INST2002 Programming 2:
Coursework Specification

Set by: Luke Dickens

Important Notice

This assessment forms part of your degree assessment. It must be done entirely on your own from start to finish:

- You must not collaborate or work with other students at any stage.
- You must not send or show other students your answers.
- You must not ask other students for help, or ask to see their answers. As well as being against regulations, this is unfair to the other student concerned, since it may lead to them being accused of plagiarism.
- You must not seek help from friends, relatives, online discussion groups other than the moodle forum for INST2002
- If you think any of the description of the task below is ambiguous or unclear, please post to the moodle forum, explaining what your concerns are, or raise it in person with your lecturer, Luke Dickens, or a INST2002 lab demonstrator.
- If you are unsure of any of the above points, please post your concern to the moodle forum.

Finally, if there is any reason you do not think you can complete this assessment in the alloted time, you should either make a formal request for an extension with your home department, or discuss your reasons with the INST2002 lecturer, Luke Dickens at `l.dickens@ucl.ac.uk`.

1 Admin & Submission

This section contains detailed instructions on where to find the provided code, how to submit your assessment, as well as some guidance.

1.1 Some Rules and Advice

To get the programme to work, you will need to define all the classes and methods described in this document. However, you are advised to do this in stages. Attempt the code in Section 3 before going on to Section 4. Implement one class at a time, and compile often, correcting compiler errors as you go. You should first compile individual classes to ensure they are self-consistent before trying to compile higher level classes (those that use other classes you have written). Remember that you

cannot be given any marks if you do not attempt something (but you can comment out code that doesn't compile).

For some methods, you have been provided with signatures either in the code or in this document. Where this is the case, you should write methods that match these signatures. This means:

- **Do not change** the return type.
- **Do not change** the visibility, e.g. `public`.
- **Do not change** the number or type of input arguments.
- **Do not change** whether a method is `static` or not.

In other cases, you will need to make a descision on what the best signature is for your method, e.g. the `equals` methods, or the constructor for `UnitaryOrder`. **You may also write your own additional methods** in any way you see fit. In some cases, it may be easier to solve the problem if you define helper methods, and you will be given credit if these are used appropriately.

You may also add attributes to any of the classes. In particular, you will have to add attributes to the `PreboxedProduct`, `PackagedProduct` and `UnitaryOrder` classes. You can add any additional attributes you deem appropriate.

1.2 Provided Code

If you go to moodle, you can access a local copy of some support code. You should use this code as a starting point for your system, but you will need to write additional files, e.g. classes etc, for your system. It is important that your final code compiles. Submissions with non-compiling code will be capped at a maximum mark of 50%. If you write something that causes the compiler to complain, then you can comment it out but leave it in place, so the markers can see what you attempted.

There is more than one way to solve this problem, and where you meet the specification you will be given credit up to a maximum of 70%: 40% for the product-list classes, and 30% for the basket classes. Additional marks will be given for coding style and comments, up to a maximum of 30%. This includes, but is not limited to:

- appropriate properties of attributes, e.g. `public`, `private`, `protected`, `static` and `final`
- appropriate use of control flow, e.g. loops, `if`/`switch` statements and recursion
- appropriate variables with good names
- code reuse and minimising duplication of code
- appropriate helper methods & additional classes
- appropriate use and handling of exceptions
- meaningful and appropriate comments
- consistent formatting of code.
- good package structure

1.3 When you are ready

When you are ready, upload a zip file of your code. This should contain all your code in a single zip file, and should preserve the folder structure that matches your package structure. Upload this to the submission webpage of moodle.

2 An Online Food Store's Product List

For this assessment, you must write code to support an online dry-food delivery service. The store has three different kinds of *products*, including *loose-products* (those sold by weight), *preboxed-products* (those already in boxes for delivery), and *packaged-products* (those in packages, that must be packed into boxes for delivery). When your code is fully implemented, the main class `FoodStoreProg` will run a simple interactive interface for a user: to see products available to them at your online-store; to select one or more products, by weight or quantity, and add them to their basket; and to see a running subtotal, postage price, and total price of their basket. Some of the code in `FoodStoreProg` has been commented out, to help you to solve part of the problem, before moving on to the other part. Initially, `FoodStoreProg` will only print a product-list to screen.

