Second Java Tick
Alan Blackwell – February 2004

This builds on instructions on the first Java tick sheet, which tells you how to copy project files from the master course directory and print out results for marking.

Chapter 2 of BlueJ book, exercises 2.1 – 2.28 (**`naive-ticket-machine`** project).
Remember to start by making your own copy of the project files in a terminal window before opening the project in BlueJ.

Create a **`TicketMachine`** object on the object bench and try its methods. The **`getPrice`** method should produce as a return value the ticket price set when this object was created. The **`insertMoney`** method followed by **`getBalance`** should confirm that the machine records the amount inserted. Try inserting different amounts. Insert the exact amount required for a ticket, then call the **`printTicket`** method. A facsimile ticket should be printed in the BlueJ terminal window.

What value is returned if you check the machine's balance after it has printed a ticket? Try inserting different amounts before printing tickets. Do you notice anything odd? What if you insert too much – do you get change? What if you do not insert enough and then try to print a ticket? Explore the ticket machine's behaviour thoroughly.

Open the source of **`TicketMachine`** in the BlueJ editor by double-clicking on the class icon. Make a list of the names of the fields, constructors and methods in the **`TicketMachine`** class. Do you notice any features of the constructor that make it different from all of the methods? What features does the declaration of the price field have in common with the other two fields? In what ways is it different? Compare the **`getBalance`** method with the **`getPrice`** method. What are the differences between them? If a call to **`getPrice`** can be characterised as "What do tickets cost?" how would you characterise a call to **`getBalance`**?

If the name of **`getBalance`** is changed to **`getAmount`**, does the return statement in the body of the method need to be changed, too? Try it out in BlueJ. Define an accessor method, **`getTotal`**, that returns the value of the total field. Try removing the return statement from the body of **`getPrice`**. What error message do you see when you compile the class? Compare the method signatures of **`getPrice`** and **`printTicket`** in **`TicketMachine`**. What is the main difference between them? Do the **`insertMoney`** and **`printTicket`** methods have return statements? Why do you think this might be? Do you notice anything about their headers to suggest an answer?

Create a new ticket machine with a ticket price of your choosing. Before doing anything else, call the **`getBalance`** method on it. Now call the **`insertMoney`** method and give a non-zero positive amount of money as the actual parameter. Now call **`getBalance`** again. The two calls to **`getBalance`** should show different output because the call to **`insertMoney`** had the effect of changing the machine's state via its **`balance`** field. Create another ticket machine and call its **`insertMoney`** method. Check the balance with a call to **`getBalance`**. Now call **`insertMoney`** and **`getBalance`** again to ensure that the new balance is the sum of the two amounts passed to the two calls to **`insertMoney`**.

Add a method called **`prompt`** to the **`TicketMachine`** class. This should have a **`void`** return type and take no parameters. The body of the method should print something like:

```
Please insert the correct amount of money.
```

Add a **`showPrice`** method to the **`TicketMachine`** class. This should have a **`void`** return type and take no parameters. The body of the method should print something like:

```
The price of a ticket is xyz cents.
```

Where ***xyz*** should be replaced by the value held in the **`price`** field when the method is called. Create two ticket machines with different priced tickets. Do calls to their **`showPrice`** methods show the same output or different? How do you explain this effect?

Implement a method, **`empty`**, that simulates the effect of removing all money from the machine. This method should have a **`void`** return type, and its body should simply set the **`total`** field to zero. Does this method need to take any parameters? Test your method by creating a machine, inserting some money, printing some tickets, checking the total and then emptying the machine. Is this method a mutator or an accessor?

Implement a method, **`setPrice`**, that is able to set the **`price`** of tickets to a new value. The new price is passed in as a parameter value to the method. Test your method by creating a machine, showing the price of tickets, changing the price, and then showing the new price. Is this method a mutator?

Chapter 2 of BlueJ book, exercises 2.29 – 2.38 (**`better-ticket-machine`** project).
Close the naïve ticket machine project, then make your own copy of the new project files in a terminal window as usual, before opening it in BlueJ.

