Chapter 2

Solutions

Note that you should have a greater degree of familiarity with the **BlueJ** environment by this stage and, as such, the provided solutions will be **less comprehensive** than those of the previous chapter.

The expectation is that you have been able to follow through the instructions as provided within the BlueJ text.

Project naïve-ticket-machine

Correct Project

Make sure that you have started with the correct project - **naïve-ticket-machine**. The purpose of the **naïve-ticket-machine** is to demonstrate **shortcomings** in terms of the **modelling** of the project and to build upon these limitations via the **introduction of sensible behaviours** - possible because of the **introduction of more Java code**.

Many Instances

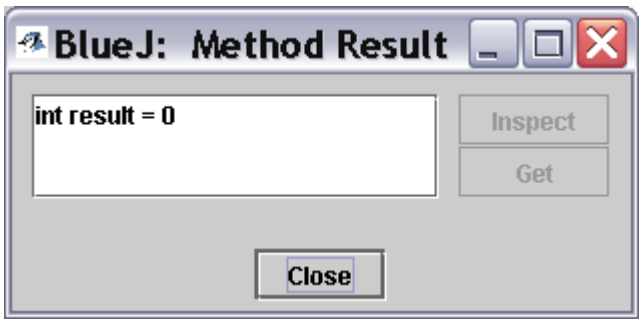
In case you are having problems visualizing ticketMachine objects consider a **multi storey car park**. A **single class definition** will provide the **'blueprint'** for the construction of the **many ticketMachine objects** that will be required i.e. **2 machines per floor, 5 floors**. Each of these instances will be created from the same single class definition.

The Limitations

Having created a ticketMachine object, you should have followed the instructions and, by doing so, encountered some of the **limitations of the project**. Some of these are dealt with below.

Ex 2.2

What value is returned if you check the machine's balance after it has printed a ticket?



Solution

Assuming that you **entered more money than the actual cost of a single ticket**, then you would expect the result to **return something greater than 0**. i.e. you would expect to see the **money left after the ticket had been paid for**, money that would then either be **refunded** or **put towards the cost of another ticket**.

However…

However, as can be seen from the above result, you have been robbed!

The result returned is **0**.

If you insert too much money into the machine, it does not warn you of this and, when you select to print a ticket, you **lose any money** that you have entered **over and above the cost of the ticket**.

You most certainly **do not receive a refund**!

It is, however, possible to **continually print tickets** by calling the **printTicket** method, regardless of how much - or more to the point, **how little** - your current balance is.

i.e. even if you have a balance of 0 it is possible to continually print tickets… maybe this ticket machine is not so bad after all ;O)

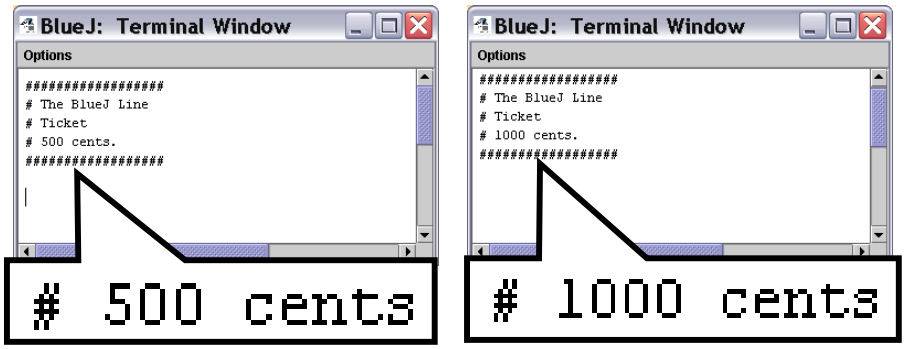
Ex 2.4

Try to obtain a good understanding of a ticket machine's behaviour by interacting with it on the object bench before we start looking at how the **TicketMachine** class is implemented in the next section.

Ex 2.5

When you create a **different ticketMachine** with a **different ticket price**, the behaviours of the ticketMachine object are, as you would expect, the same. When you print a ticket however, you will notice a difference.

If you had **two ticketMachine objects** with tickets prices of **500** and **1000** cents for example. Printing tickets from each object would provide the following result:



Original Ticket **New Ticket**

If you have a look at the code for this method then the reason for the differing amounts should be evident…

printTicket Method

```
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("####################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("####################");
    System.out.println();

    // Update the total collected with the balance.
    total += balance;
    // Clear the balance.
    balance = 0;
}
```

The line that differentiates the tickets is as follows:

```
    System.out.println("# " + price + " cents.");
```

Note the use of **concatenation** - The **actual** value of the **price** attribute will be used in the display.

Thus, we have a **printTicket** method that can print the correct ticket, regardless of the price.

Ex 2.6

Write out what you think the outer layers of the **Student** and **LabClass** classes might look like - do not worry about the inner part.

For **Student**…

```
public class Student
{
    //Inner part of the class omitted
}
```

And for **LabClass**…

```
public class LabClass
{
    //Inner part of the class omitted
}
```

Ex 2.7

Exercise 2.7 reads as follows:

From your earlier experimentation with the ticket machine objects within BlueJ you can probably remember the names of some of the methods - **printTicket**, for instance.

Look at the class definition in Code 2.1 and use this knowledge, along with the additional information about ordering we have given you, to try to make a list of the names of the fields, constructors and methods in the **TicketMachine** class. Hint: There is only **one constructor** in the class.

The fields:

```
Price
balance
total
```

The constructors

```
TicketMachine
```

The methods:

```
TicketMachine - the constructor method
getPrice
getBalance
insertMoney
printTicket
```

Ex 2.8

The constructor method is significantly different from the other methods contained within the class in that the constructor method has **no return type**. This can be seen from the signature of the constructor method shown below:

```
public TicketMachine(int ticketCost)
```

The other methods in the class all have return types, even if the return type is **void** - as in the **insertMoney** and **printTicket** methods. Look at the other methods, **getPrice** and **getBalance**. These have a return type of **int**.

The constructor is different in that it has **no return type**. This is **always** the case with the constructor method.

Note also

Note also that the majority of methods will start in **lowercase,** as is the case with, for example, the **getPrice**, **getBalance** and **printTicket** methods.

However, because the **constructor** must have the **exact same name as the class name** (remember that class names by convention should be capitalised) the constructor method will also begin in **uppercase** i.e. **Student**, **Person**, **TicketMachine** and so forth.

The code for the constructor method is shown below.

Note the capitalisation:

```
public TicketMachine(int ticketCost)
{
    price = ticketCost;
    balance = 0;
    total = 0;
}
```

The method creates a **ticketMachine** object of a set **price** - the value of **ticketCost** that is provided by the user and then assigned to the **price** attribute - and then sets **balance** and **total** to **0**.

As the ticketMachine is created new, there would be **no balance** and **no total** as such - it would be a **brand new**, **empty**, **unused** ticketMachine!

Ex 2.9

Compare the **getBalance** method with the **getPrice** method. What are the differences between them?

The **getBalance** method:

```
public int getBalance()
{
    return balance;
}
```

The **getPrice** method:

```
public int getPrice()
{
    return price;
}
```

The similarities

Both methods have an **int** specified for the **return type**.

The Difference

The difference is **what** is returned.

These are typical **accessor** or **get** methods in that they simply **return the value of the attribute in question**. Therefore, the **getBalance** method will return the value held by the **balance** attribute and the **getPrice** method will return the value held by the **price** attribute.

Ex 2.10 If a call to **getPrice** can be characterized as "What do tickets cost?" how would you characterize a call to **getBalance**?

Solution **"What is the current balance?"**

Ex 2.11 **Exercise 2.11 reads as follows:**

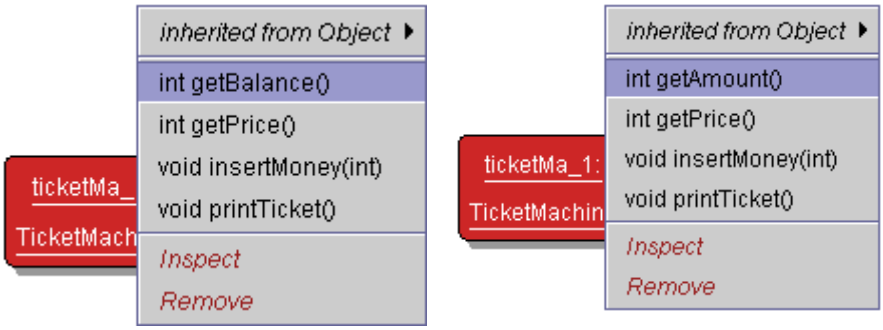
If the name of **getBalance** is changed to **getAmount**, does the return statement in the body of the method need to be changed too? Try it out within BlueJ.