You should begin by completing the implementation of `ProductList`, `Product` and the subclasses of `Product`, the requirements are described in more detail in Section 3. After which, you should attempt to implement `Basket`, `BasketItem` and the implementing classes of `BasketItem`, the requirements are described in more detail in Section 4. You have also been given a class `PriceFormatter` which gives you a simple way of converting `int` prices into `String`s. (You do not need to edit this class.) **Until you complete Section 3, you do not need to edit `FoodStoreProg`. When you get to Section 4, you should uncomment the remaining code in `FoodStoreProg`.**

2.1 Postage

One of the key challenges in this coursework is calculating appropriate postage cost for a customer's order. This section outline the rules that you should follow to achieve this. Throughout the coursework description, the text will refer you back to this section when it is relevant.

The online store follows the following rules when posting out products to customers:

- The online store uses Royal Mail's parcel service to fulfil its deliveries.
- Deliveries are sent out by Royal Mail, either as *small parcels* or as *medium parcel* (we call these non-small parcels).
- The store only delivers to the UK, and the postage costs for parcels of various weights are shown in Table 1.
- Different products are always sent out in separate parcels.
- Loose-products are always sent in small parcels, and as many full weight (2kg) parcels as possible are used. (You can assume 2kg of any loose product will fit in a small parcel.)
- Preboxed-products can either be in small or non-small parcels. If in a small parcel, then the weight will always be under 2kg. If in a non-small parcel, then the weight will be under 20kg.

parcel	max weight	postage cost
small	2kg	£3.95
non-small	2kg	£6.05
non-small	5kg	£14.85
non-small	10kg	£21.35
non-small	20kg	£29.65

Table 1: Royal Mail's UK parcel rates

- Packaged-products are packed into either small or non-small parcels and:
  - For each packaged product, there is a known maximum quantity that will fit into a small parcel, and a (larger) maximum quantity that will fit into a non-small parcel.
  - If a packaged product order can be fit into a single small parcel, then it will be.
  - When a customer orders a larger quantity of a packaged product, as many full, non-small parcels will be sent as possible.
  - When it is no longer possible to fill a non-small parcel: the remainder will be sent in a single small parcel if possible; otherwise, a partially full, non-small parcel will be used.

  For instance, one packaged-product is the *Dried mango pack*. Each pack weighs 0.13kg; a maximum of 7 packs can fit into a small-parcel; and a maximum of 100 packs can fit into a non-small parcel. If a customer orders:
  - 7 or fewer packs – will be packed into a single small parcel
  - $8-100$ packs – will be packed into a single non-small parcel
  - $101-107$ packs – one full non-small parcel and a small parcel will be used.
  - $108-200$ packs – two non-small parcels will be used (at least one will be full).
  - ...

3 Creating the Product-List

A product-list can contain different kinds of *products*, including *loose-products*, *preboxed-products*, and *packaged-products*. To capture this, you should complete the `LooseProduct` and `ProductList` classes, as well as writing two additional classes `PreboxedProduct` and `PackagedProduct`.

Look at the UML class diagram in Figure 1. This shows the classes relevant to this part of the coursework. The green class has already been implemented for you, the yellow classes have been partly implemented but require some additional implementation from you, and the orange classes should be fully implemented by you. In particular, you will need to implement `PreboxedProduct` and `PackagedProduct` classes from scratch.

The class `LooseProduct` has been partly implemented for you, and `extends` the `Product` `abstract` class. This is a good place to start. You will then need to write `PreboxedProduct` and `PackagedProduct` from scratch. You can use the existing content of `LooseProduct` to help with this. However, you

