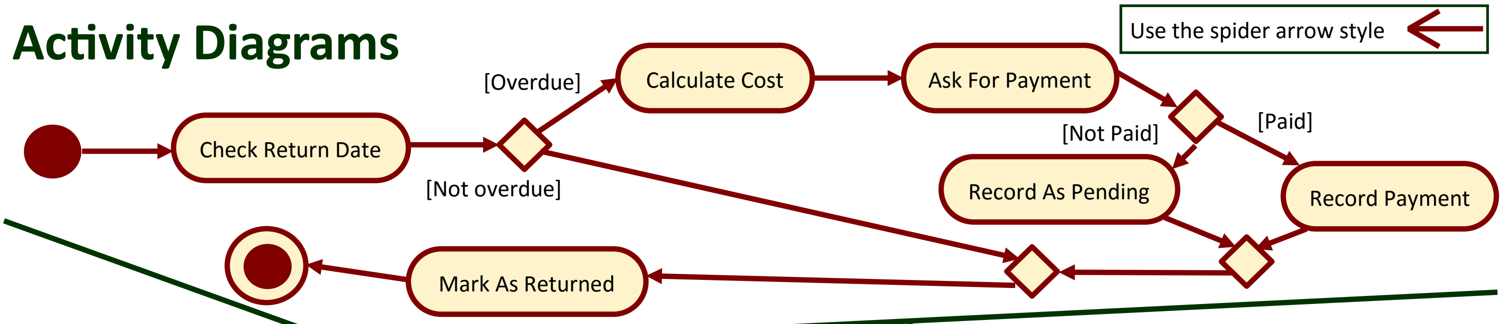
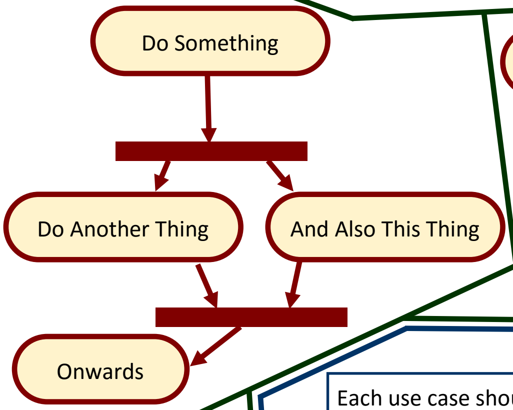
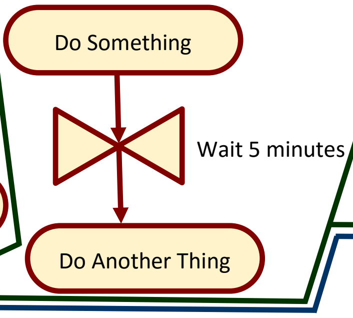
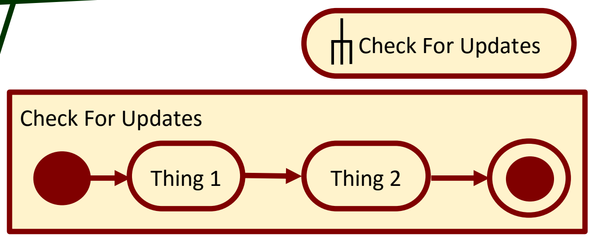
Activity Diagrams



Use the spider arrow style







Use-Case Diagrams

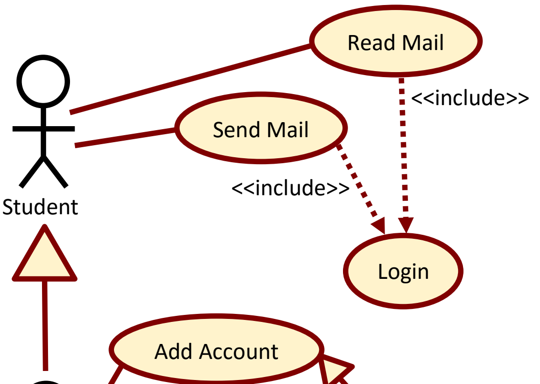
Each use case should be seen as a permission

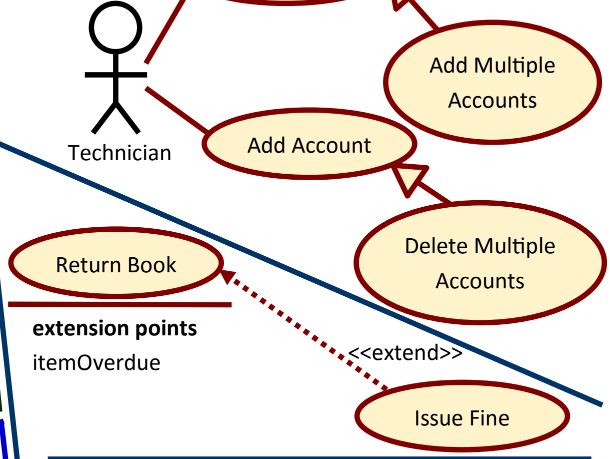
Actor generalisation, Inheritance arrow points to common similarities, technician inherits from student

