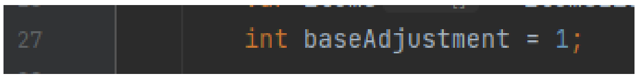
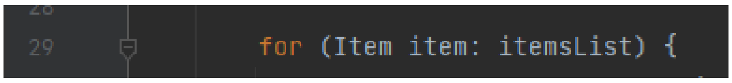
Changes to the “updateQuality” method.

- The integer variable “baseAdjustment” is added and set to 1. This is the normal degrate rate.



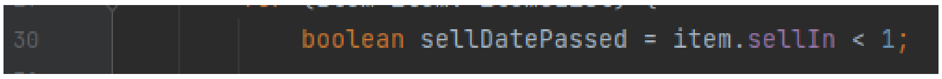
```java
int baseAdjustment = 1;
```

- The for loop is changed from looping through each element by position to looping them directly.



```java
for (Item item: itemsList) {
```

- Inside the for loop, the boolean variable “sellDatePassed” is added and stores TRUE if the sellIn attribute is less than 1, or FALSE otherwise. This help us to identify if the item is expired.

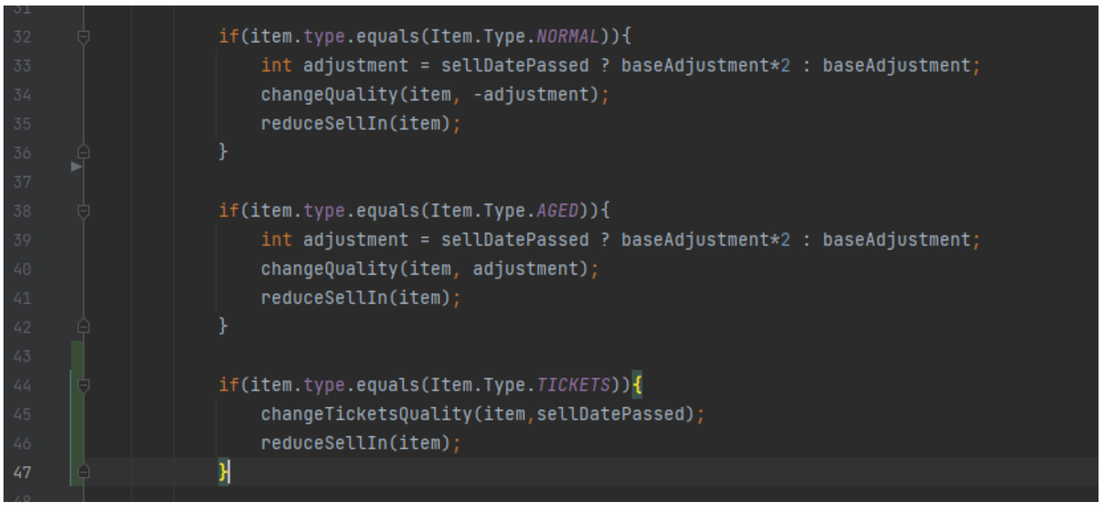


```java
boolean sellDatePassed = item.sellIn < 1;
```

- The method It’s divided into 3 possible cases depending on the type of the item without counting on the LEGENDARY type, because this one shouldn't change.

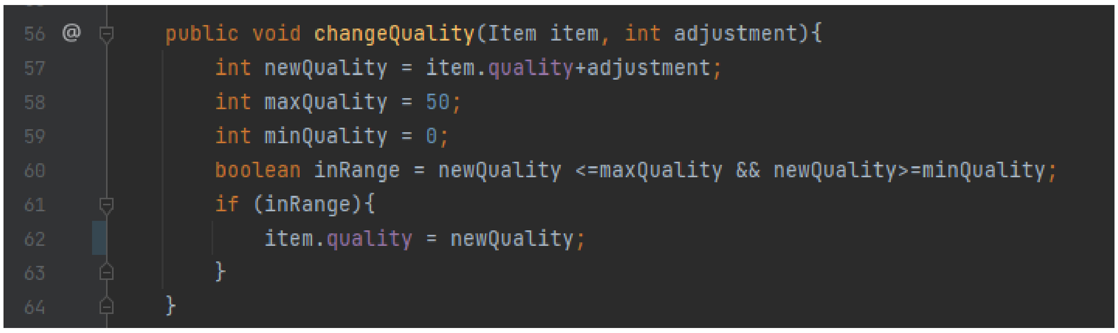
  For the cases of AGED and NORMAL types, an integer variable “adjustment” is created whose value will depend on the boolean variable “sellDatePassed”; if it is FALSE, this variable will be the same as the baseAdjustment variable, or if it is TRUE, it will be the double of the baseAdjustment variable. Then the changeQuality method will be called using the item to update and the adjustment variable (for NORMAL type a negative adjustment, and for AGED type a positive adjustment) as parameters. Finally, the “reduceSellIn “ method is called, which receives the item to update as parameter and reduces its sellIn attribute by 1.

  For the case of TICKETS type, the “changeTicketsQuality” method is called using the item to update and the “sellDatePassed” as parameters, then, it ends calling the “reduceSellIn “ method, which receives the item to update as parameter and reduces its sellIn attribute by 1.



