Zig Zag Education

Computer Science

A Level | AQA | 7517

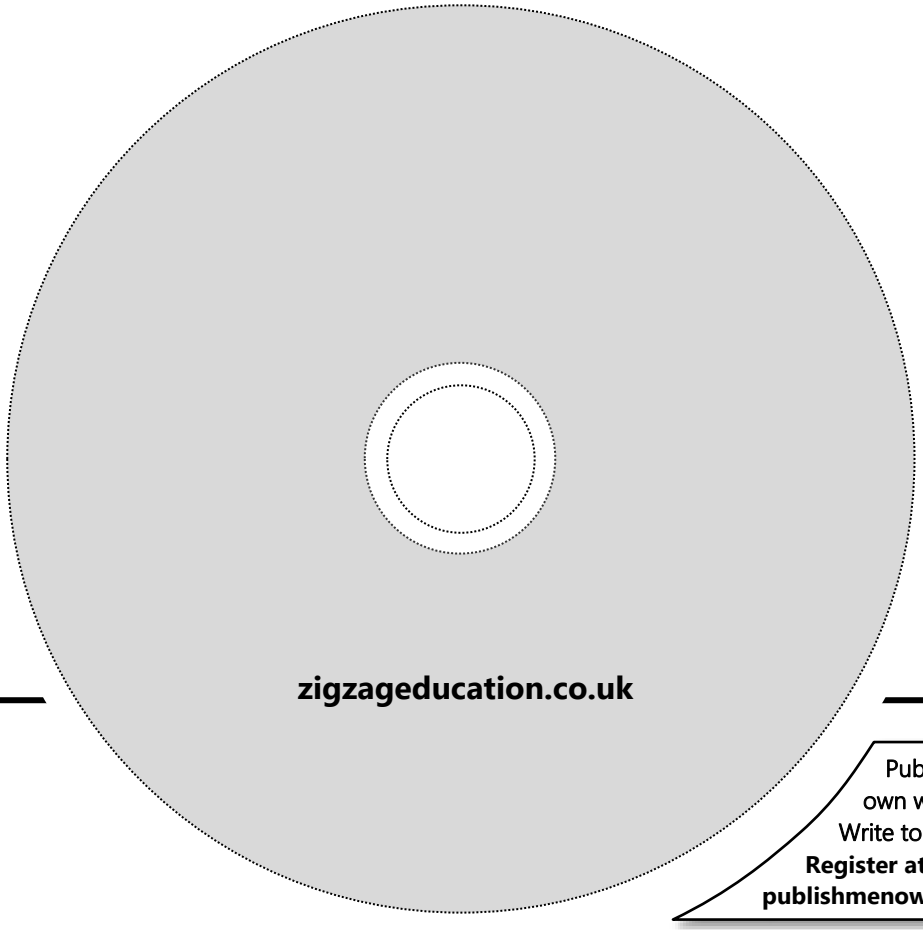
2015 specification for the 2020 exam

PAPER 1 EXAM RESOURCE PACK 2020

Food Magnate Simulation

for A Level AQA Computer Science

VB.NET EDITION



DC10/ 9841

POD 9841

zigzageducation.co.uk

Contents

Product Support

Get Product Updates

Occasionally we make improvements to resources after you receive them. Download the updated content **free of charge**.

Download Support Files

Any support files for your purchased resources are available for download. This includes HTML links pages for resources that contain lots of hyperlinks.

Send us Your Feedback

For every completed review, **get a £10 voucher** to use on your next order! Tell us what you thought, and report any issues or ideas for improvement.

Get New Resource Notifications

Opt in to receive email alerts about new resources for your subject(s).

Contact Us

zigzageducation.co.uk
sales@zigzageducation.co.uk

0117 950 3199

ZigZag Education
Unit 3 Greenway Centre
Doncaster Road
Bristol BS10 5PY

Register today via:

ZigZagEducation.co.uk → Computer Science & IT → Product Support

Quick link: zzed.uk/PS

Terms and Conditions of Use

Terms and Conditions

Please note that the **Terms and Conditions** of this resource include point 5.3, which states:

"You acknowledge that you rely on your own skill and judgement in determining the suitability of the Goods for any particular purpose."

"We do not warrant: that any of the Goods are suitable for any particular purpose (e.g. any particular qualification), or the results that may be obtained from the use of any publication, or expected exam grades, or that we are affiliated with any educational institution, or that any publication is authorised by, associated with, sponsored by or endorsed by any educational institution."

Copyright Information

Every effort is made to ensure that the information provided in this publication is accurate and up to date but no legal responsibility is accepted for any errors, omissions or misleading statements. It is ZigZag Education's policy to obtain permission for any copyright material in their publications. The publishers will be glad to make suitable arrangements with any copyright holders whom it has not been possible to contact.

Students and teachers may not use any material or content contained herein and incorporate it into a body of work without referencing/acknowledging the source of the material ("Plagiarism").

Disclaimer

This resource is intended to supplement your teaching only.

It is the teacher's responsibility to decide how to use this resource to assist themselves and their students appropriately. You may simply wish to read this material to better inform yourself and to help you prepare your lessons and to give you ideas for your teaching. You may also consider whether it is appropriate to hand out some of the sheets for reference and to use some of the activities for classwork or homework. You may also consider whether it is appropriate to hand out the booklet to be worked through by your students more independently.

As with all pre-release material, it is the teacher's responsibility to decide in what way to assist their students, and to decide how this resource in particular can be used to fit into that assistance.

The resources here are provided as an interpretation of the pre-release material.
The author does not have any special knowledge of what to expect on any particular exam.

ZigZag Education is not affiliated with AQA, Pearson, Edexcel, OCR, WJEC, CEA, International Baccalaureate Organization or DFE in any way nor is this publication authorised by, associated with, sponsored by or endorsed by these institutions unless explicitly stated on the front cover of this publication.

This publication is designed to supplement teaching only. Practice questions may be designed to follow the content of a specification and may also attempt to prepare students for the type of questions they will meet in the examination, but will not attempt to predict future examination questions. ZigZag Education do not make any warranty as to the results that may be obtained from the use of this publication, or as to the accuracy, reliability or content of the publication.

Where the teacher uses any of the material from this resource to support examinations or similar then the teacher must ensure that they are happy with the level of information and support provided pertaining to their personal point of view and to the constraints of the specification and to others involved in the delivery of the course. It is considered essential that the teacher adapt, extend and/or censor any parts of the contained material to suit their needs, the needs of the specification and the needs of the individual or group concerned. As such, the teacher must determine which parts of the material, if any, to provide to the students and which parts to use as background information for themselves. Likewise, the teacher must determine what additional material is required to cover all points on the specification and to cover each specification point to the correct depth.

Teacher's Introduction

This resource pack is designed to help you support your students taking the **A Level Computer Science Paper 1** examination. It is based on the **'Food Magnate Simulation'** preliminary material (VB.NET) – for examination June 2020.

On the CD, you will find the following:

- `FoodMagnateSim` this folder contains all of the content (PDF/DOCX) accessible via a HTML interface
- `Passwords.txt` for teacher use – this file contains all of the passwords for the protected PDFs (also listed below)

* PRINTED COPIES OF ALL THE MATERIALS IN THIS DIGITAL RESOURCE PACK ARE INCLUDED FOR REFERENCE.

Installation: Copy the entire `FoodMagnateSim` folder onto a network location that is accessible for students, and provide them with a shortcut to the index.html file. All content can be accessed from this page.

Passwords: All of the PDFs accessible via the *Solutions* web page are password-protected, so that students can only access them with your permission. Each password is a four-digit code, as follows:

`vb01-Commentary.pdf`	`1158`
`vb06a-ClassDiagramTasksMS.pdf`	`3091`
`vb06b-ClassDiagramFull.pdf`	`5630`
`vb07-TheoryQuestionsMS.pdf`	`7642`
`vb08-ProgrammingTasksMS.pdf`	`2966`

Should you wish to give students access to ALL protected-PDFs, the master password for all files is:

`zz2ghc4`

The resource pack consists of the following:

① **Pre-release Commentary** including a general explanation of how the program works (text and video), plus a more detailed, technical overview* describing each subroutine, class and attribute in turn.

> ***Note:** although this section is intended to give extra support to teachers and students, it should in no way be seen as a substitute to a student exploring the code for themselves. For this reason, this content has been placed on the 'Answers & Solutions' HTML page as a password-protected file, to allow you to control if/when students access it.*

② **Class Diagram Tasks**

Three partially complete UML class diagram tasks for students to complete while getting to grips with the skeleton program. Students must add any missing attributes, subroutines, access modifiers, parameters and return types. Completed versions are provided via the *Solutions* web page as a password-protected PDF.

③ **Written Questions**

Theory questions testing students' understanding of the skeleton program, like Section C in the exam. These questions require access to the program, but no modifications need to be made to the program. Write-on (with answer lines) and non-write-on version are available format. Suggested answers are provided via the *Solutions* web page as a password-protected PDF.

④ **Programming Tasks**

Fifteen modification exercises put students' programming skills to the test, like Section D in the exam. Example solutions with suggested mark schemes are provided via the *Solutions* web page as a password-protected PDF. Note that these are example solutions and you must use your discretion to award marks accordingly where there are valid alternative solutions.

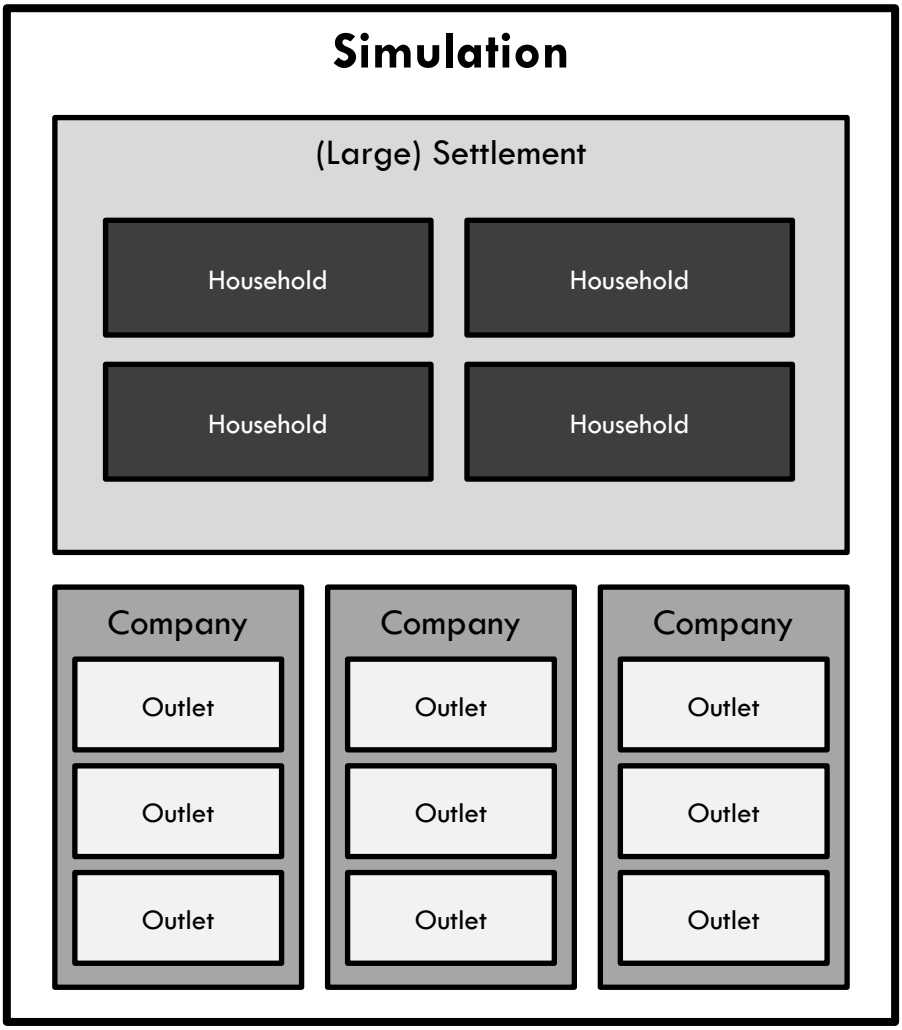
An **Electronic Answer Document (EAD)** is provided should you wish students to use it for ③ and/or ④ above.

This resource is intended to supplement your teaching only. **Please read full disclaimer (p. iii) before using it.**

Food Magnate Simulation

Introduction

The *Food Magnate Simulation* models how profitable different types of restaurant chain would be within a simulated settlement. The structure of the simulation, conceptually, is as follows:



The program contains the following objects:

- A single object of type `Simulation`, which is responsible for constructing, either directly or indirectly, the remaining objects
- A single object of type `Settlement`, which is stored within the `Simulation` object. Alternatively, this can be an object of type `LargeSettlement`, which behaves identically but can cover a larger area and contain more households.
- An unlimited number of `Household` objects, each of which is contained within the settlement. Initially, a settlement begins with 250 of these, but a `LargeSettlement` object might have more.
- An unlimited number of `Company` objects, although the default starting number of companies is three. The `Company` objects are contained directly within the `Simulation` object.
- An unlimited number of `Outlet` objects, with each outlet being stored in a particular `Company` object. A company cannot exist without at least one outlet.

Let's look at the attributes for each class in detail...

Class: Simulation

The `Main` subroutine creates the program's single `Simulation` object and calls its `Run` subroutine. From that point onwards, it is the `Simulation` object that is responsible for creating and managing all other objects. Its attributes are as follows:

- A single `Settlement` object, `SimulationSettlement`; since the `Settlement` object contains the `Household` objects, the simulation cannot interact directly with a household without calling an accessor subroutine in the `Settlement` class
- An integer variable to store the number of companies, `NoOfCompanies`
- A float called `FuelCostPerUnit`, which is used to calculate delivery costs between outlets belonging to the same company; it is passed to the constructor of the `Company` class
- A second float, called `BaseCostForDelivery`, which is also passed to the `Company` constructor and subsequently used to calculate delivery costs
- An ArrayList called `Companies`, to store objects of type `Company`; a list is a better choice than an array, since any number of companies can be stored
- A random number generator called `Rnd`, used to generate random events between days

Class: Company

A company can be either a fast-food company, a family company or a named chef company. This means that all outlets belonging to a company will also be of that type; a company cannot have, for example, some fast-food outlets and some family outlets. The `Company` class's attributes are as follows:

- A string, `Name`, to store the name of the company
- A second string, called `Category`; this stores the type of company, and can be either 'fast food', 'family' or 'named chef'
- A series of float attributes:
    - `Balance`: the amount of money a company owns, which can be positive, negative or zero
    - `ReputationScore`: a measure of how well regarded the company is; a company with a high reputation score is more likely to be visited than one with a low reputation score
    - `AvgCostPerMeal`: how much the company pays for a meal
    - `AvgPricePerMeal`: how much a customer pays for a meal when visiting one of the company's outlets
    - `DailyCosts`: single cost per day, per company, initially set to 100, but with a small chance that it will change, up or down, between days
    - `FamilyOutletCost`: the cost of opening an outlet for a 'family' company
    - `FastFoodOutletCost`: the cost of opening an outlet for a 'fast food' company
    - `NamedChefOutletCost`: the cost of opening an outlet for a 'named chef' company
    - `FuelCostPerUnit`: for companies with multiple outlets, this is part of the delivery cost from their 'main' outlet (the first one to be created) and each subsequent outlet
    - `BaseCostOfDelivery`: for companies with any number of outlets, this amount is paid once per day
- An ArrayList called `Outlets`, to store objects of type `Outlet`; again, this offers more flexibility than using an array
- A series of integer attributes:
    - `FamilyFoodOutletCapacity`: capacity for a 'family' outlet, initialised to 150
    - `FastFoodOutletCapacity`: capacity for a 'fast food' outlet, initialised to 200
    - `NamedChefOutletCapacity`: capacity for a 'named chef' outlet, initialised to 50
- A random number generator called `Rnd`, used to generate random reputation scores

Class: Outlet

An `Outlet` object is stored within a data structure in a `Company` object, modelling the fact that an outlet is owned by a single company. Outlets are also associated with the settlement, since each outlet has a location within that settlement, and households, if they choose to visit a company's outlet, will always visit the closest one.