Solution If you made the changes as indicated then the code for the method would appear as shown below:

```
public int getAmount()
{
    return balance;
}
```

To test your changes you would need to then **compile** your code, create an instance of a **TicketMachine** object and test that the **getAmount** method worked in the same way as the **getBalance** method did.

The **'before'** and **'after'** menus, obtained by right-clicking on your object, are as shown below:



Before **After**

You should see that, despite the change in name, all else is exactly as it was before. The call to **getAmount** will return the current value of the **balance** field, just as a call to **getBalance** method did before you made the changes.

So, **no difference**!

Technically however, if the field is called **balance**, as it is in our example, then **getBalance** is the most appropriate method name when creating a method with the sole purpose or returning the value held in that field.

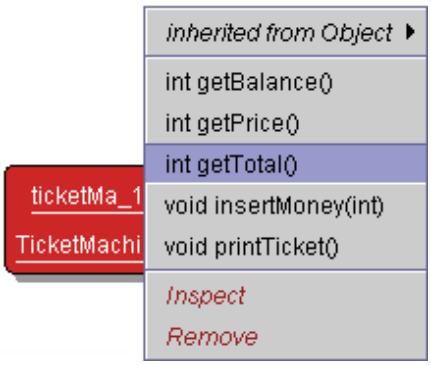
Someone used to programming in Java would likely expect a method called **getAmount** to return the current value of a field named **amount**.

Ex 2.12 Define an **accessor** method, **getTotal**, that returns the value of the **total** field.

The code for a **getTotal** method is as shown below:

```
public int getTotal()
{
    return total;
}
```

To test, compile the class and then create a new **TicketMachine** object.



Note that you will have to **insert money** and **print a ticket** to test the **getTotal** method.

The field **total** is not updated until the **printTicket** method is called.

Look again at the **printTicket** method and see where **total** is updated:

printTicket

```
public void printTicket()
{
    // Simulate the printing of a ticket.
    System.out.println("###################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("###################");
    System.out.println();

    // Update the total collected with the balance.
    total += balance;
    // Clear the balance.
    balance = 0;
}
```

Note that the **total** is altered after the ticket has been printed.

The line…

```
    total += balance;
```

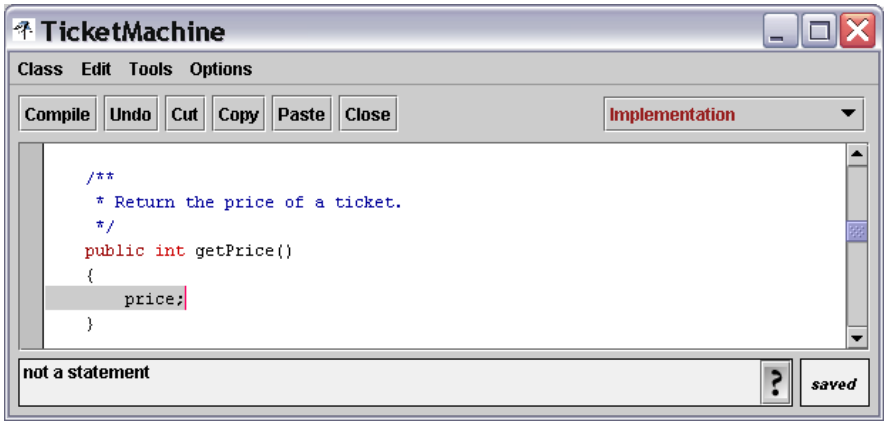
…takes the **original value of total** and adds the value held in **balance** to total. Thus, total takes the value of **what it was before plus** the **new amount** from the balance.

Ex 2.13

Try removing the return statement from the body of **getPrice**. What error message do you see now when you try compiling the class.

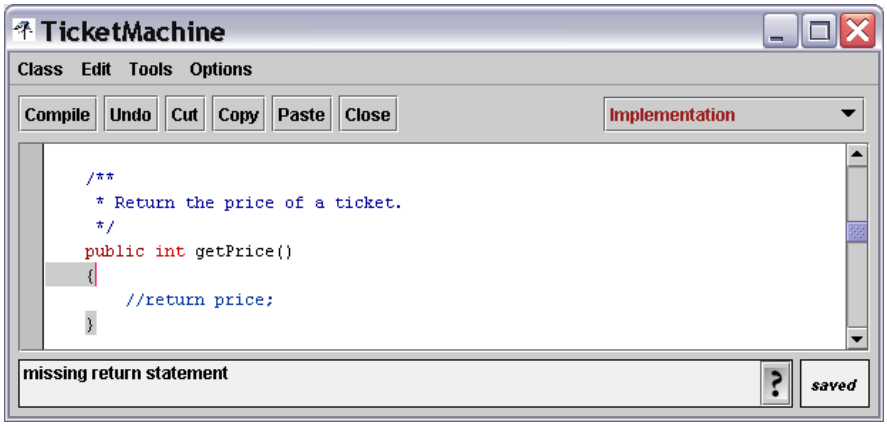
Solution:

If you remove the **return** you will get the following message when you attempt to compile the class:



The line is highlighted to alert you and you are informed that the line is '**not a statement**'.

If you remove the **whole line**, the message you will receive when attempting to compile the class is as shown below:



You are informed that your code is '**missing return statement**'. This error is triggered by the fact that you have specified an **int** for the return type and yet opted **not to return anything**.

Note

Note that I 'removed' the line of code by commenting it out as shown below. This is the best way to **temporarily remove lines - or blocks - of code** when you are **debugging** or trying something out. If you comment the code in this manner then it is simply a matter of removing your comment markers // to restore the code.

```
//return price;
```

If you were to delete the code then there is every chance that you might forget what the code was. Therefore, debugging using comments is recommended.

Remember to **restore the code** to the way it was before progressing further.

Ex 2.14

Compare the method signatures of **getPrice** and **printTicket** in code 2.1. Apart from their names, what is the main difference between them?

The respective method signatures are illustrated below:

```
public int getPrice()
```

…and…

```
public void printTicket()
```

The main difference is with regard to the **return type**.

The return type of **int** specified for the **getPrice** method indicates that an **integer** will be returned by the **getPrice** method. This **integer** will obviously relate to the **price** field of the class.

The return type of **void** specified for the **printTicket** method indicates that there will be **no value returned**.

Ex 2.15

Exercise 2.15 reads as follows:

Do the **insertMoney** and **printTicket** methods have return statements? Why do you think this might be? Do you notice anything about their headers that might suggest why they do not require return statements?

Solution:

Neither the **insertMoney** nor the **printTicket** methods have return statements.

These are **mutator** methods that are intended to perform some change to one or more of the fields of a ticket machine object.

They are **not accessor** methods, used simply to access the current value of a specific field.

Both the **insertMoney** and the **printTicket** methods have a return type of **void** in their headers and, therefore, they are **not expected to return any value**.

Ex 2.16

Exercise 2.16 reads as follows:

Create a ticket machine with a ticket price of your choosing. Before doing anything else, call the **getBalance** method on it.

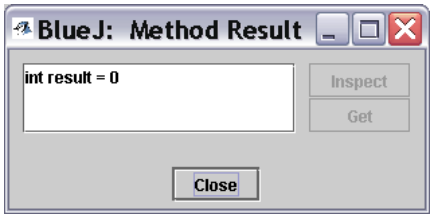
Now call the **insertMoney** method (Code 2.6) and give it a non-zero positive amount of money as the actual parameter.

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.

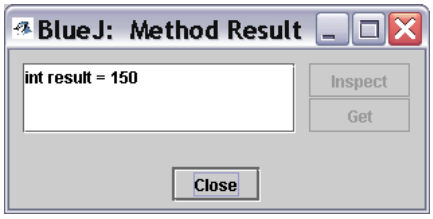
I created a ticket machine with a ticket price of **100** cents.

Invoking the **getBalance** method returns the result **0**, as illustrated on the right.



I then invoked the **insertMoney** method, entering **150** cents when prompted.

Invoking the **getBalance** method now returns a different result - 150.



