CS126 Design of Information Structures

WAFFLES

Contents

Introduction

Welcome to the CS126 coursework **WAFFLES**, **Warwick's Amazing Fast Food Logistic Engagement Service**. WAFFLES is a web application for hosting restaurant information, customer reviews and customer favourites. Interesting fact, the alternative name for this coursework was going to be WarwickTripAdvisor or Welp but that seemed to be a bit too on the nose.

Task

The WAFFLES website is not quite amazing yet and we need the help of an aspiring programmer to help reach its true potential. This coursework involves you designing and programming the data structures and methods used in the WAFFLES website.

To do so, you are given a zip, `cs126-coursework-waffles.zip`, with the files:

```
cs126-coursework-waffles.zip
└─ /WAFFLES
    ├─ /data
    ├─ /lib
    ├─ /src
    ├─ build.sh
    ├─ guide.pdf
    ├─ pom.xml
    ├─ Report.md
    ├─ run.bat
    └─ waffles.jar
```

These files will help you develop the WAFFLES website. The given `waffles.jar` file is a collection of Java class files which make up the initial WAFFLES website. Alongside that, you are given some script files, `build.sh` and `run.bat`, which enables you to compile and run the website on both **Linux/macOS** and **Windows** machines.

Initially, when running the WAFFLES website, the pages that are displayed contain no information. This is because the current code does not return any useful data for the site to display. The template code we have given you is the code the initial bare-bones WAFFLES website uses. But as you start to implement parts of the WAFFLES site, you will begin to see more and more parts of the WAFFLES website displaying relevant information.

You are tasked with the job of adding and updating files in the `/src` folder to evolve the WAFFLES website. Specifically you will be creating and implementing methods for the classes located inside the following folders:

- `/src/main/java/uk/ac/warwick/cs126/stores`
- `/src/main/java/uk/ac/warwick/cs126/structures`
- `/src/main/java/uk/ac/warwick/cs126/util`

You are free to add any additional `java` files you may use into to these 3 folders.

The files, from `/src/main/java/uk/ac/warwick/cs126`, to be completed are:

- `/stores/CustomerStore.java`
- `/stores/FavouriteStore.java`
- `/stores/RestaurantStore.java`
- `/stores/ReviewStore.java`
- `/util/ConvertToPlace.java`
- `/util/DataChecker.java`
- `/util/HaversineDistanceCalculator.java`
- `/util/KeywordChecker.java`
- `/util/StringFormatter.java`

The `CustomerStore`, `FavouriteStore` and `RestaurantStore` are the main classes for this coursework, you will be working through to complete them and the classes from `util` will help you to do that. The next section Stores gives a short description as to what they do. For an in-depth description to all the methods see the main Stores section. Also, for a helpful checklist of methods you need to complete see the Checklist section. Additionally, we tell you the Restrictions, give you some Advice and we answer the most frequently asked questions in the FAQ section.

Inside the `/structures` folder, we have given you an example structure from the labs, `MyArrayList`. This is used to show how other classes can import this structure. This is simply an example, you are free to modify this or delete this to implement your own structures. For more details on adding your own classes see the How To Add Classes section inside Setup and Installation.

The Setup and Installation section is there to help you get all the tools you need and to teach you how to run this coursework.

Furthermore, since it takes a long time to load the site to see if your code is correct, we have implemented a fast way to test your methods outside of the website. The tests are located in: `/src/main/java/uk/ac/warwick/cs126/test`. We have already written some example tests to help you get started (some are incomplete and are marked as TODO). You can run these tests using the script we gave you with the `-t` argument. For more details, see the Testing section.

The final sections cover the report (Report).

Stores

CustomerStore

The `CustomerStore` class will be used to store all the customers in the form of `Customer` objects. This class helps with

- Retrieving customer information
- Listing customers sorted by name and their ID
- Searching for customers

FavouriteStore

The `FavouriteStore` class will be used to store all the favourites from the customers in the form of `Favourite` objects. This class helps with:

- Retrieving favourite information for restaurants and customers
- Comparing favourites between customers
- Listing most favourited restaurants and which customers favourite the most

RestaurantStore

The `RestaurantStore` class will be used to store all the restaurants in the form of `Restaurant` objects. This class helps with:

- Retrieving restaurant information
- Listing restaurants sorted by name, date established and rating
- Find the closest restaurants to a given location
- Searching for restaurants

Checklist

This section lists all the methods that we require from you and that we test.

CustomerStore.java

- □ `boolean addCustomer(Customer customer)`
- □ `boolean addCustomer(Customer[] customers)`
- □ `Customer getCustomer(Long id)`
- □ `Customer[] getCustomers()`
- □ `Customer[] getCustomers(Customer[] customers)`
- □ `Customer[] getCustomersByName()`
- □ `Customer[] getCustomersByName(Customer[] customers)`
- □ `Customer[] getCustomersContaining(String searchTerm)`

FavouriteStore.java

- □ `boolean addFavourite(Favourite favourite)`
- □ `boolean addFavourite(Favourite[] favourites)`
- □ `Favourite getFavourite(Long id)`
- □ `Favourite[] getFavourites()`
- □ `Favourite[] getFavouritesByCustomerID(Long id)`
- □ `Favourite[] getFavouritesByRestaurantID(Long id)`
- □ `Long[] getCommonFavouriteRestaurants(Long customer1ID, Long customer2ID)`
- □ `Long[] getMissingFavouriteRestaurants(Long customer1ID, Long customer2ID)`
- □ `Long[] getNotCommonFavouriteRestaurants(Long customer1ID, Long customer2ID)`
- □ `Long[] getTopCustomersByFavouriteCount()`
- □ `Long[] getTopRestaurantsByFavouriteCount()`

RestaurantStore.java

- □ `boolean addRestaurant(Restaurant restaurant)`
- □ `boolean addRestaurant(Restaurant[] rs)`

```
□ Restaurant getRestaurant(Long id)

□ Restaurant[] getRestaurants()

□ Restaurant[] getRestaurants(Restaurant[] rs)

□ Restaurant[] getRestaurantsByName()

□ Restaurant[] getRestaurantsByDateEstablished()

□ Restaurant[] getRestaurantsByDateEstablished(Restaurant[] rs)

□ Restaurant[] getRestaurantsByWarwickStars()

□ Restaurant[] getRestaurantsByRating(Restaurant[] rs)

□ RestaurantDistance[] getRestaurantsByDistanceFrom(float lat,
                                                    float lon)

□ RestaurantDistance[] getRestaurantsByDistanceFrom(Restaurant[] rs,
                                                    float lat,
                                                    float lon)

□ Restaurant[] getRestaurantsContaining(String searchTerm)
```

ConvertToPlace.java

```
□ Place convert(float latitude, float longitude)
```

DataChecker.java

```
□ Long extractTrueID(String[] repeatedID)

□ boolean isValid(Long id)

□ boolean isValid(Customer customer)

□ boolean isValid(Favourite favourite)

□ boolean isValid(Restaurant restaurant)
```

HaversineDistanceCalculator.java

```
□ static float inKilometres(
          float lat1, float lon1, float lat2, float lon2)

□ static float inMiles(
          float lat1, float lon1, float lat2, float lon2)
```

StringFormatter.java

```
□ static String convertAccentsFaster(String str)
```

Restrictions

We require:

- `/stores/CustomerStore.java`
- `/stores/FavouriteStore.java`
- `/stores/RestaurantStore.java`
- `/stores/ReviewStore.java`
- `/util/ConvertToPlace.java`
- `/util/DataChecker.java`
- `/util/HaversineDistanceCalculator.java`
- `/util/KeywordChecker.java`
- `/util/StringFormatter.java`

You **cannot** change the location of any of the Java files we require.

You **cannot** create any new folders in `/src/main/java/uk/ac/warwick/cs126`.

You are **not** allowed to create new folders inside any of the `/stores`, `/structures` or `/util` folders to store anything.

You **cannot** change whether a class implements an interface in any of the required files.

You **cannot** add any files into the `/interfaces` folder or `/models` folder. So this means you **cannot** add any additional interface or model into the `/interfaces` folder or `/models` folder. We will not take anything that reside in those folders to mark. If you need to add an interface or model, add it into one of the 3 allowed folders, `/stores`, `/structures` or `/util` folder.

You **cannot** modify any existing interface or model source file that we have given you. So do **not** edit any source file from the `/interfaces` folder or `/models` folder.

You may **not** change the code of any `load*DataToArray(InputStream resource)` method from the `/stores` or `/util` classes.

You **cannot** import any of the classes from the `/test` folder in your main code. So do not call `import uk.ac.warwick.cs126.test.*;` in any code from the `/stores`, `/structures` or `/util` folders.

You are **not** allowed to use any pre-implemented data structure from the `java.util` package. Specifically, but not limited to, the following classes:

> `ArrayList`, `Arrays`, `HashMap`, `HashSet`, `Hashtable`, `IdentityHashMap`, `LinkedList`, `LinkedHashMap`, `LinkedHashSet`, `TreeMap`, `TreeSet`, `WeakHashMap`, `Vector`.

In general, you are expected to implement data structures from scratch.

You are **not** allowed to import the `java.util.Collections` package.

You **are** allowed to use `interfaces` and `exceptions` from `java.util`.

You **are** allowed to implement any of the `interfaces` found within the Java Collections Framework such as `Iterator`, `Enumeration`, `Comparable`, `List`, or `Map`.

Advice

Order of Approach

The stores vary in difficulty but are closely related.

The `CustomerStore` is closely related to the `RestaurantStore`, in a sense they have similar method designs. You can think of `RestaurantStore` being the older sibling of `CustomerStore`.

The easiest store to begin with is the `CustomerStore`, as this store has less methods to implement compared to the other stores. By completing this store first, it will give you a foundation for the other stores, especially the `RestaurantStore`.

Next should be the `RestaurantStore`, the methods are very similar to the ones in `CustomerStore` but this store differs by having more types of sorts as well as a more complicated search method.

The `FavouriteStore` differs from the previous two in that the add function is slightly more complicated, specifically, it now introduces the fact that you can replace existing objects. The main methods for the `FavouriteStore` are basically `Set` operations which you need to implement. The other methods are data retrieval tasks.

Implementation

The coursework can be completed using very simple data structures, but to get a good mark you need to improve on them and understand where and when to implement them. Below are some structure dilemmas you may face:

- `ArrayList` - It is fast for modifying elements. But searching is slow if the structure is unsorted. Keeping this structure sorted is a costly operation, and deletion in a sorted array is a very expensive operation.
- `Map` - Provides good performance for accessing elements. But you will need to account for collisions and the load maintain the good performance.
- `Tree` - Keeps data sorted and it also provides good performance on insertion and deletion. But, they will need to be balanced to maintain good performance.

As you can see there are pros and cons to every data structure, no single one is a silver bullet. Perhaps, you should not restrict yourself to one type of data structure. To decide which to use, think about what a method is asking from you, and depending on its use case, decide if the method needs to be fast, space efficient, both, or neither.

There are many ways that you can approach this coursework, finding and justifying the one that makes the most sense to you is all part of the challenge.

How To Run

We have given you scripts to help compile and run the coursework. Use `build.sh` for **Linux/macOS** (DCS machines) and use `run.bat` if you are on **Windows**.

We named the scripts different to make it easier to autocomplete in the **Linux/macOS** terminal, so when you want to run the script type `./b` and then press tab, it will expand it to `./build.sh`.

For **Windows**, the script name short instead. The tab auto-complete also works on this OS but in Command Prompt it clashes with `Report.md`, it does not clash in PowerShell. In the Command Prompt case, it is easier to type out the 3 characters.

To make it executable on **Linux/macOS** (DCS machines):

```
chmod +x build.sh
```

Then you should be able to run `-h` to see the script's documentation:

```
./build.sh -h
```

In **Windows**, double click the `.bat` file and it should open up a Command Prompt window with the script's documentation. Alternatively, in Command Prompt:

```
run -h
```

And in **Windows** PowerShell the syntax is:

```
.\run.bat -h
```

Website

To run the WAFFLES website on **Linux/macOS** (DCS machines):

```
./build.sh -r
```

This will compile your source code and use it for the WAFFLES website. The website will then be run on port **8080**. You can access it by going to `http://localhost:8080/` on your preferred web browser.

If port **8080** is in use, you can try a different port with the command:

```
./build.sh -r 9090
```

This will try to compile and run the WAFFLES website on port **9090**.

Note, if you update your source code, you need to close the website and re-run the command so that it uses your new source code.