The `Outlet` class's attributes are as follows:

- A series of integer attributes:
    - `VisitsToday`: the number of times a household has visited this outlet on the current day; it is worth noting that there is nothing preventing this exceeding the outlet's capacity
    - `XCoord` and `YCoord`: the outlet's location within the settlement
    - `Capacity`: how many seats are in the outlet, used to calculate daily costs
    - `MaxCapacity`: the maximum to which the capacity can be extended, also used to calculate daily costs
- A float, `DailyCosts`, which is how much the outlet costs to run each day, irrespective of visitors; this is not fixed, and can be changed as capacity changes. It could also be changed as a result of a call to the `Outlet` class's `AlterDailyCost` subroutine; however, this is never called, nor is `GetCapacity`.
- A random number generator called `Rnd`, used to generate `MaxCapacity` values in this class's constructor

Class: Settlement

The simulation contains a single settlement, and each household and outlet has a location within that settlement. The settlement is easy to visualise in terms of a grid, but it is not stored as a two-dimensional array. The reason for this is twofold. Firstly, most elements within a settlement array would be empty. The default settings create a settlement with one million possible locations, but only 250 households and 12 company outlets. Secondly, a settlement contains objects of different types – namely households and outlets. Instead, each outlet and each household stores its own X and Y coordinates, which must be within the bounds of the settlement.

The `Settlement` class's attributes are as follows:

- An integer variable, `StartNoOfHouseholds`, which is how many households exist at the start of the simulation. Based on existing code, this can go up but not down as the simulation runs. The default value is 250.
- Two further integers, `XSize` and `YSize`, which store between them the size of the settlement. The default value for each of these is 1,000, meaning there are one million possible locations.
- An ArrayList, `Households`, to store each `Household` object
- A random number generator called `Rnd`, used to generate random locations for new households

The `LargeSettlement` class inherits from `Settlement`, and allows the user to add to (but not subtract from) the values of `StartNoOfHouseholds`, `XSize` and `YSize`.

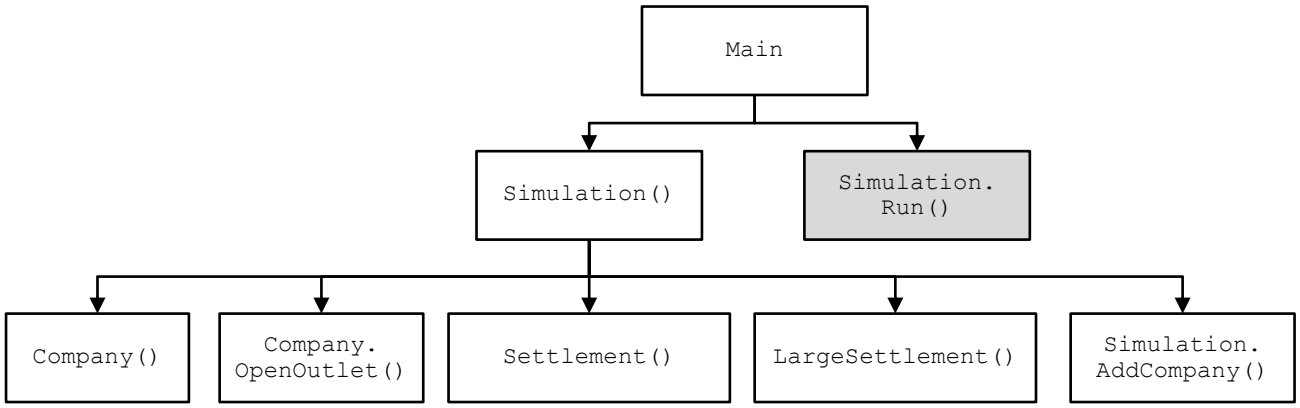
Class: Household

To all intents and purposes, this class models a consumer, since the whole household either eats out or does not. There is scope here for modelling households of different sizes, or households in which some members (but not all) go out to eat, or even households where different individuals eat out at different outlets at the same time. The `Household` class's attributes are as follows:

- A float, `ChanceEatOutPerDay`, storing a value between 0 and 1, representing the probability of the household going out to eat
- Integer variables `XCoord` and `YCoord`, representing a household's location in the settlement
- A static integer, `NextID`, which numbers each new household incrementally, starting from 1
- An additional integer, `ID`, which stores the value contained in `NextID` at point of instantiation

Overview

When the program begins:



1. A new `Simulation` object is constructed
2. That `Simulation` object constructs a new object of type `Settlement` or `LargeSettlement` (which inherits from `Settlement`), depending on user input. The `Settlement` constructor calls the constructor of `Household` repeatedly.
3. That `Simulation` object, as part of being constructed, also constructs new `Company` objects, which are either the three default companies hard-coded into the Skeleton Program or user-defined, depending on user input
4. The `AddCompany` subroutine is called if user-defined companies are selected, allowing the user to enter specific details of each company
5. The `Company` constructor will make at least one call to the `Outlet` constructor, since each company must have at least one outlet
6. The `Simulation` object's `Run` subroutine is called (see next hierarchy diagram)

```
**********************************
**********    MENU      **********
**********************************
1. Display details of households
2. Display details of companies
3. Modify company
4. Add new company
6. Advance to next day
Q. Quit

Enter your choice:
```

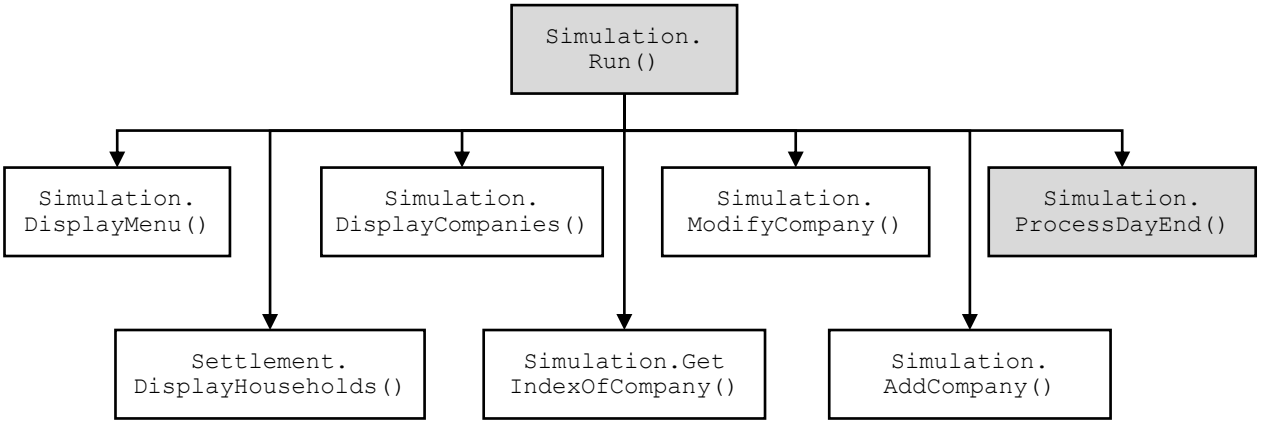
Main menu

```
**********************************
*******  MODIFY COMPANY   *******
**********************************
1. Open new outlet
2. Close outlet
3. Expand outlet

Enter your choice:
```

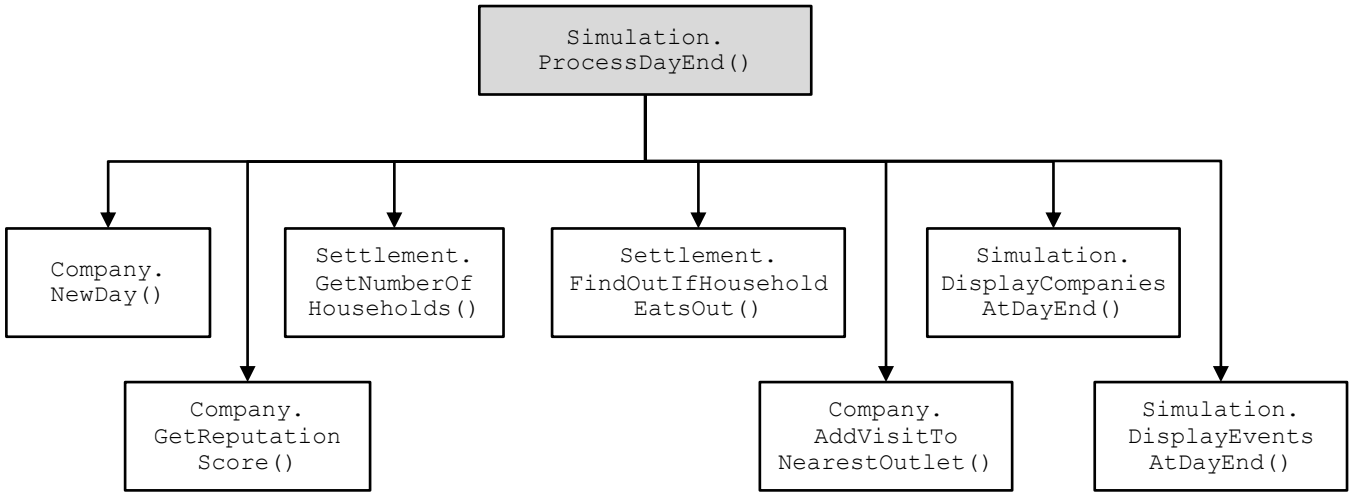
'Modify Company' menu, shown when the user selects option 3 from the main menu and enters a valid company name

When the `Run` subroutine is called:



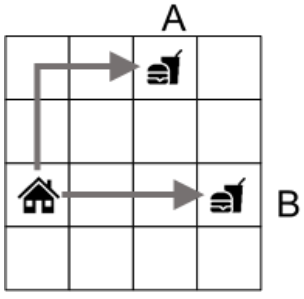
7. The user is presented with a menu as a result of a call to `DisplayMenu`
8. If the user selects option 1, a call is made to `DisplayHouseholds`, which lists details of all households in the settlement via a call to `DisplayHouseholds` in the `Settlement` class, which in turn calls `GetDetails` in each instance of the `Household` class
9. If the user selects option 2, details of all companies are displayed via a call to `DisplayCompanies`, which calls the `GetDetails` subroutine in each `Company` object. The `GetDetails` subroutine in the `Company` class calls a `GetDetails` subroutine for each outlet, meaning that selecting option 2 from the menu results in details of all companies, and their outlets, being displayed.
10. If the user selects option 3, to modify a company, they are prompted for a company name, which is then passed to `GetIndexOfCompany` in order to get the index of that company. The subsequent call to `ModifyCompany` presents the user with a second menu containing three options. They can open a new outlet (`OpenOutlet()`), close an outlet (`CloseOutlet()`) or expand an existing outlet (`ExpandOutlet()`).
11. If the user selects option 4, a new company can be created via a call to `AddCompany`. This prompts the user for details of the new company's name, type and starting balance. The new company will have a single outlet in a random location within the settlement.
12. There is no option 5, which makes it quite likely that adding an option 5 will be part of the exam
13. If the user selects option 6, a call is made to `ProcessDayEnd` in the `Simulation` class. Note that an identically named subroutine exists in the `Company` class, so be sure not to confuse the two. This is the most involved subroutine in the program and is addressed in the next hierarchy diagram.

When each day ends:



14. The call to `NewDay` (in the `Company` class) calls `NewDay` (a subroutine in each `Outlet` class) to reset the number of visits to zero
15. The reputation score of each company is accessed and added to an ArrayList, with each float value therein storing a running total of all reputation scores so far (i.e. the first value stored is for the first company, the second value stored is the sum of the reputation scores for the first two companies, the third value stored is the sum of the reputation scores for the first three companies, and so on)
16. The call to `GetNumberOfHouseholds` is to facilitate a loop through all households in a settlement
17. The call to `FindOutIfHouseholdEatsOut`, for each `Household` object, returns a Boolean, which is more likely to be true for households with a higher probability of eating out. If it is true, a company is selected at random, using the ArrayList from step 15, with companies holding a higher reputation score more likely to be chosen.
18. For the company that is chosen, the nearest outlet to the household eating out is visited. The distances to all outlets belonging to the chosen company are examined, and distances are calculated in the following way:

    - Distances are calculated by assuming movement is only possible north, south, east or west. This means that the distance from the house to outlet A is 4, while the distance from the house to outlet B is 3.
    - In the event that two outlets are of an equal distance from a house, the outlet examined first (i.e. the outlet appearing earlier in the `Outlets` list) will be the one visited.



19. The call to `DisplayCompaniesAtDayEnd` calls the `ProcessDayEnd` subroutine in the `Company` class. This subroutine calculates changes to the company's balance, which is affected by visits to each outlet per day, the price at which meals are bought and sold, and the distance between outlets for the same company, for which the delivery of ingredients incurs a cost based on the price of fuel (distances are calculated as above). The old balance and the new balance are then displayed, along with other details of the company and its outlets.
20. The call to `DisplayEventsAtDayEnd` generates either a random probability of additional households in the settlement, a change of fuel cost for a company chosen at random, a change of reputation for a company chosen at random or a change of daily costs for a company chosen at random

Program Subroutines

The program's functions, Ⓕ, and procedures, Ⓟ, are described below.

'Household' Class

Subroutine	Data	Description
`GetChanceEatOut` Ⓕ	Parameters: – Returns: `ChanceEatOutPerDay: Single` Called From: `Settlement.FindOutIfHouseholdEatsOut` Calls: –	Returns the value of `ChanceEatOutPerDay`
`GetDetails` Ⓕ	Parameters: – Returns: `Details: String` Called From: `Settlement.DisplayHouseholds` Calls: –	1. Declares an empty string 2. Populates it with all attributes other than `NextID` and `Rnd` 3. Returns the string
`GetX` Ⓕ	Parameters: – Returns: `XCoord: Integer` Called From: `Settlement.FindOutIfHouseholdEatsOut` Calls: –	Returns the value of `XCoord`
`GetY` Ⓕ	Parameters: – Returns: `YCoord: Integer` Called From: `Settlement.FindOutIfHouseholdEatsOut` Calls: –	Returns the value of `YCoord`

'Settlement' Class

Subroutine	Data	Description
`AddHousehold` Ⓟ	Parameters: – Returns: – Called From: `Settlement.CreateHouseholds` Calls: –	1. Declare integers `X` and `Y` 2. Set `X` and `Y` to random locations within the settlement 3. Create a new household object using this location 4. Add the household object to the `Households` ArrayList
`CreateHouseholds` Ⓟ	Parameters: – Returns: – Called From: `Settlement` constructor Calls: `Settlement.AddHousehold`	1. Loops `StartNoOfHouseholds` times 2. In each iteration, call `AddHousehold`

Subroutine	Data	Description
`DisplayHouseholds` Ⓟ	Parameters: – Returns: – Called From: `Simulation.Run` Calls: `Household.GetDetails`	1. Outputs 'details of all households' 2. Loops through each household in the `Households` list 3. Outputs the result of a call to the `GetDetails` subroutine of each household
`FindOutIf HouseholdEatsOut` Ⓕ	Parameters: `HouseholdNo: Integer, X: Integer, Y: Integer` Returns: `Boolean` Called From: `Simulation.ProcessDayEnd` Calls: `Household.GetX` `Household.GetY`	1. Generate random float between 0 and 1 2. If this number is less than the probability of the household's (that's the household passed as a parameter) chance of eating out, return 'true' 3. Otherwise, return 'false'
`GetNumberOf Households` Ⓕ	Parameters: – Returns: `Households.Count: Integer` Called From: `Simulation.ProcessDayEnd` Calls: –	Returns the number of items in the `Households` list
`GetRandom Location` Ⓟ	Parameters: – Returns: – Called From: `Simulation.AddCompany` `Settlement.AddHousehold` Calls: –	1. Accepts references to `X` and `Y` as parameters; as these are passed by reference, they change in the subroutine from which `GetRandomLocation` is called, even though there is no return value 2. Sets `X` and `Y` to random values within the bounds of the `Settlement` object
`GetXSize` Ⓕ	Parameters: – Returns: `XSize: Integer` Called From: `Simulation.ModifyCompany` Calls: –	Returns the value of `XSize`
`GetYSize` Ⓕ	Parameters: – Returns: `YSize: Integer` Called From: `Simulation.ModifyCompany` Calls: –	Returns the value of `YSize`