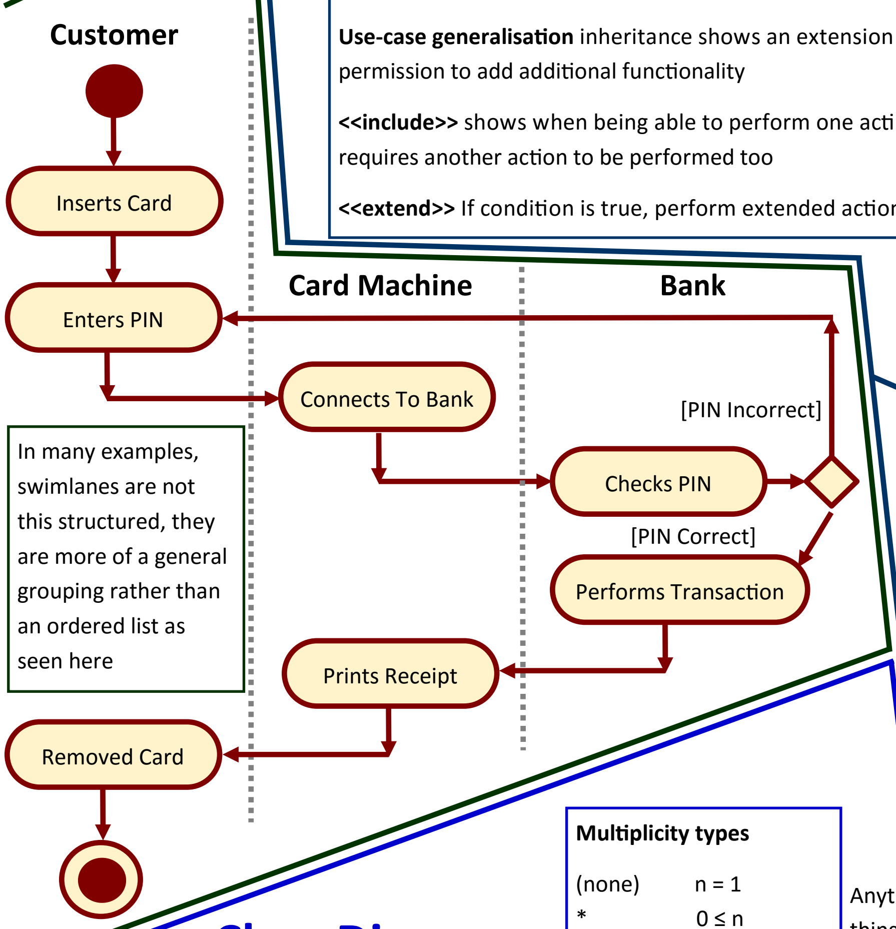
Use-case generalisation inheritance shows an extension of a permission to add additional functionality

<<include>> shows when being able to perform one action requires another action to be performed too

<<extend>> If condition is true, perform extended action too







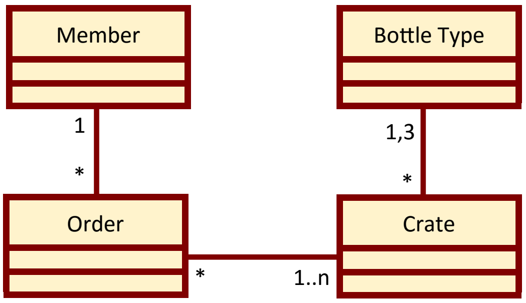
In many examples, swimlanes are not this structured, they are more of a general grouping rather than an ordered list as seen here

Both Spider arrow and dotted line ⇠

<<include>> Arrow points towards included case

<<extend>> Points from extended to base case

Class Diagrams



Multiplicity types

(none)	n = 1
*	0 ≤ n
x	n = x
x..y	x ≤ n ≤ y
1..n	1 ≤ n
x, y	n = x or n = y

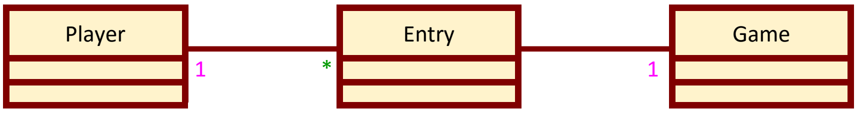
Anything can be an object, a concrete thing (person / item), or a thing that happens (where a thing does something, eg jumping)

Be aware as objects and multiplicities may change as you read through, especially if you spot something new on a second read

Prioritise getting the relationships between classes then get their multiplicities.

