LAB 4: A Java order system

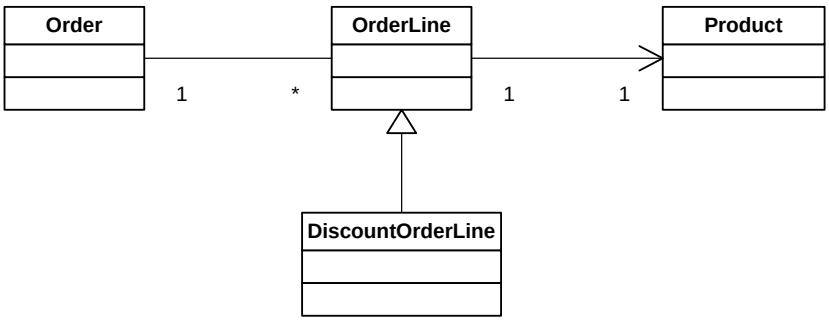
Getting started

In this lab you will create or complete some Java classes to mange orders in an ecommerce system. The completed application will consist of four classes, `Order`, `OrderLine`, `Product` and `DiscountOrderLine.`

You will be given complete code for the `Product` class. You will use a code generator tool within BlueJ to help you create the `Order` and `OrderLine` classes, and you will create the `DiscountOrderLine` class yourself.

An order object will contain a collection of order line and discount order line objects, each of which will contain a product object. A discount order line is a special version of an order line which can apply a discount to the standard product price.

The class diagram is shown below:



Task 1: Creating the Order and OrderLine classes

Installing code patterns in BlueJ:

BlueJ has a **PatternCoder** option[1] which lets you create two or more classes which work together in a commonly used way, or pattern. BlueJ creates the classes and sets up the fields which define the relationships, and some useful methods.

The classes created by BlueJ have the relationships and some common methods set up correctly, but you need to **modify them to provide the correct functionality** for your application.

1. Before you start working in BlueJ, you will need to install some files which allow BlueJ to use the patterns you need for this lab, as follows:

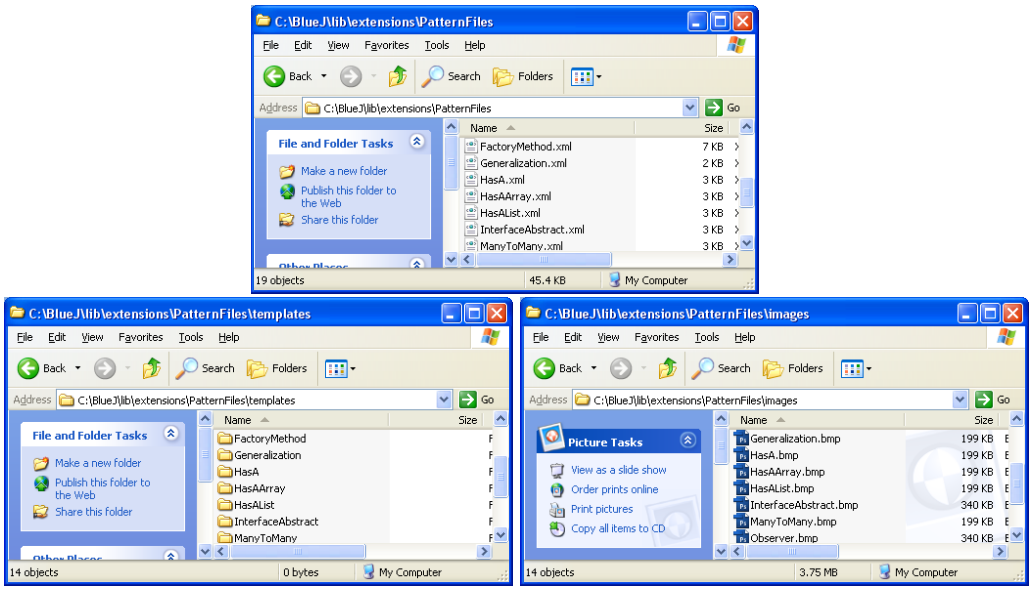
2. Download the file *PatternFiles.zip* and save it in the folder **c:\bluej\lib\extensions**. Extract the contents of the zip file into that folder.