In **Windows** Command Prompt, the commands are respectively:
`run -r` or `run -r 9090`

In **Windows** PowerShell, the commands are respectively:
`.\run.bat -r` or `.\run.bat -r 9090`

Finally, if you wish to run the initial bare-bones website, which does not use your code:

```
java -jar waffles.jar
```

Compilation - Script

If you wish to compile all the `.java` files in `stores`, `structures` and `util`. You can use the following command:

```
./build.sh -b
```

This compiles all the classes into the `target/classes` folder.

Note, you do not need to do call this before calling the run website function of the script, as that already compiles your code in the process.

Now, why would you want to use this command? It depends on if you wish to debug classes outside of how Testing does it, if so, then having a compiled class means you can run its main method.

So, to run a main method of a class after compiling, use the following syntax:

```
./build.sh -j uk.ac.warwick.cs126.test.TestRunner
```

Here this would run the `public static void main(String[] args){}` method of the `TestRunner` class.

So if you want to run the `CustomerStore` class individually, and you have set up a main method for it, you can call the following:

```
./build.sh -b
```

```
./build.sh -j uk.ac.warwick.cs126.stores.CustomerStore
```

In **Windows** Command Prompt, the commands are:

```
run -b
```

```
run -j uk.ac.warwick.cs126.stores.CustomerStore
```

In **Windows** PowerShell, the commands are:

```
.\run.bat -b
```

```
.\run.bat -j uk.ac.warwick.cs126.stores.CustomerStore
```

Compilation - Manual

Now, if you want to compile source files manually rather than use the script, as shown above, see this section. We are showing how to compile and run `TestRunner.java` using the `javac` and `java` commands directly.

Do the following for **Linux/macOS**:

First, we make a folder to store our compiled classes in:

```
mkdir -p target/classes/
```

We include the `-p` argument here for `mkdir` as it makes the parent directories if they do not exist.

Now you can compile `TestRunner.java`:

```
javac -d target/classes/ -encoding "UTF-8" -cp src/main/java/:lib/commons-
io-2.6.jar:lib/cs126-interfaces-1.2.6.jar:lib/cs126-models-1.2.6.jar: src/
main/java/uk/ac/warwick/cs126/stores/TestRunner.java
```

Note, if you try to copy and paste this from the PDF, it will include spaces at the line breaks, so make sure to remove those spaces once pasted.

The `-d` argument above for `javac` specifies where to place the compiled classes, it will not work if the folder does not exist. The `-encoding` argument tells the compiler what format to read the files. The `-cp` or `-classpath` argument tells where to look for class files, each location should be separated by a colon `:`, in this case our class files are in the `src/main/java` folder and the `jar`'s.

Finally, to run the main class of `TestRunner`, see below. The classpath has now been changed to where we compiled the classes to, `target/classes/`, but the `jar`'s remain:

```
java -cp target/classes/:lib/commons-io-2.6.jar:lib/cs126-interfaces-1.2.6
.jar:lib/cs126-models-1.2.6.jar: uk.ac.warwick.cs126.test.TestRunner
```

Note, this is what the `-t` argument in the provided scripts do.

In **Windows** the commands are a bit different, these are shown below. The `mkdir` command automatically makes parent directories if they do not exist. Also, notice how we use backslashes `\` instead, and the class locations are separated by semi-colons `;` instead of colons.

```
mkdir target\classes\
```

```
javac -d target\classes\ -encoding "UTF-8" -cp src\main\java\;lib\commons-
io-2.6.jar;lib\cs126-interfaces-1.2.6.jar;lib\cs126-models-1.2.6.jar; src\
main\java\uk\ac\warwick\cs126\test\TestRunner.java
```

```
java -cp target\classes\;lib\commons-io-2.6.jar;lib\cs126-interfaces-1.2.6
.jar;lib\cs126-models-1.2.6.jar; uk.ac.warwick.cs126.test.TestRunner
```

How To Add Classes

You are allowed to add `.java` files to any of the following folders:

- `stores`
- `structures`
- `util`

You **cannot** put them inside subfolders, only put them at the root of these folders, where all the other example `.java` files reside.

Make sure when you create a new class file you include the `package` at the top, in the first line, before the `import` statements. By adding a `package` statement you specify which folder your class is located in.

Using a `package` statement helps organise your project, it helps group similar classes together with a meaningful package name. Also, it helps avoid any naming collisions if a class has the same name, as then we would put them into different packages.

So, if you create a class in the `stores` folder, use the following `package` statement:

```
package uk.ac.warwick.cs126.stores;

import uk.ac.warwick.cs126.models.*;
import uk.ac.warwick.cs126.util.*;

public class NewStoreClass{
    //Some more code here...
}
```

Likewise, if you create a class in the `structures` folder:

```
package uk.ac.warwick.cs126.structures;

public class NewStructureClass{
    //Some more code here...
}
```

And finally, if you create a class in the `util` folder:

```
package uk.ac.warwick.cs126.util;

public class NewUtilClass{
    //Some more code here...
}
```

Models

CustomerStore

Customer

Method Summary

Modifier	Method Name and Description
`Long`	`getID()` Returns the ID of the `Customer`.
`String`	`getStringID()` Returns the ID of the `Customer` in `String` form.
`String`	`getFirstName()` Returns the first name of the `Customer`.
`String`	`getLastName()` Returns the last name of the `Customer`.
`Date`	`getDateJoined()` Returns the `Date` the `Customer` joined.
`float`	`getLatitude()` Returns the current latitude of the `Customer`.
`float`	`getLongitude()` Returns the current longitude of the `Customer`.
`void`	`setID(Long id)` Sets the ID of the `Customer` to `id`.
`void`	`setFirstName(String firstName)` Sets the first name of the `Customer` to `firstName`.
`void`	`setLastName(String lastName)` Sets the last name of the `Customer` to `lastName`.
`void`	`setDateJoined(Date dateJoined)` Sets the `Date` the `Customer` joined to `dateJoined`.
`void`	`setLatitude(float lat)` Sets the current latitude of the `Customer` to `lat`.
`void`	`setLongitude(float lon)` Sets the current longitude of the `Customer` to `lon`.
`String`	`toString()` Returns a human-readable string representation of the `Customer`.

Constructor

```
public Customer(Long id, String firstName, String lastName,
                Date dateJoined, float latitude, float longitude)
```

Constructs a new `Customer` with the given information.

Parameters:

`id`	-	The ID of the `Customer`.
`firstName`	-	The first name of the `Customer`.
`lastName`	-	The last name of the `Customer`.
`dateJoined`	-	The date the `Customer` joined.
`latitude`	-	The latitude of the `Customer`.
`longitude`	-	The longitude of the `Customer`.

FavouriteStore

Favourite

Method Summary

Modifier	Method Name and Description
`Long`	`getID()` Returns the ID of the `Favourite`.
`String`	`getStringID()` Returns the ID of the `Favourite` in `String` form.
`Long`	`getCustomerID()` Returns the ID of the `Customer` who favourited.
`Long`	`getRestaurantID()` Returns the ID of the `Restaurant` that got favourited.
`Date`	`getDateFavourited()` Returns the `Date` the `Customer` favourited the `Restaurant`.
`void`	`setID(Long id)` Sets the ID of the `Favourite` to `id`.
`void`	`setCustomerID(Long customerID)` Sets the ID of the `Customer` who favourited to `customerID`.
`void`	`setRestaurantID(Long restaurantID)` Sets the ID of the `Restaurant` that got favourited to `restaurantID`.
`void`	`setDateFavourited(Date dateFavourited)` Sets the `Date` the `Customer` favourited the `Restaurant` on to `dateFavourited`.
`String`	`toString()` Returns a human-readable string representation of the `Favourite`.

Constructor

```
public Favourite(Long id, Long customerID,
                 Long restaurantID, Date dateFavourited)
```

Constructs a new `Favourite` with the given information.

Parameters:

`id` - The ID of the `Favourite`.

`customerID` - The ID of the `Customer` who favourited.

`restaurantID` - The ID of the `Restaurant` that got favourited.

`dateFavourited` - The date the `Customer` favourited the `Restaurant`.

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.models.Favourite;
```

```java
// Create a new Favourite object
Favourite f = new Favourite(1112223334445556L,
                            1112223334445557L,
                            1112223334445558L,
                            new SimpleDateFormat("yyyy").parse("2020"));
```

RestaurantStore

Cuisine

Method Summary

Modifier	Method Name and Description
`String`	`toString()` Returns a human-readable string representation of the `Cuisine`.

List of Cuisines

- `Ale`
- `African`
- `American`
- `Brazilian`
- `British`
- `Burger`
- `Cake`
- `Caribbean`
- `Chinese`
- `Cocktails`
- `Dessert`
- `Egyptian`
- `European`
- `FishAndChips`
- `French`
- `Gelato`
- `Greek`
- `Indian`
- `Italian`
- `Jamaican`
- `Japanese`
- `Korean`
- `Lebanese`
- `Malaysian`
- `Mediterranean`
- `Mexican`
- `Moroccan`
- `Pakistani`
- `Persian`
- `Pizza`
- `Polish`
- `Romanian`
- `Salad`
- `Scandinavian`
- `Seafood`
- `Soups`
- `SouthAmerican`
- `Spanish`
- `Steakhouse`
- `Sushi`
- `Tapas`
- `Thai`
- `Turkish`
- `Vietnamese`
- `Wine`

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.models.Cuisine;
```

```java
// Assigning a Cuisine
Cuisine c = Cuisine.SouthAmerican;

// Print Cuisine c, which should come out as "South American"
System.out.println(c);

// Print Cuisine.FishAndChips, which should come out as "Fish And Chips"
System.out.println(Cuisine.FishAndChips);
```

EstablishmentType

Method Summary

Modifier	Method Name and Description
`String`	`toString()` Returns a human-readable string representation of the `EstablishmentType`.

List of Establishment Types

- `Bakery`
- `Bar`
- `Cafe`
- `DessertShop`
- `Diner`
- `FastFood`
- `MarketStall`
- `Pub`
- `Restaurant`
- `SnackBar`
- `StreetFood`
- `Takeaway`
- `Tavern`

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.models.EstablishmentType;
```

```java
// Assigning a EstablishmentType
EstablishmentType e = EstablishmentType.SnackBar;

// Print EstablishmentType e, which should come out as "Snack Bar"
System.out.println(e);

// Print EstablishmentType.StreetFood
// This should come out as "Street Food"
System.out.println(EstablishmentType.StreetFood);
```

Place

Method Summary

Modifier	Method Name and Description
`String`	`getName()` Returns the name of the `Place`.
`String`	`getPostcode()` Returns the postcode of the `Place`.
`float`	`getLatitude()` Returns the current latitude of the `Place`.
`float`	`getLongitude()` Returns the current longitude of the `Place`.
`void`	`setName(String name)` Sets the name of the `Place` to `name`.
`void`	`setPostCode(String postcode)` Sets the postcode of the `Place` to `postcode`.
`void`	`setLatitude(float lat)` Sets the current latitude of the `Place` to `lat`.
`void`	`setLongitude(float lon)` Sets the current longitude of the `Place` to `lon`.
`String`	`toString()` Returns a human-readable string representation of the `Place`.

Constructor

```
public Place(String name, String postcode,
             float latitude, float longitude)
```

Constructs a new `Place` with the given information.

Parameters:

`name`	-	The name of the `Place`.
`postcode`	-	The postcode of the `Place`.
`latitude`	-	The latitude of the `Place`.
`longitude`	-	The longitude of the `Place`.

PriceRange

Method Summary

Modifier	Method Name and Description
`String`	`toString()` Returns a human-readable string representation of the `PriceRange`.

List of Price Ranges

`CheapEats`	`MidRange`	`FineDining`

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.models.PriceRange;
```

```java
// Assigning a PriceRange
PriceRange p = PriceRange.CheapEats;

// Print PriceRange p, which should come out as "Cheap Eats"
System.out.println(p);

// Print PriceRange.FineDining, this should come out as "Fine Dining"
System.out.println(PriceRange.FineDining);
```

Restaurant

Method Summary

Modifier	Method Name and Description
`String[]`	`getRepeatedID()` Returns the repeated ID of the `Restaurant`.
`Long`	`getID()` Returns the ID of the `Restaurant`.
`String`	`getStringID()` Returns the ID of the `Restaurant` in `String` form.
`String`	`getName()` Returns the name of the `Restaurant`.
`String`	`getOwnerFirstName()` Returns the first name of the owner of the `Restaurant`.
`String`	`getOwnerLastName()` Returns the last name of the owner of the `Restaurant`.
`Cuisine`	`getCuisine()` Returns the cuisine served at the `Restaurant`.
`EstablishmentType`	`getEstablishmentType()` Returns the type of establishment of the `Restaurant`.
`PriceRange`	`getPriceRange()` Returns the price range of the `Restaurant`.
`Date`	`getDateEstablished()` Returns the `Date` the `Restaurant` was established.
`float`	`getLatitude()` Returns the current latitude of the `Restaurant`.
`float`	`getLongitude()` Returns the current longitude of the `Restaurant`.
`boolean`	`getVegetarianOptions()` Returns if the `Restaurant` has vegetarian options.
`boolean`	`getVeganOptions()` Returns if the `Restaurant` has vegan options.
`boolean`	`getGlutenFreeOptions()` Returns if the `Restaurant` has gluten-free options.
`boolean`	`getNutFreeOptions()` Returns if the `Restaurant` has nut-free options.