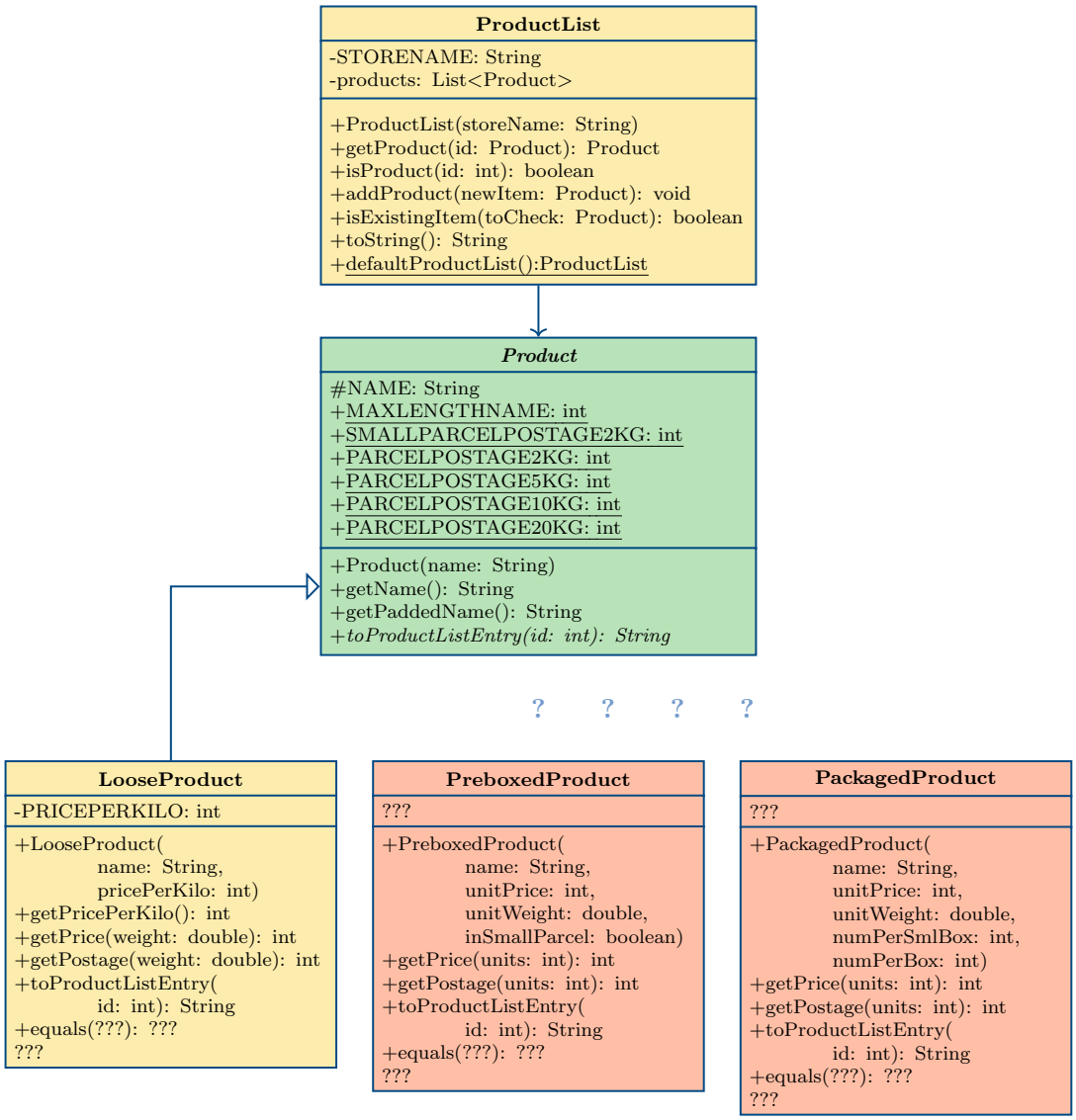

Figure 1: A UML class diagram of the `ProductList` class and its related classes and interfaces.

need to decide whether `PreboxedProduct` and `PackagedProduct` are *direct* subclasses or whether you should structure the class hierarchy differently. When these three classes are implemented, you should implement the `getDefaultProductList` static method in `ProductList`. This will populate a new `ProductList` with examples of the three kinds of products.

Once you have implemented the classes in the section, you should be able to compile and run the code, and it will create an example product-list then print it to the screen. The roles of the different classes in Figure 1, and the changes you have to make, are described in more detail below.

3.1 The `Product` class

`Product` is an `abstract` class that represents items that can appear on the product-list. **You do not have to make any changes to this class.** The attributes have the following meanings:

- `NAME` – represents the product name (or a very short description)
- `MAXLENGTHNAME` – maximum length of the product name
- `SMALLPARCELPOSTAGE2KG` – postage cost of small parcels under 2kg in weight
- `PARCELPOSTAGE2KG` – postage cost of standard (non-small) parcels under 2kg in weight
- `PARCELPOSTAGE5KG` – postage cost of standard (non-small) parcels under 5kg in weight
- `PARCELPOSTAGE10KG` – postage cost of standard (non-small) parcels under 10kg in weight
- `PARCELPOSTAGE20KG` – postage cost of standard (non-small) parcels under 20kg in weight

Key methods in this class are:

- A constructor that takes 1 argument – the `name` of the product.
- `toProductListEntry` – an `abstract` method. Concrete subclasses of `Product` should implement `toProductListEntry` so that it returns a line of text showing the product name and various other details for the item (this is described in more detail for each extending class).