   When you have done this you should check that the pattern files are in the correct place:

   - Open the folder **C:\BlueJ\lib\extensions\PatternFiles**. This folder should contain a number of XML files, including *HasAList.xml*.

   - Open the subfolder templates inside. This folder should contain a number of folders, including *HasAList*.

   - Go back up one level and open the subfolder images. This folder should contain a number of BMP files, including *HasAList.bmp*.



[1] Requires the PatternCoder extension for BlueJ (*http://www.patterncoder.org*).

Creating related classes: Order and OrderLine

1. Download, extract and open the BlueJ project *lab4*. Note that there is a single class `Product` in the project. This class is already complete. You will now create the `Order` and `OrderLine` classes.

2. The one-to-many relationship between `Order` and `OrderLine` is intended to mean that one `Order` contains many `OrderLines` Each `OrderLine` is owned by the `Order`. An `Order` **has-a** collection of `Orderlines`.

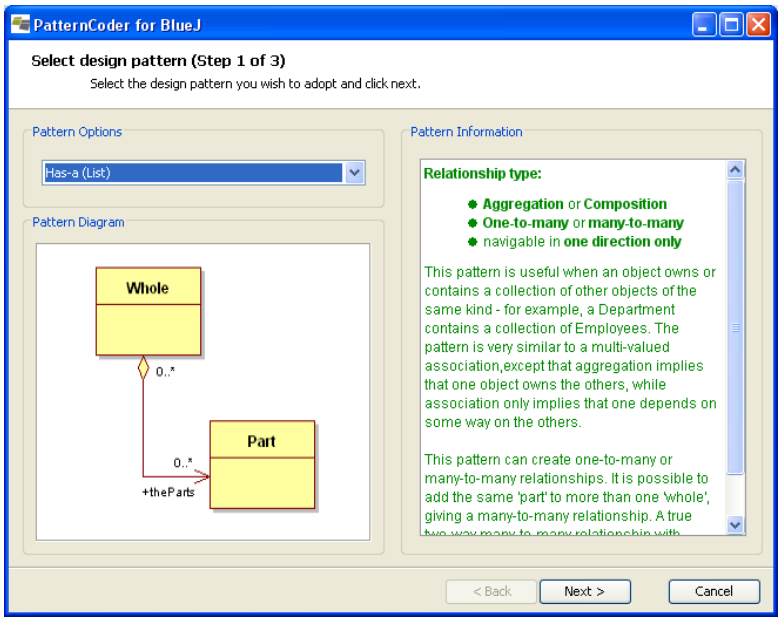
3. You have seen examples before of classes which have *has-a* relationships, and we have described the coding patterns for creating these. You could use these examples to help you create the code for `Order` and `OrderLine`. The `Order` class will need a set of methods to allow it to:

   - Add a new `OrderLine`
   - Find and return a specific `OrderLine`
   - Return the whole collection of `OrderLines`
   - Remove a specific `OrderLine`