The code for the **insertMoney** method is illustrated below:

```
public void insertMoney(int amount)
{
    balance += amount;
}
```

The **insertMoney** method expects a parameter - **amount** - of type int to be passed. The value of **amount** is then added to the current value of the **balance** field.

Note

Remember that this could also be written as:

```
    balance = balance + amount;
```

Ex 2.17

Exercise 2.17 reads as follows:

Create a 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**.

As instructed above. Ensure that the maths is correct i.e. if you invoke **insertMoney** and enter **150** cents and then invoke **insertMoney** again but enter **250** cents then you would expect a call to **getBalance** to return a value of **400**.

Ex 2.18

Now make a slight change to the **insertMoney** method via the BlueJ editor. Alter the **'+='** operator to '**=**'.

Compile the class and **re-run the method calls you just made** on a new ticket machine. What does the balance show now? (Don't forget to change the operator back to '+=' once you have finished.)

The changed method will appear as shown below:

```
public void insertMoney(int amount)
{
    balance = amount;
}
```

Do this…

Create a new **TicketMachine** object

getBalance - result is 0

insertMoney, for example, 500 cents

getBalance - result is 500

insertMoney, for example, 300 cents

getBalance - result is 300

Hang on, **where did the previous 500 go**?

What happened

Removing the **+** from the line…

```
balance += amount;
```

…had a considerable effect.

Previously, the line added whatever value was held in amount to the amount held in balance. With the change, the value held in amount is simply assigned to balance.

Remember that **assignation** is a **destructive process** and, therefore, the value previously held in **balance** is **lost**.

Remember to **change the operator back to '+=' as instructed**.

Ex 2.19

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.

The code for the **prompt** method would appear as shown below:

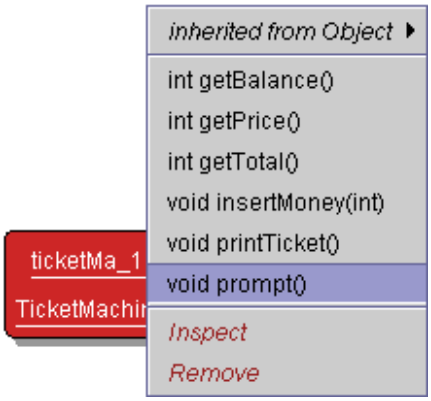
```
public void prompt()
{
  System.out.println("Please insert the correct
                      amount of money");
}
```

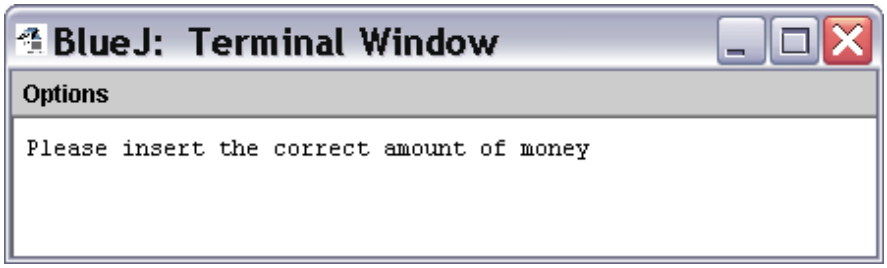
(Note that the String should all be on one line - I have split it because of space limitations)

Compile the code and then create a **TicketMachine** object.

Right click on the object that you have created and select the **prompt** method from the menu.

You will see the terminal window with the message '**Please insert the correct amount of money**' as illustrated below.





Ex 2.20

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.

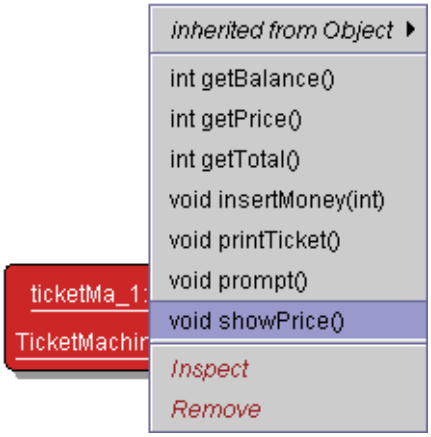
The code for the **showPrice** method would appear as shown below:

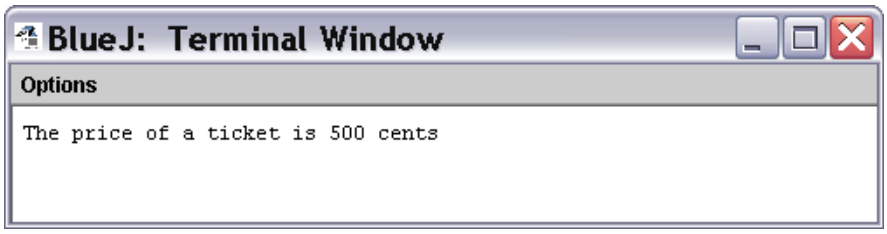
```
public void showPrice()
{
  System.out.println("The price of a ticket is"
                     + price + " cents");
}
```

Create a ticket machine object.

Right click on the object that you have created and select the **showPrice** method from the menu.

You will see the terminal window with the message '**The price of a ticket is xyz cents**' (where xyz is replaced with the actual ticket price that you entered when creating the ticket machine object) as illustrated below.





Ex 2.21

Create two ticket machines with **differently priced tickets**. Do calls to their **showPrice** methods show the **same output or different**? How do you explain this effect?

Ticket Machine 1, ticket cost **500** cents:

```
The price of a ticket is 500 cents
```

Ticket Machine 2, ticket cost **750** cents:

```
The price of a ticket is 750 cents
```

The difference in output is due to the different values of the **price** field in each of the ticket machine objects.

When you create each object, the value that you enter for the ticket price is then assigned to the **price** field.

Thus, when you have **different objects**, with **different ticket prices**, the **output will be different** when it makes use of that **price** field to determine what is displayed.

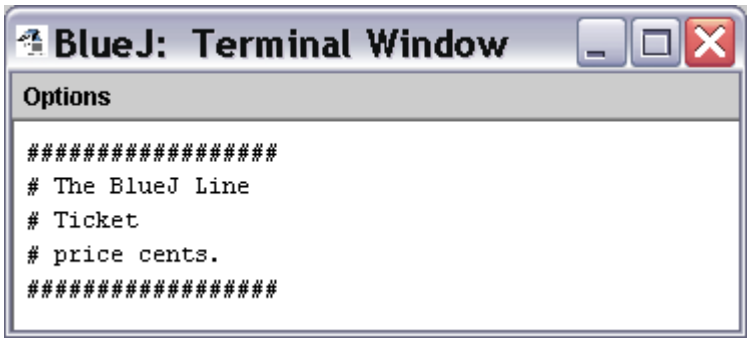
```
public void showPrice()
{
  System.out.println("The price of a ticket is"
                     + price + " cents");
}
```

Ex 2.22

What do you think would be printed if you altered the fourth statement of **printTicket** so that **price** also has quotes around it, as follows?

System.out.println("# " + "price" + " cents.");

The changes would result in the ticket shown below being displayed. As can be seen, the actual value of the **price** field is no longer displayed. Instead, **price** is treated like a string and is displayed as such.



Ex 2.23

What about the following version?

System.out.println("# price cents.");

The above code would produce exactly the same result. The code itself is different but the result is the same - **price** would again be treated as a string and the value of the **price** field would not be displayed.

The only difference in the above two examples is that the **first one makes use of concatenation** to form the string. The second example simply creates a string using one set of **"** and **"**.

Ex 2.24

Exercise 2.24 reads as follows:

Could either of the previous two versions be used to show the price of tickets in different ticket machines? Explain your answer.

I don't think that the question is all that clear. However, the answer that I think they are looking for is as follows:

The whole point of the preceding exercises was to illustrate how an instance variable - **price** - was used to display information pertaining to **a particular object**.

i.e. in this example we had two separate **TicketMachine** objects, each with **different ticket prices**.

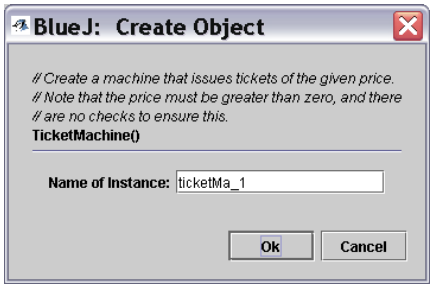
Had we not used **concatenation** to 'pull together' the standard String information that we wanted to display - i.e. the ticket - and the **actual value** held in the **price instance variable**, then we would not have been able to display accurate information for each of our TicketMachine objects.