Modifier	Method Name and Description
`boolean`	`getLactoseFreeOptions()` Returns if the `Restaurant` has lactose-free options.
`boolean`	`getHalalOptions()` Returns if the `Restaurant` has halal options.
`Date`	`getLastInspectedDate()` Returns the `Date` the `Restaurant` was last inspected.
`int`	`getFoodInspectionRating()` Returns the food inspection rating of the `Restaurant`.
`int`	`getWarwickStars()` Returns the no. of Warwick stars the `Restaurant` has.
`float`	`getCustomerRating()` Returns the customer rating of the `Restaurant`.
`void`	`setRepeatedID(String repeatedID)` Sets the repeated ID of the `Restaurant` to `repeatedID`.
`void`	`setID(Long id)` Sets the ID of the `Restaurant` to `id`.
`void`	`setName(String restaurantName)` Sets the name of the `Restaurant` to `restaurantName`.
`void`	`setOwnerFirstName(String ownerFirstName)` Sets the first name of the owner of the `Restaurant` to `ownerFirstName`.
`void`	`setOwnerLastName(String ownerlastName)` Sets the last name of the owner of the `Restaurant` to `ownerLastName`.
`void`	`setCuisine(Cuisine cuisine)` Sets the cuisine served at the `Restaurant` to `cuisine`.
`void`	`setEstablishmentType(EstablishmentType e)` Sets the type of establishment of the `Restaurant` to `e`.
`void`	`setPriceRange(PriceRange priceRange)` Sets the price range of the `Restaurant` to `priceRange`.
`void`	`setDateEstablished(Date dateEstablished)` Sets the `Date` the `Restaurant` was established to `dateEstablished`.

Modifier	Method Name and Description
`void`	`setLatitude(float lat)` Sets the current latitude of the `Restaurant` to `lat`.
`void`	`setLongitude(float lon)` Sets the current longitude of the `Restaurant` to `lon`.
`void`	`setVegetarianOptions(boolean vegetarian)` Sets if the `Restaurant` has vegetarian options to `vegetarian`.
`void`	`setVeganOptions(boolean vegan)` Sets if the `Restaurant` has vegan options to `vegan`.
`void`	`setGlutenFreeOptions(boolean glutenFree)` Sets if the `Restaurant` has gluten-free options to `glutenFree`.
`void`	`setNutFreeOptions(boolean nutFree)` Sets if the `Restaurant` has nut-free options to `nutFree`.
`void`	`setLactoseFreeOptions(boolean lactoseFree)` Sets if the `Restaurant` has lactose-free options to `lactoseFree`.
`void`	`setHalalOptions(boolean halal)` Sets if the `Restaurant` has halal options to `halal`.
`void`	`setLastInspectedDate(Date lastInspected)` Sets the `Date` the `Restaurant` was last inspected to `lastInspected`.
`void`	`setFoodInspectionRating(int inspectionRating)` Sets the food inspection rating of the `Restaurant` to `inspectionRating`.
`void`	`setWarwickStars(int warwickStars)` Sets the no. of Warwick stars the `Restaurant` has to `warwickStars`.
`void`	`setCustomerRating(float rating)` Sets the customer rating of the `Restaurant` to `rating`.
`String`	`toString()` Returns a human-readable string representation of the `Restaurant`.

Constructors

```
public Restaurant(String repeatedID,
                  String name,
                  String ownerFirstName,
                  String ownerLastName,
                  Cuisine cuisine,
                  EstablishmentType establishmentType,
                  PriceRange priceRange,
                  Date dateEstablished,
                  float latitude,
                  float longitude,
                  boolean vegetarianOptions,
                  boolean veganOptions,
                  boolean glutenFreeOptions,
                  boolean nutFreeOptions,
                  boolean lactoseFreeOptions,
                  boolean halalOptions,
                  Date lastInspectedDate,
                  int foodInspectionRating,
                  int warwickStars,
                  float customerRating)
```

Constructs a new `Restaurant` with the given information.
The initial `ID` of the restaurant is set to `-1L`.

Parameters:

`repeatedID` - The repeated ID of the `Restaurant`.
`name` - The name of the `Restaurant`.
`ownerFirstName` - The first name of the owner of the `Restaurant`.
`ownerLastName` - The last name of the owner of the `Restaurant`.
`cuisine` - The cuisine served at the `Restaurant`.
`establishmentType` - The establishment type of the `Restaurant`.
`priceRange` - The price range of the `Restaurant`.
`dateEstablished` - The date the `Restaurant` was established.
`latitude` - The latitude of the `Restaurant`.
`longitude` - The longitude of the `Restaurant`.
`vegetarianOptions` - If the `Restaurant` has vegetarian options.
`veganOptions` - If the `Restaurant` has vegan options.
`glutenFreeOptions` - If the `Restaurant` has gluten-free options.
`nutFreeOptions` - If the `Restaurant` has nut-free options.
`lactoseFreeOptions` - If the `Restaurant` has lactose-free options.
`halalOptions` - If the `Restaurant` has halal options.
`lastInspectedDate` - The date the `Restaurant` was last inspected.
`foodInspectionRating` - The food inspection rating of the `Restaurant`.
`warwickStars` - The no. of Warwick stars the `Restaurant` has.
`customerRating` - The customer rating of the `Restaurant`.

```java
public Restaurant(String repeatedID,
                  String name,
                  String ownerFirstName,
                  String ownerLastName,
                  Cuisine cuisine,
                  EstablishmentType establishmentType,
                  PriceRange priceRange,
                  Date dateEstablished,
                  float latitude,
                  float longitude,
                  boolean vegetarianOptions,
                  boolean veganOptions,
                  boolean glutenFreeOptions,
                  boolean nutFreeOptions,
                  boolean lactoseFreeOptions,
                  boolean halalOptions,
                  Date lastInspectedDate,
                  int foodInspectionRating,
                  int warwickStars)
```

Constructs a new `Restaurant` with the given information.
The initial `ID` of the restaurant is set to `-1L`.
The initial `rating` of the restaurant is set to `0.0f`.

Parameters:

`repeatedID` - The repeated ID of the `Restaurant`.
`name` - The name of the `Restaurant`.
`ownerFirstName` - The first name of the owner of the `Restaurant`.
`ownerLastName` - The last name of the owner of the `Restaurant`.
`cuisine` - The cuisine served at the `Restaurant`.
`establishmentType` - The establishment type of the `Restaurant`.
`priceRange` - The price range of the `Restaurant`.
`dateEstablished` - The date the `Restaurant` was established.
`latitude` - The latitude of the `Restaurant`.
`longitude` - The longitude of the `Restaurant`.
`vegetarianOptions` - If the `Restaurant` has vegetarian options.
`veganOptions` - If the `Restaurant` has vegan options.
`glutenFreeOptions` - If the `Restaurant` has gluten-free options.
`nutFreeOptions` - If the `Restaurant` has nut-free options.
`lactoseFreeOptions` - If the `Restaurant` has lactose-free options.
`halalOptions` - If the `Restaurant` has halal options.
`lastInspectedDate` - The date the `Restaurant` was last inspected.
`foodInspectionRating` - The food inspection rating of the `Restaurant`.
`warwickStars` - The no. of Warwick stars the `Restaurant` has.

Method Notes

- `String[] getRepeatedID()`
  This method splits the repeated ID `String` after every 16 characters and returns a `String` array that is of length 3 or higher. So if a repeated ID `String` contains 20 characters, you get a `String` array with the first `String` being of length 16, the second `String` being of length 4, and the third `String` is `null`.

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.models.Restaurant;
```

```java
// Create a new Restaurant object
Restaurant r = new Restaurant(
    "111222333444555611122233344455561112223334445556",
    "DCS",
    "Tux",
    "",
    Cuisine.British,
    EstablishmentType.Pub,
    PriceRange.FineDining,
    new SimpleDateFormat("yyyy").parse("1996"),
    52.3838f,
    -1.560065f,
    true,
    true,
    true,
    true,
    true,
    true,
    new SimpleDateFormat("yyyy").parse("2020"),
    5,
    3,
    5.0f);

// Manually calculated ID from repeated ID and set it
r.setID(1112223334445556L);

// Get rating of the Restaurant r
float restaurantRating = r.getCustomerRating();

// Print out Restaurant r's data
System.out.println(r);
```

RestaurantDistance

Method Summary

Modifier	Method Name and Description
`Restaurant`	`getRestaurant()` Returns the `Restaurant`.
`float`	`getDistance()` Returns the distance away, in kilometres, from the `Restaurant`.
`void`	`setRestaurant(Restaurant restaurant)` Sets the `Restaurant` to `restaurant`.
`void`	`setDistance(float distance)` Set the distance away, in kilometres, from the `Restaurant` to `distance`.
`String`	`toString()` Returns a human-readable string representation of the `RestaurantDistance`.

Constructor

```
public RestaurantDistance(Restaurant restaurant, float distance)
```

Constructs a new `RestaurantDistance` with the given information.

Parameters:

`restaurant`	-	The `Restaurant`.
`distance`	-	The distance away, in kilometres, from the `Restaurant`.

Example Code

```
// The import statement
import uk.ac.warwick.cs126.models.RestaurantDistance;
```

```
// Create a new RestaurantDistance object
RestaurantDistance r = new RestaurantDistance(null, 8046.7f);

// Get Restaurant from the RestaurantDistance r, should be null
Restaurant restaurant = r.getRestaurant();
System.out.println(restaurant);

// Set distance for RestaurantDistance r
r.setDistance(5000.0f);
System.out.println(r);
```

Stores

CustomerStore

Method Summary

Modifier	Method Name and Description
`Customer[]`	`loadCustomerDataToArray(InputStream resource)` Returns a `Customer[]` loaded from the `resource`.
`boolean`	`addCustomer(Customer customer)` Adds a valid `customer` to the store. Returns `true` if added successfully, `false` otherwise.
`boolean`	`addCustomer(Customer[] customers)` Adds all valid customers from `customers` to the store. Returns `true` if all added successfully, `false` otherwise.
`Customer`	`getCustomer(Long id)` Gets the `Customer` with the corresponding ID `id`. Returns the `Customer` if found, `null` otherwise.
`Customer[]`	`getCustomers()` Returns an array of all customers in the store, sorted in ascending order of ID.
`Customer[]`	`getCustomers(Customer[] customers)` Returns the input array `customers`, sorted in ascending order of ID.
`Customer[]`	`getCustomersByName()` Returns an array of all customers in the store, sorted in alphabetical order of their Last Name , then First Name, then ID.
`Customer[]`	`getCustomersByName(Customer[] customers)` Returns the input array `customers`, sorted in alphabetical order of Last Name, then First Name, then ID.
`Customer[]`	`getCustomersContaining(String str)` Return an array of all the customers whose First Name and Last Name contain the given query `str`. The returned array is sorted in alphabetical order of their Last Name, then First Name, then ID.

Constructor

```
public CustomerStore()
```

Constructs a new `CustomerStore`.

Method Notes

In most methods, make sure you check for `null` objects, if not otherwise stated.

```
Customer[] loadCustomerDataToArray(InputStream resource)
```

Loads data from a CSV file containing the `Customer` data into a `Customer` array, parsing the attributes where required.

Returns a `Customer[]` loaded from the `resource`.

Note, this is already implemented for you, do **not** change.

Parameters:

`resource` - The CSV data in the form of an `InputStream`.

Returns:

`Customer[]` - The customers loaded.
`Customer[]` - A `Customer[]` of length 0, if failed to load.