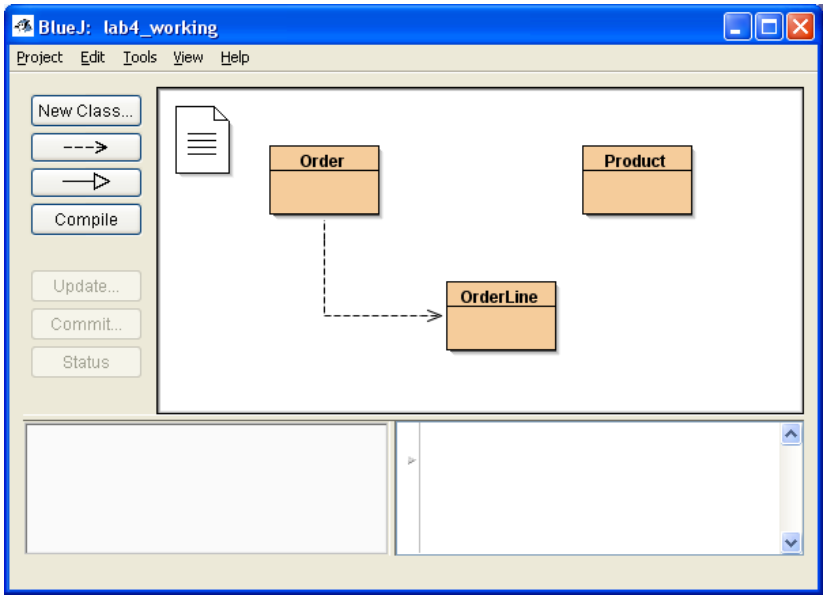
   To make this a bit easier, you can use the PatternCoder option to generate classes with the correct relationship already set up.

4. Select the Tools > PatterCoder menu option and choose the best option for this relationship in the Pattern Options list. There are two possibilities for a one-to-many relationship like this: *Has-a (Array)* and *Has-a (List)*. We will choose *Has-a (List)* as this will give the advantages of using an `ArrayList`.

   The PatternCoder dialog shows a diagram and some additional information about the pattern.



5. In the following steps, replace the name `Whole` with `Order`, and `Part` with `OrderLine`, and click Finish. The two classes, `Order` and `OrderLine` are added to your BlueJ project. You can drag the classes in the class diagram to position them so that you can see them clearly, and then click the Compile button. The classes should compile successfully.



Looking at the basic classes:

1. Double-click the `Order` class to see in the editor the code which has been created

   The one-to-many relationship between `Order` and `OrderLine` has been implemented by giving the `Order` class a **multivalued field**. This is an `ArrayList` which holds `OrderLine` objects.

```
public class Order
{
    private List<OrderLine> theOrderLines;
```

2. Look through the rest of the code for the `Order` class. `Order` has been given standard methods to add a new `OrderLine`, to find a specified `OrderLine` and to remove a specified `OrderLine`. Note that these last two methods require the target `OrderLine` to be specified using its `description` field value, and will need to be slightly modified later.

3. Open the `OrderLine` class in the editor. `OrderLine` has no reference to any `Order` objects. It simply has a `description` field.

Testing the basic classes:

1. Before you go on and modify these classes, you should test that they work. Create the following objects in the Object Bench

order1	new `Order`, `description` = "example order"
orderLin1	new `OrderLine`, `description` = "example orderline 1"
orderLin2	new `OrderLine`, `description` = "example orderline 2"
orderLin3	new `OrderLine`, `description` = "example orderline 3"

2. Call the `addOrderLine` method of *order1*, and supply *orderLin1* as the parameter. Repeat for the other `OrderLine` objects.

3. Call the `printDetails` method of *order1*. You should get the following output in the BlueJ terminal window:

   ```
   This Order object (description: example order) contains the
   following OrderLine objects:
   description: example orderline 1
   description: example orderline 2
   description: example orderline 3
   ```

   This demonstrates that the objects are correctly related and that `addOrderline` works as expected.

4. Call the `getOrderLine` method of *order1*, and supply *"example orderline 2"* as the parameter. Check that the appropriate `OrderLine` object is returned. Repeat for an invalid value of the parameter – what is returned?

5. Call the `removeOrderLine` method of *order1*, and supply *"example orderline 2"* as the parameter. Call `printDetails` again and check that the object was removed.

6. This demonstrates that the `getOrderLine` and `removeOrderLine` methods work as expected.

   You have achieved quite a lot at this point. You should have two related classes with all aspects of the relationship fully working. And you haven't actually written any code yet! You are, however, now ready to add the code which will make these classes properly represent Orders and OrderLines. You will need to write this code.

