# Self-Checkout Machine

## 1 Introduction

This assignment will cover creating, throwing, and handling exceptions as well as review object-oriented programming design principles.

## 2 Problem Description

A new convenience store near Tech called *TechConvenience* is opening soon and wants you to write software for their self-checkout machines. They want these machines to be extremely reliable, so it must handle every possible problem.

There are two special features the store wants you to implement. The first is a way to detect people swapping barcode labels on items. This machine weighs every item as it is scanned. You'll be using this to your advantage.

The second feature they want is a way to prevent competitors from using this software if they somehow got their hands on it.

They already have a server in place that handles some functionality, but unfortunately it is slightly buggy. Create the self-checkout machine software and then a demo to show off all of its features to the convenience store executives.

## 3 Solution Description

A description of each class that you should write follows. Some have been either partially or wholly provided to you. Make sure to check out the Tips section too.

### 3.1 Exceptions

- `ServerException` has been provided to you. The `Server` (see below) throws this exception when something has gone wrong in one of its methods. Take note that this is a *checked* exception.

- `PaymentFailedException` is a checked exception that is thrown when a payment goes wrong.
- `CardExpiredException` is a `PaymentFailedException`. It should be constructed with a `java.time.LocalDate` that represents the expiration date of the card, and should create a message about expiration date, e.g, “Card expired on 2015-01-02.” That message should be used as the `Exception`’s message.
- `NotEnoughFundsException` is a `PaymentFailedException`. When created, it should take in a message documenting that there aren’t enough funds from the specific `PaymentMethod` that threw this exception (see below).
- `InvalidItemException` is a checked exception that should have a simple message along the lines of “This item is not valid.”
- `WrongStoreError` is an *unchecked* exception. It will be thrown if a competing store makes a `CheckoutMachine` object.

## 3.2 Core Structure

### 3.2.1 Partially or Wholly Provided

- `Server` has been provided. While its source isn’t available, you have access to its javadocs to see what it does. You will be calling it from your `CheckoutMachine`. It will read valid items from the “database” (csv file).
- The `PaymentMethod` interface has been provided for you. Everything that can act as a `PaymentMethod` has a method to pay a specified amount of money.
- `Item` has two constructors whose headers have been provided. It has a name, weight, price, and barcode. Make sure to utilize constructor chaining. If unspecified, the name should be `null` and price should be 0.0. `Item` should have a getter method for price called `getPrice()`. It should also have a properly written `equals` method which compares items solely on their weight and barcode. If something goes wrong while paying, it should throw a `PaymentFailedException`.

### 3.2.2 Payment

- `Cash` can be used as a `PaymentMethod`. `Cash` only has one attribute: cashOnHand – the total amount of cash on hand. You can assume this will never be negative. The `pay(double amount)` method should be declared to throw a `NotEnoughFundsException`, which it will throw when there isn’t enough cashOnHand to cover the amount. It will subtract the specified amount from cashOnHand otherwise.

- `Card` can be used as a `PaymentMethod`, but it can't be instantiated. A `Card` has two attributes: ownerName and balance. Its `pay(double amount)` method should check if there is enough of a balance to pay the amount, and remove it from balance if there is. If there isn't enough balance to cover the payment, throw a `NotEnoughFundsException`.
- `CreditCard` is a `Card` that has an additional attribute: `expirationDate`, which is a `java.time.LocalDate`. The `pay` method will throw a `CardExpiredException` if the `CreditCard` is passed expiration. You will need to compare the expirationDate with the current date. If it isn't expired, the `pay` method works just like that of `Card`.
- `BuzzCard` is a `Card`. That's exactly how it works. There is nothing else about it, really.

#### 3.2.3 `CheckoutMachine`

`CheckoutMachine` has a `String` storeName, an `ArrayList<Item>` for validItems, and an `ArrayList<Item>` cart. The storeName is passed in through the constructor, cart starts out as an empty `ArrayList<Item>`, and validItems is gotten from the `Server`. The `Server` may throw a `ServerException`, so you should keep trying until you get the `ArrayList<Item>` of validItems. The storeName should also be checked against the `Server` to see if it is valid. Once again it may throw exceptions, so be sure to handle that. If the storeName isn't valid, the constructor should throw the unchecked `WrongStoreError`.

`CheckoutMachine` has a `scan(Item item)` method that doesn't return anything but may throw an `InvalidItemException`. The `Item` passed in will only have a weight and barcode because that is all the `CheckoutMachine` can determine. It then goes through all the valid items – comparing it with the passed in item. If it finds an `Item` that is equal, it will add the item from the validItems list – *not* the item passed in as a parameter – to the cart. This is important because the item added to the cart is guaranteed to have the right price because it is from the `Server`. If item is not in the validItems list, it throws a `InvalidItemException`.

`CheckoutMachine` has a `getTotalPrice()` method which sums all the prices of the items in the cart and returns that sum. It also has a `payForCart(PaymentMethod method)` method that doesn't return anything but may throw a `PaymentFailedException`. It tries to pay the cart's total price with the specified `PaymentMethod` and if it succeeds, empties the cart.

#### 3.2.4 `Demo`

The `Demo` is a way to show those convenience store executives how your `CheckoutMachine` works and that their money was well spent. You don't have to go all out – non-technical execs are easily impressed by programs – but show them that it handles their intermittently-functioning server and throws its own errors when needed. A simple format may be to print out what you will be testing, test it, catch the exception, and print its message to show that everything worked as expected.

## 4 Tips

- Look at the Java API for `Exceptions`. Take note of what constructors an `Exception` has. Which constructors should you override, if any? Will you be overloading the constructor with new parameters? Which constructor will you be making super calls to?
- Look at the Java API for `java.time.LocalDate`. Several methods here may be useful. The static `now` method returns the current date. The static `of` method allows you to create an arbitrary date, and the methods `isBefore` or `isAfter` may be helpful as well.
- `CreditCard`'s `pay` method does the same thing as its parent's `pay` method if it isn't expired. Is there a way to do this without just copying the relevant code?
- Taking a peek at the CSV file could be useful when you test your codCSV file could be useful when you test your code in `Demo`

## 5 Javadocs

For this assignment you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to have are `@author, @version, @param,` and `@return`. Here is an example of a properly Javadoc'd class:

```java
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

   /**
    * Creates an awesome dog (NOT a dawg!)
    */
   public Dog(){
      ...
   }

   /**
    * This method takes in two ints and returns their sum
    * @param a first number
    * @param b second number
    * @return sum of a and b
    */
   public int add(int a, int b){
        ...
    }
}
```

Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@version` tag included. The comments for a class start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

## 5.1 Javadoc and Checkstyle

You can use the Checkstyle jar mentioned in the following section to test your javadocs for completeness. Simply add -j to the checkstyle command, like this:

```
$ java -jar checkstyle-6.2.1.jar -j *.java
Audit done. Errors (potential points off):
0
```

## 6 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **100** points.
Review the Style Guide and download the Checkstyle jar. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-6.2.1.jar *.java
Audit done. Errors (potential points off):
0
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **100** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

## 7 Turn-in Procedure

Submit all of the Java source files you modified and resources your program requires to run to T-Square. Do not submit any compiled bytecode (`.class` files) or the Checkstyle jar file. When you're ready, double-check that you have submitted and not just saved a draft.

**Please remember to run your code through Checkstyle!**

**Verify the Success of Your Submission to T-Square**
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
   (c) Helps find last minute causes of files not compiling and/or running.

[1]Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!