OOP supervision 2

See the Github repo at https://github.com/MarcellPerger1/CST-1A-OOP-Supervision2 for the full code.

Section 4: Inheritance

4.1

Inheritance should only be used when the subclass has the same methods and attributes as the base class with some extra (i.e. the subclass extends the functionality). The classes should also follow the Liskov substitution principle: anywhere where the superclass can be used, the subclass should also be allowed. However, a `Vector3` cannot be used everywhere where a `Vector2` can be used (e.g. you can add a `Vector2` to a given `Vector2` but you can add a `Vector3` to that `Vector2`). The misunderstanding is that it's not enough for the data to be the same with some stuff added, the methods/behaviour also has to be the same but with some stuff added.

4.2

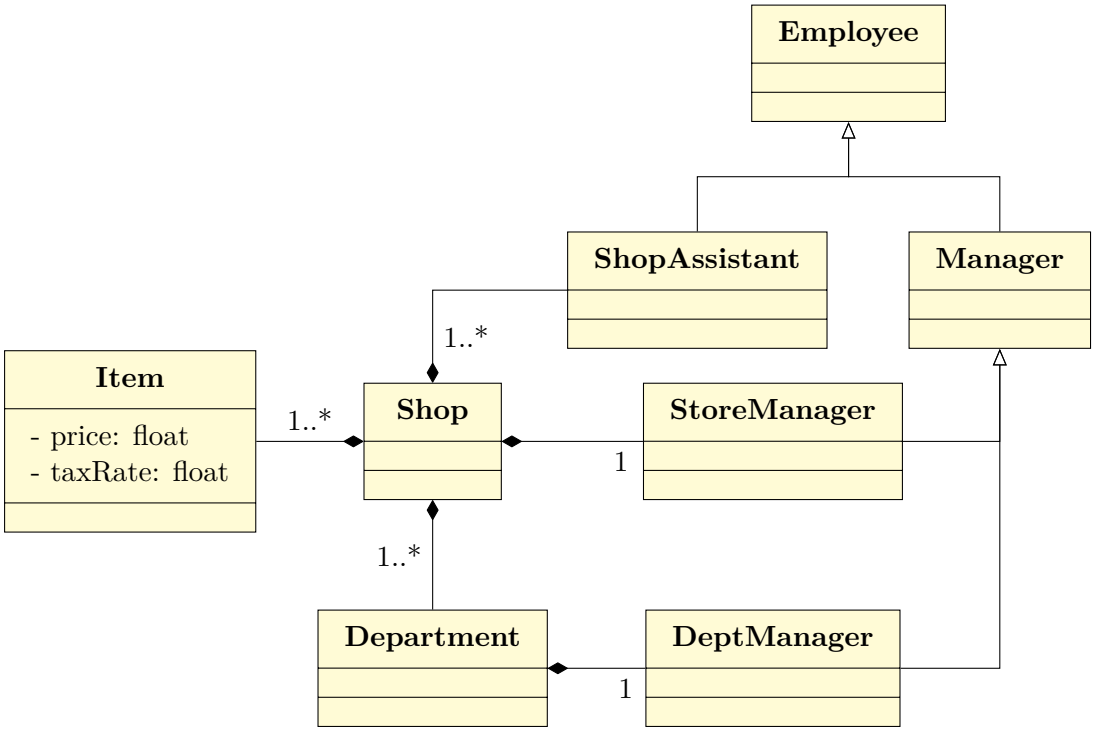
If you don't specify and access modifier, the default is package-private - it can be accessed by anything in its class and anything in the same package as it. See `Q4P2/**` for demo (commented-out code will make the compilation fail if it is uncommented)

4.3

See `Q4P3/NoConstructor/*.java` for classes without any constructor was added.
See `Q4P3/UnaryConstructor/*.java` for how it had to be modified when `A`'s constructor now has to take one argument (I had to add a constructor in all the subclasses to correctly call `A`'s constructor with the passed argument).