'Outlet' Class

Subroutine	Data	Description
`AlterCapacity` Ⓕ	Parameters: `Change: Integer` Returns: `Change: Integer` Called From: `Company.ExpandOutlet` Calls: –	1. Stores the current capacity in a local variable, `OldCapacity` 2. Adds the parameter to the `Capacity` attribute; the parameter can be a negative number, in which case `Capacity` is decreased 3. If the capacity has, as a result of step (2), risen above `MaxCapacity`, set `Capacity` to `MaxCapacity` and return the difference between `OldCapacity` and `MaxCapacity` (i.e. the amount of increase after capacity was limited) 4. If the capacity has fallen below zero, set `Capacity` to zero 5. Recalculate the daily costs based on the new capacity and return the value of change **NB If there is a negative change attempted that is larger than possible, e.g. an outlet with a capacity of 3 has this subroutine called with a parameter of -5, it is -5 that is then returned, even though the capacity only goes down by three.**
`AlterDailyCost` Ⓟ	Parameters: `Amount: Single` Returns: – Called From: – Calls: –	Accepts a value as a parameter and adds this value to the `DailyCosts` attribute **NB This subroutine is not called from anywhere.**
`CalculateDaily ProfitLoss` Ⓕ	Parameters: `AvgCostPerMeal: Single, AvgPricePerMeal: Single` Returns: `Single` Called From: `Company.ProcessDayEnd` Calls: –	The calculation of daily profit or loss entails calculating the profit/loss for a single meal, multiplying by the number of meals, then subtracting the outlet's daily costs
`GetCapacity` Ⓕ	Parameters: – Returns: `Capacity: Integer` Called From: – Calls: –	Returns the value of `Capacity`
`GetDetails` Ⓕ	Parameters: – Returns: `Details: String` Called From: `Company.GetDetails` Calls: –	Returns a string containing an intelligible representation of each of the attributes – a toString subroutine in all but name

Subroutine	Data	Description
`GetX` Ⓕ	Parameters: – Returns: `XCoord: Integer` Called From: `Company.AddVisitToNearestOutlet` `Company.GetDistanceBetweenTwoOutlets` Calls: –	Returns the value of `XCoord`
`GetY` Ⓕ	Parameters: – Returns: `YCoord: Integer` Called From: `Company.AddVisitToNearestOutlet` `Company.GetDistanceBetweenTwoOutlets` Calls: –	Returns the value of `YCoord`
`IncrementVisits` Ⓟ	Parameters: – Returns: – Called From: `Company.AddVisitToNearestOutlet` Calls: –	Adds 1 to the `VisitsToday` attribute
`NewDay` Ⓟ	Parameters: – Returns: – Called From: `Company.NewDay` `Outlet` constructor Calls: –	Sets the `VisitsToday` attribute to zero

'Company' Class

Subroutine	Data	Description

`AddVisitTo NearestOutlet` Ⓟ	Parameters: `X: Integer, Y: Integer`
	Returns: –
	Called From: `Simulation.ProcessDayEnd`
	Calls: `Outlet.GetX`
	`Outlet.GetY`
	`Outlet.IncrementVisits`
		1. The parameters, `X` and `Y`, are the coordinates of a household whose occupant(s) will eat out on the current day
		2. Initialise local variable `NearestOutlet` to zero and declare two floats
		3. Initialise `NearestOutlet` to first outlet in the list (see step 5 below)
		4. Loops through each outlet in the `Outlets` list
		5. Calculates the distance from the household to the outlet. This is only along a straight line if the outlet and the household are in the same row or column of the settlement; otherwise, there will be a combination of either up or down with either left or right, along with a 90-degree turn.
		6. If the current outlet being examined is closer than the closest found so far, store the index of the current outlet in `NearestOutlet`
		7. After the loop, access the outlet indexed by `NearestOutlet` and call its `IncrementVisits` subroutine

`AlterAvgCost PerMeal` Ⓟ	Parameters: `Change: Single` Returns: – Called From: `Simulation.ProcessCostChangeEvent` Calls: –	Accepts a value as a parameter and adds this value to the `AvgCostPerMeal` attribute
`AlterDailyCosts` Ⓟ	Parameters: `Change: Single` Returns: – Called From: `Simulation.ProcessCostChangeEvent` Calls: –	Accepts a value as a parameter and adds this value to the `DailyCosts` attribute
`AlterReputation` Ⓟ	Parameters: `Change: Single` Returns: – Called From: `Simulation.ProcessReputationChangeEvent` Calls: –	Accepts a value as a parameter and adds this value to the `ReputationScore` attribute

Subroutine	Data	Description
`AlterFuelCost PerUnit` Ⓟ	Parameters: `Change: Single` Returns: – Called From: `Simulation.ProcessCostOfFuelChangeEvent` Calls: –	Accepts a value as a parameter and adds this value to the `FuelCostPerUnit` attribute

`Calculate DeliveryCost` Ⓕ	Parameters: –
	Returns: `TotalCost: Single`
	Called From: `Company.GetDetails`
	`Company.ProcessDayEnd`
	Calls: `Company.GetListOfOutlets`
	`Company.GetDistanceBetweenTwoOutlets`
		1. Calls `GetListOfOutlets` to receive an integer ArrayList, with one integer for each outlet, beginning with zero
		2. Declares an integer, `TotalDistance`, initialised to zero
		3. Loops once per outlet minus 1 (this subroutine is actually not called if a company has only one outlet)
		4. Calculates the distance between the current outlet (0 on the first iteration, 1 on the second, 2 on the third, etc.) and the outlet indexed one value higher (distance between outlets 0 and 1, distance between outlets 1 and 2, distance between outlets 2 and 3, etc.)
		5. For each run through the loop, add this distance to `TotalDistance`
		6. Return the product of `TotalDistance` and the `FuelCostPerUnit` attribute

`CloseOutlet` Ⓕ	Parameters: `ID: Integer` Returns: `CloseCompany: Boolean` Called From: `Simulation.ModifyCompany` Calls: –	1. The outlet, identified by an `ID` integer passed as a parameter, is removed from the `Outlets` list 2. If the list is now empty, return true; otherwise, return false
`ExpandOutlet` Ⓟ	Parameters: `ID: Integer` Returns: – Called From: `Simulation.ModifyCompany` Calls: `Outlet.AlterCapacity`	1. The parameter is the index of the outlet to be expanded 2. User is prompted for how much they want to expand capacity 3. The subsequent input is passed as a parameter to the outlet's `AlterCapacity` subroutine 4. If the expansion took place in full (i.e. did not attempt to take the capacity of the outlet beyond its maximum capacity), the message 'capacity adjusted' is output 5. Otherwise, a message stating that the outlet is now at maximum capacity is output

Subroutine	Data	Description
GetReputationScore Ⓕ	Parameters: – Returns: ReputationScore: Single Called From: Simulation.ProcessDayEnd Calls: –	Returns the value of ReputationScore
GetDetails Ⓕ	Parameters: – Returns: Details: String Called From: Simulation.DisplayCompanies Calls: Company.CalculateDeliveryCost Outlet.GetDetails	1. Declares an empty string 2. Appends all attributes to this string, including the contents of the Outlets list, which are addressed in turn via a for loop 3. Returns the string – this is essentially a toString subroutine
GetDistanceBetween TwoOutlets Ⓕ	Parameters: Outlet1: Integer, Outlet2: Integer Returns: Single Called From: Company.CalculateDeliveryCost Calls: –	Returns the distance between two outlets. Each outlet has a grid position, and the distance between outlets is the sum of the horizontal difference (between the column in which each one exists) and the vertical distance (between the row in which each one exists), i.e. not a straight line unless they share a row or a column.
GetListOfOutlets Ⓕ	Parameters: – Returns: Temp: ArrayList Called From: Company.CalculateDeliveryCost Calls: –	1. A new ArrayList object is created 2. A loop iterates through each Outlet in the Outlets list 3. The zero-based index of each outlet is added to the ArrayList 4. The ArrayList is returned
GetName Ⓕ	Parameters: – Returns: Name: String Called From: Simulation.DisplayCompaniesAtDayEnd Simulation.ProcessCostOfFuelChangeEvent Simulation.ProcessReputationChangeEvent Simulation.ProcessCostChangeEvent Simulation.GetIndexOfCompany Calls: –	Returns the value of Name
GetNumberOfOutlets Ⓕ	Parameters: – Returns: Integer Called From: Simulation.ModifyCompany Calls: –	Returns the number of outlets in the Outlets list

Subroutine	Data	Description
`NewDay` Ⓟ	Parameters: – Returns: – Called From: `Simulation.ProcessDayEnd` Calls: `Outlet.NewDay`	1. Loops through each outlet in the `Outlets` list 2. Calls the `NewDay` subroutine for each outlet, resetting the number of visits for each to zero
`OpenOutlet` Ⓟ	Parameters: `X: Integer, Y: Integer` Returns: – Called From: `Simulation` constructor `Simulation.ModifyCompany` `Company` constructor Calls: `Outlet` constructor	1. Coordinates are passed as parameters 2. Depending on the type of company (fast-food, family, named chef), the company's balance is updated by subtracting the start-up costs 3. A new `Outlet` object is constructed using the coordinates which were passed in as parameters as well as the capacity (which is an integer variable, set according to company category) 4. The new outlet is added to the `Outlets` list
`ProcessDayEnd` Ⓕ	Parameters: – Returns: `Details: String` Called From: `Simulation.DisplayCompaniesAtDayEnd` Calls: `Company.CalculateDeliveryCost` `Outlet.CalculateDailyProfitLoss`	1. Declares an empty string, `Details` 2. Declares `Single` variable to track profit/loss overall and a further variable to track profit/loss for each outlet in turn, as well as one to track delivery costs 3. If there is only one outlet, `DeliveryCosts` is set to the `BaseCostOfDelivery` attribute 4. If there is more than one outlet, `DeliveryCosts` is set to the `BaseCostOfDelivery` attribute plus a call to `CalculateDeliveryCost` 5. `DeliveryCosts` is then appended to `details` 6. Loops through each outlet in the `outlets` list, calling `CalculateDailyProfitLoss` for each outlet 7. The return from this call is appended to the `Details` string and added to the total profit/loss variable 8. Once the loop is over, the previous balance is appended to the string, and the new balance is calculated 9. The `Details` string is returned

'Simulation' Class

Subroutine	Data	Description
`AddCompany` Ⓟ	Parameters: – Returns: – Called From: `Simulation.Run` Calls: `Company.GetReputationScore`	1. Local variables declared to store details of the new company 2. User is prompted for name and starting balance for company 3. User is repeatedly prompted to enter 1, 2 or 3 (fast-food, family restaurant, named chef respectively) until either 1, 2 or 3 is entered 4. Depending on user input, the local variable `TypeOfCompany` is set to either 'fast food', 'family' or 'named chef' 5. A call to `GetRandomLocation` sets the new `X` and `Y` coordinates 6. New `Company` object is constructed and added to the `Companies` list
`DisplayCompanies AtDayEnd` Ⓟ	Parameters: – Returns: – Called From: `Simulation.ProcessDayEnd` Calls: `Company.GetName` `Company.ProcessDayEnd`	1. Outputs 'Companies:' to the console 2. Loops through each company in the `companies` list 3. Outputs the company's name 4. Outputs the return from a call to the `ProcessDayEnd` subroutine

`DisplayEvents AtDayEnd` Ⓟ	Parameters: –
	Returns: –
	Called From: `Simulation.ProcessDayEnd`
	Calls: `Simulation.ProcessAddHouseholdEvent`
	`Simulation.ProcessCostOfFuelChangeEvent`
	`Simulation.ProcessReputationChangeEvent`
	`Simulation.ProcessCostChangeEvent`
		1. Writes 'Events:' to the console
		2. Generates a random float, with 25% chance of entering a selection structure
		3. If the selection structure is entered, another random float is generated, giving a 25% chance of a call to `ProcessAddHouseholdEvent`
		4. Another random float is generated, still within the selection structure described in step 2; this one causes a 50% chance of a call to `ProcessCostOfFuelChangeEvent`
		5. Still within the selection structure from step 2, another random float is generated, causing a 50% chance of a call to `ProcessReputationChangeEvent`
		6. Still within the selection structure from step 2, a final random float is generated, causing a 50% chance of a call to `ProcessCostChangeEvent`
		7. If the original random float (step 2) did not cause the initial selection structure to be entered, 'No events' is output to the console
		NB The events in steps 3 to 6 are independent of one another, meaning either they might all happen, none of them will happen, or any combination of events will happen. If none of the events happens, but the initial selection structure from step 2 is entered, 'No events' will not be displayed.

Subroutine	Data	Description
`DisplayCompanies` Ⓟ	Parameters: – Returns: – Called From: `Simulation.Run` Calls: `Company.GetDetails`	1. Text 'details of all companies:' is output to the console 2. Loop initiated to run once per object in the `companies` ArrayList 3. Call to `GetDetails` for each company is made
`DisplayMenu` Ⓟ	Parameters: – Returns: – Called From: `Simulation.Run` Calls: –	Displays menu items, one on each line, and prompts the user for a choice; this subroutine does not validate input or even store the user's response; user input is handled in run **NB The menu has no option 5**
`GetIndexOfCompany` Ⓕ	Parameters: `CompanyName: String` Returns: `Index: Integer` Called From: `Simulation.Run` Calls: `Company.GetName`	1. Integer variable declared and set to -1 2. Loops through all companies in turn 3. If a company name matches the passed parameter, return the zero-based index of that company 4. If the loop terminates without a match, return -1

`ModifyCompany` Ⓟ	Parameters: `Index: Integer`
	Returns: –
	Called From: `Simulation.Run`
	Calls: `Company.GetNumberOfOutlets`
	`Company.CloseOutlet`
	`Company.ExpandOutlet`
	`Settlement.GetXSize`
	`Settlement.GetYSize`
	`Company.OpenOutlet`
		1. Submenu is displayed, with three options, and user input is accepted
		2. If input is 2 (close outlet) or 3 (expand outlet), the user is prompted for the ID of the outlet; if input is 1 (open outlet), jump to step 6
		3. If 2 (close outlet) was entered, a call to `CloseOutlet` is made, and if that was the last outlet for that company, the company is removed
		4. If 3 (expand outlet) was entered, a call to `ExpandOutlet` is made
		5. If an out-of-range ID is entered in step 3, display error message
		6. If input is 1 (open new outlet), prompt user for `X` and `Y` coordinates
		7. If the coordinates are within the grid bounds, call `OpenOutlet`
		8. If not, display an error message
		NB No error is displayed if an entry other than 1, 2 or 3 is made.