Ex 2.25

Modify the constructor of **TicketMachine** so that it no longer has a parameter. Instead, the price of tickets should be fixed at 1000 cents. What effect does this have when you construct ticket machine objects within BlueJ?

```
public TicketMachine()
{
    price = 1000;
    balance = 0;
    total = 0;
}
```

Create a new instance of a ticket machine object.



You will see that you are **no longer prompted to enter a value** for the price of your ticket.

Calling either the **getPrice** or **showPrice** methods will reveal that the price of the ticket is **1000** cents.

If you alter the constructor of **TicketMachine** in this way, **every** ticket machine object that you create will have a **ticket price of 1000 cents**.

Note: Important

This arguably makes the ticket machine class definition less versatile.

Class definition for Flat

To elaborate:

Lets create a class definition for a **Flat**.

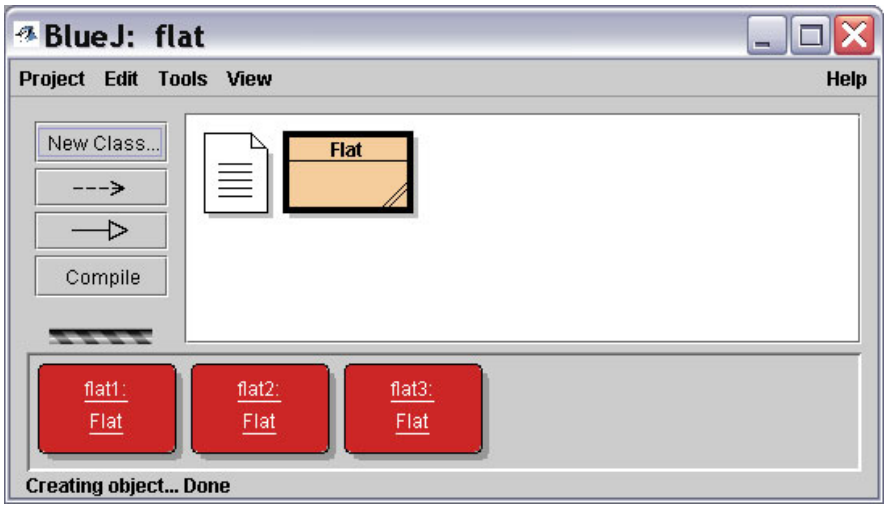
Lets keep things real simple and say that we have only one instance variable, a **String** named **address** that will hold the **address of the flat**.

The code for our Flat class is very simple as shown below:

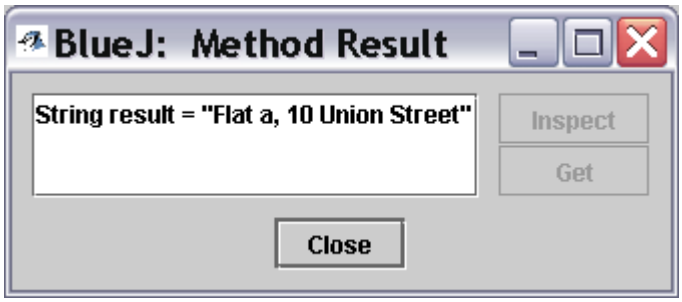
Flat class definition

```
public class Flat
{
    // instance variables
    private String address;

    public Flat()
    {
        // initialise instance variables
        address = "Flat a, 10 Union Street";
    }

    public String getAddress()
    {
        return address;
    }

    public void setAddress(String newAddress)
    {
        address = newAddress;
    }
}
```

Having compiled our Flat class, we can then **create instances of our flat** - In modelling terms we could be **constructing flats** or merely **representing an existing street** for example.



Invoke the **getAddress** method for each instance of a flat object - the display is always the same **"Flat a, 10 Union Street"**



Because we **'hard coded'** information into the constructor method for our Flat class definition, we end up with instances of flat objects that all have the **same address** - in terms of modelling multiple flats that is **useless!**

Each flat would be expected to have its own unique address.

The same can be said for **TicketMachine** objects - If we leave the ability to determine the **price** open then we have a **much more useful class definition**, one that can accommodate the creation **of many many TicketMachine** objects, with **varying ticket prices**.

Note

Ensure that you change your constructor method back so that it again **prompts for the entry of a value for the ticket price**.

Ex 2.26

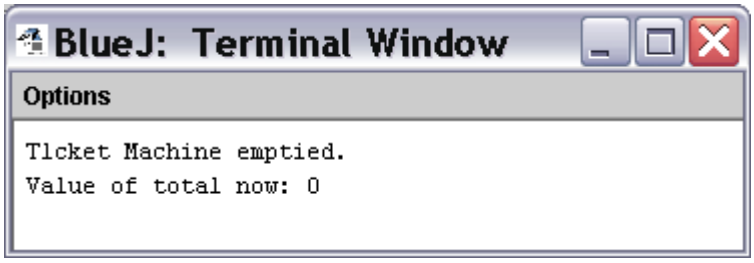
Implement a method, **empty**, that simulates the effect of removing all money from the machine. This method should have a **void** return type, and its body should simply set the **total** field to zero. Does this method need to take any parameters? Test your method by creating a machine, inserting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

The code in its simplest form:

```
public void empty()
{
    total = 0;
}
```

Alternatively, using the **terminal window** to inform the user that the machine has been emptied. Note that by referencing the **total** variable, we essentially 'check' to ensure that it has been emptied i.e. we would expect the display always to be **0**.

```
public void empty()
{
   total = 0;
   System.out.println("Ticket machine emptied");
   System.out.println("Value of total now: " + total);
}
```



Ex 2.27

Implement a method, **setPrice**, which 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 **TicketMachine**, **showing** the **price** of tickets, **changing** the **price**, and then **showing** the **new price**.

Is this method a mutator?

The code for the **setPrice** method in its simplest form is shown below:

```
public void setPrice(int newPrice)
{
    price = newPrice;
}
```

Alternatively, we could ensure that the suggested **newPrice** is a **sensible value** i.e. that it is **greater than 0**, a sensible check.

If the newPrice is **not greater than 0** then **the current price should remain** and a message to inform the user that **no change was made** should be shown. The code for this is shown below:

```
public void setPrice(int newPrice)
{
    if(newPrice > 0)
        price = newPrice;
    else
        System.out.println("Ticket Price must be
                            greater than 0!");
}
```

(Note that the String is split over two lines due to space limitations)

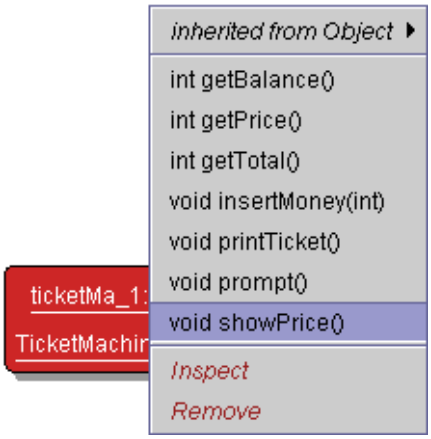
Note that **conditional statements** are covered in greater detail in the forthcoming pages of the chapter.

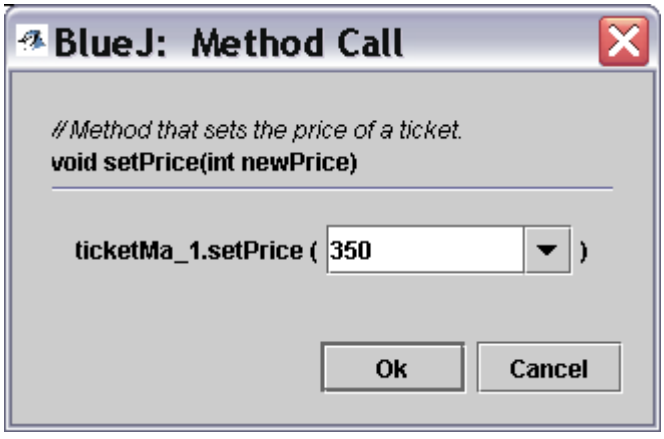
Testing this…

Create a new ticket machine object.