```
boolean addCustomer(Customer customer)
```

Attempts to add `customer` to the store.

The `customer` should not be added if it is not valid. See method `DataChecker > isValid(Customer customer)` for more details on whether a `Customer` is valid or not.

A valid `customer` should not be added if a `Customer` with the same ID already exists in the store.

If a duplicate ID is encountered from a valid `customer`, the `customer` is not added and the existing `Customer` with that ID should be removed from the store. Finally, the duplicate ID should be blacklisted from further use.

A `customer` with a blacklisted ID should not be added.

Return `true` if `customer` is successfully added to the store, otherwise `false`.

Note that there is no ordering on `Customer` objects coming into this method, i.e. the next one may be older or newer, or may have a higher or lower ID than the previously recieved `Customer`.

Parameters:

`customer` - The `Customer` to be added into the store.

Returns:

`boolean` - `true` if `customer` is added.
`boolean` - `false` if `customer` is not added.

Example:

These examples are similar to the ones shown in the other stores.

In the store at the beginning:

`Customer A with ID:1112223334445556L`

`Customer B with ID:1112223334445557L`

`Customer C with ID:1112223334445558L`

Now, we try to add `Customer D with ID:1112223334445555L`, this fails because it has an invalid ID.

After, we try to add `Customer E with ID:1112223334445557L` and its first name field is `null`, this fails because there is a `null` field. Note, we do not blacklist the ID even though the ID exists in the store because `Customer E` is an invalid `Customer`.

Next, we try to add `Customer F with DateJoined:null`, this fails because there is a `null` field.

Then, we try to add `Customer G with ID:1112223334445557L`, assume the other fields are valid too, this fails because it is a duplicate ID. We remove `Customer B` from the store. We blacklist the ID `1112223334445557L`.

After that, we try to add `Customer H with ID:1112223334445557L`, this fails because it is a blacklisted ID.

Finally, we try to add `Customer I with ID:1112223334445559L`, assume the other fields are valid too, this succeeds and is added to the store.

The store at the end:

`Customer A with ID:1112223334445556L`

`Customer C with ID:1112223334445558L`

`Customer I with ID:1112223334445559L`

```
boolean addCustomer(Customer[] customers)
```

Attempts to add valid `Customer` objects from the `customers` input array to the store.

These customers are added under the same conditions as specified in above method: `addCustomer(Customer customer)`.

Return `true` if the all the `customers` are all successfully added to the data store, otherwise `false`.

Hint: You can loop through the `customers` array and on each customer you can call the `addCustomer(Customer customer)` method. You still need to do some other checks in this method, like checking for `null`.

Parameters:

`customers`	-	The input `Customer` array.

Returns:

`boolean`	-	`true` if all the customers from `customers` are added.
`boolean`	-	`false` if any `customer` from `customers` is not added.

```
Customer getCustomer(Long id)
```

Returns the `Customer` with the matching ID `id` from the store, otherwise this method should return `null` if not found.

Parameters:

`id`	-	The ID of the customer you wish to get.

Returns:

`Customer`	-	The found `Customer`.
`Customer`	-	`null` if not found.

```
Customer[] getCustomers()
```

Returns an array of all customers in the store, sorted in ascending order of **ID**.

Returns:

`Customer[]`	-	All stored customers, sorted in ascending order of **ID**.
`Customer[]`	-	A `Customer[]` of length 0, if otherwise.

Example:

Index	ID
`0`	`1112223334445556L`
`1`	`2223334445556667L`
`2`	`3334445556667778L`

```
Customer[] getCustomers(Customer[] customers)
```

Returns the input array `customers` sorted in ascending order of **ID**.

Hint: **DO NOT USE BUBBLESORT**. In general, for this coursework anything that has an average time of $O(n^2)$ is awful.

Parameters:

`customers`	-	The input `Customer` array.

Returns:

`Customer[]`	-	Input `customers` sorted in ascending order of **ID**.
`Customer[]`	-	A `Customer[]` of length 0, if otherwise.

`Customer[] getCustomersByName()`

Returns an array of all customers in the store, sorted alphabetically by **Last Name**, if they have same **Last Name** then alphabetically by **First Name**.

If they have the same **Last Name** and **First Name**, then it is sorted in ascending order of **ID**.

In sorting, **Last Name** and **First Name** fields are case-insensitive.

Note, when we say in sorting a field is **case-insensitive**, is take the **First Name** strings `"Alice"` and `"ALICE"`, comparatively we say that the strings are equal (even though they are in a different case), so then you must compare using their next field, in this case you must sort by their **ID**. This is not detailed again in the guide, so assume we mean that when we say in sorting such and such field is **case-insensitive**.

Returns:

`Customer[]` - All stored customers, sorted alphabetically by **Last Name**, if same then by **First Name**, if same then in ascending order of **ID**.

`Customer[]` - A `Customer[]` of length 0, if otherwise.

Example:

Index	First Name	Last Name	ID
0	"Alice"	""	4445556667778889L
1	"Billy"	"Bob"	8884445556667779L
2	"The"	"Bob"	2223334445556667L
3	"The"	"Bob"	3334445556667778L
4	"JAY"	"Z"	1112223334445556L
5	"jay"	"z"	2225556667778881L
6	"JAY"	"Z"	2225556667778883L
7	""	"Öreo"	9995556667778882L
8	"Anne"	"Öreo"	4445556667778882L

`Customer[] getCustomersByName(Customer[] customers)`

Returns the input array `customers` sorted alphabetically by **Last Name**, if they have same **Last Name** then alphabetically by **First Name**.

If they have the same **Last Name** and **First Name**, then it is sorted in ascending order of **ID**.

In sorting, **Last Name** and **First Name** fields are case-insensitive.

Parameters:

`customers` - The input `Customer` array.

Returns:

`Customer[]`	-	Input `customers` sorted alphabetically by **Last Name**, if same then by **First Name**, if same then by **ID** (low to high).
`Customer[]`	-	A `Customer[]` of length 0, if otherwise.

```
Customer[] getCustomersContaining(String str)
```

Return an array of all the customers from the store whose **First Name** and **Last Name** contain the given query `str`.

Search queries are **accent-insensitive** and **case-insensitive**. Ignore leading and trailing spaces. Also, ignore multiple spaces, only use the one space.

When we say a search query is **accent-insensitive**, we mean that when we are searching for `"Amélie"`, what we are really searching for is `"Amelie"`, so if there is a customer in the store with the **First Name** `"Amélie"` it should yield that customer. Furthermore, if in the store there is a customer with the **First Name** `"Amelie"` it should also yield that customer as well.

When looking for a customer with the **First Name** `"John"` and **Last Name** `"Smith"`, if a user queries the term `"ohn Smi"`, the customer with the **First Name** `"John"` and **Last Name** `"Smith"` should be included in the results. If a user queries the term `"John Smith"`, the results should yield the customer. However, if a user queries the term `"JohnSmith"`, this should **not** yield the customer. Also, if a user queries the term `"Smith John"`, this should **not** yield the customer. And if a user queries the term `"ith Joh"` or `"ith ohn"` or `"Smi ohn"` or `"Smi Joh"` or `"Joh Smi"` or `"Joh ith"` or `"ohn ith"`, this should **not** yield the customer.

Implement the `StringFormatter` > `convertAccentsFaster(String str)` method to strip off accents in this method.

The returned array is sorted the same as `getCustomersByName()`. So, sorted alphabetically by **Last Name**, if they have same **Last Name**, then alphabetically by **First Name**. If they have the same **Last Name** and **First Name**, then it is sorted in ascending order of **ID**. In sorting, **Last Name** and **First Name** fields are case-insensitive.

The empty string `""` query should return a `Customer[]` of length 0.

Note, the returned customers array should have their original names, not their names with no accents and in the wrong case. Also, the returned sort is **not accent-insensitive**, do **not** make the returned sort be **accent-insensitive**. The sort is exactly like how we explained in `getCustomersByName()`, in this entire coursework **none** of the returned sorts are **accent-insensitive**.

Hint: If your output needs sorting after searching, isn't there a method which you implemented that sorts a `Customer` array by name? But should you need to use that method though?

Parameters:

`str` - Search the **First Name** and **Last Name** fields to see if it contains this query `str`.

Returns:

`Customer[]` - Array of customers whose name contains the input query `str`, ordered by their **Last Name**, then if same by their **First Name**, then if same by ascending order of **ID**.

`Customer[]` - A `Customer[]` of length 0, if otherwise.

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.stores.CustomerStore;
```

```java
// Constructs CustomerStore
CustomerStore c = new CustomerStore();

// Add null customer, should return false
boolean addedCustomer = c.addCustomer((Customer) null);
System.out.println(addedCustomer);

// Tries to get Customer with ID 1112223334445556L, should return null
Customer foundCustomer = c.getCustomer(1112223334445556L);
System.out.println(foundCustomer);
```

Related Model

- `Customer`

Related Methods

- `DataChecker` > `isValid(Customer customer)`
- `StringFormatter` > `convertAccentsFaster(String str)`

FavouriteStore

Method Summary

Modifier	Method Name and Description
`Favourite[]`	`loadFavouriteDataToArray(InputStream resource)` Returns a `Favourite[]` loaded from the `resource`.
`boolean`	`addFavourite(Favourite favourite)` Adds a valid `favourite` to the store. Returns `true` if added successfully, `false` otherwise.
`boolean`	`addFavourite(Favourite[] favourites)` Adds all valid favourites from `favourites` to the store. Returns `true` if all added successfully, `false` otherwise.
`Favourite`	`getFavourite(Long id)` Gets the `Favourite` with the corresponding ID `id`. Returns the `Favourite` if found, `null` otherwise.
`Favourite[]`	`getFavourites()` Returns an array of all favourites in the store, sorted in ascending order of ID.
`Favourite[]`	`getFavouritesByCustomerID(Long id)` Gets the favourites that corresponds to the `Customer` ID `id`. Returns the `Favourite[]` of found favourites.
`Favourite[]`	`getFavouritesByRestaurantID(Long id)` Gets the favourites that corresponds to the `Restaurant` ID `id`. Returns the `Favourite[]` of found favourites.
`Long[]`	`getCommonFavouriteRestaurants(Long id1, Long id2)` Returns the Restaurant IDs from the favourites in common between `Customer` 1 with `id1` and `Customer` 2 with `id2`.
`Long[]`	`getMissingFavouriteRestaurants(Long id1, Long id2)` Returns the Restaurant IDs from the favourites that are favourited by `Customer` 1 with `id1` but not favourited by `Customer` 2 with `id2`.
`Long[]`	`getNotCommonFavouriteRestaurants(Long id1, Long id2)` Returns the Restaurant IDs from the favourites that are favourited by `Customer` 1 with `id1` but not favourited by `Customer` 2 with `id2`, and the favourites that are favourited by `Customer` 2 with `id2` but not favourited by `Customer` 1 with `id1`.

Modifier	Method Name and Description
`Long[]`	`getTopCustomersByFavouriteCount()` Returns the ID's of top 20 customers that favourited the most.
`Long[]`	`getTopRestaurantsByFavouriteCount()` Returns the ID's of top 20 restaurant with the most favourites.

Constructor

```
public FavouriteStore()
```

Constructs a new `FavouriteStore`.

Method Notes

In most methods, make sure you check for `null` objects, if not otherwise stated.

```
Favourite[] loadFavouriteDataToArray(InputStream resource)
```

Loads data from a CSV file containing the `Favourite` data into a `Favourite` array, parsing the attributes where required.

Returns a `Favourite[]` loaded from the `resource`.

Note, this is already implemented for you, do **not** change.

Parameters:

`resource` - The CSV data in the form of an `InputStream`.

Returns:

`Favourite[]` - The favourite data loaded.
`Favourite[]` - A `Favourite[]` of length 0, if failed to load.

```
boolean addFavourite(Favourite favourite)
```

Attempts to add the `favourite` to the store.

The `favourite` should not be added if it is not valid. See the `DataChecker > isValid(Favourite favourite)` for more details on whether a inputted `Favourite` is valid or not.

A valid `favourite` should not be added to the store if a `Favourite` with the same ID already exists in the store.

If a duplicate ID is encountered from a valid `favourite`, the `favourite` is not added and then the existing `Favourite` with that ID should be removed from the store. Finally, the duplicate ID should be blacklisted from further use.

A `favourite` with a blacklisted ID should not be added.

Note that there is no ordering on `Favourite` objects coming into this method, i.e. the next one may be older or newer, or may have a higher or lower ID than the previously recieved `Favourite`.

Now for the twist!