Subroutine	Data	Description
`ProcessReputationChangeEvent` Ⓟ	Parameters: – Returns: – Called From: `Simulation.DisplayEventsAtDayEnd` Calls: `Company.GetName` `Company.AlterReputation`	1. Generates a random number, always to a single decimal place, between 0.1 and 0.9 2. Generates a random integer of either 1 or 0 called `UpOrDown` 3. Generates a random company index 4. If `UpOrDown` is zero, the randomly chosen company's reputation score goes up by the decimal amount; otherwise, it goes down by that amount 5. Step 4 is implemented by outputting the result and by calling the company's `AlterReputation` subroutine
`ProcessAddHouseholdEvent` Ⓟ	Parameters: – Returns: – Called From: `Simulation.DisplayEventsAtDayEnd` Calls: `Settlement.AddHousehold`	1. Generates a random integer between 1 and 4 2. Loops that many times, calling the `S`ettlement class's `AddHousehold` subroutine, effectively creating that many new `Household` objects 3. Output to console the number of new households created
`ProcessCostOfFuelChangeEvent` Ⓟ	Parameters: – Returns: – Called From: `Simulation.DisplayEventsAtDayEnd` Calls: `Company.GetName` `Company.AlterFuelCostPerUnit`	1. Generates a random number, always to a single decimal place, between 0.1 and 0.9 2. Generates a random integer of either 1 or 0 called `UpOrDown` 3. Generates a random company index 4. If `UpOrDown` is zero, the randomly chosen company's fuel cost per unit goes up by the decimal amount; otherwise, it goes down by that amount 5. Step 4 is implemented by outputting the result and by calling the company's `AlterFuelCostPerUnit` subroutine

`ProcessCostChangeEvent` Ⓟ	Parameters: –
	Returns: –
	Called From: `Simulation.DisplayEventsAtDayEnd`
	Calls: `Company.GetName`
	`Company.AlterDailyCosts`
	`Company.AlterAvgCostPerMeal`
		1. Integer variables `CostToChange` and `UpOrDown` are randomly initialised to either zero or one
		2. A further integer, `CompanyNo`, is initialised to a randomly selected index of a company
		3. If `CostToChange` was initialised to zero:
		a. Generate a random one-decimal-place number between 0.1 and 1.9
		b. If `UpOrDown` is zero, daily costs go up by the decimal amount
		c. Otherwise, they go down by the decimal amount
		4. Otherwise, if `CostToChange` was initialised to one:
		a. Generate a random one-decimal-place number between 0.1 and 0.9
		b. If `UpOrDown` is zero, the average cost per meal goes up by the decimal amount
		c. Otherwise, it does down by the decimal amount

Subroutine	Data	Description

`ProcessDayEnd`
Ⓟ
	Parameters: –
	Returns: –
	Called From: `Simulation.Run`
	Calls: `Company.NewDay`
	`Company.GetReputationScore`
	`Settlement.GetNumberOfHouseholds`
	`Settlement.FindOutIfHouseholdEatsOut`
	`Company.AddVisitToNearestOutlet`
	`Simulation.DisplayCompaniesAtDayEnd`
	`Simulation.DisplayEventsAtDayEnd`
		1. Loop through each company in the `Companies` list
		2. Call the `NewDay` subroutine for each company, resetting the number of visits for each company to zero
		3. Get each company's reputation score in turn, adding it to a `TotalReputation` local variable
		4. For each company, append the cumulative reputation total to an ArrayList
		5. Loop through each household in the simulation
		6. For each household, if an occupant will now visit a restaurant (determined using a random number in `FindOutIfHouseholdEatsOut`), set a local variable to a random integer between zero and the cumulative reputation score
		7. Loop through the cumulative reputation ArrayList, incrementing a zero-initialised variable `Current` with each iteration
		8. If the randomly generated number (step 6) is less than the value in the ArrayList at index `Current`, a visit is added to the company indexed by `Current` in the `Companies` list, with the household's coordinates passed to `AddVisitToNearestOutlet`

`Run`
Ⓟ
	Parameters: –
	Returns: –
	Called From: `Module1.Main`
	Calls: `Simulation.DisplayMenu`
	`Settlement.DisplayHouseholds`
	`Simulation.DisplayCompanies`
	`Simulation.DetIndexOfCompany`
	`Simulation.ModifyCompany`
	`Simulation.AddCompany`
	`Simulation.ProcessDayEnd`
		1. Initiates a loop that will repeat until the user enters 'Q' at the prompt
		2. If the user enters '1', call `DisplayHouseholds` (which is in the Settlement class)
		3. If the user enters '2', call `DisplayCompanies`
		4. If the user enters '3', prompt the user for a company name, which is passed to `DetIndexOfCompany`. The call to this subroutine will return either the integer index of the company, if it exists, or -1, if it doesn't. If -1 is returned, the user is prompted for a company name until -1 is not returned.
		5. If the user enters '4', call `AddCompany`
		6. If the user enters '6', call `ProcessDayEnd`
		7. If the user enters 'Q', an exit message is displayed and the program ends
		NB There is no option 5 on the menu.

Program Classes

The classes defined in the program, and their attributes, are described below.

Class	Description
`Household`	An individual household within a settlement; each household has its own unique location, and each household has a probability of a person eating out each day
`Settlement`	A 1,000 by 1,000 grid, with some spaces within the grid (initially 250) occupied by households. Households are stored within a list, meaning the empty 'cells' of the grid are not themselves represented.
`LargeSettlement`	Inherits from settlement, but the grid can be larger than 1,000 by 1,000, and the starting number of households can be larger than 250
`Outlet`	An individual restaurant, belonging to a company. Although each outlet is a distinct object, all outlets of a company are of a particular category (i.e. all fast-food, all family or all named chef – never a combination).
`Company`	Each company consists of one or more outlets, each of which is stored in a list. A company can close an outlet, expand an outlet or open a new outlet.
`Simulation`	'Manages' each day, processing events which are generated via random numbers. These events include the construction of new households within a settlement as well as changes to a company in cost or reputation.
`Module1`	Contains the `Main` subroutine to run the program, which entails creating a new `Simulation` object

'Household' Attributes

Attribute	Type	Default Value	Description
`ChanceEatOutPerDay`	Single	Random, between 0 and 1	Represents the probability that a member of the household will eat out on a given day
`XCoord`	Integer	Passed as a parameter to constructor	The X coordinate of the household's location within the settlement
`YCoord`	Integer	Passed as a parameter to constructor	The Y coordinate of the household's location within the settlement
`NextID`	Integer	Static/shared; initialised to 1	Unique identifier for each household, with the first household numbered '1', the second '2', and so forth
`ID`	Integer	Current value of `NextID`	Value in `NextID` is stored in `ID` when a `Household` object is constructed

'Settlement' Attributes

The `LargeSettlement` class inherits from the Settlement class, but does not include any additional subroutines or attributes. Instead, it includes three additional parameters to the constructor, allowing any calling class to vary the initialisation values of `XSize`, `YSize` and `StartNoOfHouseholds`.

Attribute	Type	Default Value	Description
`StartNoOf Households`	Integer	250	The number of households in the settlement at the beginning of the simulation
`XSize`	Integer	1,000	One dimension of the size of the settlement 'grid'. As it is the 'x' dimension, it might be helpful to visualise this as the settlement's width, but the program does not store the grid in any form, such as a 2D array.
`YSize`	Integer	1,000	The other dimension for the size of the settlement
`StartNoOf Households`	Integer	250	The number of households in the settlement at the beginning of the simulation

'Outlet' Attributes

Attribute	Type	Default Value	Description
`VisitsToday`	Integer	0	The number of times the outlet has been visited on the current day, which is incremented with each visit
`XCoord`	Integer	Passed as a parameter to constructor	The X coordinate of the outlet's location within the settlement
`YCoord`	Integer	Passed as a parameter to constructor	The Y coordinate of the outlet's location within the settlement
`Capacity`	Integer	See →	The maximum number of visits that an outlet can receive, initialised to 60% of `MaxCapacityBase`, which is a parameter passed to the Outlet class's constructor
`MaxCapacity`	Integer	See →	The highest value that capacity can be increased to by way of calls to the class's `AlterCapacity` subroutine. This is initialised to the `MaxCapacityBase` parameter, with a random integer between 0 and 50 added to it, then another random number in the same range subtracted from it.
`DailyCosts`	Single	See →	The cost of running the outlet, irrespective of customer footfall. This is initialised to a sum of the following: • £0.50 per seat within capacity • £0.20 per seat or potential seat up to `MaxCapacityBase` (in addition to £0.50 that has already been paid) • £100 one-off

'Company' Attributes

Attribute	Type	Default Value	Description
`Name`	String	Passed as a parameter to constructor	The name of the company
`Category`	String	Passed as a parameter to constructor	The type of company – one of 'fast food', 'family' or 'named chef'
`Balance`	Single	Passed as a parameter to constructor	The amount of money the company has
`ReputationScore`	Single	See →	A measure of a company's reputation. It is initialised to 100, then altered depending on the category of the company, with named chef companies likely to have a higher score than family companies, which, in turn, are likely to have a higher score than fast-food companies. This is not guaranteed, however, as random numbers are used in these calculations.
`AvgCostPerMeal`	Single	See →	The average cost of a meal (as bought by the company), set to 5, 12 and 20 for fast-food, family and named chef companies respectively
`AvgPricePerMeal`	Single	See →	The average price of a meal (as sold to a customer), set to 10, 14 and 40 for fast-food, family and named chef companies respectively
`DailyCosts`	Single	100	Part of a company's expenses, regardless of the number of outlets
`FamilyOutletCost`	Single	1,000	The cost of a family company opening a new outlet
`FastFoodOutletCost`	Single	2,000	The cost of a fast-food company opening a new outlet
`NamedChefOutletCost`	Single	15,000	The cost of a named chef company opening a new outlet
`FuelCostPerUnit`	Single	Passed as a parameter to constructor	Used in the calculation of delivery to each outlet
`BaseCostOfDelivery`	Single	Passed as a parameter to constructor	Used in the calculation of delivery to a company, regardless of the number of outlets
`Outlets`	ArrayList of `Outlet` objects	Empty	Collection of all the company's outlets
`FamilyFoodOutletCapacity`	Integer	150	The base maximum capacity for a family outlet
`FastFoodOutletCapacity`	Integer	200	The base maximum capacity for a fast-food outlet
`FamedChefOutletCapacity`	Integer	50	The base maximum capacity for a named chef outlet
`Name`	String	Passed as a parameter to constructor	The name of the company

Simulation Attributes

Attribute	Type	Default Value	Description
`SimulationSettlement`	Settlement	See →	The single settlement for the game, which will be initialised to either a new instance of `Settlement` or a new instance of `LargeSettlement`, depending on user input
`NoOfCompanies`	Integer	See →	The number of companies in the simulation, which is initialised to either 3 or a user-input value, depending on a menu selection
`FuelCostPerUnit`	Single	0.0098	Passed to the `Company` constructor to subsequently be used in calculations of delivery for each of a company's outlets
`BaseCostForDelivery`	Single	100	Passed to the `Company` constructor to subsequently be used in calculations of delivery costs for a company
`Companies`	ArrayList of `Company` objects	Empty	Collection of all of the simulation's companies

The class containing the *`Main`* subroutine has no attributes

Food Magnate Simulation

Class Diagram Tasks

Complete each of the following unfinished UML class diagrams. Each one is missing a combination of attributes, subroutines, access modifiers, parameters and return types.

Household
__ ChanceEatOutPerDay: float # XCoord: _____ # _____: int # ID: _____ # NextID: int
+ Household(int, int) + GetDetails(): _____ __ GetChanceEatOut(): _____ + GetX(): int __ _____(): _____

Settlement
StartNoOfHouseholds: _____ # XSize: _____ # YSize: _____ __ Households: Household[0..*]
+ Settlement() + GetNumberOfHouseholds(): _____ + _____(): int + _____(): int __ GetRandomLocation(int, int): _____ + CreateHouseholds(): _____ + AddHousehold(): _____ __ DisplayHouseholds(): void + FindOutIfHouseholdEatsOut(int, int, _____): _____

Complete the UML class diagram for the `Company` class.
The first attribute and the first subroutine are provided for you.

Company
Name: String
+ Company(String, String, float, int, int, float, float) + GetName(): String

Food Magnate Simulation

MARKS **/50**

Programming Theory Questions

These questions refer to the preliminary material and require you to load the skeleton program, but <u>do not</u> require any additional programming.

1. State the name of an identifier for:

   a) An attribute in the `Household` class that would **not** be instantiated for each new object [1]

   .....

   b) A subroutine in the `Settlement` class that accepts parameters by reference [1]

   .....

   c) A subclass [1]

   .....

   d) A local variable that is used to return a Boolean [1]

   .....

   e) Two subroutines from the `Company` class that **cannot** be called from outside the `Company` class [2]

   .....

   f) A library string function called from the `GetIndexOfCompany` subroutine in the `Simulation` class [1]

   .....

   g) A collection attribute in the `Company` class [1]

   .....

   h) An instance of `Settlement` [1]

   .....

2. Showing and explaining your working, give the probability of a call to `ProcessCostOfFuelChangeEvent` being made from the `DisplayEventsAtDayEnd` subroutine in the `Simulation` class. [3]

   .....

   .....

   .....

   .....

   .....

3. Explain how validation might be added to the `OpenOutlet` subroutine of the `Company` class to prevent a new outlet being created beyond the bounds of the settlement. Remember, you do **not** need to write any code. [3]

.....

.....

.....

.....

.....

4. Each `Household` object is stored within an ArrayList called `Households`. Describe how a *Dictionary* could have been used instead to store `Household` objects. [3]

.....

.....

.....

.....

.....

5. Describe in full how the `GetDistanceBetweenTwoOutlets` subroutine of the `Company` class calculates the distance between two outlets. [4]

.....

.....

.....

.....

.....

.....

6. Explain the role of the object of type `Random` in the `Household` class. [2]

.....

.....

.....

7. Explain the role of the variable `UpOrDown` in the `ProcessCostOfFuelChangeEvent` subroutine of the `Simulation` class. [3]

.....

.....

.....

.....

.....

8. In the `Simulation` constructor, the integer literals 100000, 200 and 203 are passed to the `Company` constructor when creating the 'AQA Burgers' company. State the role of each of these integer literals. [3]

.....

.....

.....

.....

.....

9. Describe in full the operation of the `GetIndexOfCompany` subroutine in the `Simulation` class. [5]

.....

.....

.....

.....

.....

.....

.....

.....

10. Describe the circumstances under which the `ModifyCompany` subroutine of the `Simulation` class would output the text 'Invalid coordinates'. [3]

.....

.....

.....

.....

.....

11. Currently, a call to the `LargeSettlement` constructor could not result in a settlement that is smaller than 1,000 by 1,000. This is true even if negative numbers are entered by the user when prompted for additional x and y values.

    Explain how a call to the `LargeSettlement` constructor never results in a smaller settlement size. [3]

    .....

    .....

    .....

    .....

    .....

12. Describe the concept of constructor overloading, and explain how constructor overloading could have been used instead of inheritance for the creation of a new large settlement. [4]

    .....

    .....

    .....

    .....

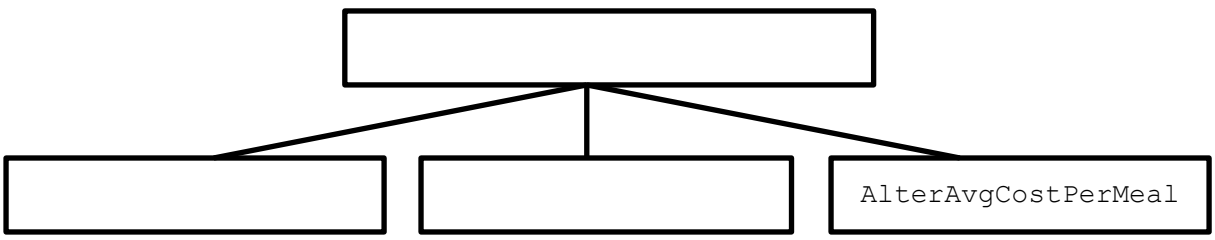
    .....

    .....

    .....

    .....

13. Complete the following hierarchy chart for part of the `Simulation` class of the Skeleton Program. You should **not** include calls to any library subroutines. [3]



14. Describe how the program would respond to a call to the `Company` constructor using a category that is neither 'fast food', 'family' nor 'named chef'. [2]

    .....

    .....

    .....

Food Magnate Simulation

MARKS /50

Programming Theory Questions

These questions refer to the preliminary material and require you to load the skeleton program, but <u>do not</u> require any additional programming.

1. State the name of an identifier for:
   a) An attribute in the `Household` class that would **not** be instantiated for each new object [1]
   b) A subroutine in the `Settlement` class that accepts parameters by reference [1]
   c) A subclass [1]
   d) A local variable that is used to return a Boolean [1]
   e) Two subroutines from the `Company` class that **cannot** be called from outside the `Company` class [2]
   f) A library string function called from the `GetIndexOfCompany` subroutine in the `Simulation` class [1]
   g) A collection attribute in the `Company` class [1]
   h) An instance of `Settlement` [1]

2. Showing and explaining your working, give the probability of a call to `ProcessCostOfFuelChangeEvent` being made from the `DisplayEventsAtDayEnd` subroutine in the `Simulation` class. [3]

3. Explain how validation might be added to the `OpenOutlet` subroutine of the `Company` class to prevent a new outlet being created beyond the bounds of the settlement. You do **not** need to write any code. [3]

4. Each `Household` object is stored within an ArrayList called `Households`. Describe how a *Dictionary* could have been used instead to store `Household` objects. [3]

5. Describe in full how the `GetDistanceBetweenTwoOutlets` subroutine of the `Company` class calculates the distance between two outlets. [4]

6. Explain the role of the object of type `Random` in the `Household` class. [2]

7. Explain the role of the variable `UpOrDown` in the `ProcessCostOfFuelChangeEvent` subroutine of the `Simulation` class. [3]

8. In the `Simulation` constructor, the integer literals 100000, 200 and 203 are passed to the `Company` constructor when creating the 'AQA Burgers' company. State the role of each of these integer literals. [3]

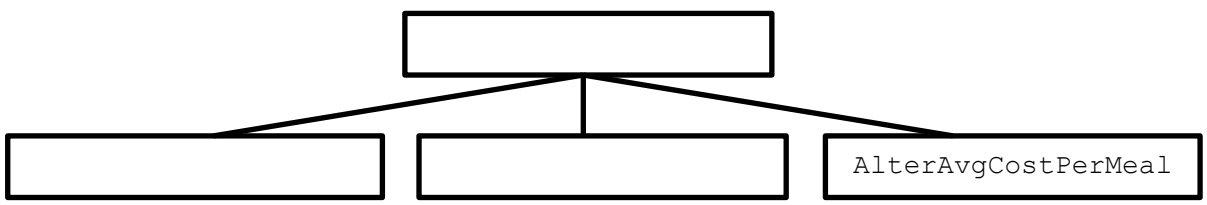
9. Describe in full the operation of the `GetIndexOfCompany` subroutine in the `Simulation` class. [5]

10. Describe the circumstances under which the `ModifyCompany` subroutine of the `Simulation` class would output the text 'Invalid coordinates'. [3]

11. Currently, a call to the `LargeSettlement` constructor could not result in a settlement that is smaller than 1,000 by 1,000. This is true even if negative numbers are entered by the user when prompted for additional x and y values. Explain how a call to the `LargeSettlement` constructor never results in a smaller settlement size. [3]

12. Describe the concept of constructor overloading, and explain how constructor overloading could have been used instead of inheritance for the creation of a new large settlement. [4]

13. Complete the following hierarchy chart for part of the `Simulation` class of the Skeleton Program. You should **not** include calls to any library subroutines. [3]



14. Describe how the program would respond to a call to the `Company` constructor using a category that is neither 'fast food', 'family' nor 'named chef'. [2]

Food Magnate Simulation

Programming Tasks

The following require you to open the skeleton program and make modifications

Task 1 (max. 6 marks)

This question refers to the subroutine `ModifyCompany` within the `Simulation` class.