   THINK ABOUT IT: You could have done these tests with the help of the unit testing tools in BlueJ. How would you have done this?

NOTE

To help you understand a coding pattern, you can use PatternCoder to create more examples to study. You could, for example, use the *Has-a (List)* pattern to create other pairs of classes, such as `Department` and `Employee`, `Team` and `Player`, `Customer` and `Booking`, and so on. Look at the code which is created – what changes, and what stays the same? Try using the *Has-a (Array)* pattern instead – how does the code differ? The more examples you see, the easier it will be to write the full code yourself to implement a pattern in future.

Adding the real functionality for OrderLine:

Your two classes are not very useful yet. You will have to modify them to do something a bit more useful.

1. Remove the declaration of the example instance variable `description` and replace this with declarations of the following instance variables:

   - an int variable called `lineNumber`
   - an int variable called `quantity`
   - a `Product` variable called `product`

2. Modify the constructor of `OrderLine` so that it takes three parameters and sets the values of the three new instance variables.

3. Add another constructor for `OrderLine` which takes one parameter and sets the value of the `lineNumber` variable. (*Note*: *this is needed by the methods in `Order` which get and remove `OrderLines`*).

4. Add a default constructor for `OrderLine` which takes no parameters and doesn't do anything at all, like this:

   ```
   public OrderLine() {}
   ```

   (*Note*: *this is needed for part of task 2 of this lab to work correctly)*

5. Modify the javadoc comment for the constructor to match the modified code.

6. Remove the getter and setter methods for `description`, and replace these with getters (no setters) for the three new instance variables.

7. Add a new method `getCost` which takes not parameters and returns a double value calculated by multiplying the cost of the product by `quantity`. You will need to look at the `Product` class to see how to get the cost of a product.

8. Replace the *return* statement in the `toString` method with the following:

```
return String.format(
      "Line: %d, Product: %s, Quantity: %d, Cost: £%6.2f",
       lineNumber,
       product.getProductName(),
       quantity,
       getCost());
```

9. Find the `equals` method. This method is needed so that the `ArrayList` in `Order` can search through the `OrderLine` objects it contains. As it is currently defined, two `OrderLine` objects are considered equal if their `description` values are equal. You need to change this to identify `OrderLine` objects by their `lineNumber` values.

   Change the line:

```
if (this.description.compareTo(test.description) == 0)
```

   to

```
if (this.lineNumber==test.lineNumber)
```

10. Compile the `OrderLine` class.

11. If you have time, add Javadoc comments for the methods you have created in `OrderLine`.

Modifying Order to use OrderLine:

Your Order class will now not compile because it searches by `description` when getting and removing OrderLines. You will need to change this so that it searches by lineNumber instead

1. Find the `getOrderLine` method. Replace the lines:

```
   public OrderLine getOrderLine(String description)
    {
          OrderLine target = new OrderLine(description);
```

   with:

```
    public OrderLine getOrderLine(int lineNumber)
    {
          OrderLine target = new OrderLine(lineNumber);
```

2. Make similar changes to the `removeOrderLine` method.

3. Compile the `Order` class.

Testing the modified classes:

1. Create the following objects in the Object Bench

   order1
   new `Order`, `description` = "example order"

   product1
   new `Product`, `productCode` = "P1", `productName` = "widget", `cost` = 3.00
   product2
   new `Product`, `productCode` = "P2", `productName` = "gadget", `cost` = 5.00
   product3
   new `Product`, `productCode` = "P3", `productName` = "sprocket", `cost` = 7.00

   orderLin1
   new `OrderLine`, `lineNumber`=1,`product` = ***product1***, `quantity` = 5
   orderLin2
   new `OrderLine`, `lineNumber`=2,`product` = ***product2***, `quantity` = 2
   orderLin3
   new `OrderLine`, `lineNumber`=3,`product` = ***product3***, `quantity` = 10

2. Create a test class for `Order` and copy these objects from Object Bench to Test Fixture so that you can repeat tests easily if necessary. Copy back from Test Fixture to Object Bench to continue testing.