If the `favourite` is valid and does not have an ID that has been blacklisted, is a duplicate, or is invalid: if there exists a `Favourite` already inside the store with the same **Customer ID** and **Restaurant ID**, and if this `favourite` is **older** than the one in the store, you must replace it with this `favourite`. If this replace happens, the ID of the `Favourite` originally in the store should be blacklisted from further use.

In laymen's term, if a customer has already favourited a restaurant before, if everything is valid, choose the **older** favourite.

Return `true` if the `favourite` is successfully added to the store, otherwise return `false`.

The twist comes with many edge cases you must explore:

For example, if we replace `Favourite A 2018` with `Favourite B - 2017`, but after, `Favourite B - 2017` gets blacklisted when we add in a duplicate ID that came from `Favourite C - 2020`. Then, we must un-blacklist and add back `Favourite A - 2018`, otherwise a restaurant ends up incorrectly missing a favourite.

Another example, if the data had been added in a different order:

`Favourite B - 2017`, `Favourite A - 2018`, `Favourite C - 2020`.

Then `Favourite A - 2018` never gets added to the store. And when `B` gets removed because of `C`, no favourites were added at all. But this is wrong, `Favourite A - 2018` should exist in the store.

There are some more edge cases which we will leave for you to explore.

Parameters:

`favourite` - The `Favourite` to be added into the store.

Returns:

`boolean` - `true` if `favourite` is added.

`boolean` - `false` if `favourite` is not added.

Example:

These examples are similar to the ones shown in the other stores.

In the store at the beginning:

`Favourite A with ID:1112223334445556L`

`Favourite B with ID:1112223334445557L`

`Favourite C with ID:11122233344445558L`

Now, we try to add `Favourite D with ID:1112223334445555L`, this fails because it has an invalid ID.

After, we try to add `Favourite E with ID:11122233344445557L` and its name field is `null`, this fails because there is a `null` field. Note, we do not blacklist the ID even though the ID exists in the store because `Favourite E` is an invalid `Favourite`.

Next, we try to add `Favourite F with Name:null`, this fails because there is a `null` field.

Then, we try to add `Favourite G with ID:11122233344445557L`, assume the other fields are valid too, this fails because it is a duplicate ID. We remove `Favourite B` from the store. We blacklist the ID `11122233344445557L`.

After that, we try to add `Favourite H with ID:11122233344445557L`, this fails because it is a blacklisted ID.

At the end, we try to add `Favourite I with ID:11122233344445559L`, we assume the other fields are valid too, this succeeds and is added to the store.

The store at the end:

`Favourite A with ID:11122233344445556L`

`Favourite C with ID:11122233344445558L`

`Favourite I with ID:11122233344445559L`

The edge case examples for the twist are explained before this, and so they are not explained again here.

```
boolean addFavourite(Favourite[] favourites)
```

Attempts to add valid `Favourite` objects from the `favourites` input array to the store.

These favourites are added under the same conditions as specified in above method: `addFavourite(Favourite favourite)`.

Return true if the all the `favourites` are all successfully added to the data store, otherwise `false`.

Parameters:

`favourites` - The `Favourite` array.

Returns:

`boolean` - `true` if all the favourites from `favourites` are added.
`boolean` - `false` if any `favourite` from `favourites` is not added.

```
Favourite getFavourite(Long id)
```

Returns the `Favourite` with the matching ID `id` from the store, otherwise this method should return `null` if not found.

Parameters:

`id` - The ID of the favourite you wish to get.

Returns:

`Favourite` - The found `Favourite`.

`Favourite` - `null` if not found.

```
Favourite[] getFavourites()
```

Returns an array of all the favourites in the store, sorted in ascending order of **ID**.

Returns:

`Favourite[]` - All stored favourites, sorted in ascending order of **ID**.

`Favourite[]` - A `Favourite[]` of length 0, if otherwise.

```
Favourite[] getFavouritesByCustomerID(Long id)
```

Return a favourite array with all the favourites from the store that have `id` for its **Customer ID**.

The returned array should be sorted by **Date Favourited**, from newest to oldest.

Compare dates to the millisecond.

If they have the same **Date Favourited**, then it is sorted in ascending order of their **ID**.

If the customer does not exist, or otherwise, return a `Favourite[]` of length 0.

Parameters:

`id` - The ID of the customer you wish to get all favourites for.

Returns:

`Favourite[]` - The favourites belonging to `Customer` with ID `id`, sorted by **Date Favourited**, from newest to oldest, if same then in ascending order of **ID**.

`Favourite[]` - A `Favourite[]` of length 0, if otherwise.

Example:

Index	Date Favourited	ID
`0`	`2020-01-01 16:27:11.000`	`4445556667778889L`
`1`	`2019-01-01 00:00:00.000`	`1114445556667779L`
`2`	`2019-01-01 00:00:00.000`	`2223334445556667L`
`3`	`2018-01-01 14:44:44.000`	`9994445556667778L`

`Favourite[] getFavouritesByRestaurantID(Long id)`

Return a favourite array with all the favourites from the store that have `id` for its **Restaurant ID**.

The returned array should be sorted by **Date Favourited**, from newest to oldest.

Compare dates to the millisecond.

If they have the same **Date Favourited**, then it is sorted in ascending order of their **ID**.

If the restaurant does not exist, or otherwise, return a `Favourite[]` of length 0.

Parameters:

`id` - The ID of the restaurant you wish to get all favourites for.

Returns:

`Favourite[]`	-	The favourites belonging to `Restaurant` with ID `id`, sorted by **Date Favourited**, from newest to oldest, if same then in ascending order of **ID**.
`Favourite[]`	-	A `Favourite[]` of length 0, if otherwise.

`Long[] getCommonFavouriteRestaurants(Long id1, Long id2)`

Returns the **Restaurant IDs** from the favourites in-common between `Customer 1` with ID `id1` and `Customer 2` with ID `id2`.

In essence, this is the set intersection operation.

We label favourites as in-common, if `Customer 1` has a `Favourite A` with **Restaurant ID** `r` and `Customer 2` also has `Favourite B` with **Restaurant ID** `r`. Then `Favourite A` and `Favourite B` are in-common.

For each in-common favourite scenario, use the favourite that has the latest **Date Favourited** between the two in-common. For example, if `Favourite A` was favourited in 2020 and `Favourite B` was favourited in 2010, we keep `Favourite A`. If they have the same date, choose any, it does not matter as we do **not** use the **Favourite ID**.

The resulting in-common favourites should be sorted by **Date Favourited**, from newest to oldest.

Compare dates to the millisecond.

If they have the same **Date Favourited**, then it is sorted in ascending order of their **Restaurant ID**.

Return a `Long[]` of all the **Restaurant IDs** from the resulting sorted in-common favourites. The ordering should still be the same as the sorted in-common favourites. Think of it like we are stripping away all the other fields from the `Favourite` leaving only the **Restaurant ID** field.

If otherwise, return a `Long[]` of length 0.

Parameters:

`id1` - The ID of `Customer 1`.

`id2` - The ID of `Customer 2`.

Returns:

`Long[]` - The **Restaurant ID**'s from the common favourites between `Customer 1` with ID `id1` and `Customer 2` with `id2`, sorted by **Date Favourited**, newest to oldest, if same then in ascending order of **Restaurant ID**.

`Long[]` - A `Long[]` of length 0, if otherwise.

Example:

If `Customer 1` with `id1` has:

Favourite ID	Date Favourited	Restaurant ID
2223334445556668L	2019	2223334445556667L
3334445556667779L	2018	3334445556667778L
4445556667778881L	2017	4445556667778889L
6667778889991113L	2015	6667778889991112L

If `Customer 2` with `id2` has:

Favourite ID	Date Favourited	Restaurant ID
7778889991112224L	2020	6667778889991112L
7778889992223334L	2019	8889991112223334L
8889991112223335L	2018	4445556667778889L
8889992223334445L	2017	3334445556667778L

Then if we call `getCommonFavouriteRestaurants(id1, id2)`:

Favourite ID	Date Favourited	Restaurant ID
7778889991112224L	2020	6667778889991112L
3334445556667779L	2018	3334445556667778L
8889991112223335L	2018	4445556667778889L

Finally, we return only the **Restaurant IDs**.

```
Long[] getMissingFavouriteRestaurants(Long id1, Long id2)
```

Returns the **Restaurant IDs** from the favourites that are favourited by `Customer 1` with ID `id1` but not favourited by `Customer 2` with ID `id2`.

In essence, this is the set difference operation.

Favourites are labelled as missing, if `Customer 1` has a `Favourite A` with **Restaurant ID** `restaurantID` but `Customer 2` does not have a `Favourite` with **Restaurant ID** `restaurantID`. Then `Favourite A` is missing.

The missing favourites should be sorted by **Date Favourited**, from newest to oldest.

Compare dates to the millisecond.

If they have the same **Date Favourited**, then it is sorted in ascending order of their **Restaurant ID**.

Return a `Long[]` of all the **Restaurant IDs** from the resulting sorted missing favourites. The ordering should still be the same as the sorted missing favourites. Think of it like we are stripping away all the other fields from the `Favourite` leaving only the **Restaurant ID** field.

If otherwise, return a `Long[]` of length 0.

Parameters:

`id1`	-	The ID of `Customer 1`.
`id2`	-	The ID of `Customer 2`.

Returns:

`Long[]`	-	The **Restaurant ID**'s from the missing favourites, sorted by **Date Favourited**, newest to oldest, if same then in ascending order of **Restaurant ID**.
`Long[]`	-	A `Long[]` of length 0, if otherwise.

Example:

If `Customer 1` with `id1` has:

Favourite ID	Date Favourited	Restaurant ID
`1112223334445557L`	`2020`	`1112223334445556L`
`2223334445556668L`	`2019`	`2223334445556667L`
`3334445556667779L`	`2018`	`3334445556667778L`
`5556667778889992L`	`2016`	`5556667778889991L`

If `Customer 2` with `id2` has:

Favourite ID	Date Favourited	Restaurant ID
`7778889992223334L`	`2019`	`8889991112223334L`
`8889992223334445L`	`2017`	`3334445556667778L`
`9991112223334446L`	`2016`	`7778889991112223L`
`9992223334445556L`	`2015`	`9991112223334445L`

Then if we call `getMissingFavouriteRestaurants(id1, id2)`:

Favourite ID	Date Favourited	Restaurant ID
`1112223334445557L`	`2020`	`1112223334445556L`
`2223334445556668L`	`2019`	`2223334445556667L`
`5556667778889992L`	`2016`	`5556667778889991L`

Finally, we return only the **Restaurant IDs**.

```
Long[] getNotCommonFavouriteRestaurants(Long id1, Long id2)
```

Returns the **Restaurant IDs** from the favourites that are favourited by `Customer 1` with ID `id1` but not favourited by `Customer 2` with ID `id2`, as well as the favourites that are favourited by `Customer 2` with ID `id2` but not favourited by `Customer 1` with ID `id1`.

In essence, this is the set symmetric difference operation.

We label favourites as not-common, if `Customer 1` has a `Favourite A` with **Restaurant ID** `restaurantID` but `Customer 2` does not have a `Favourite` with **Restaurant ID** `restaurantID`. Then `Favourite A` is not-common.

Also, favourites are not-common, if `Customer 2` has a `Favourite B` with **Restaurant ID** `restaurantID` but `Customer 1` does not have a `Favourite` with **Restaurant ID** `restaurantID`. Then `Favourite B` is not-common.

The resulting not-common favourites should be sorted by **Date Favourited**, from newest to oldest.

If they have the same **Date Favourited**, then it is sorted in ascending order of their **Restaurant ID**.

Compare dates to the millisecond.

Return a `Long[]` of all the **Restaurant IDs** extracted from the resulting sorted not-common favourites. The ordering should still be the same as the sorted not-common favourites. Think of it like we are stripping away all the other fields from the `Favourite` leaving only the **Restaurant ID** field.

If otherwise, return a `Long[]` of length 0.

Parameters:

`id1` - The ID of `Customer` `1`.
`id2` - The ID of `Customer` `2`.

Returns:

`Long[]` - The **Restaurant ID**’s from the not-common favourites, sorted by **Date Favourited**, newest to oldest, if same then in ascending order of **Restaurant ID**.
`Long[]` - A `Long[]` of length 0, if otherwise.