Currently, the user is prompted to enter a value of 1, 2 or 3, but if nothing is entered by the user, the program responds by outputting a blank line.

Change the subroutine `ModifyCompany` to present the user with an additional choice: "C. Cancel". If the user enters anything other than 1, 2, 3 or an upper-case 'C', the menu should be redisplayed repeatedly until either 1, 2, 3 or C is selected. If 1, 2 or 3 is entered, `ModifyProgram` should behave as normal. If an upper-case 'C' is entered, the program should output 'Operation Cancelled', and `ModifyCompany` should terminate without executing any additional code.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 3 for 'modify company'
- enter 'AQA Burgers' when prompted for a company name
- enter 'X' at the first prompt of the 'modify company' submenu
- enter 'C' at the second prompt of the 'modify company' submenu

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `ModifyCompany`

b. SCREEN CAPTURE(S) showing the required test

Task 2 (max. 9 marks)

This question refers to the subroutine `GetRandomLocation` within the `Settlement` class.

This subroutine generates a random location within the bounds of the settlement that is used to position a new household. Currently, there is no mechanism for ensuring that a new household is not assigned the location of an existing household.

Change the subroutine `GetRandomLocation` to ensure that only unoccupied locations are returned. Prior to returning the location, a check should be made to determine whether the location is already occupied by a household. If it is already occupied, a new location should be generated, repeatedly if necessary.

Test that the changes you have made work:

- modify the Settlement constructor in the following ways:
  - change `XSize = 1000` to `XSize = 3`
  - change `YSize = 1000` to `YSize = 3`
  - change `StartNoOfHouseholds = 250` to `StartNoOfHouseholds = 8`

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 1 for 'display details of households'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `GetRandomLocation`

b. SCREEN CAPTURE(S) showing the required test

Task 3 (max. 7 marks)

This question refers to the subroutine `ExpandOutlet` within the `Company` class, as well as a new subroutine, `ExtendCapacity`, within the `Outlet` class.

Currently, each outlet has a limit, beyond which it cannot be expanded. Any attempt to expand beyond the limit results in the capacity being set at the limit itself.

Create a new subroutine in the `Outlet` class called `ExtendCapacity` which takes a single integer parameter. When this subroutine is called, the value of the attribute `MaxCapacity` should be multiplied by the value of this parameter.

Change the `ExpandOutlet` subroutine so that, in the event of an attempt to increase an outlet beyond its `maxCapacity` value, a random number is generated. Using this number, there should be a 40% chance of calling the new subroutine, `ExtendCapacity`, and passing a value of 2, a 35% chance of calling it and passing a value of 3, and a 25% chance of calling it and passing a value of 4. The message 'Outlet max capacity expanded' should be displayed in these circumstances, with no other message.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 2 for 'display details of companies', ensuring that outlet 4 for 'Paltry Poultry' is visible in your screenshot
- enter 3 for 'modify company'
- enter 'Paltry Poultry' when prompted for a company name
- enter 3 for 'expand outlet'
- enter 4 when prompted for an ID
- enter 1000 when asked for an amount
- enter 2 for 'display details of companies', ensuring that outlet 4 for 'Paltry Poultry' is visible in your screenshot

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `ExpandOutlet`

b. Your PROGRAM SOURCE CODE for the new subroutine `ExtendCapacity`

c. SCREEN CAPTURE(S) showing the required test

Task 4 (max. 8 marks)

This question refers to the subroutine `ProcessDayEnd` within the `Simulation` class, as well as a new subroutine, `ProcessLeavers`, within the `Settlement` class.

Currently, there is no mechanism for households to leave a settlement.

Create a new subroutine in the `Settlement` class called `ProcessLeavers`. This subroutine should accept no parameters and should return an integer value. Each household in the settlement should be subject to a random 2% chance of leaving the settlement. The `ProcessLeavers` subroutine should remove households as applicable from the `Households` data structure, and return the number of households that were removed.

Modify the subroutine `ProcessDayEnd`, so that the final instructions are to call `ProcessLeavers` and output the number of households that left the settlement.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 6 for 'advance to next day'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `ProcessDayEnd`

b. Your PROGRAM SOURCE CODE for the new subroutine `ProcessLeavers`

c. SCREEN CAPTURE(S) showing the required test

Task 5 (max. 7 marks)

This question refers to the subroutine `Run` within the `Simulation` class.

Currently, when modifying a company, the user needs to enter a company name in full, for which they must either remember it or scroll up the console window to see it previously displayed.

Change the subroutine `Run` so that when the user enters option 3 from the main menu (modify company), they are presented with a numbered list of names of companies. The first company to be displayed in the list should be displayed next to a number 1, even though its index in the `Companies` data structure will be 0.

When the user enters the number next to the company name, the program should respond in the same way as it would have done had the company's name been entered.

Entering the company's name should no longer be effective, and the input message should read 'enter the number next to the company you wish to modify'.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 3 for 'modify company'
- enter 3 when asked for a number
- enter 3 for 'expand outlet'
- enter 4 for the ID
- enter 1000 for the amount
- enter 2 for 'display details of companies'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `Run`

b. SCREEN CAPTURE(S) showing the required test

Task 6 (max. 5 marks)

This question refers to an additional constructor for the `Outlet` class.

Currently, when a new outlet is constructed, this takes place by way of X and Y coordinates being passed to the `Outlet` constructor. In this task, you will create an additional constructor that facilitates random placement.

Without making any changes to the existing constructor, create an additional constructor for the `Outlet` class. This constructor should take as parameters three integers – `MaxX`, `MaxY` and `MaxCapacityBase` – and a Boolean, `IsRandom`. The `MaxX` and `MaxY` integers represent the highest values for X and Y that an outlet could possibly have within its settlement.

The constructor should contain code that sets `XCoord` to a random integer value between 0 and `MaxX` inclusive, as well as code that sets `YCoord` to a random integer value between 0 and `MaxY` inclusive. All other attributes of the `Outlet` class should be set identically to those of the existing constructor.

Test that the changes you have made work:

- Modify the call to the `Outlet` class's constructor within the subroutine `OpenOutlet` of the `Company` class; it should read as follows:
  - `Dim NewOutlet As New Outlet(5, 10, Capacity, True)`
- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 2 for 'display details of companies'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the new `Outlet` constructor

b. SCREEN CAPTURE(S) showing the required test

Task 7 (max. 7 marks)

This question refers to the subroutines `DisplayMenu` and `Run` within the `Simulation` class.

Currently, the program allows the user to advance the simulation for multiple days only by repeatedly selecting option 6 from the menu.

Change `DisplayMenu` to include an additional option: '5. Advance'.

Change `Run` so that if option 5 is selected, the user is prompted for the number of days they wish the simulation to advance. This number, which does not require validation, will be the number of times that the subroutine `ProcessDayEnd` is called.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 5 for 'advance'
- enter 3 when asked for a number

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `Run`

b. Your PROGRAM SOURCE CODE for the amended subroutine `DisplayMenu`

c. SCREEN CAPTURE(S) showing the required test

Task 8 (max. 9 marks)

This question refers to the subroutine `AddCompany` within the `Simulation` class.

Functionality will be added to allow a company type to be assigned at random.

Change the `AddCompany` subroutine so that the user is prompted to enter 1, 2, 3 or 4. The existing prompt should be updated to indicate that '4' means a randomly chosen type of company.

If 1, 2 or 3 is entered, the program should continue as before. If '4' is entered, a random number should be used to determine which of the three types of company will be created. Each type of company should be equally likely to be created.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- leave the second prompt blank, to indicate user-defined companies
- enter 1 when asked for a number of companies
- enter the company name 'Random Restaurant'
- enter a starting balance of 50000
- enter 4 to indicate a 'random' new company
- enter 2 for 'display details of companies'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `AddCompany`

b. SCREEN CAPTURE(S) showing the required test

Task 9 (max. 6 marks)

This question refers to the subroutine `ProcessDayEnd` in the `Company` class, as well as a new subroutine, `CloseAllOutlets`, also in the `Company` class.

Currently, a company can continue operating irrespective of how far below zero their balance falls.

Create a new subroutine called `CloseAllOutlets`, in the `Company` class, which iterates through all outlets belonging to the company, and closing them with a call to `CloseOutlet`.

Modify the subroutine `ProcessDayEnd` in the `Company` class, so that immediately before the return statement, the value of the `balance` field is checked. If it is below zero, a call to `CloseAllOutlets` is made, and a message is output to notify the user that all outlets have closed.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- leave the second prompt blank, to indicate user-defined companies
- enter 1 when asked for a number of companies
- enter the company name 'Bankrupt Burgers'
- enter a starting balance of 1000
- enter 2 to indicate a family restaurant
- enter 6 in the main menu, 'advance to next day'
- enter 2 in the main menu, 'display details of companies'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `ModifyCompany`

b. Your PROGRAM SOURCE CODE for the new subroutine `CloseAllOutlets`

c. SCREEN CAPTURE(S) showing the required test

Task 10 (max. 11 marks)

This question refers to a new class, called `FoodTruck`, as well as the subroutine `ProcessDayEnd` in the `Company` class.

Create a new class, called `FoodTruck`, which inherits from the class `Outlet`. As well as the inherited subroutines and attributes, `FoodTruck` should include a new subroutine called `Move`. Movement should take place in a random direction, moving one 'square' north, south, east or west. There is no need to validate the food truck's movement, which is permitted to leave the settlement as a result of its `XCoord` and `YCoord` values being beyond the settlement's bounds.

The constructor for `FoodTruck` should take `XCoord` and `YCoord` integers as parameters, then pass these to the superclass constructor along with a value of 10 for capacity.

Change `ProcessDayEnd` so that the `move` subroutine is called for any outlets that are of type `FoodTruck`.

Test that the changes you have made work:

- Modify the call to the `Outlet` class's constructor within the `OpenOutlet` subroutine of the `Company` class; it should read as follows:

  ```
  Dim NewOutlet As New FoodTruck(X, Y)
  ```

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- leave the second prompt blank, to indicate user-defined companies
- enter 1 when asked for a number of companies
- enter the company name 'Taco Truck'
- enter a starting balance of 30000
- enter 1 to indicate a fast-food restaurant
- enter 2 to display details of companies
- enter 6 to advance to the next day
- enter 2 to display details of companies

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the new class `FoodTruck`

b. Your PROGRAM SOURCE CODE for the amended subroutine `ProcessDayEnd`

c. SCREEN CAPTURE(S) showing the required test, ensuring that the location of the food truck is visible after each entry of '2' in the main menu

Task 11 (max. 15 marks)

This question refers to the subroutine `GetIndexOfCompany` within the `Simulation` class.

Currently, when a company is searched for using this subroutine, the whole company name is required in order to generate a match.

Change `GetIndexOfCompany` so that if the user enters a search term that is contained within the name of one company, the index of that company is returned. If the text is contained within the names of multiple companies, the user should be presented with all matching company names before being asked to type one of them in full in order to select it.

The subroutine should continue to be non-case-sensitive, and a search for a company that finds nothing, or an attempt to select a matching company that doesn't actually match one of the search results, should still return a value of -1.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 3 for 'modify company'
- enter a lower-case 't' for the company name
- type 'Paltry Poultry' when asked to type the name of a company

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `GetIndexOfCompany`

b. SCREEN CAPTURE(S) showing the required test

Task 12 (max. 7 marks)

This question refers to the subroutine `CloseOutlet` in the `Company` class.

Currently, closing an outlet incurs no expense on the part of the company.

Change `CloseOutlet` so that a company's balance decreases for each outlet that is closed. The cost of closing the outlet depends on both the type of the company and the capacity of the outlet being closed. Taking 'capacity' as being the number of seats in an outlet, the costs of closing an outlet are as follows:

Fast-food outlet: 75 per seat

Family outlet: 50 per seat

Named chef outlet: 150 per seat

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 2 for 'display details of companies
- enter 3 for 'modify company'
- enter 'Paltry Poultry'
- enter 2 for 'close outlet'
- enter 4 when prompted for an ID
- enter 2 for 'display details of companies'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `CloseOutlet`

b. SCREEN CAPTURE(S) showing the required test, ensuring all details for Paltry Poultry's outlets are visible after each request of 'display details of companies'

Task 13 (max. 8 marks)

This question refers to the subroutines `DisplayMenu` and `Run` in the `Simulation` class.

Currently, it is possible to add a company to a simulation, but not to remove one.

Change `DisplayMenu` to include an additional option: '5. Remove company'.

Change `Run` so that if option 5 is selected, the user is prompted for the name of the company they wish to remove. After the user has entered the name of the company, that company's index should be obtained via `GetIndexOfCompany`. The program should repeatedly ask them for a company name until either a valid name has been entered, or 'cancel' (any combination of upper case and lower case) has been entered.

If 'cancel' is entered, the user should be returned to the main menu. Otherwise, the company with a name matching the user entry should be removed from the simulation, and the user should be returned to the main menu.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 5 for 'remove company'
- type CANCEL (all in upper case) when prompted for the name of a company
- enter 5 for 'remove company'
- type 'Paltry Poultry' when prompted again for the name of a company
- type 2 for 'display details of companies'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `DisplayMenu`

b. Your PROGRAM SOURCE CODE for the amended subroutine `Run`

c. SCREEN CAPTURE(S) showing the required test

Task 14 (max. 20 marks)

This question refers to the subroutines `DisplayMenu` and `Run` in the `Simulation` class, as well as two new subroutines: `RunToTarget` in the `Simulation` class, and `GetBalance` in the `Company` class.

Currently, the simulation runs day by day, regardless of the effects of any changes.

Create a new subroutine in the `Company` class called `GetBalance`, which should accept no parameters and return the value of the balance attribute.

Create a new subroutine in the `Simulation` class called `RunToTarget`. This subroutine should prompt the user for upper and lower limits, storing them in integer variables called `UpperLimit` and `LowerLimit`. No validation is required for user input.

The simulation should run, via repeated calls to `ProcessDayEnd` in the `Simulation` class, until one company has a balance either equal to or above `UpperLimit` or equal to or below `LowerLimit`, using calls to the new subroutine `GetBalance`. At this point, there should be no additional calls to `ProcessDayEnd`, and the program should output the name of the company, its balance and the number of days that have elapsed since the beginning of the simulation.

In the event that multiple companies reach `UpperLimit` and/or `LowerLimit` at the same time, the program need only display the details of one of the companies.

Change `DisplayMenu` to include an additional option: '5. Run to target'.

Change `Run` so that if the user enters option 5, a call is made to `RunToTarget`.

Test that the changes you have made work:

- run the **Skeleton Program**
- leave the first prompt blank, to indicate a normal-sized settlement
- enter D at the next prompt for default companies
- enter 5 for 'run to target'
- enter 0 when prompted for the lower limit
- enter 100000 (one hundred thousand) when prompted for the upper limit

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended subroutine `DisplayMenu`

b. Your PROGRAM SOURCE CODE for the amended subroutine `Run`

c. Your PROGRAM SOURCE CODE for the new subroutine `RunToTarget`

d. Your PROGRAM SOURCE CODE for the new subroutine `GetBalance`

e. SCREEN CAPTURE(S) showing the required test; only the final 'events' section needs to be included

Task 15 (max. 14 marks)

This question refers to a new class called `CitySettlement`, as well as the constructor of the `Simulation` class.

