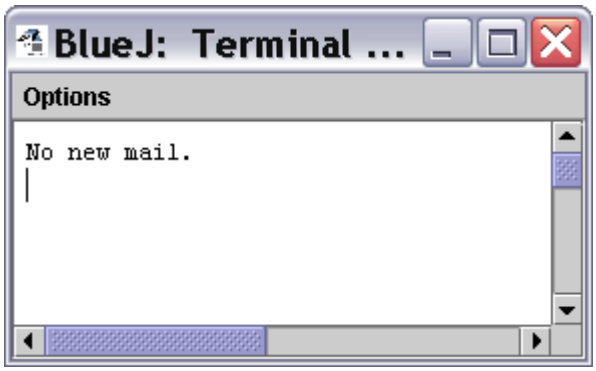
Ex 3.27

Call the same method (**printNextMailItem**) again. Step through the method again, as before. What do you observe? Explain why this is.

Next time round, because there is no email, **item** does **not** point to a **MailItem** object. Thus, we expect the following line to be performed:

```
System.out.println("No new mail.");
```

…with the following result…



Ex 3.28

Set up the same test situation as we did before. That is, send a message from Sophie to Juan. Then invoke the **printNextMailItem** message of Juan's mail client again. Step forward as before. This time, when you reach the line:

```
item.print( )
```

use the **Step Into** command instead of the **Step** command. Make sure you can see the text terminal window as you step forward. What do you observe? Explain what you see.

You are taken **step by step** through the actual **print** method in the **MailItem** class. When you reach the end of this method, then you will see that the arrow indicating your position in the code returns to the **MailClient** class. This helps us to **see exactly how the classes interact with each other**.

Ex 3.29

Set a breakpoint in the first line of the **sendMessage** method in the **MailClient** class. Then invoke this message. Use the **Step Into** function to step into the constructor of the mail item. In the debugger display for the **MailItem** object, you can see the instance variables and local variables that have the same names, as discussed in section 3.12.2. Step further to see the instance variables get initialized.

Ex 3.30

Use a combination of code reading, execution of methods, breakpoints, and single stepping to familiarize yourself with the **MailItem** and **MailClient** classes. Note that we have not discussed enough for you to understand the implementation of the **MailServer** class just yet, so you can ignore this for now. (You can, of course, look at it if you feel adventurous, but don't be surprised if you find it slightly baffling…) Explain in writing how the **MailClient** and **MailItem** classes interact. Draw object diagrams as part of your explanations.

The **MailClient** performs as you would expect - it retrieves any email messages for a particular user (**MailItem**) from the **MailServer** and, further, it enables a particular user to **send messages**, again using the **MailServer**.

Ex 3.31 Use the debugger to investigate the **clock-display** project. Set break-points in the **ClockDisplay** constructor and each of the methods, and then single step through them. Does it behave as you expected? Did this give you new insights? If so, what were they?

Ex 3.32 Use the debugger to investigate the **insertMoney** method of the **better-ticket-machine** project from Chapter 2. Conduct tests that cause both branches of the if statement to be executed.

Ex 3.33 Add a subject line for an email to mail items in the **mail-system** project. Make sure printing messages also prints the subject line. Modify the mail client accordingly.

This involves adding another **instance variable** to the **MailItem** class definition - **subject**, of type **String**.

The **constructor** and **print** methods must then be adapted accordingly.

The **sendMessage** method in the **MailClient** class definition would also need to be modified to take into account the **additional parameter** that is **expected** for a **MailItem** object.