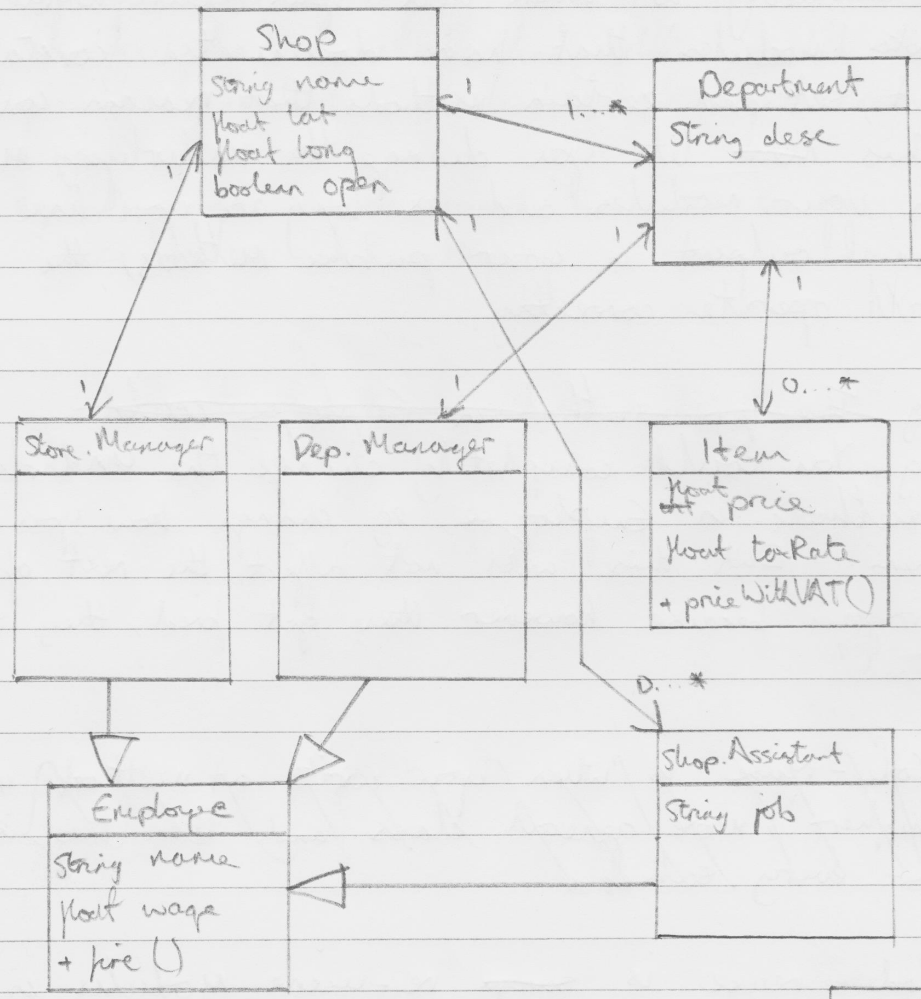
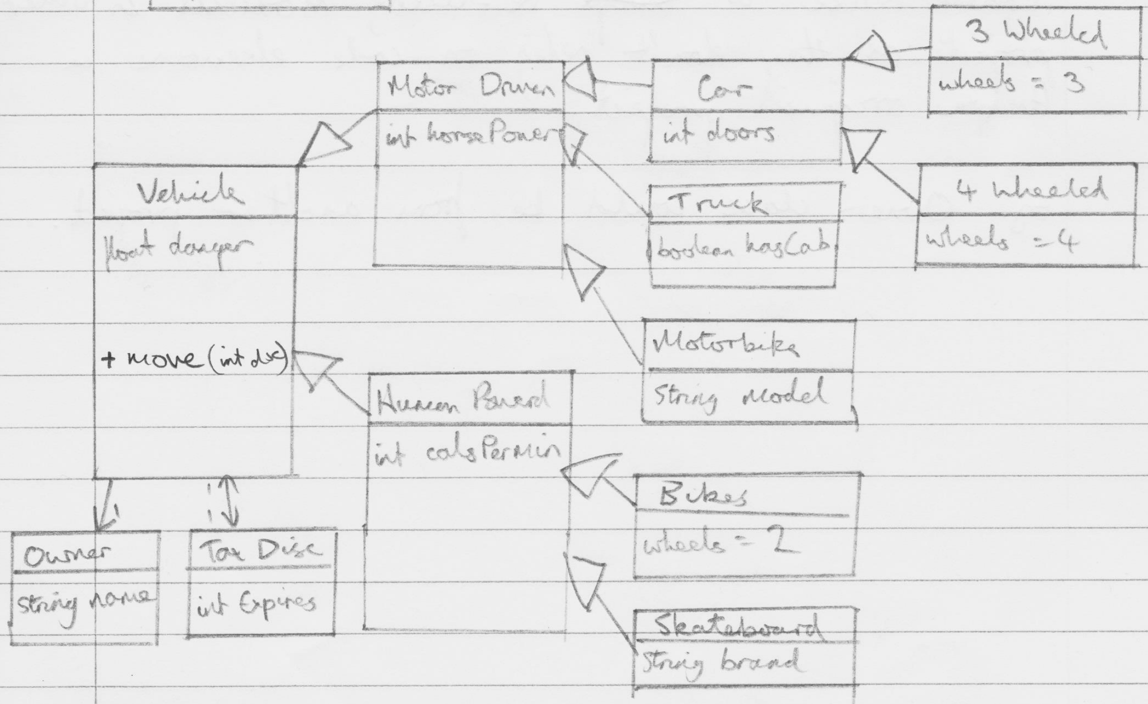
1.