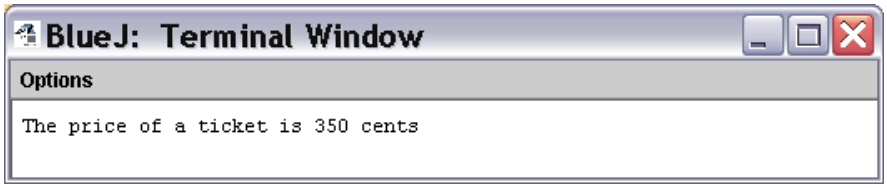
Call the **setPrice** method for this ticket machine object.

Enter an appropriate price when prompted, as illustrated below:





Invoke the **showPrice** method for your ticket machine object. Your new ticket price will be displayed in the terminal window.

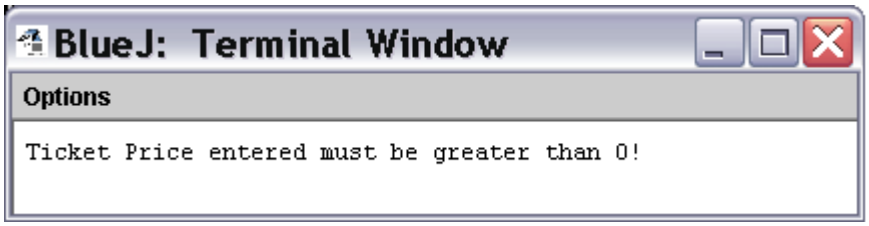


Negative Price

Invoke the **setPrice** method but this time enter a **negative** amount

i.e. **-100**.

What happens? You should see the message shown below. Further, if you now invoke the **showPrice** method, you should see that the current ticket price **remains the same** as it was before you attempted to set a negative value.



Ex 2.28

Exercise 2.28 reads as follows:

Give the class two constructors. One should take a single parameter that specifies the price, and the other 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.

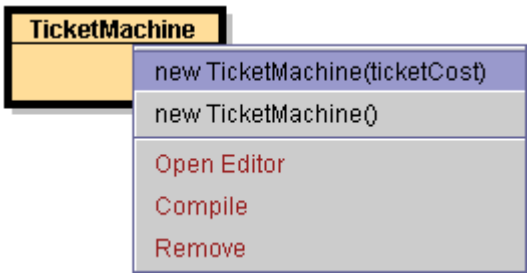
Your class definition already contains one constructor method:

```
public TicketMachine(int ticketCost)
{
    price = ticketCost;
    balance = 0;
    total = 0;
}
```

Add a further constructor such as the one illustrated below:

```
public TicketMachine()
{
    price = 1000;
    balance = 0;
    total = 0;
}
```

When you attempt to create a new ticket machine object, you will see that you **now have a choice** - a ticket machine whereby **you provide the ticket cost**, and a **ticket machine that expects no input**:



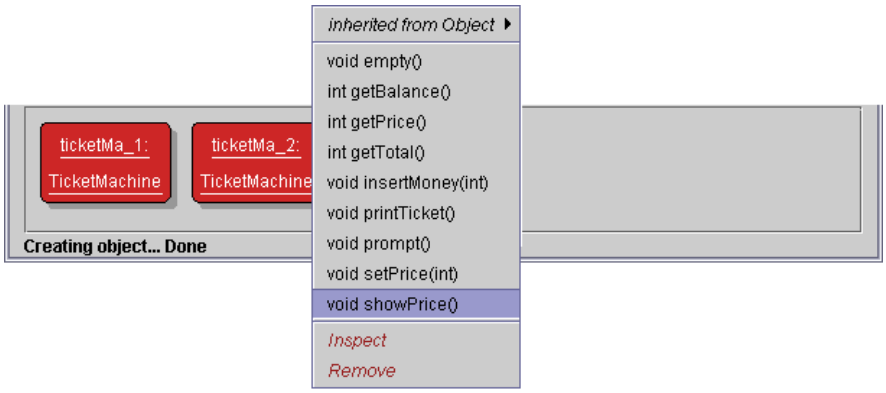
Create **two ticket machine objects**, **one with** and **one without** a provided ticket cost, and then **test** each ticket machine's **showPrice** method to see what is displayed.

One of the ticket machine objects should indicate a ticket price that is the **same as the price that you hard-coded**

i.e. the ticket machine that **does not expect to be provided with a ticket price parameter**.

The **other ticket machine** should indicate a ticket price that is the **same as the price you provided when you created the object**.

i.e. when you were **prompted for and entered a parameter**.



Correct Project

Make sure that you have the correct project open:

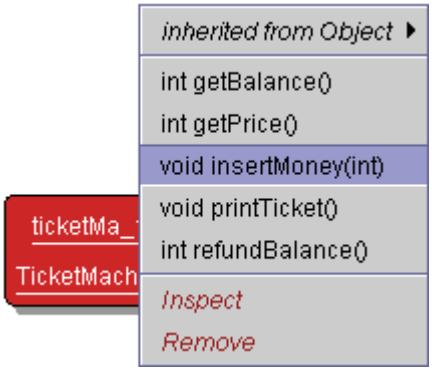
better-ticket-machine.

Ex 2.29

Exercise 2.29 reads as follows:

Check that the behaviour we have discussed here is accurate by creating a **TicketMachine** instance and calling **insertMoney** with various actual parameter values.

Check the balance both before and after calling **insertMoney**.



You start with an **Initial balance** of **0**. **Insert** some money i.e. **1000**. **Check** the **balance**.

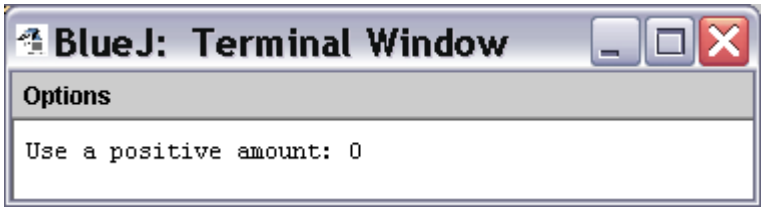
You should see that the **balance** is **1000**.

Insert some more money i.e. **350**.

Check the **balance** yet again. The **balance** should read **1350**.

Does the balance ever change in the cases where an error message is printed? Try to predict what will happen if you enter the value zero as the parameter and then see if you are right.

The **better-ticket-machine** is, as the name suggests, better! It introduces sensible behaviour. For example, if an error message is reported, no changes are made. Further, if you enter **0** as the parameter to the **insertMoney** method, you will see the following display:



Check the code of the **insertMoney** method to see why this happens:

```
public void insertMoney(int amount)
{
    if(amount > 0){
        balance += amount;
    }
    else{
        System.out.println("Use a positive amount: " +
                           amount);
    }
}
```

The **insertMoney** method makes use of a **conditional statement** - an **if** .. **else** - to check that the amount of money entered is **greater than 0** i.e. that the amount entered is **not a negative amount**.

As long as the amount of money entered is greater than 0, then the amount is **added to the value of the current balance**.

Remember

Remember that the line…

```
balance += amount;
```

Could be written as follows….

```
balance = balance + amount;
```

The second format is, perhaps, more obvious in terms of exactly what is happening to the balance.

If the amount of money entered is not greater than 0, then an appropriate message - '**Use a positive amount**' - is displayed in the terminal window.

Note also…

Note also that, where the **alternative** action is performed, there is **no change** made to the **balance** whatsoever - it remains the same as before.

Ex 2.30

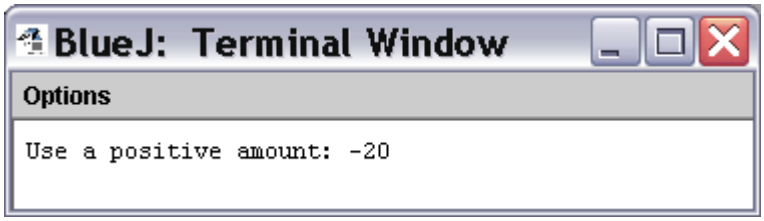
Predict what you think will happen if you change the text in **insertMoney** to use the greater-than or equal-to operator.

if(amount >= 0)

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

The **insertMoney** method no longer objects to the value **0** being entered. Subsequent calls to the **getBalance** method show that no changes are made to the value held in **balance** when 0 is entered.

The error message experienced previously is now displayed only when the user attempts to enter a **negative amount** as a ticket price.



