Challenge - PHP OOP

For this week's challenge we need to write a code for managing a public market. Please consider the following general guidelines when writing your code:

- Do not duplicate your code.
- Try to use the parent methods whenever possible.

Part 1: Create a class named Product.
When instantiating an object from this class, we need to set the values of the following properties: name, price and sellingByKg.

Example: `$orange = new Product('Orange', 35, true);`

Note: the property sellingByKg can accept a value of true or false. If the value is true, that means that the product is sold by kilograms, if the value is false, that means that the product is sold by a gunny sack.

You need to make a method that returns the price for the product.

Part 2: Create a class named MarketStall.
This class should have 1 public property for storing products.
The value for the property products should be set through a constructor.

The property products must only accept an associative array where the keys of the array are the names of the products and the values of the array will be objects of the class Product.
Ex: `$marketStall = new MarketStall( ['orange' => $orange, 'potato' => $potato, 'nuts' => $nuts] )`

Make a method *addProductToMarket* so we can add products to the market after its instantiation. When we call this method, it has to append the object it received to the property products, which, remember, is an associative array.

Make a method getItem that should be called as follows:

```
$marketStall->getItem('orange', 4);
```

When called, this method must check in the products array if the name we have provided exists as a key. If it doesn't, it should return false.

Hint: look for a native php function that checks if a key exists in an array.

If it does exist, it should return an associative array with "amount" and "item" as keys (ex. return ['amount' => 4, 'item' => $item])

The key "amount" will have as value the amount we have provided when calling the method. The key "item" will have the object from the products array corresponding to the name we have provided when calling the method.

Notice: Don't worry about the amount, we don't store how much of each item we have, we only store which products we have. We use the amount value in order to be able to calculate the final price of the cart contents.

Part 3: Create a class named Cart.

This class should have one private property cartItems.

The value for the property cartItmes should be set with a method called addToCart.

This method will be called as follows:

```
$cart = New Cart();
$cart->addToCart( $marketStall1->getItem('orange', 3) );
//As you call the method addToCart, you append each array to $cartItems.
```

Make another method called printReceipt. In this method you will need to implement a logic that checks if the property cartItmes is empty, if it is, it should return: “your cart is empty”, if not, depending on how many products it has, it should print something like this:

```
Raspberry | 3 gunny sacks | total= 1665 denars
Pepper | 1 kgs | total= 330 denars

Final price amount: 1995
```

Part 4: Let's see all this in action!

Create objects from the class Product, each representing a different type of product:

```
new Product('Orange', 35, true);
new Product('Potato', 240, false);
new Product('Nuts', 850, true);
new Product('Kiwi', 670, false);
new Product('Pepper', 330, true);
new Product('Raspberry', 555, false);
```

Create 2 objects from the class MarketStall, each representing a different market stall. Add half of the product types in one market stall and the other in the second market stall.

Add some products to the shopping cart and call the method printReceipt.

Bonus:

Take this a step further and make each product as a separate class, for example class Kiwi, that extends the interface Product, and make all the classes as separate files.

Deadline:

One week after the day of presentation (23:59).