The constructor is already implemented. Additionally, a standard getter method, and the method `getPaddedName` have also been implemented for you. `getPaddedName` produces a fixed length string (24 characters) containing the `Product`'s name, and is to help you write the `toProductListEntry` method. While there is no need to edit this class, you may find that it is a good place to define helper methods for other classes. This is left to your judgement.

3.2 The `LooseProduct` class

`LooseProduct` is a direct concrete subclass of `Product` that represents loose (unpackaged) products available at your store. Customers order these by weight, and for simplicity, we allow any positive weight to be ordered. As well as inheriting a name, a `LooseProduct` records the price per kilo (in the attribute `PRICEPERKILO`). Some details of the key methods are as follows:

- A constructor that takes arguments: `name` and `pricePerKilo` (with the expected meanings).
- `getPricePerKilo` – a simple getter method for the `PRICEPERKILO` attribute.

- `toProductListEntry` – This overrides the corresponding `abstract` method in `Product`. It takes an `int` input argument `id`, and returns a line of text showing (in the following order): the product id, the name, the string `"(per kg)"`, and the price per kilo. For example:

```
LooseProduct loose1 = new LooseProduct("Star Anise", 1990);
LooseProduct loose2 = new LooseProduct("Methi Seeds", 2490);

System.out.println(loose1.toProductListEntry(6));
System.out.println(loose2.toProductListEntry(17));
```

  should produce output similar to:

```
006: Star Anise              (per kg) 19.90
017: Methi Seeds             (per kg) 24.90
```

- `getPrice` – This takes a `double` `weight` as input, and returns the price, in pence, of the given weight of the loose-product.
- `getPostage` – This takes a `double` `weight` as input, and returns the postage price, in pence, of the given weight of the loose-product. Refer to Section 2.1 for a description of how this is to be calculated.
- `equals` – You should implement a standard equals method to allow any `LooseProduct` to be compared to any other object. This should return true if and only if the other object is a `LooseProduct` and has identical attribute values. **You may choose to implement an additional method to adhere to good coding style.**

Look at the code or Figure 1 for more details.

3.3 The `PreboxedProduct` class

You will need to create this class and write all the specified methods. You may add any fields or additional methods you require. `PreboxedProduct` is a concrete subclass of `Product` that represents preboxed products (those already in boxes suitable as parcels). Customers order these in terms of units (which must be whole numbers). As well as inheriting a name, a `PreboxedProduct` must record: the unit-price, the unit-weight, and whether it is sent as a small-parcel or not. The details of the required methods are as follows:

- A constructor that takes arguments: `name`, `unitPrice`, `unitWeight`, and `inSmallParcel` with the expected meanings. (See the UML diagram for the type information.)
- Simple getter methods for any attributes.
- `toProductListEntry` – This overrides the corresponding `abstract` method in `Product`. It takes an `int` input argument `id`, and returns a line of text showing (in the following order): the product id, the name, the weight in brackets and the unit price. For example, the code:

```
PreboxedProduct prebox1 = new PreboxedProduct(
                                  "Red Camargue Rice", 1400, 2.0, true);
PreboxedProduct prebox2 = new PreboxedProduct(
                                  "Kettle Barbeque", 3500, 5.0, false);

System.out.println(prebox1.toProductListEntry(1));
System.out.println(prebox2.toProductListEntry(22));
```

should produce output similar to:

```
001: Red Camargue Rice        (2.00kg) 14.00
022: Kettle Barbeque          (5.00kg) 35.00
```

- `getPrice` – This takes an `int` `units` as input, and returns the price, in pence, of the given number of units of the preboxed-product. **Note that this method has a different signature to `LooseProduct.getPrice` (as it takes an `int` not a `double`).**
- `getPostage` – This takes a `int` `units` as input, and returns the postage price, in pence, of the given number of units of the preboxed-product. Refer to Section 2.1 for a description of how this is to be calculated. **Note that this method has a different signature to `LooseProduct.getPostage` (as it takes an `int` not a `double`).**
- `equals` – You should implement a standard equals method to allow any `PreboxedProduct` to be compared to any other object. This should return true if and only if the other object is a `PreboxedProduct` and has identical attribute values. **You may choose to implement an additional method to adhere to good coding style.**

Look at the code or Figure 1 for more details.