Currently, a new `Settlement` object is constructed by way of calls to the constructor of either `Settlement` or `LargeSettlement`.

Create a new class called `CitySettlement`, which inherits from `LargeSettlement`. Its constructor should have the same parameters as the `LargeSettlement` constructor, and should call the `LargeSettlement` constructor, passing on its own parameters.

`CitySettlement` should include a subroutine called `AddHousehold`, which should override the `AddHousehold` subroutine of the `Settlement` class. This new subroutine should select a random location within the city, then select a random integer between 2 and 20 inclusive. This number of new households should then be created and added to the settlement at the single location generated.

This is to model families living in apartment blocks. In this class, creating a settlement with 100 households should have the effect of creating 100 such apartment blocks, with each block containing between 2 and 20 households. This means that a 250-household `CitySettlement` will contain more than 250 households, as it should contain 250 blocks of households.

Change the constructor of the `Simulation` class to present the user with a choice of 'a large settlement', 'a normal-sized settlement' or 'a city settlement' (the order is unimportant). If a city settlement is selected, the user should be prompted for additional x-size, y-size and households values. These values should then be sent as parameters to a call to the new `CitySettlement` constructor.

Test that the changes you have made work:

- run the **Skeleton Program**
- at the prompt, indicate a city settlement
- enter a value of zero for each of the additional prompts
- enter D for default companies
- enter 1 for 'Display details of households'

Evidence that you need to provide:

a. Your PROGRAM SOURCE CODE for the amended `Simulation` constructor

b. Your PROGRAM SOURCE CODE for the new class `CitySettlement`

c. Your PROGRAM SOURCE CODE for the `AddHousehold` subroutine of the `Settlement` class

d. SCREEN CAPTURE(S) showing the required test, ensuring at least the last five households are visible

Food Magnate Simulation

Additional Programming Tasks (Extension)

These challenges are presented without solutions, and offer to further explore and understand the skeleton program.

1. Add validation any user input to ensure that something is entered and a message displayed if nothing is entered

2. Add validation according to type, such that inputs required to be numeric are, indeed, numeric

3. Create an additional category of restaurant, with a new company of that type created as part of the default companies

4. Generate random likelihood of an outlet being closed as a day progresses

5. Create a random instance of a company or an outlet running a promotion, during which time its expenses go up but its reputation score also rises

6. Close an outlet that runs at a loss for five consecutive days, adding a notification that this has happened to the events

7. Display details of the restaurant which, during a day, was either the most profitable, the most visited or the one with the highest reputation rating

8. Prevent a new outlet being opened within a certain distance of another outlet, or another outlet of the same type

9. Redesign `Company` to be an abstract class, with categories of company each being a subclass

10. Generate a random budget and store it as an attribute within each `Household` object; a household that eats out will only eat out with a company whose prices are within their budget

11. Incorporate weather into the simulation; it can rain at random, in which case the probabilities of eating out are all halved, and the presence of rain is indicated within the events

12. Generate random events, such as power cuts, fuel shortages and festivals, each of which can have an impact on the probability of each house eating out

13. Incorporate a text file from which initial companies and outlets are created (where they differ from the default companies), rather than having the user manually enter them each time the simulation is run

14. Add a feature in which meals can be delivered, rather than customers visiting outlets; households who choose not to eat out might, according to random chance, order food to be delivered, although this costs the company extra in fuel according to the distance of delivery

15. The named chef outlets generally make a profit, while other categories make a loss; write a subroutine to determine, for each type of company, how much an outlet should have charged per meal during the previous day in order to have operated at a profit

Food Magnate Simulation

Class Diagram Tasks – Solutions

Shaded areas indicate the correct answers. Any differences in with case and spacing are acceptable, but any misspellings should not be awarded a mark. Accept 'integer' in lieu of 'int'.

Household	**(10 marks)**
# ChanceEatOutPerDay: float # XCoord: **int** # **YCoord**: int # ID: **int** # NextID: int	
+ Household(int, int) + GetDetails(): **String** **+** GetChanceEatOut(): **float** + GetX(): int **+** **GetY**(): **int**	

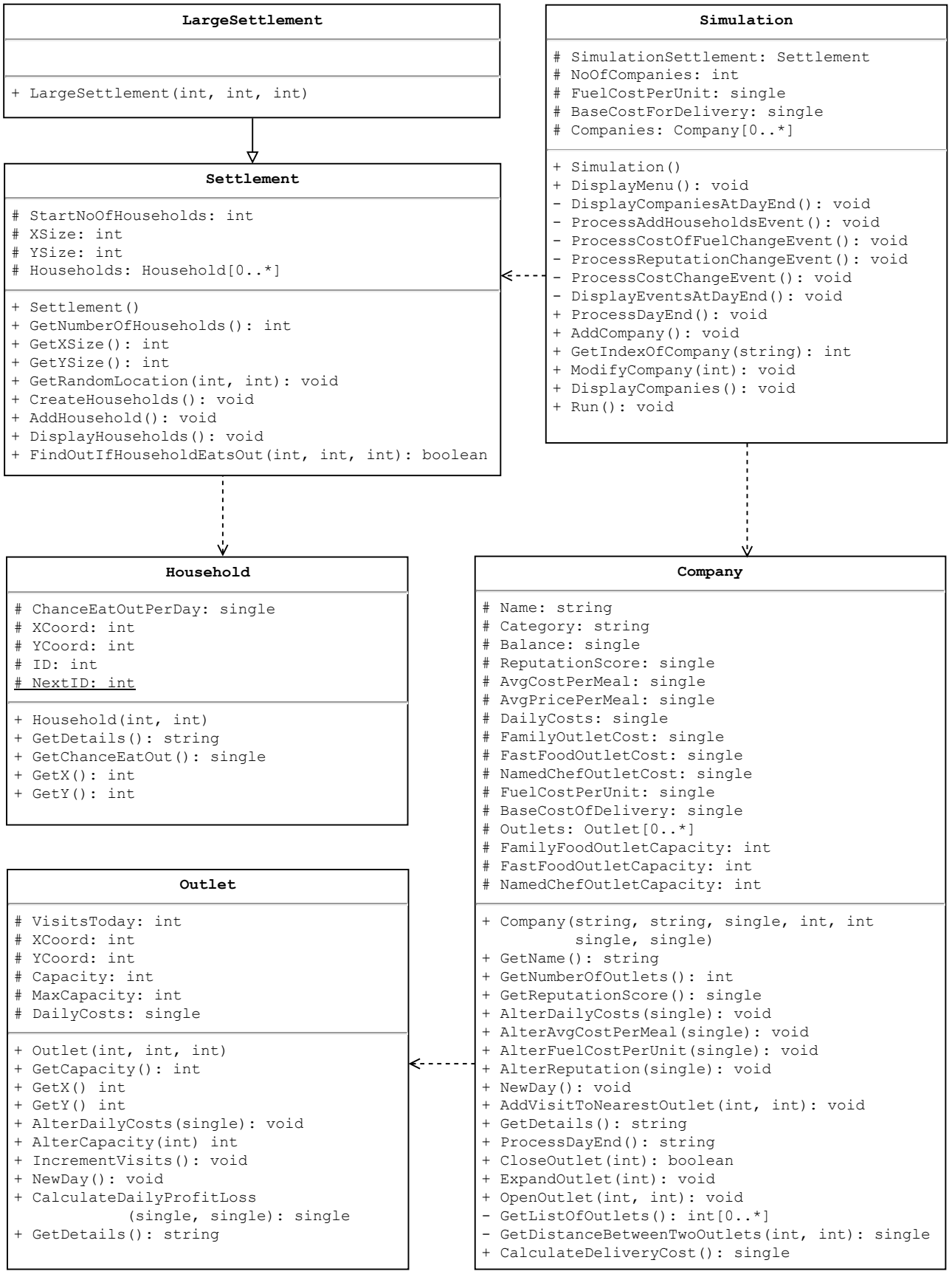
Settlement	**(14 marks)**
# StartNoOfHouseholds: **int** # XSize: **int** # YSize: **int** **#** Households: Household[0..*]	
+ Settlement() + GetNumberOfHouseholds(): **int** + **GetXSize**(): int + **GetYSize**(): int **fine if getXSize and getYSize are the other way around* **+** GetRandomLocation(int, int): **void** + CreateHouseholds(): **void** + AddHousehold(): **void** **+** DisplayHouseholds(): void + FindOutIfHouseholdEatsOut(int, int, **int**): **boolean**	

For the `Company` class, the first attribute and the first two subroutines are included in the question and should not be marked. All other lines are worth two marks each, one for the content before the colon and one for the content after the colon. You can accept 'integer' in lieu of 'int'. The alternating grey rows are only here to make this easier to read. Penalise for misspellings and any extra content within a line; do not penalise for capitalisation or spacing errors. Your attention is drawn to the private access level of one attribute and two subroutines. Peer assessment is recommended.

Company **(62 marks)**

```
# Name: String
# Category: String
# Balance: float
# ReputationScore: float
# AvgCostPerMeal: float
# AvgPricePerMeal: float
# DailyCosts: float
# FamilyOutletCost: float
# FastFoodOutletCost: float
# NamedChefOutletCost: float
# FuelCostPerUnit: float
# BaseCostOfDelivery: float
# Outlets: Outlet[0..*]
# FamilyFoodOutletCapacity: int
# FastFoodOutletCapacity: int
# NamedChefOutletCapacity: int
```

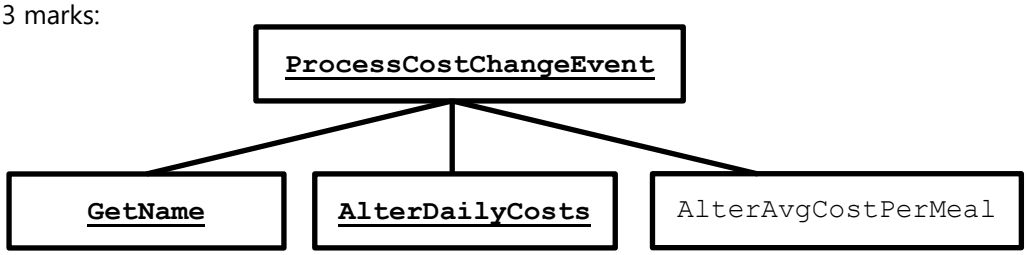
```
+ Company(String, String, float, int, int, float, float)
+ GetName(): String
+ GetNumberOfOutlets(): int
+ GetReputationScore(): float
+ AlterDailyCosts(float): void
+ AlterAvgCostPerMeal(float): void
+ AlterFuelCostPerUnit(float): void
+ AlterReputation(float): void
+ NewDay(): void
+ AddVisitToNearestOutlet(int, int): void
+ GetDetails(): String
+ ProcessDayEnd(): String
+ CloseOutlet(int): boolean
+ ExpandOutlet(int): void
+ OpenOutlet(int, int): void
- GetListOfOutlets(): int[0..*]
- GetDistanceBetweenTwoOutlets(int, int): float
+ CalculateDeliveryCost(): float
```


LargeSettlement
+ LargeSettlement(int, int, int)
Simulation
SimulationSettlement: Settlement
NoOfCompanies: int
FuelCostPerUnit: single
BaseCostForDelivery: single
Companies: Company[0..*]
+ Simulation()
+ DisplayMenu(): void
- DisplayCompaniesAtDayEnd(): void
- ProcessAddHouseholdsEvent(): void
- ProcessCostOfFuelChangeEvent(): void
- ProcessReputationChangeEvent(): void
- ProcessCostChangeEvent(): void
- DisplayEventsAtDayEnd(): void
+ ProcessDayEnd(): void
+ AddCompany(): void
+ GetIndexOfCompany(string): int
+ ModifyCompany(int): void
+ DisplayCompanies(): void
+ Run(): void
Settlement
StartNoOfHouseholds: int
XSize: int
YSize: int
Households: Household[0..*]
+ Settlement()
+ GetNumberOfHouseholds(): int
+ GetXSize(): int
+ GetYSize(): int
+ GetRandomLocation(int, int): void
+ CreateHouseholds(): void
+ AddHousehold(): void
+ DisplayHouseholds(): void
+ FindOutIfHouseholdEatsOut(int, int, int): boolean
Household
ChanceEatOutPerDay: single
XCoord: int
YCoord: int
ID: int
NextID: int
+ Household(int, int)
+ GetDetails(): string
+ GetChanceEatOut(): single
+ GetX(): int
+ GetY(): int
Company
Name: string
Category: string
Balance: single
ReputationScore: single
AvgCostPerMeal: single
AvgPricePerMeal: single
DailyCosts: single
FamilyOutletCost: single
FastFoodOutletCost: single
NamedChefOutletCost: single
FuelCostPerUnit: single
BaseCostOfDelivery: single
Outlets: Outlet[0..*]
FamilyFoodOutletCapacity: int
FastFoodOutletCapacity: int
NamedChefOutletCapacity: int
+ Company(string, string, single, int, int
single, single)
+ GetName(): string
+ GetNumberOfOutlets(): int
+ GetReputationScore(): single
+ AlterDailyCosts(single): void
+ AlterAvgCostPerMeal(single): void
+ AlterFuelCostPerUnit(single): void
+ AlterReputation(single): void
+ NewDay(): void
+ AddVisitToNearestOutlet(int, int): void
+ GetDetails(): string
+ ProcessDayEnd(): string
+ CloseOutlet(int): boolean
+ ExpandOutlet(int): void
+ OpenOutlet(int, int): void
- GetListOfOutlets(): int[0..*]
- GetDistanceBetweenTwoOutlets(int, int): single
+ CalculateDeliveryCost(): single
Outlet
VisitsToday: int
XCoord: int
YCoord: int
Capacity: int
MaxCapacity: int
DailyCosts: single
+ Outlet(int, int, int)
+ GetCapacity(): int
+ GetX() int
+ GetY() int
+ AlterDailyCosts(single): void
+ AlterCapacity(int) int
+ IncrementVisits(): void
+ NewDay(): void
+ CalculateDailyProfitLoss
(single, single): single
+ GetDetails(): string

Food Magnate Simulation

Programming Theory Questions – Mark Scheme

Q	Answer/Guidance	Marks
1a	1 mark: • `NextID`	1
1b	1 mark for either of: • `GetRandomLocation` • `FindOutIfHouseholdEatsOut`	1
1c	1 mark: • `LargeSettlement`	1
1d	1 mark: • `CloseCompany`	1
1e	1 mark for each of: • `GetListOfOutlets` • `GetDistanceBetweenTwoOutlets`	2
1f	1 mark: • `ToLower`	1
1g	1 mark: • `Outlets`	1
1h	1 mark: • `SimulationSettlement`	1
2	3 marks: • 1 mark for showing calculation: 0.25 * 0.5 • 1 mark for explaining: 0.5 probability nested in selection statement for 0.25 probability // any explanation that makes clear that the 0.5 probability is only evaluated 25% of the time • 1 mark for correct probability: 0.125 / 12.5% / $^1/_8$ / 1 in 8	3
3	3 marks: • (Dimensions of) Settlement must be passed (as a parameter) • Selection/if statement to compare x and y to dimensions of settlement • Creation/construction of outlet inside the selection/if structure (credit can be given for code that is described, but not code in isolation)	3
4	3 marks: • *Dictionary* uses key and value pairs • An index would be the key; the Household object would be the value • Households would be unordered // no guarantee of order	3
5	4 marks: • One X (or Y) coordinate is subtracted from the other • Result of this is squared • Square root of this is found (award this mark and the previous mark if the answer makes it clear that the distance is always positive) • Result is added to the result of the same process performed on Y (or X) coordinate	4
6	2 marks: • Generates a random float/fraction/single • To set `ChanceEatOutPerDay` // to set the probability of eating out	2

Q	Answer/Guidance	Marks
7	3 marks: • Randomly set to either 0 or 1 • Determines whether fuel cost goes up or down // if it is set to zero, fuel cost goes up • If it is set to 1, fuel cost goes down (last two marks can be given if reference is instead made to variable `FuelCostChange`)	3
8	3 marks: • 100,000: company's balance • 200: x position of company / first outlet • 203: y position of company / first outlet (last 2 marks cannot be awarded if they simply refer to 'x' and 'y' – these must be in the correct context)	3
9	5 marks: • Loop through all companies (in companies list) • Convert search term / parameter / company name to lower case • Compare this with (lower-case version of) each company's name / call to `GetName` • If there is a match, the index / value of current / company's location within list is returned • If loop ends without a match, -1 is returned	5
10	3 marks: • User enters 1 / selects 'open new outlet' at the menu • X coordinate is outside the range of the settlement • **or** Y coordinate is within the range of the settlement (for last 2 marks, there must be a clear expression that either x or y being beyond the bounds of the settlement would trigger the message)	3
11	3 marks: • `LargeSettlement` constructor calls `Settlement`/superclass constructor • `Settlement` constructor uses (x and y) values of 1,000 when placing houses • Reduction of `XSize`/`YSize` would take place after this // houses are already positioned according to value of 1,000, even if those values are subsequently reduced	3
12	4 marks: • Constructor overloading is multiple constructors in a single class • Each constructor requires a different signature / order of parameter types • `Settlement` class could have used constructor overloading • `LargeSettlement` would need (in addition to `Settlement` constructor parameters) parameters for (additional) size and number of households	4
13	3 marks:  1 mark per underlined term; order of `GetName` and `AlterDailyCosts` is unimportant, but each must be on the bottom row for the mark.	3
14	2 marks: • 'else' clause would be executed • Values for a named chef restaurant would be used // variables would be assigned values 20, 40 and a random float multiplied by 50	2
TOTAL MARKS		**50**

Food Magnate Simulation

Programming Tasks – Suggested Mark Scheme

Note that the following are recommended solutions, and not an exhaustive list of all possible solutions to each task. The marking guidance should be used as a guide only. Discretion should be used in awarding credit where alternative solutions are given.