Example:

If `Customer` `1` with `id1` has:

Favourite ID	Date Favourited	Restaurant ID
`2223334445556668L`	`2019`	`2223334445556667L`
`3334445556667779L`	`2018`	`3334445556667778L`
`5556667778889992L`	`2016`	`5556667778889991L`
`6667778889991113L`	`2015`	`6667778889991112L`

If `Customer` `2` with `id2` has:

Favourite ID	Date Favourited	Restaurant ID
`7778889991112224L`	`2020`	`6667778889991112L`
`7778889992223334L`	`2019`	`8889991112223334L`
`8889992223334445L`	`2017`	`3334445556667778L`
`9991112223334446L`	`2016`	`7778889991112223L`

Then if we call `getNotCommonFavouriteRestaurants(id1, id2)`:

Favourite ID	Date Favourited	Restaurant ID
`2223334445556668L`	`2019`	`2223334445556667L`
`7778889992223334L`	`2019`	`8889991112223334L`
`5556667778889992L`	`2016`	`5556667778889991L`
`9991112223334446L`	`2016`	`7778889991112223L`

Finally, we return only the **Restaurant IDs**.

```
Long[] getTopCustomersByFavouriteCount()
```

Returns the **Customer ID’s** of the top 20 customers who favourited the most.

Here, we order the customers by the number of favourites each of them have favourited, from highest to lowest, and select the top 20.

If customers have the same favourite count, then it is sorted by the date of their latest favourite, from oldest to newest.

In essence, this means the `Customer` who first reached that occurrence count will come out on top of another who reached it later.

If the customers then have the same favourite count and have the same latest date favourited, it is sorted in ascending order of **ID**.

Compare dates to the millisecond.

Return a `Long[]` of length 20, with the **Customer ID's** of the top customers.

If there are less than 20 customers, the empty elements should remain `null`.

After all that, if otherwise, return a new `Long[]` of length 20.

Returns:

`Long[]` - The top 20 customers who favourite the most.

`Long[]` - A `Long[]` of length 20 of the top *n* customers who favourite the most, where (*n* < 20), the remaining elements should be `null`.

`Long[]` - A new `Long[]` of length 20, if otherwise.

Example:

Index	Favourite Count	Latest Date Favourited	ID
0	9	2012	4445556667778889L
1	8	2010	1114445556667779L
2	7	2018	9994445556667778L
3	7	2019	3334445556667778L
...	...	...	...
19	0	2010	1115556667778883L

`Long[] getTopRestaurantsByFavouriteCount()`

Returns the **Restaurant ID's** of top 20 restaurants that have the most favourites.

Here, we order the restaurants by the number of favourites each of them have, from highest to lowest, and select the top 20.

If restaurants have the same favourite count, then it is sorted by the date of their latest favourite, from oldest to newest.

In essence, this means the `Restaurant` who first reached that occurrence count will come out on top of another who reached it later.

If the restaurants then have the same favourite count and have the same latest date favourited, it is sorted in ascending order of **ID**.

Compare dates to the millisecond.

Return a `Long[]` of length 20, with the **Restaurant ID's** of the top restaurants.

If there are less than 20 restaurants, the empty elements should remain `null`.

After all that, if otherwise, return a new `Long[]` of length 20.

Returns:

`Long[]` - The top 20 restaurants which have the most favourites.

`Long[]` - A `Long[]` of length 20 of the top *n* restaurants which have the most favourites, where (*n* < 20), the remaining elements should be `null`.

`Long[]` - A new `Long[]` of length 20, if otherwise.

Example:

Index	Favourite Count	Latest Date Favourited	ID
0	9	2012	4445556667778889L
1	8	2010	1114445556667779L
2	7	2018	9994445556667778L
3	7	2019	3334445556667778L
4	N/A	N/A	null
...	...	...	...
19	N/A	N/A	null

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.stores.FavouriteStore;
```

```java
// Constructs FavouriteStore
FavouriteStore f = new FavouriteStore();

// Get all favourites sorted by ID
Favourite[] gotFavourites = f.getFavourites();

// gotFavourites should be an array of length 0
if (gotFavourites.length == 0) {
    System.out.println("Got no favourites!");
}
```

Related Model

- `Favourite`

Related Method

- `DataChecker` > `isValid(Favourite favourite)`

RestaurantStore

Method Summary

Modifier	Method Name and Description
`Restaurant[]`	`loadRestaurantDataToArray(InputStream resource)` Returns a `Restaurant[]` loaded from the `resource`.
`boolean`	`addRestaurant(Restaurant restaurant)` Adds a valid `restaurant` to the store. Returns `true` if added successfully, `false` otherwise.
`boolean`	`addRestaurant(Restaurant[] restaurants)` Adds all valid restaurants from `restaurants` to the store. Returns `true` if all added successfully, `false` otherwise.
`Restaurant`	`getRestaurant(Long id)` Gets the `Restaurant` with the corresponding ID `id`. Returns the `Restaurant` if found, `null` otherwise.
`Restaurant[]`	`getRestaurants()` Returns an array of all restaurants in the store, sorted in ascending order of ID.
`Restaurant[]`	`getRestaurants(Restaurant[] restaurants)` Returns the input array `restaurants`, sorted in ascending order of ID.
`Restaurant[]`	`getRestaurantsByName()` Returns an array of all restaurants in the store, sorted in alphabetical order of `Restaurant` name.
`Restaurant[]`	`getRestaurantsByDateEstablished()` Returns an array of all restaurants in the store, sorted by date established (oldest first).
`Restaurant[]`	`getRestaurantsByDateEstablished(Restaurant[] r)` Returns the input array `r`, sorted by date established (oldest first).
`Restaurant[]`	`getRestaurantsByWarwickStars()` Returns an array of all restaurants in the store that have at least 1 Warwick Star, sorted in descending order of Warwick Stars.

Modifier	Method Name and Description
`Restaurant[]`	`getRestaurantsByRating(Restaurant[] restaurants)` Returns the input array `restaurants`, sorted by rating.
`RestaurantDistance[]`	`getRestaurantsByDistanceFrom(float lat, float lon)` Returns an array of `RestaurantDistance`, sorted in ascending order of distance from the input coordinates, for all the restaurants in the store.
`RestaurantDistance[]`	`getRestaurantsByDistanceFrom( Restaurant[] r, float lat, float lon)` Returns an array of `RestaurantDistance`, sorted in ascending order of distance from the input coordinates, for the given input restaurants `r`.
`Restaurant[]`	`getRestaurantsContaining(String str)` Return an array of all the restaurants whose name, cuisine or place name contain the given query `str`. The returned array is sorted alphabetically by `Restaurant` name.

Constructor

```
public RestaurantStore()
```

Constructs a new `RestaurantStore`.

Method Notes

In most methods, make sure you check for `null` objects, if not otherwise stated.

```
Restaurant[] loadRestaurantDataToArray(InputStream resource)
```

Loads data from a CSV file containing the `Restaurant` data into a `Restaurant` array, parsing the attributes where required.

Returns a `Restaurant[]` loaded from the `resource`.

Note, this is already implemented for you, do **not** change.

Parameters:

`resource` - The CSV data in the form of an `InputStream`.

Returns:

`Restaurant[]` - The restaurants loaded.

`Restaurant[]` - A `Restaurant[]` of length 0, if failed to load.

```
boolean addRestaurant(Restaurant restaurant)
```

Attempts to add `restaurant` to the store.

Trust no intial **ID** from the `restaurant`, the **ID must** be recalculated and set from the **Repeated ID** field. If you cannot get an **ID** from the **Repeated ID** field do not add the `restaurant`.

You should use the `DataChecker` > `extractTrueID(String[] repeatedID)` method to help you extract the true **ID** to use for the `restaurant`.

The `restaurant` should not be added if it is not valid. See the `DataChecker` > `isValid(Restaurant restaurant)` for more details on whether a inputted `Restaurant` is valid or not.

A valid `restaurant` should not be added if a `Restaurant` with the same ID already exists in the store.

If a duplicate ID is encountered from a valid `restaurant`, the `restaurant` is not added and then the existing `Restaurant` with that ID should be removed from the store. Finally, the duplicate ID should be blacklisted from further use.

A `restaurant` with a blacklisted ID should not be added.

Return `true` if the `restaurant` is successfully added to the store, otherwise return `false`.

Note that there is no ordering on `Restaurant` objects coming into this method, i.e. the next one may be older or newer, or may have a higher or lower ID than the previously recieved `Restaurant`.

Parameters:

`restaurant` - The `Restaurant` to be added into the store.

Returns:

`boolean` - `true` if `restaurant` is added.

`boolean` - `false` if `restaurant` is not added.

Example:

These examples are similar to the ones shown in the other stores.

In the store at the beginning:

`Restaurant A with ID:1112223334445556L`

`Restaurant B with ID:1112223334445557L`

`Restaurant C with ID:1112223334445558L`

Now, we try to add `Restaurant D with ID:1112223334445555L`, this fails because it has an invalid ID.

After, we try to add `Restaurant E with ID:1112223334445557L` and its name field is `null`, this fails because there is a `null` field. Note, we do not blacklist the ID even though the ID exists in the store because `Restaurant E` is an invalid `Restaurant`.

Next, we try to add `Restaurant F with Name:null`, this fails because there is a `null` field.

Then, we try to add `Restaurant G with ID:1112223334445557L`, assume the other fields are valid too, this fails because it is a duplicate ID. We remove `Restaurant B` from the store. We blacklist the ID `1112223334445557L`.

After that, we try to add `Restaurant H with ID:1112223334445557L`, this fails because it is a blacklisted ID.

At the end, we try to add `Restaurant I with ID:1112223334445559L`, we assume the other fields are valid too, this succeeds and is added to the store.

The store at the end:

- `Restaurant A with ID:1112223334445556L`
- `Restaurant C with ID:1112223334445558L`
- `Restaurant I with ID:1112223334445559L`

```
boolean addRestaurant(Restaurant[] restaurants)
```

Attempts to add valid `Restaurant` objects from the `restaurants` input array to the store.

These restaurants are added under the same conditions as specified in above method: `addRestaurant(Restaurant restaurant)`.

Return true if the all the `restaurants` are all successfully added to the data store, otherwise `false`.

Parameters:

`restaurants`	-	The input `Restaurant` array.

Returns:

`boolean`	-	`true` if all the restaurants from `restaurants` are added.
`boolean`	-	`false` if any `restaurant` from `restaurants` is not added.

```
Restaurant getRestaurant(Long id)
```

Returns the `Restaurant` with the matching ID `id` from the store, otherwise this method should return `null` if not found.

Parameters:

`id`	-	The ID of the `Restaurant` you wish to get.

Returns:

`Restaurant`	-	The found `Restaurant`.
`Restaurant`	-	`null` if not found.

```
Restaurant[] getRestaurants()
```

Returns an array of all the restaurants in the store, sorted in ascending order of **ID**.

Returns:

`Restaurant[]` - All stored restaurants, sorted in ascending order of **ID**.

`Restaurant[]` - A `Restaurant[]` of length 0, if otherwise.

```
Restaurant[] getRestaurants(Restaurant[] restaurants)
```

Returns the input array `restaurants` sorted in ascending order of **ID**.

Parameters:

`restaurants` - The input `Restaurant` array.

Returns:

`Restaurant[]` - Input `restaurants` sorted in ascending order of **ID**.

`Restaurant[]` - A `Restaurant[]` of length 0, if otherwise.

```
Restaurant[] getRestaurantsByName()
```

Returns an array of all the restaurants in the store, the returned array should be sorted alphabetically by restaurant **Name**.

If they have the same restaurant **Name**, then it is sorted in ascending order of **ID**.

In sorting, the **Name** field is case-insensitive.

Returns:

`Restaurant[]` - All stored restaurants, sorted alphabetically by **Name**, if same then in ascending order of **ID**.

`Restaurant[]` - A `Restaurant[]` of length 0, if otherwise.

Example:

Index	Name	ID
0	""	4445556667778889L
1	"Alamo Freeze"	8884445556667779L
2	"Bob's Burgers"	2223334445556667L
3	"Bob's Burgers"	3334445556667778L
4	"MacLaren's Pub"	2225556667778883L

```
Restaurant[] getRestaurantsByDateEstablished()
```

Returns an array of all the restaurants in the store, sorted by **Date Established**, from oldest to most recent. Compare dates to the millisecond.

If they have the same **Date Established**, then it is sorted alphabetically by the restaurant **Name**. If they have the same restaurant **Name**, then it is sorted in ascending order of their **ID**.

In sorting, the **Name** field is case-insensitive.

Returns:

`Restaurant[]` - All stored restaurants, sorted by **Date Established**, from old to new, if same then alphabetically by **Name**, if same then in ascending order of **ID**.

`Restaurant[]` - A `Restaurant[]` of length 0, if otherwise.

Example:

Index	Date Est.	Name	ID
`0`	`2004`	`"The Three Broomsticks"`	`8884445556667779L`
`1`	`2008`	`"Pizza Planet"`	`2223334445556667L`
`2`	`2008`	`"Pizza Planet"`	`3334445556667778L`
`3`	`2020`	`"El Jefe"`	`2225556667778883L`
`4`	`2020`	`"The Krusty Krab"`	`1115556667778882L`

`Restaurant[] getRestaurantsByDateEstablished(Restaurant[] r)`

Returns the input array `Restaurant[] r` sorted by **Date Established**, from oldest to most recent. Compare dates to the millisecond.

If they have the same **Date Established**, then it is sorted alphabetically by the restaurant **Name**. If they have the same restaurant **Name**, then it is sorted in ascending order of their **ID**.

In sorting, the **Name** field is case-insensitive.

Parameters:

`r` - The input `Restaurant` array.

Returns:

`Restaurant[]` - Input `r` sorted by **Date Established**, from old to new, if same then alphabetically by **Name**, if same then in ascending order of **ID**.

`Restaurant[]` - A `restaurants[]` of length 0, if otherwise.