Ex 2.31

Exercise 2.31 reads as follows:

In the **shapes** project we looked at in Chapter 1 we used a **boolean** field to control a feature of the circle objects. What was that feature? Was it well suited to being controlled by a type with only two different values?

The **boolean** field used in the circle class definition of the **shapes** project was **isVisible**.

```
private boolean isVisible;
```

The name of the field speaks for itself - It describes the **visibility** (or otherwise) of the shape.

If we look at the **constructor method** for the **Circle** class, we see that the **initial visibility** is set to **false**.

Thus, when working with the **Circle** and, indeed, any of the other shapes, we have to **actively set the visibility to true** when we want to see an instance of the shape in question.

```
public Circle()
{
    diamter = 30;
    xPosition = 20;
    yPosition = 60;
    color = "blue"
    isVisible = false;
}
```

This **is** an appropriate use of a boolean.

After all, the circle could **only ever be visible or invisible** - there is **no in-between status** that could apply.

Ex 2.32

Exercise 2.32 reads as follows:

In this version of **printTicket** we also do something slightly different with the **total** and **balance** fields. Compare the implementation of the method in Code 2.1 with that in Code 2.8 to see whether you can tell what those differences are. Then check your understanding by experimenting within BlueJ.

The code for the method in question is as follows:

```
public void printTicket()
{
    if(balance >= price) {
        // Simulate the printing of a ticket.
        System.out.println("####################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("####################");
        System.out.println();

        // Update the total collected with the price.
        total += price;
        // Reduce the balance by the price.
        balance -= price;
    }
    else {
       System.out.println("You must insert at least: "
                          + (price - balance) +
                          " more cents.");
    }
}
```

(Note that the code to be performed as the alternative action has been split over three lines due to space limitations)

The 'new' printTicket method differs in such that the code incorporates sensible checks.

For example, before printing a ticket, the **printTicket** method actually checks to see if the current value of the balance is greater than or equal to the price of a ticket.

```
if(balance >= price)
```

This is sensible behaviour - after all, where in life do you get away with paying less than the actual cost of something?

Further, The **balance** is not simply set to 0 once a ticket has been printed. The actual cost of the ticket is deducted from the balance.

```
balance -= price;
```

Remember that this could also be written as follows:

```
balance = balance - price;
```

This may be **more obvious** in terms of the action performed. The actual **price** of a ticket is **subtracted** from the **balance** and this **new value is then assigned as the value of balance**. Thus, we maintain an **accurate** (and **honest**!) account of the balance.

If the balance were greater than the ticket price, the user would find that the **difference remained**. They would no longer be 'ripped off'!

This situation is further attained via the provision of a **refundBalance** method, enabling the user to obtain any money left in the **balance** field.

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

A **local variable** is declared - **amountToRefund**.

This does exactly as its name suggests. It holds the value that is to be refunded.

If we are requesting a refund then obviously the refund is the **current value of balance**.

Thus, the value held in **balance** is assigned to **amountToRefund**. Having done this, we can **safely set balance to 0** without fear of short-changing our customer!

We then return the **amountToRefund**.

The **insertMoney** method is also different. The amount of money inserted is checked to ensure that it is a positive amount - i.e. **amount** is **greater than 0** - This represents another sensible check. After all, it is not possible to insert a negative amount of money!

```
public void insertMoney(int amount)
{
    if(amount > 0) {
        balance += amount;
    }
    else {
        System.out.println("Use a positive amount: "
                           + amount);
    }
}
```

Ex 2.33

Exercise 2.33 reads as follows:

After a ticket has been printed, could the value in the **balance** field ever be set to a **negative value** by subtracting **price** from it? Justify your answer.

No. There is no opportunity for this situation to occur. The sensible checks introduced in the **better-ticket-machine** prevent this kind of behaviour from occuring.

One of these is a check to ensure that a ticket is only printed **if** the balance is greater than or equal to the actual cost of a ticket.

```
public void printTicket()
{
     if(balance >= price) {
       //code here…
```

The first thing that the 'new' **printTicket** method does is to **check** that the **balance** is either **greater than** or **equal to** the **price** of a ticket.

As a result, there should **never** be a situation whereby a ticket is printed **without sufficient funds** to pay for it - the user may end up with a **balance of 0** but that is **as low as balance will go** and is perfectly acceptable. There are no opportunities for a negative balance.

Ex 2.34

Why does the following version of **refundBalance** not give the same result as the original?

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

The original version of **refundBalance** is shown below:

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

The **amended version** of **refundBalance** sets the variable **balance** to a value of 0 and **then** returns **balance**, complete with its **0** value - hardly a refund!

The **original version** makes use of a local variable, **amountToRefund**, which is defined within the **refundBalance** method and is then used to hold the value of **balance**.

To simulate an actual refund, **balance** is then set to **0** and the value that was previously held in **balance**, but that is now held in **amountToRefund**, is returned.

Which tests can you run to demonstrate that it does not?

Make the changes to the **refundBalance** method as shown below. I would recomend **commenting out the original code** so that it can be **easily restored** once you have finished testing.

Make the changes…

```
public int refundBalance()
{
   /** int amountToRefund;
   *   amountToRefund = balance;
   *   balance = 0;
   */  return amountToRefund;

   balance = 0;
   return balance;
}
```

Compile…

Compile the class and then create a **TicketMachine** object.

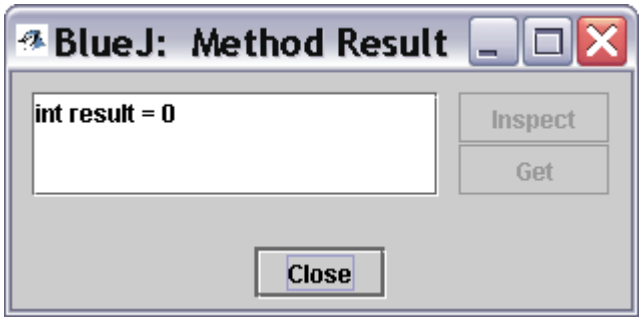
Insert **more than enough** money to pay for a ticket.

Invoke the **printTicket** method. This should buy you a ticket.

You **should** then have some money left over.

Invoke the **refundBalance** method.

Thanks to the changes that you have made, you should see the following display. Consider yourself robbed!



With the previous example of the code, you would, at this point, have seen the **actual remaining money** once you had paid for your ticket.

Restore

Remember to **restore** the method to its original state.

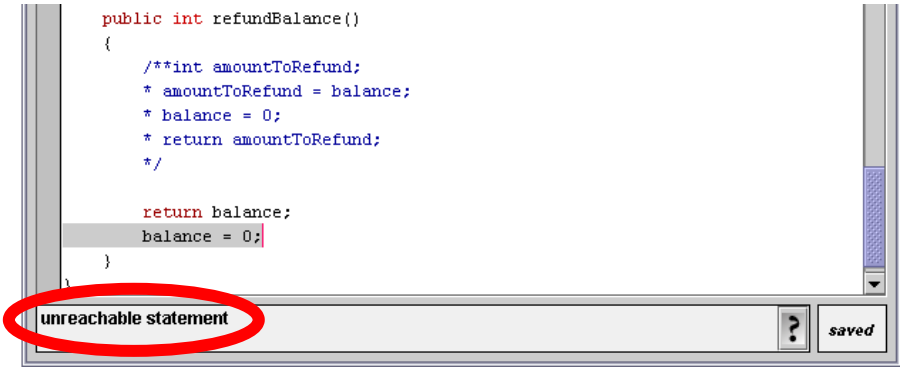
Ex 2.35

Exercise 2.35 reads as follows:

What happens if you try to compile the **TicketMachine** class with the following version of **refundBalance**:

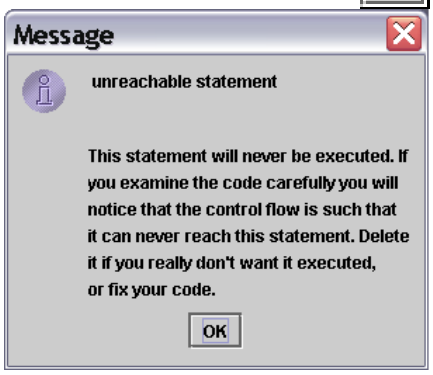
```
public int refundBalance()
{
  return balance;
  balance = 0;
}
```

With the code shown above, you will get the following result when you attempt to compile. (Note that I have again maintained the original code using comments so that it can easily be restored at a later point).