Task 1 (max. 6 marks)

1 mark loop set up in `ModifyCompany` to repeat until something other than 'C', '1', '2' or '3' is entered

1 mark loop must contain call to menu display and user input, and must not begin with `choice` uninitialised

```
Public Sub ModifyCompany(ByVal Index As Integer)
    Dim Choice As String = ""
    Dim OutletIndex, X, Y As Integer
    Dim CloseCompany As Boolean
    While Choice <> "C" And Choice <> "1" And Choice <> "2" And Choice <> "3"
        Console.WriteLine(Environment.NewLine & "**********************************")
        Console.WriteLine("*******  MODIFY COMPANY   *******")
        Console.WriteLine("**********************************")
```

1 mark selection statement to catch entry of an upper-case 'C'

1 mark 'operation cancelled' displayed within selection clause

1 mark all inputs, old and new, dealt with correctly

```
            Else
                Console.WriteLine("Invalid coordinates.")
            End If
        ElseIf Choice = "C" Then
            Console.WriteLine("Operation Cancelled")
        End If
    End While
    Console.WriteLine()
End Sub
```

1 mark screenshot showing 'X' causing a repeat of the loop and 'C' causing redisplay of main menu:

```
Enter your choice: X


**********************************
*******  MODIFY COMPANY   *******
**********************************
1. Open new outlet
2. Close outlet
3. Expand outlet

Enter your choice: C

Operation Cancelled


**********************************
**********    MENU     **********
**********************************
1. Display details of households
2. Display details of companies
3. Modify company
4. Add new company
6. Advance to next day
Q. Quit
```

Task 2 (max. 9 marks)

1 mark Boolean or equivalent to store whether a collision has occurred

1 mark outer loop attempts to continue until valid location is found, even if unsuccessful (intent must be clear)

1 mark random location generated inside outer loop

1 mark inner loop to iterate through `Households` list

1 mark selection statement to determine whether location is already occupied

1 mark flag set in inner loop to store that a location was already occupied

1 mark Boolean (`placed` in this code) set to reflect no collision occurring

1 mark when outer loop ends, X and Y will always represent a non-colliding location

```
Public Sub GetRandomLocation(ByRef X As Integer, ByRef Y As Integer)
    Dim placed As Boolean = False
    Dim collision As Boolean

    While placed = False

        collision = False
        X = Int(Rnd() * XSize)
        Y = Int(Rnd() * YSize)

        For count = 0 To Households.Count - 1
            If Households(count).GetX = X And
                Households(count).GetY = Y Then
                collision = True
            End If
        Next
        If collision = False Then
            placed = True
        End If
    End While
End Sub
```

1 mark eight household locations are all different:

```
1     Coordinates: (1, 1)     Eat out probability: 0.7859395
2     Coordinates: (0, 2)     Eat out probability: 0.3263758
3     Coordinates: (0, 1)     Eat out probability: 0.545325
4     Coordinates: (2, 1)     Eat out probability: 0.8064888
5     Coordinates: (1, 0)     Eat out probability: 0.4022227
6     Coordinates: (1, 2)     Eat out probability: 0.2681935
7     Coordinates: (0, 0)     Eat out probability: 0.1292226
8     Coordinates: (2, 0)     Eat out probability: 0.9387147
```

Task 3 (max. 7 marks)

1 mark new subroutine declared correctly in the `Outlet` class

1 mark `MaxCapacity` multiplied by the parameter (name of parameter unimportant)

```
Public Sub ExtendCapacity(ByVal Multiplier As Integer)
    MaxCapacity *= Multiplier
End Sub
```

1 mark call to generate a random number that could be appropriate to the 40/35/25 distribution in `ExpandOutlet`

1 mark probabilities are actually 40%, 35% and 25%

1 mark correct parameter passed to `ExtendCapacity` in all circumstances

1 mark only output message in these circumstances is 'outlet max capacity expanded'

```
    If Result = Change Then
        Console.WriteLine("Capacity adjusted.")
    Else
        Dim Multiplier As Integer = Int(Rnd() * 100)
        If Multiplier < 40 Then
            Outlets(ID).ExtendCapacity(2)
        ElseIf Multiplier < 75 Then
            Outlets(ID).ExtendCapacity(3)
        Else
            Outlets(ID).ExtendCapacity(4)
        End If
        Console.WriteLine("Outlet max capacity expanded")
    End If
End Sub
```

1 mark correct message output, and Paltry Poultry outlet 4 max capacity doubled, tripled or quadrupled:

```
Number of outlets: 4
Outlets
1. Coordinates: (800, 390)     Capacity: 120      Maximum Capacity: 173
2. Coordinates: (400, 390)     Capacity: 120      Maximum Capacity: 198
3. Coordinates: (820, 370)     Capacity: 120      Maximum Capacity: 201
4. Coordinates: (800, 600)     Capacity: 120      Maximum Capacity: 182
```

```
Enter ID of outlet: 4
Enter amount you would like to expand the capacity by: 1000
Outlet max capacity expanded
```

```
Number of outlets: 4
Outlets
1. Coordinates: (800, 390)     Capacity: 120      Maximum Capacity: 173
2. Coordinates: (400, 390)     Capacity: 120      Maximum Capacity: 198
3. Coordinates: (820, 370)     Capacity: 120      Maximum Capacity: 201
4. Coordinates: (800, 600)     Capacity: 182      Maximum Capacity: 364
```

Task 4 (max. 8 marks)

1 mark new subroutine, correct name, type and parameters in the `Settlement` class

1 mark integer to store number of leavers

1 mark loop to iterate over all households

1 mark selection structure, based on 2% probability

1 mark removal of household from list, inside selection clause

1 mark integer variable incremented inside selection clause, and returned after loop

```
Public Function ProcessLeavers() As Integer
    Dim Leavers As Integer = 0
    For index = 0 To Households.Count - 1
        If Int(Rnd() * 100) < 2 Then
            Households.RemoveAt(index)
            Leavers += 1
        End If
    Next
    Return Leavers
End Function
```

1 mark call to `ProcessLeavers` as the final instruction in `ProcessDayEnd`

```
    DisplayCompaniesAtDayEnd()
    DisplayEventsAtDayEnd()
    Console.WriteLine(SimulationSettlement.ProcessLeavers() &
                      " households left the settlement")
End Sub
```

1 mark screenshot displays number of households removed, which might be the number by itself (i.e. without the text 'households left the settlement') and/or zero:

```
************************
*****   Events:   *****
************************

No events.
4 households left the settlement

************************************
**********    MENU     **********
************************************
1. Display details of households
2. Display details of companies
3. Modify company
4. Add new company
6. Advance to next day
Q. Quit
```

Task 5 (max. 7 marks)

1 mark new input message within code executed when user enters '3'

1 mark loop to iterate over all companies in the list

1 mark output to contain call to company's `GetName` subroutine

1 mark output to display 1-based indices instead of 0-based indices

1 mark user input stored in the `index` variable

1 mark user input decremented

```
Case "3"
    Do
        Console.WriteLine("Enter the number next to the company you wish to modify: ")
        For i = 0 To Companies.Count - 1
            Console.WriteLine((i + 1) & ": " & Companies(i).getName())
        Next
        Index = Console.ReadLine() - 1
    Loop Until Index <> -1
    ModifyCompany(Index)
Case "4"
```

1 mark user input should now be based on a list, with outlets numbered 1, 2, 3; final outlet of 'Paltry Poultry' should have a capacity equal to its max capacity:

```
**********************************
**********     MENU     **********
**********************************
1. Display details of households
2. Display details of companies
3. Modify company
4. Add new company
6. Advance to next day
Q. Quit

Enter your choice: 3
Enter the number next to the company you wish to modify:
1: AQA Burgers
2: Ben Thor Cuisine
3: Paltry Poultry
3
```

```
Number of outlets: 4
Outlets
1. Coordinates: (800, 390)     Capacity: 120      Maximum Capacity: 186
2. Coordinates: (400, 390)     Capacity: 120      Maximum Capacity: 168
3. Coordinates: (820, 370)     Capacity: 120      Maximum Capacity: 180
4. Coordinates: (800, 600)     Capacity: 200      Maximum Capacity: 200
```

Task 6 (max. 5 marks)

1 mark subroutine correctly declared as a constructor

1 mark correct parameters declared

1 mark `XCoord` and `YCoord` set to random integers between 0 and `MaxX`/`MaxY` inclusive

```
Public Sub New(ByVal MaxX As Integer, MaxY As Integer,
               MaxCapacityBase As Integer, IsRandom As Boolean)

    XCoord = Int(Rnd() * (MaxX + 1))
    YCoord = Int(Rnd() * (MaxY + 1))
```

1 mark the rest of the subroutine executes as normal. **A.** call to a new subroutine (to which existing constructor might call) that contains the extra lines. **R.** if call is made to other constructor.

```
    Capacity = Int(MaxCapacityBase * 0.6)
    MaxCapacity = MaxCapacityBase + Int(Rnd() * 50) - Int(Rnd() * 50)
    DailyCosts = MaxCapacityBase * 0.2 + Capacity * 0.5 + 100
    NewDay()
End Sub
```

1 mark screen capture shows values for outlet coordinates that are all 10 or lower:

```
Name: Paltry Poultry
Type of business: fast food
Current balance: 17000
Average cost per meal: 5
Average price per meal: 10
Daily costs: 100
Delivery costs: 0.07916134
Reputation: 98.63483

Number of outlets: 4
Outlets
1. Coordinates: (2, 3)     Capacity: 120
2. Coordinates: (0, 6)     Capacity: 120
3. Coordinates: (1, 8)     Capacity: 120
4. Coordinates: (3, 9)     Capacity: 120
```

Task 7 (max. 7 marks)

1 mark option 5 correctly added in `DisplayMenu`

```
Console.WriteLine("4. Add new company")
Console.WriteLine("5. Advance")
Console.WriteLine("6. Advance to next day")
Console.WriteLine("Q. Quit")
```

1 mark selection structure in `Run` captures an input of 5

1 mark any suitable prompt for the user to enter a number of days

1 mark integer variable set to user input

1 mark loop that runs the correct number of times

1 mark call to `ProcessDayEnd` inside the loop and no other code

```
Case "4"
    AddCompany()
Case "5"
    Console.Write("Enter number of days: ")
    Dim NumberOfDays As Integer = Console.ReadLine
    For x = 1 To NumberOfDays
        ProcessDayEnd()
    Next
Case "6"
```

1 mark three days' worth of financials and events should be displayed, although if the previous balance for Paltry Poultry is anything other than 17000, and would match code, credit can be given:

```
Paltry Poultry

Daily costs for company: 100
Cost for delivering produce to outlets: 110.3032
Outlet 1 profit/loss: -180
Outlet 2 profit/loss: -45
Outlet 3 profit/loss: -195
Outlet 4 profit/loss: -155
Previous balance for company: 15344.39
New balance for company: 14559.09
```

Task 8 (max. 9 marks)

1 mark prompt updated to include reference to '4' generating a random company

1 mark loop updated to include '4' as a terminating condition

```
Do
    Console.Write("Enter 1 for a fast food company, 2 for a family company, " &
                  "3 for a named chef company or 4 for a company chosen at random: ")
    TypeOfCompany = Console.ReadLine()
Loop Until TypeOfCompany = "1" Or TypeOfCompany = "2" Or
    TypeOfCompany = "3" Or TypeOfCompany = "4"
```

1 mark updating selection structure to ensure 3 results in 'named chef'

1 mark addition to selection structure to catch 4 or 'else' (i.e. not 1, 2 or 3)

1 mark random number generated, even if likelihoods are not evenly distributed

1 mark selection structure sets restaurant type according to random number

1 mark three company types are equally likely to be created

1 mark additional code does not impede code from the Skeleton Program

```
ElseIf TypeOfCompany = "3" Then
    TypeOfCompany = "named chef"
Else
    Dim companyType As Integer = Int(Rnd() * 3)
    If companyType = 0 Then
        TypeOfCompany = "fast food"
    ElseIf companyType = 1 Then
        typeOfCompany = "family"
    Else
        TypeOfCompany = "named chef"
    End If
End If
SimulationSettlement.GetRandomLocation(X, Y)
```

1 mark input of '4', 'Random Restaurant' and '50000', resulting in a company of any type being created

```
Enter a name for the company: Random Restaurant
Enter the starting balance for the company: 50000
Enter 1 for a fast food company, 2 for a family company,
3 for a named chef company or 4 for a company chosen at random:
4
```

```
Name: Random Restaurant
Type of business: named chef
Current balance: 35000
Average cost per meal: 20
Average price per meal: 40
Daily costs: 100
Delivery costs: 0
Reputation: 104.639
```

Task 9 (max. 6 marks)

1 mark new subroutine created, with no parameters and no return

1 mark loop to iterate over all outlets in the `Outlets` list

1 mark call to `CloseOutlet` for each outlet

```
Public Sub CloseAllOutlets()
    For current = 0 To Outlets.Count - 1
        CloseOutlet(current)
    Next
End Sub
```

1 mark selection statement in `ProcessDayEnd` to check for balance of less than zero, immediately before return statement

1 mark call to `CloseAllOutlets` in selection structure

```
    Balance += ProfitLossFromOutlets - DailyCosts - DeliveryCosts
    Details &= "New balance for company: " & Balance.ToString()
    If Balance < 0 Then
        CloseAllOutlets()
    End If
    Return Details
End Function
```

1 mark entering '2' in the main menu should reveal that 'Bankrupt Burgers' has no outlets:

```
Name: Bankrupt Burgers
Type of business: family
Current balance: -159
Average cost per meal: 12
Average price per meal: 14
Daily costs: 100
Delivery costs: 0
Reputation: 104.545

Number of outlets: 0
Outlets
```

Task 10 (max. 11 marks)

1 mark class definition, which includes inheritance

1 mark constructor correctly declared, with two parameters

1 mark call within constructor to the `Outlet` constructor, passing correct values

1 mark `Move` subroutine declared

1 mark random number generated

1 mark selection statement uses four different possible values of random number

1 mark each of north, south, east and west correctly simulated

```
Class FoodTruck
    Inherits Outlet

    Public Sub New(XCoord As Integer, YCoord As Integer)
        MyBase.New(XCoord, YCoord, 10)
    End Sub

    Public Sub Move()
        Dim direction As Integer = Int(Rnd() * 4)
        If direction = 0 Then
            XCoord += 1
        ElseIf direction = 1 Then
            XCoord -= 1
        ElseIf direction = 2 Then
            YCoord += 1
        Else
            YCoord -= 1
        End If
    End Sub
End Class
```

1 mark either creation of loop to iterate through outlets in `ProcessDayEnd` subroutine, or an attempt to use the existing loop to call `move`, even if syntactically invalid

1 mark selection statement to check whether an outlet is an instance of a `FoodTruck`

1 mark `move` subroutine called for all `FoodTruck` objects and only `FoodTruck` objects

```
For Current = 0 To Outlets.Count - 1
    If Outlets(Current).GetType Is GetType(FoodTruck) Then
        Outlets(Current).Move()
    End If
Next
```