3. **Add** the three `OrderLine` objects to ***order1***.

   Call the `printDetails` method of ***order1***. You should get the following output in the BlueJ terminal window:

```
This Order object (description: example order) contains the
following OrderLine objects:
Line: 1, Product: widget, Quantity: 5, Cost: £ 15.00
Line: 2, Product: gadget, Quantity: 2, Cost: £ 10.00
Line: 3, Product: sprocket, Quantity: 10, Cost: £ 70.00
```

4. Remove the order line with line number **2** from ***order1***.

   Call the `printDetails` method of ***order1***. You should get the following output in the BlueJ terminal window:

   ```
   This Order object (description: example order) contains the
   following OrderLine objects:
   Line: 1, Product: widget, Quantity: 5, Cost: £ 15.00
   Line: 3, Product: sprocket, Quantity: 10, Cost: £ 70.00
   ```

   You have now completed your `Order` and `OrderLine` classes.

   THINK ABOUT IT: We haven't modified the Order class very much. What properties and methods do you think we could have put in that class to make it represent an order more realistically?

Follow up:

Open assignment *Lab 4* in Blackboard.

Copy and paste your Java code for the `OrderLine` class into the appropriate box in the assignment, and answer the two questions which follow.

Task 2: Creating the DiscountOrderLine class

The class `DiscountOrderLine` will be a **subclass** of `OrderLine`, and is an example of the concept of **inheritance** in object-oriented programming.

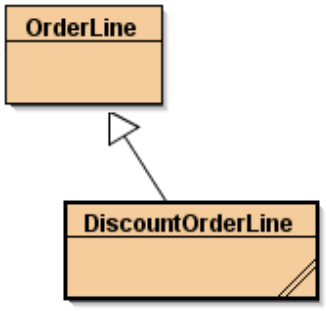
Creating a subclass:

1. Add a new class called `DiscountOrderLine` and open it in the editor.

2. Make this class a **subclass** of `OrderLine` by modifying the class declaration to:

```
public class DiscountOrderLine extends OrderLine
```

3. Compile the new class and check that the relationship between `OrderLine` and `DiscountOrderLine` is shown in BlueJ like this:



4. Delete the existing sample code within `DiscountOrderLine`.

5. Add a new instance variable of type char, called `discountCode`. Note that a `DiscountOrderLine` object now has this new instance variable and also the instance variables of the `OrderLine` class which it **inherits**.

6. Add a new constructor which takes four parameters:

   - `lineNumber` of type int
   - `product` of type Product
   - `quantity` of type int
   - `discountCode` of type char

   The constructor should use the `discountCode` parameter to set the relevant instance variable

   It should pass the other parameters to the constructor of the superclass, `OrderLine`, like this:

```
super(lineNumber,product,quantity);
```

7. Add a new method called `getDiscount` which takes no parameters and has a return type of double.

   Write code in this method to return a value which represents a percentage discount. The value returned should depend on the `discountCode` instance variable, as follows:

discountCode	getDiscount
A	5
B	10
C	15

8. Add a method called `getCost` which takes no parameters and returns a double value. This method has the same signature as a method in `OrderLine`, and overrides that method.

   As in `OrderLine`, this method should multiply the cost of the product by `quantity`, and should **also apply the percentage discount** obtained by calling `getDiscount` before returning the final value.

9. Compile the `DiscountOrderLine` class. You should get a compilation error:

   product has private access in OrderLine

10. Open the `OrderLine` class in the editor. Change the key word in the declaration of the instance variables `product` and quantity from **`private`** to **`protected`**. This gives subclasses (but not any other classes) direct access to these variables.

    Both classes should now compile successfully.