3.4 The `PackagedProduct` class

You will need to create this class and write all the specified methods. You may add any fields or additional methods you require. `PackagedProduct` is a concrete subclass of `Product` that represents packaged products (those needing to be packaged into parcels for delivery). Customers order these in terms of units (which must be whole numbers). As well as inheriting a name, a `PackagedProduct` must record: the unit-price, the unit-weight, and whether it is sent as a small-parcel or not. The details of the required methods are as follows:

- A constructor that takes arguments: `name`, `unitPrice`, `unitWeight`, `maxPerSmlParcel` and `maxPerParcel` (for non-small parcels) with the expected meanings. (See the UML diagram for the type information.)
- Simple getter methods for any attributes.
- `toProductListEntry` – This has the same requirements `PreboxedProduct.toProductListEntry`, but applied to a `PackagedProduct`. For example:

```
PackagedProduct packaged1 = new PackagedProduct(
                                "Dried Mangosteen Pack", 269, 0.13, 7,
                                    100);
PackagedProduct packaged2 = new PackagedProduct(
                                "Dried Durian Pack", 550, 0.26, 3, 50);

System.out.println(packaged1.toProductListEntry(9));
System.out.println(packaged2.toProductListEntry(10));
```

should produce output similar to:

```
009: Dried Mangosteen Pack    (0.13kg) 2.69
010: Dried Durian Pack        (0.26kg) 5.50
```

- `getPrice` and `getPostage` – These have the same requirements as `PreboxedProduct.getPrice` and `PreboxedProduct.getPostage` respectively, but applied to a `PackagedProduct`.
- `equals` – You should implement a standard equals method to allow any `PackagedProduct` to be compared to any other object. This should return true if and only if the other object is a `PackagedProduct` and has identical attribute values. **You may choose to implement an additional method to adhere to good coding style.**

Look at the code or Figure 1 for more details.

3.5 The `ProductList` class

`ProductList` is a simple class for holding a collection of `Products`, displaying them to screen, and selecting them based on their `id`s. You will need to edit the `createDefaultProductList` to create a number of loose, preboxed and packaged product descriptions for your online store. **You only need to edit one method in this class.** You may add any fields or additional methods you require.

`createExampleProductList` is a static method that creates a new `ProductList` object, and populates it with `PreboxedProduct`, `PackagedProduct` and `LooseProduct` objects. You should choose a name for your online store, and add at least 4 loose products, 3 packaged products and 2 preboxed products to the list (remember to include a variety). There are some examples of preboxed products, packaged products and loose products to guide you; please replace these with your own descriptions.

4 A Customer's Basket

Do not attempt this part until you have attempted to implement the classes required by Section 3. You should now implement the classes that will allow customers to order items from the product-list, and to keep track of these orders. Begin by looking at Figure 2. The interface in green has been provided for you, and does not need any changes. The classes in yellow will require