1 mark screen captures show a difference of 1 in **either** X **or** Y coordinates between days:

```
Enter number of companies that exist at start of simulation: 1
Enter a name for the company: Taco Truck
Enter the starting balance for the company: 30000
Enter 1 for a fast food company, 2 for a family company or 3 for a named chef company: 1
```

```
Name: Taco Truck
Type of business: fast food
Current balance: 28000.0
Average cost per meal: 5.0
Average price per meal: 10.0
Daily costs: 100.0
Delivery costs: 0.0
Reputation: 93.62324

Number of outlets: 1
Outlets
1. Coordinates: (595, 285)
```

```
Name: Taco Truck
Type of business: fast food
Current balance: 28355.0
Average cost per meal: 5.0
Average price per meal: 10.0
Daily costs: 100.0
Delivery costs: 0.0
Reputation: 93.62324

Number of outlets: 1
Outlets
1. Coordinates: (594, 285)
```

Task 11 (max. 15 marks)

1 mark data structure created to store indexes

1 mark selection structure has been changed from 'equals' to 'contains' or equivalent

1 mark inside selection structure, adding the index to the data structure

```
Dim indexes As New ArrayList

For current = 0 To Companies.Count - 1
    If Companies(current).GetName().ToLower().contains(CompanyName.ToLower()) Then
        indexes.Add(current)
    End If
Next
```

1 mark selection structure to check for only a single match, **A.** if 'else' by process of elimination

1 mark correct index returned for a single match

```
If indexes.Count = 1 Then
    Return indexes(0)
```

1 mark selection structure to check for multiple matches, **A.** if 'else' by process of elimination

1 mark prompt for user entry, either before or after attempt to display matches

1 mark loop to iterate through all matches in an attempt to display them

1 mark name of each matching outlet displayed

1 mark user input requested only if multiple matches have been found

1 mark loop to iterate through the matches in an attempt to compare with user input

1 mark comparison ('equals', not 'contains') is made inside the loop, with case ignored

1 mark correct index returned in the event of a match

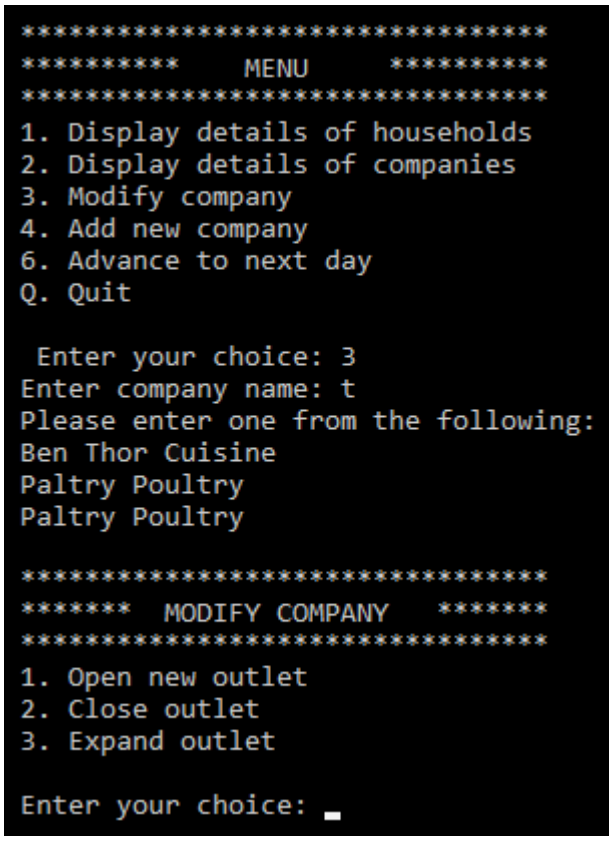
1 mark value of -1 returned if no matches are found or if second entry does not match

```
ElseIf indexes.Count > 1 Then
    Console.WriteLine("Please enter one from the following:")
    For i = 0 To indexes.Count - 1
        Console.WriteLine(Companies(indexes(i)).GetName())
    Next
    Dim input As String = Console.ReadLine()
    For i = 0 To indexes.Count - 1
        If input.ToLower().Equals(Companies(indexes(i)).GetName().ToLower()) Then
            Return indexes(i)
        End If
    Next
End If

Return -1
```

(continues on next page)

1 mark screen capture displays 'Paltry Poultry' and 'Ben Thor Cuisine', with 'Paltry Poultry' entered and accepted:



Task 12 (max. 7 marks)

1 mark outlet with a matching `ID` is accessed

1 mark number of seats is set to a call to `GetCapacity`

1 mark selection structure uses `Category` attribute of the `Company` class

1 mark cost per seat is set correctly for all categories

1 mark balance is decremented by the cost per seat multiplied by the number of seats

1 mark the original code, to remove the outlet and check for zero outlets, should run unchanged

```
Public Function CloseOutlet(ByVal ID As Integer) As Boolean

    Dim o As Outlet = Outlets(ID)
    Dim seats As Integer = o.GetCapacity()
    Dim costPerSeat As Integer
    If Category = "fast food" Then
        costPerSeat = 75
    ElseIf Category = "family" Then
        costPerSeat = 50
    Else
        costPerSeat = 150
    End If
    Balance -= (seats * costPerSeat)

    Dim CloseCompany As Boolean = False
    Outlets.RemoveAt(ID)
    If Outlets.Count = 0 Then
        CloseCompany = True
    End If
    Return CloseCompany
End Function
```

1 mark current balance of Paltry Poultry should change from 17000 to 8000, and there should be one outlet fewer:

```
Name: Paltry Poultry
Type of business: fast food
Current balance: 17000
Average cost per meal: 5
Average price per meal: 10
Daily costs: 100
Delivery costs: 10.30317
Reputation: 95.64574

Number of outlets: 4
Outlets
1. Coordinates: (800, 390)     Capacity: 120
2. Coordinates: (400, 390)     Capacity: 120
3. Coordinates: (820, 370)     Capacity: 120
4. Coordinates: (800, 600)     Capacity: 120
```

```
Name: Paltry Poultry
Type of business: fast food
Current balance: 8000
Average cost per meal: 5
Average price per meal: 10
Daily costs: 100
Delivery costs: 8.040665
Reputation: 95.64574

Number of outlets: 3
Outlets
1. Coordinates: (800, 390)     Capacity: 120
2. Coordinates: (400, 390)     Capacity: 120
3. Coordinates: (820, 370)     Capacity: 120
```

Task 13 (max. 8 marks)

1 mark additional line added to `DisplayMenu` with correct number and text

```
Console.WriteLine("4. Add new company")
Console.WriteLine("5. Remove company")
Console.WriteLine("6. Advance to next day")
Console.WriteLine("Q. Quit")
```

1 mark addition of '5' to the selection structure in `run`

1 mark prompt for company name and storage of user input in a string variable

1 mark selection structure to check whether the index either is or is not -1

1 mark removal of company at the correct index, only if the index is not -1

1 mark loop will terminate if the loop is not -1

1 mark loop will terminate if 'cancel', in any combination of upper case / lower case, is entered

```
case "4":
    addCompany();
    break;
case "5":
    String nameOfCompany;
    do {
        Console.write("Enter company name: ");
        nameOfCompany = Console.readLine();
        index = getIndexOfCompany(nameOfCompany);
        if (index > -1) {
            companies.remove(index);
        }
    } while (index == -1 && !(nameOfCompany.toUpperCase().equals("CANCEL")));
    break;
case "6":
    processDayEnd();
    break;
```

1 mark entering 'CANCEL' returns user to the main menu; entering 'Paltry Poultry' results in only two companies being present in the simulation – AQA Burgers and Ben Thor Cuisine:

```
Enter your choice: 5
Enter company name: CANCEL

************************************
**********     MENU      **********
************************************
1. Display details of households
2. Display details of companies
3. Modify company
4. Add new company
5. Remove company
6. Advance to next day
Q. Quit
```

```
*******************************
*** Details of all companies: ***
*******************************

Name: AQA Burgers
Type of business: fast food
Current balance: 86000.0
Average cost per meal: 5.0
Average price per meal: 10.0
Daily costs: 100.0
Delivery costs: 33.661526
Reputation: 94.51912

Number of outlets: 7
Outlets
1. Coordinates: (200, 203)     Capacity: 120
2. Coordinates: (300, 987)     Capacity: 120
3. Coordinates: (500, 500)     Capacity: 120
4. Coordinates: (305, 303)     Capacity: 120
5. Coordinates: (874, 456)     Capacity: 120
6. Coordinates: (23, 408)     Capacity: 120
7. Coordinates: (412, 318)     Capacity: 120


Name: Ben Thor Cuisine
Type of business: named chef
Current balance: 85400.0
Average cost per meal: 20.0
Average price per meal: 40.0
Daily costs: 100.0
Delivery costs: 0.0
Reputation: 102.05345

Number of outlets: 1
Outlets
1. Coordinates: (390, 800)     Capacity: 30
```

Task 14 (max. 20 marks)

1 mark `GetBalance` declared in the `Company` class, with no parameters and correct type

1 mark correct return statement, **R.** if any additional code

```
Public Function GetBalance() As Single
    Return Balance
End Function
```

1 mark additional option added to `DisplayMenu`

```
Console.WriteLine("4. Add new company")
Console.WriteLine("5. Run to target")
Console.WriteLine("6. Advance to next day")
```

1 mark selection structure modified to include '5'

1 mark call to new `RunToTarget` subroutine if '5' is entered

```
Case "4"
    AddCompany()
Case "5"
    RunToTarget()
Case "6"
    ProcessDayEnd()
```

1 mark new `RunToTarget` subroutine declared

1 mark user prompted for an upper limit and a lower limit

1 mark each user input stored as a separate integer

1 mark variable names `UpperLimit` and `LowerLimit` used as instructed

1 mark variables for number of days, balance and company name, of appropriate types, declared

```
Public Sub RunToTarget()

    Console.Write("Upper Limit: ")
    Dim UpperLimit As Integer = Console.ReadLine
    Console.Write("Lower Limit: ")
    Dim LowerLimit As Integer = Console.ReadLine
    Dim NumberOfDays As Integer = 0
    Dim c As Company
    Dim CompanyName As String = ""
    Dim Balance As Single = 0
```

1 mark loop to run until a balance is equal to or above the upper limit, or equal to or below the lower limit

1 mark loop to iterate over each company in the simulation

1 mark comparison with both upper and lower limits

1 mark call to `ProcessDayEnd` once within each iteration

1 mark no call to `ProcessDayEnd` if a balance has already reached termination condition, including if the termination condition was already reached by the start of the first loop (i.e. zero days should be a possibility)

1 mark number of days incremented within each iteration

(continues on next page)

1 mark calls to `GetName` and `GetBalance` have occurred before loop terminates

```
Do
    For i = 0 To Companies.Count - 1
        c = Companies(i)
        If c.GetBalance() >= UpperLimit Or c.GetBalance() <= LowerLimit Then
            CompanyName = c.GetName()
            Balance = c.GetBalance()
        End If
        If CompanyName = "" Then
            NumberOfDays += 1
            ProcessDayEnd()
        End If
    Next
Loop While CompanyName = ""
```

1 mark values for the company's name and balance, and the number of days, are output

1 mark output values are correct under all circumstances

```
Console.WriteLine("Company: " & CompanyName)
Console.WriteLine("Balance: " & Balance)
Console.WriteLine("Days:    " & NumberOfDays)
```

1 mark screen evidence showing that multiple days have passed, and the balance is either >= 100000 or <= 0:

```
************************
*****   Events:   *****
************************

Company: Ben Thor Cuisine
Balance: 100505
Days:    19
```

Task 15 (max. 14 marks)

1 mark class definition with correct identifier, which inherits from `LargeSettlement`

1 mark valid constructor with three integer parameters

1 mark valid call within constructor to constructor of superclass

1 mark subroutine `AddHousehold` declared with `Overrides` modifier

1 mark generation of random number, within `AddHousehold`, between 2 and 20 inclusive

1 mark generation of random X and Y coordinates within the settlement (easiest via a call to the `Settlement` class's `GetRandomLocation` subroutine, but any valid approach can be credited)

1 mark loop set up that will iterate once for each household in this location (integer between 2 and 20)

1 mark each new household, within the loop, added to the `Households` list

```
Class CitySettlement
    Inherits LargeSettlement

    Public Sub New(ByVal ExtraXSize As Integer,
                   ExtraYSize As Integer, ExtraHouseholds As Integer)
        MyBase.New(ExtraXSize, ExtraYSize, ExtraHouseholds)
    End Sub

    Public Overrides Sub AddHousehold()
        Dim NumberOfHouseholds As Integer = Int(Rnd() * 19) + 2
        Dim X, Y As Integer
        GetRandomLocation(X, Y)
        For i = 0 To NumberOfHouseholds - 1
            Households.Add(New Household(X, Y))
        Next
    End Sub

End Class
```

1 mark Overridable modifier added to pre-existing `AddHouseholds` subroutine

```
Public Overridable Sub AddHousehold()
    Dim X, Y As Integer
    GetRandomLocation(X, Y)
    Dim Temp As New Household(X, Y)
    Households.Add(Temp)
End Sub
```

(continues on next page)

1 mark first prompt of the simulation amended to allow user to select a city settlement

1 mark user selecting a city settlement results in prompts for additional X and Y coordinates and number of households

1 mark user selecting a city settlement results in call to `CitySettlement` constructor

1 mark other inputs should continue to work as previously (i.e. new code does not disrupt old code)

```
Public Sub New()
    FuelCostPerUnit = 0.0098
    BaseCostForDelivery = 100
    Dim Choice As String
    Console.Write("Enter L for a large settlement, C for a city settlement, " &
                  "anything else for a normal size settlement: ")
    Choice = Console.ReadLine()
    If Choice = "L" Or Choice = "C" Then
        Dim ExtraX, ExtraY, ExtraHouseholds As Integer
        Console.Write("Enter additional amount to add to X size of settlement: ")
        ExtraX = Console.ReadLine()
        Console.Write("Enter additional amount to add to Y size of settlement: ")
        ExtraY = Console.ReadLine()
        Console.Write("Enter additional number of households to add to settlement: ")
        ExtraHouseholds = Console.ReadLine()
        If Choice = "L" Then
            SimulationSettlement = New LargeSettlement(ExtraX, ExtraY, ExtraHouseholds)
        Else
            SimulationSettlement = New CitySettlement(ExtraX, ExtraY, ExtraHouseholds)
        End If
    Else
        SimulationSettlement = New Settlement()
    End If
```

1 mark screen capture should show consecutive households at the same location

```
2765     Coordinates: (617, 150)     Eat out probability: 0.1691158
2766     Coordinates: (617, 150)     Eat out probability: 0.6980169
2767     Coordinates: (617, 150)     Eat out probability: 0.04484648
2768     Coordinates: (617, 150)     Eat out probability: 0.1412596
2769     Coordinates: (617, 150)     Eat out probability: 0.6288148
```

Name	

ZigZag Education supporting

A Level AQA Computer Science Paper 1

Summer 2020

Food Magnate Simulation

Electronic Answer Document (EAD)

Instructions

- Enter your name in the box at the top of this page
- Answer **all** questions by entering your answers into this document
- Remember to **save** this document regularly
- Save and print this document and any additional pages

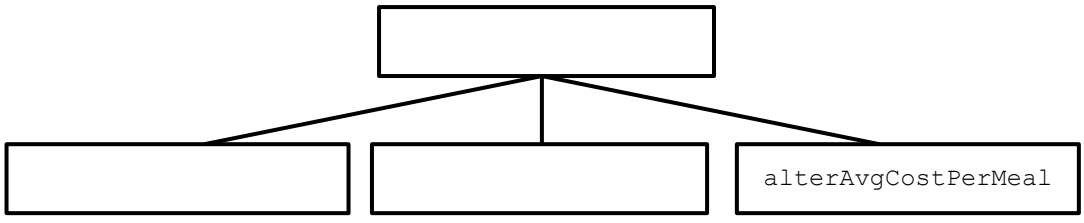
- Answer **all** questions
- The marks available for each question are shown in brackets

- You will need:
  - □ access to a computer
  - □ access to a printer
  - □ access to appropriate software
  - □ electronic copies of the required skeleton code
  - □ EAD (Electronic Answer Document)

Total marks:

Programming Theory Questions

Answer all questions. Remember to save this document regularly.

Q		Answer	*Mark* *(leave blank)*
1	(a)		
	(b)		
	(c)		
	(d)		
	(e)		
	(f)		
	(g)		
	(h)		
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			

Programming Tasks

Answer all questions. Remember to save this document regularly.

Q	Answer	*Mark* *(leave blank)*
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		