Click on the question mark button in the bottom right hand corner of the BlueJ window:

You have an **unreachable statement**. This is because the return statement in the previous line essentially **quits the method**.



The line…

```
balance = 0;
```

…is never reached!

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

Solution…

To quote from the book:

"Where a method contains a return statement, it is always the final statement of that method, because no further statements in that method will be executed once the return statement is executed."[1]

This requires no further explanation!

Ex 2.36

Exercise 2.36 reads as follows:

Add a new method, **emptyMachine**, that is designed to simulate **emptying the machine of money**.

It should both return the value in **total** and reset **total** to be zero.

```
public int emptyMachine()
{
    int amountEmptied = total;
    total = 0;
    return amountEmptied;
}
```

The way I have done this is similar to the approach employed in the **refundBalance** method. A **local variable** is used in the same way as in the **refundBalance** method.

[1] Barnes, D.J. & Kolling, M; Objects First With Java, A Practical Introduction Using BlueJ; p28

Ex 2.37

Is **emptyMachine** an **accessor**, a **mutator**, or **both**?

The **emptyMachine** method does indeed **access** a value held in one of the objects instance variables - **total**.

However, it does this **indirectly**, through the assignation of the value held to another, **locally defined variable**.

The instance variable **total** is reset to zero. Thus, the method could be thought of as a **mutator** - it **mutates/changes the value of total**.

Ex 2.38

Exercise 2.38 reads as follows:

Rewrite the **printTicket** method so that it declares a local variable, **amountLeftToPay**. This should then be initialized to contain the difference between **price** and **balance**.

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.

Solution

The code for this is as shown below:

```
public void printTicket()
{
    int amountLeftToPay = price - balance;
    if(amountLeftToPay <= 0) {
        // Simulate the printing of a ticket.
        System.out.println("####################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("####################");
        System.out.println();

        // Update the total collected with the price.
        total += price;
        // Reduce the balance by the prince.
        balance -= price;
    }
    else {
        System.out.println("You have "
                           + amountLeftToPay
                           + " still to pay.");
    }
}
```

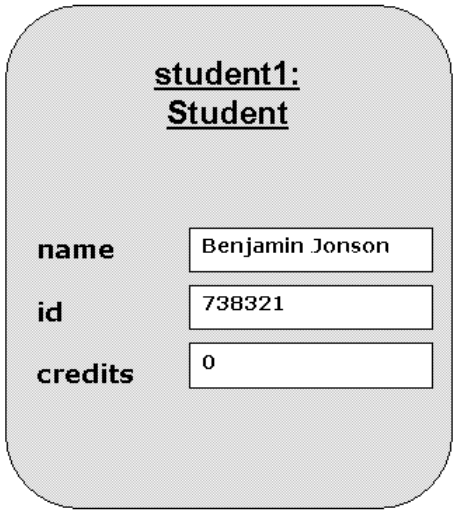
Ex 2.39

Exercise 2.39 reads as follows:

Draw a picture of the form shown in Figure 2.3 representing the initial state of a **student** object following its construction with the following actual parameter values:

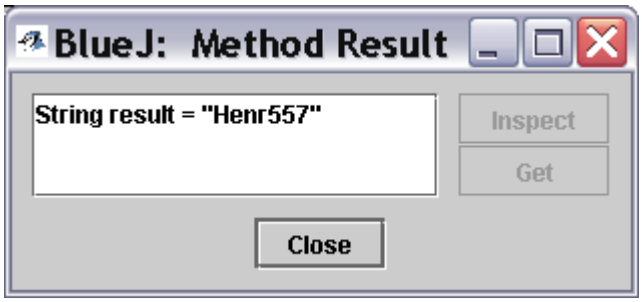
new Student("Benjamin Jonson", "738321")

Solution



Ex 2.40

What would be returned by **getLoginName** for a student with the **name "Henry Moore"** and the **id "557214"**?



As shown above, the login name of **Henr557** is returned for a student named **"Henry Moore"** with the id **"557214"**.

Why?

```
public String getLoginName()
{
    return name.substring(0,4) + id.substring(0,3);
}
```

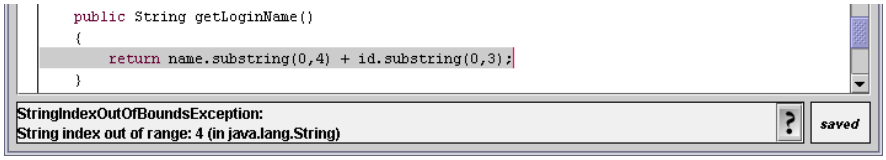
The **getLoginName** method takes the **name** and the **id** Strings and then, using **substring**, obtains **parts of the Strings** i.e. starting at position **0**, **4 characters** are taken from the **name** String. Starting at position **0**, **three characters** are taken from the **id** String.

Thus, we end up with: **Henr557**

Ex 2.41

Create a **student** with **name "djb"** and **id "859012"**. What happens when **getLoginName** is called on this student? Why do you think this is?

When you attempt to call the **getLoginName** method for the student with **name "djb"** and **id "859012"** you get the following error message:



The problem is that you are attempting to access a character within the **name** String that **does not exist** i.e. there are only 3 characters in the String **djb**.

Ex 2.42

Exercise 2.42 reads as follows:

The **String** class defines a **length** accessor method with the following signature:

```
/**
 * Return the number of characters in this string.
 * /
 public int length( )
```

Add conditional statements to the constructor of **Student** to print an error message if either the **length** of the **fullName** parameter is **less than four characters** or the **length** of the **studentId** parameter is **less than three characters**.

However, the constructor should **still use those parameters** to set the **name** and **id** fields, even if the error message is printed. Hint: use if statements of the following form (that is, having no else part) to print the error messages.

```
if (perform a test on one of the parameters)
{
  print an error message if the test gave a true result
}
```

See Appendix C for further details of the different types of if statement, if necessary.

Ambiguity

This question is perhaps slightly ambiguous??? I read it and produced the following as my **suggested solution**:

Note that my suggested solution checks both the **fullName** and the **studentID** parameters that are passed to the constructor method and **displays messages accordingly** if they are **less than 4** and **less than 3** characters respectively.

Note, however, that I **still assign the parameters**, irrespective of their length.

Amended Student constructor method

```
public Student(String fullName, String studentID)
{
    if(fullName.length() < 4)
        System.out.println("Name provided is too
                            short");
    name = fullName;
    if(studentID.length() < 3)
        System.out.println("ID provided is too
                            short");
    id = studentID;
    credits = 0;
}
```

Ex 2.43

Challenge Exercise: Modify the **getLoginName** method of **Student** so that it always generates a login name, even if either of the **name** and **id** fields is not strictly long enough. For strings shorter than the required length, use the whole string.

Solution

My **suggested solution** is shown below:

```
public String getLoginName()
{
    String loginName = "";
    if(name.length() >= 4)
        loginName = name.substring(0,4);
    else
        loginName = name;
    if(id.length() >= 3)
        loginName += id.substring(0,3);
    else
        loginName += id;
    return loginName;
}
```

The first thing that I have done is to establish a **local variable**, of type **String**, called **loginName**.

I then perform some simple tests to build the necessary login name.

If the respective **name** or **id** String is of the **correct length or above**, then I make use of this in the generation of the login name.

In the event that the **provided details are inadequate**, I simply **add** the information that **has** been provided to the login name string.

Finally, I return the **loginName** String.

Ex 2.44

Below is the outline for a **Book** class, which can be found in the **book-exercise** project. The outline already defines **two fields** and a **constructor** to initialize the fields. In this exercise and the next few, you will **add further features to the class outline**.

The provided code is as follows:

```
/**
 * A class that maintains information on a book.
 * This might form part of a larger application such
 * as a library system, for instance.
 *
 * @author (Insert your name here.)
 * @version (Insert today's date here.)
 */
class Book
{
    // The fields.
    private String author;
    private String title;

    /**
     * Set the author and title fields when this object
     * is constructed.
     */
    public Book(String bookAuthor, String bookTitle)
    {
        author = bookAuthor;
        title = bookTitle;
    }

    // Add the methods here ...
}
```

Add two accessor methods to the class - **getAuthor** and **getTitle** - that return the **author** and **title** fields as their respective results. Test your class by creating some instances and calling the methods.

getAuthor

```
public String getAuthor()
{
    return author;
}
```

getTitle

```
public String getTitle()
{
    return title;
}
```

Ex 2.45

Add two methods, **printAuthor** and **printTitle**, to the outline **Book** class. These should print the author and title fields, respectively, to the terminal window.

printAuthor

```
public void printAuthor()
{
    System.out.println("Author: " + author);
}
```

printTitle

```
public void printTitle()
{
    System.out.println("Title: " + title);
}
```

Ex 2.46

Add a further field, **pages**, to the **Book** class to store the number of pages. This should be of type **int**, and its initial value should be passed to the single constructor, along with the **author** and **title** strings. Include an appropriate **getPages** accessor method for this field.