```
Restaurant[] getRestaurantsByWarwickStars()
```

Returns an array of all the restaurants in the store that have **at least** 1 Warwick Star, sorted in descending order of **Warwick Stars**.

If they have the same **Warwick Stars**, then it is sorted alphabetically by the restaurant **Name**. If they have the same restaurant **Name**, then it is sorted in ascending order of their **ID**.

In sorting, the **Name** field is case-insensitive.

Returns:

`Restaurant[]` - All stored restaurants with at least 1 Warwick Star, sorted in descending order of **Warwick Stars**, if same then alphabetically by **Name**, if same then in ascending order of **ID**.

`Restaurant[]` - A `Restaurant[]` of length 0, if otherwise.

Example:

Index	Warwick Stars	Name	ID
0	3	"Moe's"	8884445556667779L
1	3	"The Queen Victoria"	2224445556667778L
2	2	"The Banana Stand"	1114445556667778L
3	2	"The Banana Stand"	2225556667778884L

```
Restaurant[] getRestaurantsByRating(Restaurant[] restaurants)
```

Returns the input array `restaurants` sorted in descending order of **Rating**.

If they have the same **Rating**, then it is sorted alphabetically by the restaurant **Name**. If they have the same restaurant **Name**, then it is sorted in ascending order of their **ID**.

In sorting, the **Name** field is case-insensitive.

Parameters:

`restaurants` - The input `Restaurant` array.

Returns:

`Restaurant[]` - Input restaurants sorted in descending order of **Rating**, if same then alphabetically by **Name**, if same then in ascending order of **ID**.

`Restaurant[]` - A `Restaurant[]` of length 0, if otherwise.

Example:

Index	Rating	Name	ID
`0`	`4.0`	`"Ten Forward"`	`8884445556667779L`
`1`	`3.5`	`"The Cafeteria"`	`1112223334445556L`
`2`	`3.5`	`"The Cafeteria"`	`3334445556667778L`
`3`	`2.1`	`"Monk's Cafe"`	`2225556667778884L`

```
RestaurantDistance[] getRestaurantsByDistanceFrom (float lat,
                                                   float lon)
```

Returns an array of `RestaurantDistance`, that is sorted in ascending order of distance from the input coordinates, `lat` and `lon`, the returned array is calculated using all the restaurants in the store.

You should implement the method `inKilometres(float lat, float lon)` from `HaversineDistanceCalculator` to help you calculate the distance in kilometres between two locations given their latitudes and longitudes.

If they have the same **Distance**, then it is sorted in ascending order of their **ID**.

Parameters:

`lat` - The latitude of the location where you want the distance from.

`lon` - The longitude of the location where you want the distance from.

Returns:

`RestaurantDistance[]` - All stored restaurants with distance in km from the input coordinates, sorted in ascending **Distance**, if same then in ascending order of **ID**.

`RestaurantDistance[]` - A `RestaurantDistance[]` of length 0, otherwise.

Example:

Index	Distance (km)	ID
`0`	`0.6`	`8882223334445556L`
`1`	`0.7`	`1112223334445556L`
`2`	`0.7`	`7772223334445556L`

```
RestaurantDistance[] getRestaurantsByDistanceFrom(Restaurant[] r,
                                                  float lat,
                                                  float lon)
```

Returns an array of `RestaurantDistance`, that is sorted in ascending order of distance from the input coordinates, `lat` and `lon`, the returned array is calculated using the input array `r` — so only the restaurants from the input array `r` will be processed.

You should implement the method `inKilometres(float lat, float lon)` from `HaversineDistanceCalculator` to help you calculate the distance in kilometres between two locations given their latitudes and longitudes.

If they have the same **Distance**, then it is sorted in ascending order of their **ID**.

If any restaurant from the array you are given is an invalid restaurant you should not proceed and return a `RestaurantDistance[]` of length 0.

Make sure you recalculate the **ID** from the **Repeated ID** for each restaurant you are given.

Parameters:

`lat`	-	The latitude of the location where you want the distance from.
`lon`	-	The longitude of the location where you want the distance from.

Returns:

`RestaurantDistance[]`	-	The input restaurants with distance in km from the input coordinates, sorted in ascending **Distance**, if same then in ascending order of **ID**.
`RestaurantDistance[]`	-	A `RestaurantDistance[]` of length 0, otherwise.

```
Restaurant[] getRestaurantsContaining(String str)
```

Return an array of all the restaurants from the store whose **Name**, `Cuisine` or `Place` name contain the given query `str`.

Search queries are **accent-insensitive** and **case-insensitive**. Ignore leading and trailing spaces. Also, ignore multiple spaces, only use the one space.

The `Cuisine` `FishAndChips` and `Cuisine` `SouthAmerican` should be found when searching for `"Fish And Chips"` and `"South American"` respectively. Searching for `"FishAndChips"` with no spaces should yield no results, unless the restaurant or place name includes that.

You should use the `ConvertToPlace` > `convert(float lat, float lon)` method to get the `Place` data of a restaurant.

Use the `StringFormatter` > `convertAccentsFaster(String str)` method to strip off accents.

The returned array is sorted alphabetically by **Name**, if they have the same **Name**, then it is sorted in ascending order of **ID**. In sorting, the **Name** field is case-insensitive.

The empty string `""` query should return a `Restaurant[]` of length 0.

Parameters:

`str`	-	Search the **Name**, `Cuisine` or `Place` fields of all the restaurants to see if it contains this query `String`.

Returns:

`Restaurant[]` - Array of restaurants whose **Name**, `Cuisine` or `Place` name contains the input query `str`, ordered by their **Name**, if same by ascending order of **ID**.

`Restaurant[]` - A `Restaurant[]` of length 0, if otherwise.

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.stores.RestaurantStore;
```

```java
// Constructs RestaurantStore
RestaurantStore r = new RestaurantStore();

// Add null restaurant, should return false
boolean addedRestaurant = r.addRestaurant((Restaurant) null);
System.out.println(addedRestaurant);

// Tries to get sorted by ID restaurants, should return 0-length array
Restaurant[] sortedRestaurants = r.getRestaurants();
System.out.println(sortedRestaurants.length);
```

Related Models

- `Restaurant`
- `RestaurantDistance`
- `Place`
- `Cuisine`
- `EstablishmentType`
- `PriceRange`

Related Methods

- `DataChecker` > `extractTrueID(String[] repeatedID)`
- `DataChecker` > `isValid(Restaurant restaurant)`
- `HavesineDistanceCalculator` > `inKilometres(float lat1, loat lon1, float lat2, float lon2)`
- `StringFormatter` > `convertAccentsFaster(String str)`
- `ConvertToPlace` > `convert(float lat, float lon)`

Util

ConvertToPlace

Method Summary

Modifier	Method Name and Description
`Place`	`convert(float latitude, float longitude)` Returns the `Place` corresponding to the given `latitude` and `longitude`.
`Place[]`	`getPlacesArray()` Returns all the places you can search for in the form of a `Place` array.

Constructor

```
public ConvertToPlace()
```

Constructs a new `ConvertToPlace`.

Method Notes

```
Place convert(float latitude, float longitude)
```

Searches through all the places to find a match with the given `latitude` and `longitude`.

If found, returns the `Place` that matches.

If no matching `Place` found, return the default `Place`:

```
new Place("", "", 0.0f, 0.0f);
```

The data we have given you is unique, meaning there are no duplicate `latitude` and `longitude` pairs.

Parameters:

`latitude` - The latitude to be found.
`longitude` - The longitude to be found.

Returns:

`Place` - The `Place` found.
`Place` - If no match found, returns the default `Place`.

```
Place[] getPlacesArray()
```

Returns a `Place` array of all the places you can search through.

Hint: You should initialize this in your constructor, as you do not want to load this every time you convert since this is a very expensive operation.

Returns:

`Place[]` - `Place` array of all the places.

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.util.ConvertToPlace;
```

```java
// Construct ConvertToPlace
ConvertToPlace c = new ConvertToPlace();

// Load places array
Place[] places = c.getPlacesArray();

// Find place from given latitude and longitude
Place foundPlace = c.convert(places[0].getLatitude(),
                             places[0].getLongitude());

// Print found place
System.out.println(foundPlace);
```

Related Model

- `Place`

Related Method

- `RestaurantStore` > `getRestaurantsContaining(String str)`

DataChecker

Method Summary

Modifier	Method Name and Description
`long`	`extractTrueID(String[] repeatedID)` Returns the true ID extracted from the repeated ID, if it exists.
`boolean`	`isValid(Long id)` Returns if the `id` is valid.
`boolean`	`isValid(Customer customer)` Returns if the `customer` is valid.
`boolean`	`isValid(Favourite favourite)` Returns if the `favourite` is valid.
`boolean`	`isValid(Restaurant restaurant)` Returns if the `restaurant` is valid.

Constructor

```
public DataChecker()
```

Constructs a new `DataChecker`.

Method Notes

```
Long extractTrueID(String[] repeatedID)
```

Returns the true ID extracted from the repeated ID, if it exists.

You are given repeated ID in `String[]` format, each element in the `repeatedID` array is supposed to be a `String` of length 16, but in practice this may vary, it may even be `null`.

If `repeatedID` has more or less than 3 elements, return `null`.

There should be 3 `String` elements in the `repeatedID` array. Compare these elements, the element that appears the most is the true ID.

Formally, if there exists an element that appears at least twice in the `repeatedID` array, we say there is a consensus among the 3, return that element in `Long` format.

If there is no consensus among the 3, return `null`.

You may assume the strings provided will only contains numbers, `"0"` to `"9"`, if not `null`.

Note, this method only extracts the true ID, it does not check that the ID is valid, that is what `isValid(Long id)` is for.

Examples:

`"1112223334445556"` `"1112223334445557"` `"1112223334445556"`

We see that `"1112223334445556"` appears at twice, so then it reaches a consensus. Therefore, we return `1112223334445556L`.

`"1112223334445556"` `"1112223334445557"` `"1112223334445558"`

No string reaches a consensus so we return `null`.

`"1112223334445559"` `null` `"1112223334445559"`

We see that `"1112223334445559"` appears at twice, so then it reaches a consensus. Therefore, we return `1112223334445559L`.

`"12345"` `"2468"` `"12345"`

We see that `"12345"` appears at twice, so then it reaches a consensus. So we return `12345L`.

Parameters:

`repeatedID`	-	The repeated ID in `String[]` format.

Returns:

`Long`	-	The extracted true ID.
`Long`	-	`null` if no ID could be extracted.

```
boolean isValid(Long id)
```

Returns if `id` is valid.

In our case, a valid single-digit number is 1, 2, 3, 4, 5, 6, 7, 8, or 9.

An `id` is valid if it contains 16 valid single-digit numbers and no valid single-digit number appears more that 3 times.

Return `true` if the given `id` is valid, `false` otherwise.

Example:

`1112223334445556L` is valid as no number appears more than 3 times and there are 16 digits.

`1112223334445555L` is invalid as 5 appears more than 3 times.

`1112223334445550L` is invalid as it contains a 0, which is not a valid single-digit number.

`111222333444555L` is invalid as there are only 15 digits.

Parameters:

`id` - The ID.

Returns:

`boolean` - `true` if `id` is valid.
`boolean` - `false` if `id` is invalid.