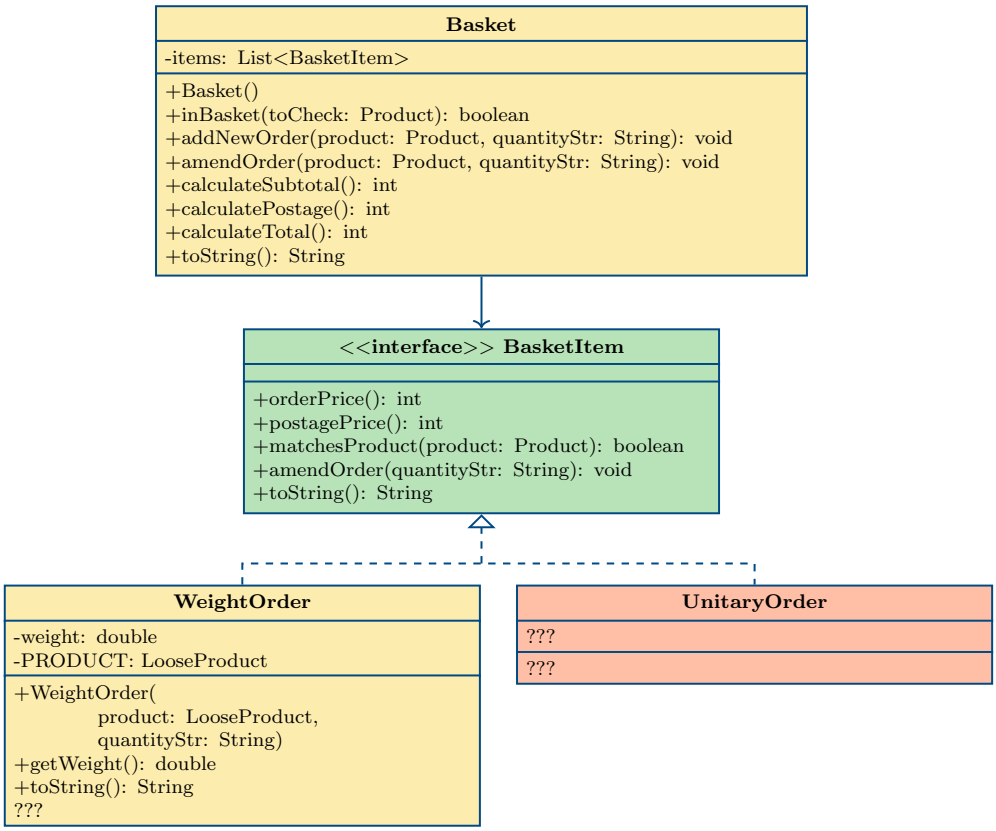

Figure 2: A UML class diagram of the `Basket` class and its related classes and interfaces.

some implementation from you. The class in orange must be written from scratch. **To test your changes in this section you should uncomment the remaining code in `FoodStoreProg`.**

4.1 The `BasketItem` `interface`

This interface has been written for you and you should not make any changes to it.

Any class that `implements` the `BasketItem` `interface` represents an entry in a customers basket and corresponds to an amount of a given `Product`. As described below, this can be `WeightOrder`, which represents an order for some weight of a `LooseProduct`. Alternatively, a basket-item can be for a `UnitaryOrder` which represents an order for a number of units of either a `PreboxedProduct` or a `PackagedProduct`. `WeightOrder` has been partly written for you and is a good place to start. You will need to implement `UnitaryOrder` from scratch.

Key methods that implementing classes will need to define are:

- `orderPrice`: returns the price in pence of the order.
- `postagePrice`: returns the postage price in pence of the order.
- `matchesProduct`: takes any `Product` as input, and returns `true` if this matches the basket-item's `Product`, and `false` otherwise.

- `amendOrder`: takes a String `quantityStr` corresponding to a new amount for the order. `quantityStr` must represent real value for a `WeightOrder`, or an integer value for a `UnitaryOrder`.
- `toString`: returns a String with the order-price followed by a description of the order. The order-entry price should be that calculated by `orderPrice` method. The postage-price is not displayed here.

Hint: You should try to make use of the methods you have already defined in Section 3 when writing the implementing classes. You may also want to make use of the library methods `Integer.parseInt` or `Double.parseDouble` when implementing `amendOrder`.

4.2 The `WeightOrder` class

An object of type `WeightOrder` represents a customer's desire to purchase some weight of a particular `LooseProduct`. This class should implement the `BasketItem` interface. The class has been partly implemented for you. It includes two attributes: `PRODUCT` – the `LooseProduct` being ordered; and `weight` – the ordered weight in kilogrammes.

4.3 The `UnitaryOrder` class

You will need to create this class and write all the specified methods. You may add any fields or additional methods you require.

An object of type `UnitaryOrder` represents a customer's desire to purchase one or more units of a particular `PreboxedProduct` or `PackagedProduct`. This class should implement the `BasketItem` interface, and should store the product and number of units ordered.

4.4 The `Basket` class

`Basket` is a simple class for holding a customer's basket in terms of a collection of `BasketItem`s. It should display this basket to screen, calculate the total price and do so taking account of any discounts. Key methods that you will need to implement in this class are:

- `inBasket`: takes a `Product` as input. Method returns `true` if a basket-item matches this product, and `false` otherwise.
- `addNewOrder`: adds a `BasketItem` to the `Basket`. The input argument `product` represents the `Product` desired. If `product` is a `LooseProduct` then `quantityStr` represents the weight desired. If `product` is a `PreboxedProduct` or `PackagedProduct` then `quantityStr` represents the number of units desired.

  Hint: To check whether the type of a `Product` you will find the `instanceof` keyword helpful. You may also need to downcast.
- `amendOrder`: Changes the order quantity for an existing basket-item. The input arguments have the same meaning as in `addNewOrder`.
- `calculateSubtotal`: takes no arguments and returns the total-price before postage of the current order (in pence).

Hint: You may want to make use of each basket item's `orderPrice` *method.*

- `calculatePostage`: takes no arguments and returns the total-postage price of the current order (in pence), taking account of any discounts applied.

  Hint: You may want to make use of each basket item's `postagePrice` *method.*

A constructor, `calculateTotal` and the `toString` method have been implemented for you.