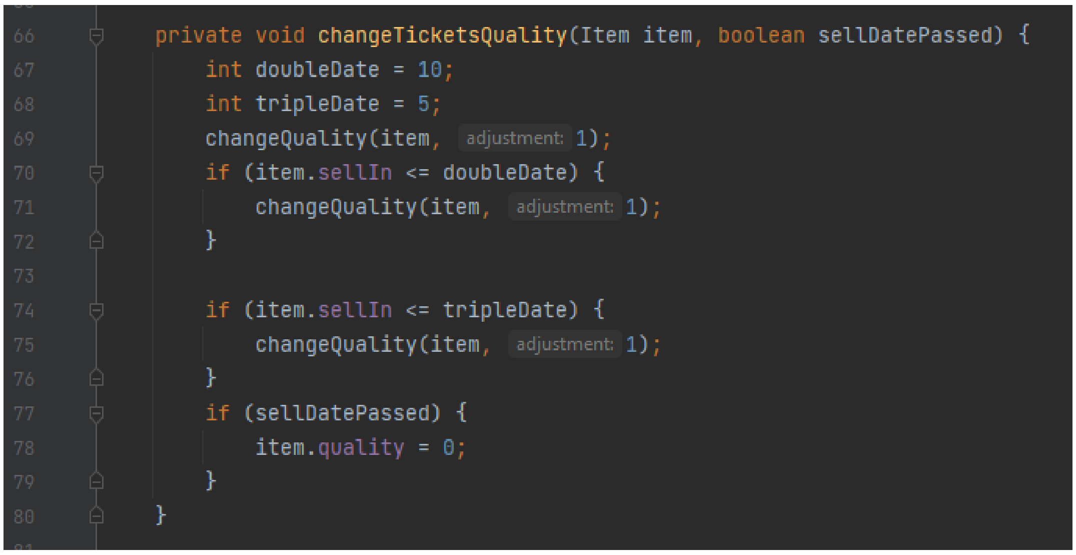
```java
        if(item.type.equals(Item.Type.NORMAL)){
            int adjustment = sellDatePassed ? baseAdjustment*2 : baseAdjustment;
            changeQuality(item, -adjustment);
            reduceSellIn(item);
        }

        if(item.type.equals(Item.Type.AGED)){
            int adjustment = sellDatePassed ? baseAdjustment*2 : baseAdjustment;
            changeQuality(item, adjustment);
            reduceSellIn(item);
        }

        if(item.type.equals(Item.Type.TICKETS)){
            changeTicketsQuality(item,sellDatePassed);
            reduceSellIn(item);
        }
```

- The changeQuality method is created and uses as its parameters an instance of Item class and an integer. It has newQuality,maxQuality and minQuality as its integer variables. The newQuality variable stores the item quality plus the desired adjustment. The minQuality and maxQuality variables store the lower and top limit that the item quality can not surpass. Also, a boolean variable "inRange" that checks if "newQuality" is between "maxQuality" and "minQuality", then an "if" that apply the adjustment if "inRange" is TRUE.



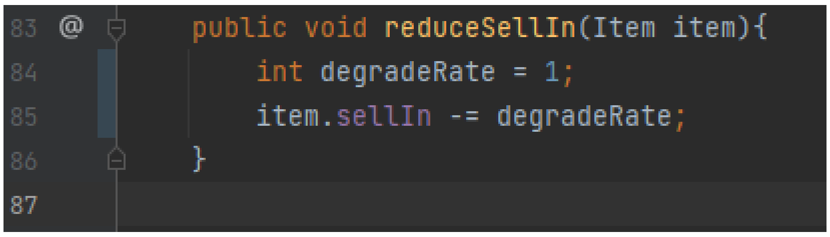
```java
    public void changeQuality(Item item, int adjustment){
        int newQuality = item.quality+adjustment;
        int maxQuality = 50;
        int minQuality = 0;
        boolean inRange = newQuality <=maxQuality && newQuality>=minQuality;
        if (inRange){
            item.quality = newQuality;
        }
    }
```

- The changeTicketsQuality method has as integer variables "doubleDate" that establishes in 10 the number of remaining days in which the quality is going to be increased by 2, and tripleDate in 5 days for when the increase is by 3. Then, the changeQuality method is called that receives the item to update and the adjustment to be made set to 1. Additionally, it proceeds to check if it fulfills the conditions that it has 10 days or less and/or 5 days or less and does the adjustments via the changeQuality method, or if it is already expired proceed to set the quality to 0.



```java
private void changeTicketsQuality(Item item, boolean sellDatePassed) {
    int doubleDate = 10;
    int tripleDate = 5;
    changeQuality(item, adjustment: 1);
    if (item.sellIn <= doubleDate) {
        changeQuality(item, adjustment: 1);
    }

    if (item.sellIn <= tripleDate) {
        changeQuality(item, adjustment: 1);
    }
    if (sellDatePassed) {
        item.quality = 0;
    }
}
```

- The method reduceSellIn is in charge of reducing the sellIn variable of the item, using the degradeRate that in all cases should be 1.



```java
@   public void reduceSellIn(Item item){
        int degradeRate = 1;
        item.sellIn -= degradeRate;
    }
```

Conclusion: In this case, most of the changes obey the CLEAN CODE principles, in order to increase readability, changing the abstract numbers and turn them into variables that represent in a better way the meaning of these.
In the case of SOLID principles the one that is mostly applied is the first one, SINGLE RESPONSIBILITY, since most of the changes and the creation of new methods are focused on doing just one thing, instead of having everything together in a single block of code, we created a method that focus on the changing of quality variable and checks if it satisfies the maximum and minimum values in the requirements. We also created one that checks the tickets conditions that are different from the rest. And in the updateQuality method, we just checked the 3 main possibilities, instead of having a lot of nested Ifs.

In the rest of the code, we try the option of using the second SOLID principle, but to do it, we need it to change the way the items were created, guided by the factory creational pattern, sadly we weren't able to make it work and fix the mistakes that this generated, so we leave it just like it was.
The rest of the principles doesn't apply to the code, since the lack of interfaces and inheritance

As for other CLEAN CODE modifications in the code, we leave it just like it was, because we think it is quite readable and doesn't need urgent changes as needed in the updateQuality method.