Create some **`TicketMachine`** objects in the new project, and experiment with their behaviour. Compare this with those defined in the **`naive-ticket-machine`** project. Combine your experimentation with a browse through the source of the class. You will notice a new language feature in some methods: the conditional statement **`if`**. In particular, check what happens when negative amounts of money are inserted, or an attempt is made to print a ticket before enough money has been inserted.

Give the class a second constructor. The new one should take no parameter and set the price to be a default value of your choosing. Test your implementation by creating machines via the two different constructors. Predict what you think will happen if you change the test in **`insertMoney`** to use the greater-than or equal-to operator:

```
if(amount >= 0)
```

Check your predictions by running some tests. What difference does it make to the behaviour of the method?

In this version of printTicket, we also do something slightly different with the **`total`** and **`balance`** fields. Compare the implementation of the method in the naive ticket machine with this version to see if you can tell what those differences are. Then check your understanding by experimenting in BlueJ.

Why does the following version of refundBalance not give the same results as the one defined in the better ticket machine?

```
public int refundBalance()
    {
        balance = 0;
        return balance;
    }
```

What tests can you run to demonstrate that it does not? What happens if you try to compile the TicketMachine class with the following version of refundBalance:

```
public int refundBalance()
    {
        return balance;
        balance = 0;
    }
```

What do you know about return statements that helps to explain why this version does not compile?

Add a new method, **`emptyMachine`**, that is designed to simulate emptying the machine of money. It should return the value in total and reset total to be zero. Is **`emptyMachine`** an accessor, a mutator, or both?

Rewrite the **`printTicket`** method so that it defines a local variable, **`amountLeftToPay`**, as its first statement. This should be initialised to contain the difference between **`price`** and **`balance`**. Then rewrite the test in the conditional statement to check the value of **`amountLeftToPay`**. If its value is less than or equal to zero, a ticket should be printed, otherwise an error message should be printed stating the amount still required. Test your version to ensure that it behaves in exactly the same way as the original version.

Print out the window showing your test results, using the method described in the first tick instructions.

Chapter 3 of BlueJ book, exercises 3.2 – 3.20 (clock-display project).
Remember to start by making your own copy of the project files in a terminal window before opening BlueJ.

Open the **`clock-display`** example and experiment with it. Create a **`ClockDisplay`** object, then open an inspector window for this object. With the inspector open, call the object's methods. Watch the **`displayString`** field in the inspector. Read the project comment (by double-clicking the text note icon on the main screen) to get more information. Create a new **`NumberDisplay`** object with a rollover limit of your choosing. What happens when the **`setValue`** method is called with an illegal value? Is this a good solution? Can you think of a better solution? Does the **`getDisplayValue`** method work correctly in all circumstances? What assumptions are made within it? Is there any difference in the result of writing

```
    return value + "";
```

rather than

```
    return "" + value;
```

in the **`getDisplayValue`** method? Try it.

Explain the modulo operator. What is the result of the expression (8 % 3)? What are all possible results of the expression (n % 5), where n is an integer variable? Explain in detail how the increment method works. Rewrite the increment method without the modulo operator, using an if statement. Which solution is better?

Test the NumberDisplay class by creating a few NumberDisplay objects and calling their methods. Create a ClockDisplay object by selecting the 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?

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?

Look at the second constructor in ClockDisplay's source code. Explain what it does and how it does it. Identify the similarities and differences between the two constructors. Why is there no call to updateDisplay in the second constructor, for instance?

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!

There are (at least) two ways that 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, and just change the display string of the clock display 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?

Print out the window showing your results, and the editor window showing how you achieved them, to hand in for your tick.

Note that you can capture both windows to a single file using a single "import" command using the following method: rather than clicking once in a single window (as you did for tick number 1), click at the top left of the area you want to print, and drag (while holding the mouse button down) to the bottom right of the area you want. This saves a whole region of the screen rather than a single window. You may want to arrange the two windows near each other first so that the final result looks tidy. You can check the image before printing it by using the following command:

```
        Ghostview tempOut.ps &
```

Remember to delete the file tempOut.ps before you log out.