Multiplicities are the “#x / y” eg:(30 PCs / room) would give you “|Room|----30|PC|”

“A member can make many orders” - * Orders / Member, “An order can include more than one type of create” - 1..n Crates / Order, “A crate is typically 12 bottles of the same type but some consist of 3 types of 4 bottles” - 1 or 3 Bottle Type / Crate



"(1)| Players can enter as many games as they wish | (2)| but only one entry is allowed per player per game|"

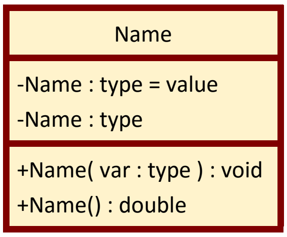
Many entries / player (player can enter multiple game instances

One player / entry (Each entry represents one player)

one game / entry (A player can only be entered into a specific game once)

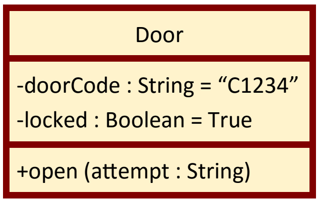
This is a class, we have the name at the top, followed by it's attributes (variables) which are private, and finally it's operations (methods) which are typically public.

You can make up your own types here to suit the needs of the class



Most of the info you get is context and does not need to be modelled - ignore the external stuff

"A locked door is opened by a digital keypad where the door code is set to C1234. The door code is entered and ten the user presses enter. If the attempt is entered correctly it unlocks the door, if it is entered incorrectly it displays the message "Invalid entry" and it remains locked



```
open()
     If attempt == doorCode
          locked = false
     Else
          Display "Invalid entry"
```

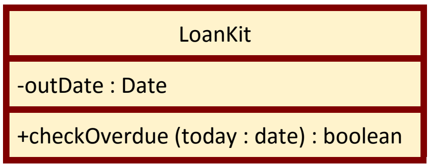
Do not model any external data (eg: getting user input or other types used by your class), pretend they exist and use them

Spot the variables, are they standard or a special type?

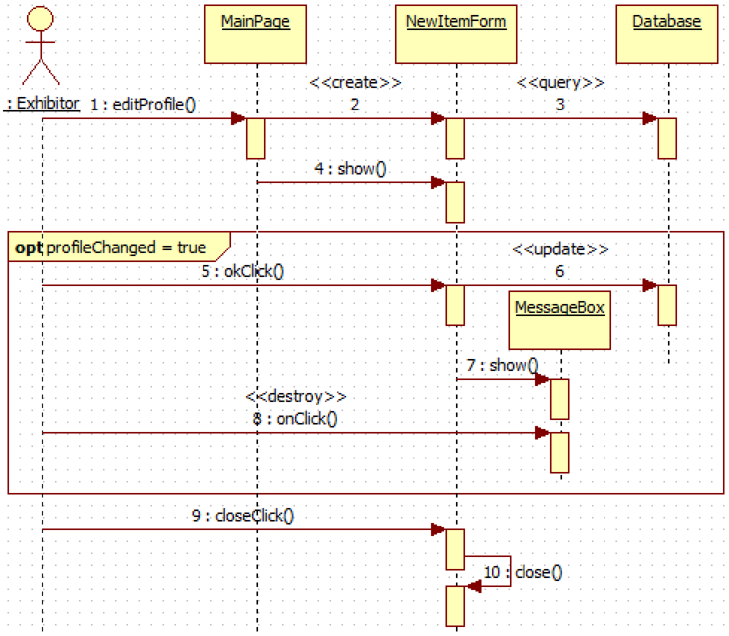
Spot the methods, what do they need to work and do they return anything?

Write the DD pseudocode, keep it basic and to the point

When an iPad is taken out on loan a record (LoanKit) must be created. This will hold the date the item is taken out. All such loans are valid for 14 days after which they are said to be overdue. A LoanKit should be able to answer the query "are you overdue" given today's date



```
checkOverdue()
     If today - 14 days > outdate
          return true
     Else
          return false
```



Sequence Diagrams

Loop [min, max] a for loop

Opt [if x] Perform contents if x

Alt [if x] Perform contents if x else..

Alt [else] Perform contents if !x

Go through and spot the different objects (the user, windows, message boxes, databases), and actions (clicking a button, pressing enter)

"An exhibitor will select from a menu on the Main page to edit their profile". If clicking buttons opens up a new window, call it the action the user wants to perform (editProfile), otherwise call it (xxxClick)

Do not get bogged down in the specifics of naming or semantics, just model the high level overview basics, the windows and their interactions, then you are done. It's much simpler than it looks

<<create>> Creating a new window, typically followed by show()

<<query>> Used when connecting to external sources, ignore content / return

<<destroy>> Used to kill windows, mark with **X** at end of line

<<update>>, **<<insert>>** Updating / modifying the contents of a table

<<table>> #name Used as top level object representing a table with #name

Do not get bogged down in the specifics of naming or semantics, just model the high level overview basics, the windows and their interactions, then you are done. It's much simpler than it looks

On top level objects, name them as such ": NameOfObject" unless they are databases in which case name them "<<table>>\n: NameOfTable" **[Scrap that, found alternatives in past answers]**

It's more important to be consistent that right in this exam, letter capitalisation for instance does not matter as long as it's consistent