Shop
String name
float lat
float long
boolean open
1
1...*
Department
String desc
1
1
1
1
0...*
Store. Manager
Dep. Manager
Item
~~int~~ float price
float taxRate
+ priceWithVAT()
0...*
Shop. Assistant
String job
Employee
String name
float wage
+ fire()


Motor Driven
int horsePower
Car
int doors
3 Wheeled
wheels = 3
4 Wheeled
wheels = 4
Truck
boolean hasCab
Motorbike
String model
Vehicle
float danger
+ move(int dx)
Human Powered
int calsPerMin
Bikes
wheels = 2
Skateboard
String brand
1
1
Owner
string name
Tax Disc
int Expires

2. a) Modularity is when you split your program into 'modules' that have documented, constant get, set and certain methods that remain constant. This ~~mea~~ lets you change the underlying methods in your modules and as long as you have the same output in your public methods, the module will operate correctly.

~~eg The fire method in Employee could~~
eg. You could completely change the underlying methods in Employee, eg change how you fire them. But this will not affect the rest of the program because however they get fired, they still get fired!

b) ~~Code-reuse is when you write eg method in a 'higher', more general class and then this is used for every subclass~~

Code-reuse is ~~swap~~ re-using modules in other projects as they don't rely on code elsewhere in their original project.

Eg. Owner class could be from another project.

c) Encapsulation is keeping variables private and only having get and set methods when needed. This way your can control there limits or logic when set.

eg. Employee wage must be > 0 so you can enforce this in a set method.

&.

3. A class implements methods with the concrete code for those methods. It can contain abstract methods, but is designed to be an object. An abstract class however is designed to be extended to other concrete classes rather than be used as objects themselves. They can have concrete methods but these will be used in the classes that extend the abstract class. Interfaces however are all abstract methods, you must implement every signature in an interface.

Class:

```
class Car extends Vehicle{
      String name;
      void run(){
            ….
      }
}
```

Abstract Class:

```
abstract class Vehicle{
      int age;
      int getAge(){
            return age;
      }

      abstract void run();
}
```

Interface:

```
interface Vehicle{
      void run();
}
```

4. Dynamic polymorphism is when you can reference a class with a type that the class extends. E.g. say you had a class B that extends A, you could have ‘B v = new B()’ and ‘A v = new B()’.

5. You could have a ‘NinjaFace’ interface, that defines what the behaviour will be. You then make the class Ninja implement NinjaFace, implementing the methods. You then also make NinjaEmployee that extends Employee and implements NinjaFace. This means it gets the state from Employee and behaviour from NinjaFace.

Pretty sure this is wrong, because you still have to rewrite Ninja methods in NinjaEmployee, but not really sure how to do this!