Testing the subclass:

1. Create the following objects in the Object Bench

   product1
   new `Product`, `productCode` = “P1”, `productName` = “widget”, `cost` = 5.00

   discount1
   new `DiscountOrderLine`, `lineNumber`=1,`product` = ***product1***, `quantity` = 10, discountCode = ‘B’

2. Call the `getCost` method of ***discount1***. Check that a 10% discount has been applied, giving a cost of 45.00.

   This demonstrates that the `DiscountOrderLine` class is working correctly. Now we need to test that it works with the `Order` class.

3. Create the following additional objects in the Object Bench
   order1
   new `Order`, `description` = "example order"

   orderLin1
   new `OrderLine`, `lineNumber`=2,`product` = ***product1***, `quantity` = 10

   Note that this is the same product and quantity as in the discount order line.

4. Add ***orderLin1*** to ***order1***

5. Add ***discount1*** to ***order1*** – you can do this as a discount order line **is-a** type of order line.

6. Call the `printDetails` method of ***order1***. You should get the following output in the BlueJ terminal window:

```
This Order object (description: example order) contains the
following OrderLine objects:
Line: 2, Product: widget , Quantity: 10, Cost: £ 50.00
Line: 1, Product: widget , Quantity: 10, Cost: £ 45.00
```

   this is actually a DiscountOrderLine object

   This demonstrates that an `Order` can contain a mixture of `OrderLine` and `DiscountOrderLine` objects – this is **polymorphism** in action!

Follow up:

Continue with assignment *Lab 4*in Blackboard.

Copy and paste your Java code for the `DiscountOrderLine` class into the appropriate box in the assignment, and answer the two questions which follow.

If you have not completed all tasks, then paste the code as it is at the point you have reached.

Extra Task: A GUI application which uses these classes

In this lab you have created and tested objects, but you have not built an application. If you wish, you can try out a simple GUI application which makes use of the classes you have developed.

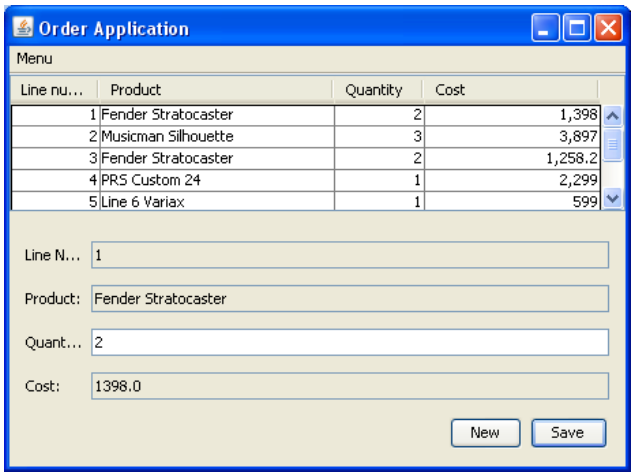
This application displays the order lines for an order, and allows new order lines to be added and the quantity value in existing order lines to be updated. It's not really a complete application, but might be a window for editing orders which is forms part of a larger application.

1. Add two new classes `OrderForm` and `About` to your project. Replace the code in these with the contents of the files *OrderForm.java* and *About.java* which you can download.

   The code in these classes is rather complicated, so don't worry about how it all works. However, you should find and look at the `createOrder` method. You can see that this uses your own classes to create some objects.

   Compile the new classes.

2. Right-click on `OrderForm` in the BlueJ class diagram and call the *main* method. Click OK in the Method Call dialog. An application window should appear, displaying details of the objects created in the `createOrder` method.



   You can experiment viewing, updating and adding order lines. Call the Menu > Exit option to quit the application.

3. Select the Project > Create Jar File... menu option in BlueJ. Choose `OrderForm` as the main class in the Create Jar File dialog and save on the desktop as *orderform.jar*.

4. Find the *orderform.jar* icon on the desktop and double-click to run the application.