```
boolean isValid(Customer customer)
```

Returns if `customer` is valid.

A valid `Customer` is not null, nor should any of its fields be null.

A valid `Customer` has a valid ID. See `isValid(Long id)`.

Return `true` if the given `customer` is valid, `false` otherwise.

Parameters:

`customer` - The customer.

Returns:

`boolean` - `true` if `customer` is valid.
`boolean` - `false` if `customer` is invalid.

```
boolean isValid(Favourite favourite)
```

Returns if `favourite` is valid.

A valid `Favourite` is not null, nor should any of its fields be null.

A valid `Favourite` has a valid ID, a valid `Customer` ID and a valid `Restaurant` ID. See `isValid(Long id)`.

Return `true` if the given `favourite` is valid, `false` otherwise.

Parameters:

`favourite` - The favourite.

Returns:

`boolean` - `true` if `favourite` is valid.
`boolean` - `false` if `favourite` is invalid.

```
boolean isValid(Restaurant restaurant)
```

Returns if `restaurant` is valid.

A valid `Restaurant` is not null, nor should any of its fields be null.

A valid `Restaurant` has a valid ID. See `isValid(Long id)`.

A valid `Restaurant` cannot have a last inspected date be before the date it was established.

The food inspection rating of a valid `Restaurant` can only be: 0, 1, 2, 3, 4, 5.

The number of Warwick Stars a valid `Restaurant` has can only be: 0, 1, 2, 3.

The customer rating of a valid `Restaurant` can only be 0.0f or be between 1.0f and 5.0f inclusive.

Note, when you call this, make sure you have set the ID by getting the true ID from the repeated ID, otherwise the ID would remain the default at -1 and this method would return `false` every time.

Return `true` if the given `restaurant` is valid, `false` otherwise.

Parameters:

`restaurant`	-	The restaurant.

Returns:

`boolean`	-	`true` if `restaurant` is valid.
`boolean`	-	`false` if `restaurant` is invalid.

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.util.DataChecker;
```

```java
// Constructs DataChecker
DataChecker d = new DataChecker();

// Extracts true ID
Long trueID = d.extractTrueID(new String[]{"1112223334445556",
                                           "1112223334445556",
                                           "1112223334445557"});
System.out.println(trueID);

// Check if valid
boolean isIDValid = d.isValid((Long) null);
boolean isCustomerValid = d.isValid((Customer) null);
boolean isFavouriteValid = d.isValid((Favourite) null);

if (!isIDValid && !isCustomerValid && !isFavouriteValid) {
        System.out.println("Everything is null!");
}
```

Related Models

- `Customer`
- `Favourite`
- `Restaurant`

Related Methods

- `CustomerStore` > `addCustomer(Customer c)`
- `CustomerStore` > `addCustomer(Customer[] c)`
- `FavouriteStore` > `addFavourite(Favourite f)`
- `FavouriteStore` > `addFavourite(Favourite[] f)`
- `RestaurantStore` > `addRestaurant(Restaurant r)`
- `RestaurantStore` > `addRestaurant(Restaurant[] r)`

HaversineDistanceCalculator

Method Summary

Modifier	Method Name and Description
`float`	`inKilometres(float lat1, float lon1, float lat2, float lon2)` Returns the distance in kilometres (to 1 dp) between location 1 defined by `lat1` and `lon1` and location 2 defined by `lat2` and `lon2`.
`float`	`inMiles(float lat1, float lon1, float lat2, float lon2)` Returns the distance in miles (to 1 dp) between location 1 defined by `lat1` and `lon1` and location 2 defined by `lat2` and `lon2`.

Method Notes

```
static float inKilometres(float lat1, float lon1,
                          float lat2, float lon2)
```

Returns the distance in kilometres (to 1 dp) between location 1 defined by `lat1` and `lon1` and location 2 defined by `lat2` and `lon2`.

The formula for calculating the distance in kilometres is:

$$a = sin^2(\frac{\varphi_2 - \varphi_1}{2}) + cos(\varphi_1) * cos(\varphi_2) * sin^2(\frac{\lambda_2 - \lambda_1}{2})$$

$$c = 2 * arcsin(\sqrt{a})$$

$$d = R * c$$

Where:

φ_1 - The latitude of location 1, in radians.

φ_2 - The latitude of location 2, in radians.

λ_1 - The longitude of location 1, in radians.

λ_2 - The longitude of location 2, in radians.

R - The Earth's radius, 6372.8 km.

d - The calculated distance in km between location 1 and 2.

We want the distance to be in kilometres and to be rounded to 1 decimal place.

Note, the input parameters must be in degrees but the formula uses radians.

Parameters:

`lat1` - The latitude of location 1, in degrees.

`lon1` - The longitude of location 1, in degrees.

`lat1` - The latitude of location 2, in degrees.

`lon2` - The longitude of location 2, in degrees.

Returns:

`float` - The distance in kilometres (to 1 dp) between the two locations.

```
static float inMiles(float lat1, float lon1,
                     float lat2, float lon2)
```

Returns the distance in miles (to 1 dp) between location 1 defined by `lat1` and `lon1` and location 2 defined by `lat2` and `lon2`.

See the `inKilometres` method for the formula to calculate the distance in kilometres between the locations, then convert it into miles by dividing by the value `kilometresInAMile`, which is `1.609344f`.

Return the distance in miles, rounded to 1 decimal place.

Important note, you cannot call the `inKilometres` method directly and then do the division, because you will lose precision.

Parameters:

`lat1`	-	The latitude of location 1, in degrees.
`lon1`	-	The longitude of location 1, in degrees.
`lat1`	-	The latitude of location 2, in degrees.
`lon2`	-	The longitude of location 2, in degrees.

Returns:

`float`	-	The distance in miles (to 1 dp) between the two locations.

Example Code

```
// The import statement
import uk.ac.warwick.cs126.util.HaversineDistanceCalculator;
```

```
// Calculate distance in kilometres, should be 0.5 km
float distanceInKM = HaversineDistanceCalculator.inKilometres(
                52.3838f, -1.560065f, 52.379049f, -1.560898f);
System.out.println(distanceInKM);

// Calculate distance in miles, should be 0.3 miles
float distanceInMiles = HaversineDistanceCalculator.inMiles(
                52.3838f, -1.560065f, 52.379049f, -1.560898f);
System.out.println(distanceInMiles);
```

Related Methods

- `RestaurantStore` > `getRestaurantsByDistanceFrom(float lat, float lon)`
- `RestaurantStore` > `getRestaurantsByDistanceFrom(Restaurant[] rs, float lat, float lon)`

StringFormatter

Method Summary

Modifier	Method Name and Description
`String`	`convertAccents(String str)` Returns the `String str` but with accents removed.
`String`	`convertAccentsFaster(String str)` Same as `convertAccents(String str)` but faster.

Method Notes

```
static String convertAccents(String str)
```

Returns the `String str` but with accents removed.

A hard-coded multi-dimensional `String` array, `accentAndConvertedAccent`, shows you what an accent converts to.

This method loops through the array and replaces any accents in the `str`.

If the input `str` is `null`, this method returns the empty `String`, `""`.

Parameters:

`str`	-	The `String` to be converted.

Returns:

`String`	-	The `String str` converted with no accents.
`String`	-	If the input `str` is `null`, returns the empty `String`, `""`.

```
static String convertAccentsFaster(String str)
```

The `convertAccents` method is slow, find a faster way of converting accents.

We are looking for at least a 2.5x speed up.

The output of this method should still be the same as the `convertAccents` method.

Hint: Use the `static` initializer block to initialize something to help you.

Parameters:

`str`	-	The `String` to be converted.

Returns:

`String`	-	The `String str` converted with no accents.
`String`	-	If the input `str` is `null`, returns the empty `String`, `""`.

Example Code

```java
// The import statement
import uk.ac.warwick.cs126.util.StringFormatter;
```

```java
// Converts Á to A
String convertedString = StringFormatter.convertAccents("Á");
System.out.println(convertedString);

// Converts Á to A but faster
String convertedStringFast = StringFormatter.convertAccentsFaster("Á");
System.out.println(convertedStringFast);
```

Related Methods

- `CustomerStore` > `getCustomersContaining(String str)`
- `RestaurantStore` > `getRestaurantsContaining(String str)`

Testing

To help speed up the process of debugging your code we have written some tests for you to use and adapt. Note, some test methods are incomplete, these are labelled as TODO. Additionally, you should write more tests than the ones provided, we have only given you the bare minimal to get started.

This part of the coursework is **not** assessed, it is simply here to aid you in its design.

The tests, located inside `/src/main/java/uk/ac/warwick/cs126/test/`, are:

- `TestRunner.java`
- `TestTheConstructorsAndInitializers.java`
- `TestTheCustomerStore.java`
- `TestTheFavouriteStore.java`
- `TestTheRestaurantStore.java`
- `TestTheUtils.java`

You are free to modify or add any classes in the `/test` folder.

To run the tests in **Linux/macOS** Terminal:

```
./build.sh -t
```

In **Windows** Command Prompt:

```
run -t
```

In **Windows** PowerShell:

```
.\run.bat -t
```

The given script file will compile the `TestRunner` class and its dependencies, then it will run the main class of `TestRunner`.

Loading Test Data

As creating new objects in code can get tedious we have made it so you can load your own data files from the `/data` folder. For example, in `TestTheCustomerStore.java` we can load customers with:

```
        Customer[] customers = customerStore.loadCustomerDataToArray(
            loadData("/test-customer/customer-10.csv"));
```

The `loadData(String s)` function gives you a relative path to the `/data` folder, then combined with the input string `s` you can define a file to load from the `/data` folder.

Take a look at each `test-*` folder files to see the format for each store. Note, `Review` TSV data fields and `placeData.tsv` fields are separated by tabs, the rest are separated by commas.

Additionally, `placeData.tsv` cannot be moved from where it resides, otherwise it will not load in your `ConvertToPlace` tests, you can modify its contents but you cannot move it from that location. The rest of the data files have no restriction to where they are placed as long as they reside in the `/data` folder.

If you wish look at the full data the website loads, run the script with argument `-d`:

In **Linux/macOS** Terminal:

```
./build.sh -d
```

In **Windows** Command Prompt:

```
run -d
```

In **Windows** PowerShell:

```
.\run.bat -d
```

This will copy the full data into the `/data/full` directory. The `placeData.tsv` you have is already the full data, but if you want to cut it down so that your tests will load faster, know that you can get back the original file from running the script with that argument.

Report

For this coursework you are required to write a short report to summarise your solution. In this report you should give:

- A brief explanation of your design choices for each store and util implementation.
- Space complexity details for the required classes.
- Time complexity details for the required methods.

This document is to help consolidate in one place how you did your solution, so that we will be able to understand the advantages and disadvantages of your solution.

Mark Breakdown

The marks for this coursework will be allocated as follows:

Area	Mark Allocation
CustomerStore	25%
FavouriteStore	25%
RestaurantStore	25%
Report, Comments and Coding Practices	25%

Util Allocation

The `util` classes will be marked in conjunction with the stores, in the following way:

- `CustomerStore`
  - `StringFormatter`
- `RestaurantStore`
  - `HaversineDistanceCalculator`
  - `ConvertToPlace`

Store Mark Criteria

The stores will be marked according to the following criteria:

- **Correctness**

  Checks whether the solution follows the given specification. This will be assessed via automated tests. These tests checks for all the various cases that could occur for an implemented method. In these tests, each solution is given an appropriate amount of time to run, if a solution exceeds this time limit for a single test that will result in a failed test — the time limit is generous so if it fails from that, the method it tested must have been very inefficient.

- **Design, Understanding and Efficiency**

  Looks to see if appropriate design decisions have been made and if the student shows understanding. Code is looked at via inspection and tests to see if the coursework is efficiently designed, and that the student has justifiable time and space complexity for each parts of the solution.

Report, Comments and Coding Practices Criteria

This area of the coursework will be marked according to the following criteria:

- **Report**

  Looks to see if a solution was explained well and in a succinct manner. Looks to see if the student shows understanding on their solution's complexity.

- **Comments**

  Looks to see if a solution source files are properly documented, and that there are relevant comments where code gets complicated or ambiguous. Looks to see that the student avoids obvious comments.

- **Coding Practices**

  Looks to see if a solution follows good coding practices. This means consistent indentation, relevant and consistent names for variables/methods/classes, and have proper bracing. Also, it means that code is encapsulated properly and code is modularised so that there is no repeated code.

Note, the comments and coding practices mark is a total mark for all the stores. So if you did not complete a store, it will negatively affect the mark in this area too.