6. Every object in the heap has a count of the references linked to it. When this becomes zero, there is no reference to the object so there is no way of getting to it anymore, i.e. it is useless. This means it is eligible for garbage collection.

I do not think ‘finalize’ is guaranteed to be called. For example, when System.exit() is called, finalizers are not called.

7. println is actually calling Person’s toString() method, so changing this in Person would change the output of the program.

e.g
if you had

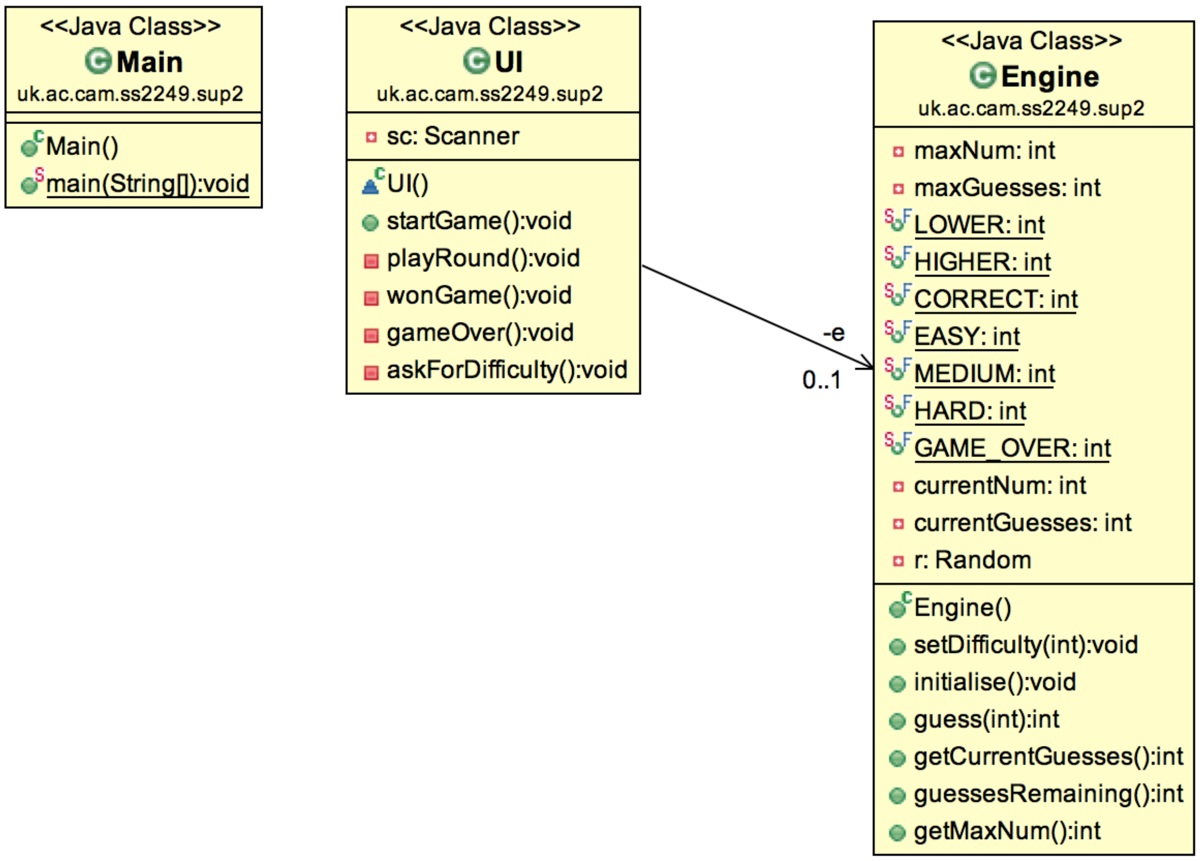
```
class Person{
	String firstName;
	String lastName;
	Person(String f, String l){
		firstName = f;
		lastName = l;
	}
	String toString(){
		return firstName + “ “ + lastName;
	}
}
```

then the program would print “Joe Bloggs”
but you could change toString() to be:

```
String toString(){
	return lastName + “ “ + firstName;
}
```

in which case it would print “Bloggs Joe”.

8.



Code is on github at: https://github.com/samsnyder/OOP-supervisions/tree/master/Sup2