First, add a **pages** attribute to the list of instance variables for the class:

```
// The fields.
private String author;
private String title;
private int pages;
```

Next, amend the **constructor** method for the class so that information pertaining to the number of **pages** can be passed to the **constructor**:

```
public Book(String bookAuthor, String bookTitle, int numberOfPages)
    {
        author = bookAuthor;
        title = bookTitle;
        pages = numberOfPages;
    }
```

Ex 2.47

Add a method, **printDetails**, to the **Book** class. This should print details of the **author**, **title** and **pages** to the **terminal window**.

It is your choice how the details are formatted. For instance, all three items could be printed on a single line, or each could be printed on a separate line. You might also choose to include some explanatory text to help a user work out which is the author and which is the title, for example:

Title: Robinson Crusoe, Author: Daniel Defoe, Pages: 232

```
public void printDetails()
{
    System.out.println("Title: " + title + ", Author: "
                       + author + ", Pages: " + pages);
}
```

Ex 2.48

Add a library field, **refNumber**, to the **Book** class.

This field can store a reference number for a library, for example. It should be of type **String** and initialized to the zero length string (" ") in the constructor as its initial value is **not passed in a parameter to the constructor**.

Instead, define a **mutator** for it with the following signature:

public void setRefNumber(String ref)

The body of this method should assign the value of the parameter to the **refNumber** field. Add a corresponding **getRefNumber** accessor to help you check that the mutator works correctly.

First, add **refNumber** to the instance variables for the class:

```
private String refNumber;
```

Next, **modify** the **constructor** method for the class so that **refNumber** is initialised to be a **blank String**:

```
public Book(String bookAuthor, String bookTitle, int numberOfPages)
    {
        author = bookAuthor;
        title = bookTitle;
        pages = numberOfPages;
        refNumber = "";
    }
```

The required mutator and accessor methods are included below:

```
public void setRefNumber(String ref)
{
    refNumber = ref;
}

public String getRefNumber()
{
    return refNumber;
}
```

Ex 2.49

Modify your **printDetails** method to include printing the reference number. However, the method should print the reference number only if it has been set - that is, the **refNumber** string has a non-zero length. If it has not been set, then print the string "zzz" instead. Hint: Use a conditional statement whose test calls the **length** method on the **refNumber** string.

The **suggested solution** for the **printDetails** method is shown below:

```
public void printDetails()
{
    String printThisString = "Title: " + title + ", Author: "
                             + author + ", Pages: " + pages
                             + ", Reference Number: ";
    if(refNumber.length() != 0)
        printThisString += refNumber;
    else
        printThisString += "zzz";
    System.out.println(printThisString);
}
```

Ex 2.50

Modify your **setRefNumber** mutator so that it sets the **refNumber** field only if the parameter is a string of at least three characters. If it is less than three, then print an error message and leave the field unchanged.

The **suggested solution** for the modified **setRefNumber** method is provided below: The thing to note is that, in this method, it is the parameter that is passed into the method - **ref** - that is being **checked for length**.

```
public void setRefNumber(String ref)
{
   if(ref.length() < 3)
      System.out.println("Error, Reference is too short");
   else
      refNumber = ref;
}
```

Ex 2.51

Add a further integer field, **borrowed**, to the **Book** class. This keeps a count of the **number of times a book has been borrowed**.

```
private int borrowed;
```

Note - Assumption

Note that the book does not explicitly state to do this but I would recommend that you set **borrowed** to **0** within the **constructor** method for the class. After all, when the book object is **first created**, it will **not yet have been borrowed**.

Add the following line to the constructor method:

```
borrowed = 0;
```

Add a **mutator**, **borrow**, to the class. This should update the field by 1 each time it is called.

```
public void borrow()
{
    borrowed += 1;
}
```

Include an accessor, **getBorrowed**, that returns the value of this new field as its result.

```
public int getBorrowed()
{
    return borrowed;
}
```

Modify **printDetails** so that it includes the value of this field with an explanatory piece of text. Note that I have done this on a separate line. The String starts with **\n** to force a new line.

```
public void printDetails()
{
    String printThisString = "Title: " + title + ", Author: "
                             + author + ", Pages: " + pages
                             + ", Reference Number: ";
    if(refNumber.length() != 0)
        printThisString += refNumber;
    else
        printThisString += "zzz";
    printThisString += "\nThis book has been borrowed "
                       + borrowed + " times.";
    System.out.println(printThisString);
}
```

Ex 2.52

Challenge Exercise: Create a new project, **heater exercise**, within BlueJ.

Edit the details in the project description - the text note you see in the diagram.

Create a class, **Heater**, that contains a single integer field, **temperature**. Define a constructor that takes no parameters. The **temperature** field should be set to the value 15 in the constructor. Define the mutators **warmer** and **cooler**, whose effect is to increase or decrease the value of temperature by 5° respectively. Define an accessor method to return the value of **temperature**.

My **suggested solution** is shown below:

```
/**
 * Simulation of a simple Heater.
 *
 * @author (Jonathan J Mackintosh)
 * @version (v1.0, 28th Oct 2003)
 */
public class Heater
{
    // instance variables
    private int temperature;

    /**
     * Constructor for objects of class Heater
     */
    public Heater()
    {
        // initialise instance variables
        temperature = 15;
    }

    public void warmer()
    {
        temperature += 5;
    }

    public void colder()
    {
        temperature -= 5;
    }

    public int getTemperature()
    {
        return temperature;
    }
}
```

Ex 2.53

Challenge Exercise: modify your **Heater** class to define three new integer fields: **min**, **max** and **increment**. The values of **min** and **max** should be set by parameters passed to the constructor. The value of **increment** should be set to 5 in the constructor. Modify the definitions of **warmer** and **cooler** so that they use the value of increment rather than an explicit value of 5. Before proceeding further with this exercise, check that everything works as before. Now modify the **warmer** method so that it will not allow the temperature to be set to a value greater than **max**.

Similarly modify **cooler** so that it will not allow **temperature** to be set to a value less than **min**. Check that the class works properly.

Now add a method, **setIncrement**, that takes a single integer parameter and uses it to set the value of increment. Once again, test that the class works as you would expect it to by creating some Heater objects within BlueJ.

My suggested solution is shown below:

```
/**
 * Simulation of a simple Heater.
 *
 * @author (Jonathan J Mackintosh)
 * @version (v1.0, 28th Oct 2003)
 */
public class Heater
{
      // instance variables
      private int temperature;
      private int min;
      private int max;
      private int increment;

      /**
       * Constructor for objects of class Heater
       */
      public Heater(int minValue, int maxValue)
      {
            // initialise instance variables
            min = minValue;
            max = maxValue;
            increment = 5;
            temperature = 15;
      }

      public void warmer()
      {
          if(temperature + increment > max)
              System.out.println("Unable to increase temperature");
          else
              temperature += increment;
      }

      public void colder()
      {
          if(temperature - increment < min)
              System.out.println("Unable to decrease temperature");
          else
              temperature -= increment;
      }

      public int getTemperature()
      {
          return temperature;
      }

      public void setIncrement(int incrementAmount)
      {
          increment = incrementAmount;
      }
}
```

Do things still work as expected if a negative value is passed to the **setIncrement** method?

Things most definitely **do not work as expected** if a negative value is passed to the **setIncrement** method.

If a negative value is passed and then assigned, invoking the **colder** method has the effect of **increasing** the temperature and invoking the **warmer** method has the effect of **decreasing** the temperature.

Add a check to this method to prevent a negative value from being assigned to increment.

My **suggested solution** is shown below:

```
public void setIncrement(int incrementAmount)
{
    if(incrementAmount < 0)
        System.out.println("Unable to set increment amount.");
    else
        increment = incrementAmount;
}
```