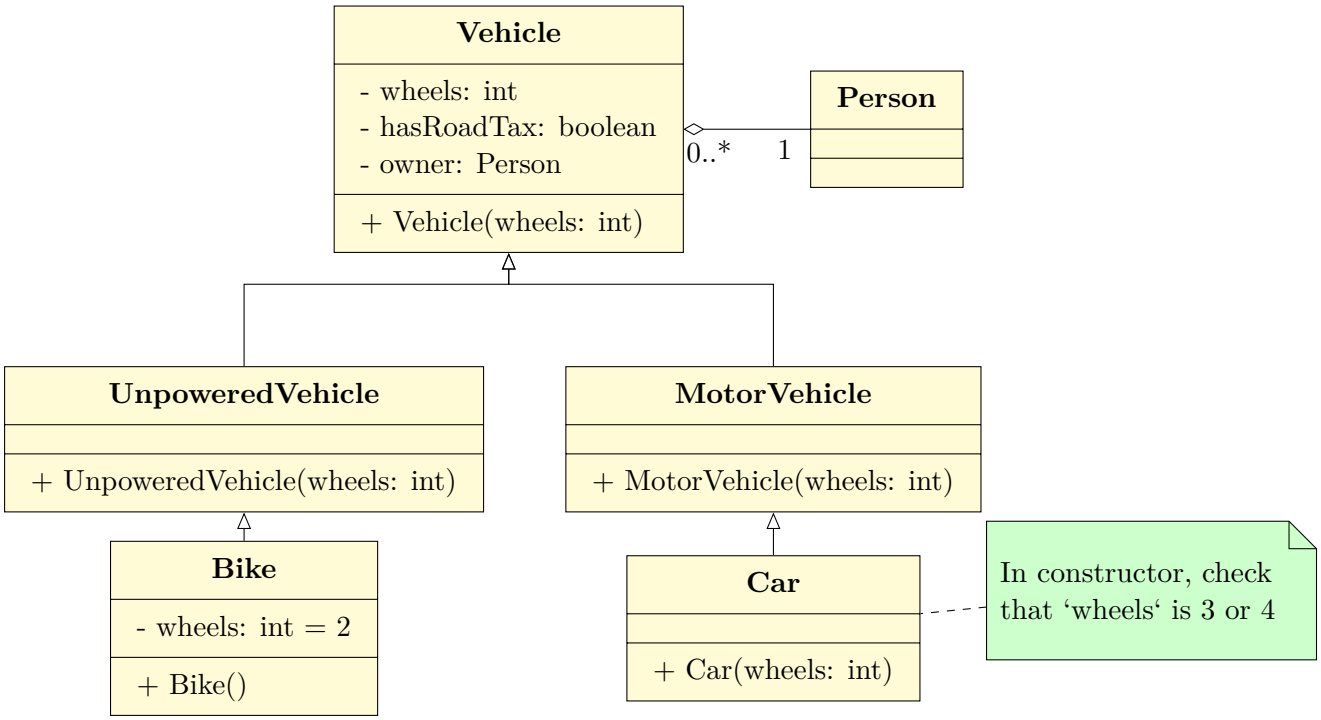
4.4

a



Getters, setters and any attributes for the relationships have been omitted for conciseness. Though this architecture means that an employee cannot be promoted without having to delete and recreate the object.

b



4.5

There will never be any runtime errors, only compile-time errors (as Java is smart enough to figure out at compile-time if you're not allowed to be accessing something)

	`public`	`protected`	unspecified	`private`
a	OK	OK	OK	Error
b	OK	OK	Error	Error
c	OK	OK	OK	Error
d	OK	Error	Error	Error

4.6

It will print `a.value = Super` and then `Sub.printValue: Sub`. This is because attributes exhibit static polymorphism so it will look at the declared type of `a` (`Super`) and access the attributes of that object. However, when methods exhibit dynamic polymorphism so it will look at the runtime type of `a` (`Sub`) and call `Sub.printValue()` and in that method, the compiler knows that the type of `this` will be `Sub` so will use `Sub.value`. This is because method dispatch is implemented using vtable, a hidden attribute that hold information about the actual type about the actual runtime type of an object and the method call uses this table to lookup the address of the correct `printValue` function to call.

4.7 - 4.8

See `ooplists/*.java`

Section 5: Polymorphism

5.1

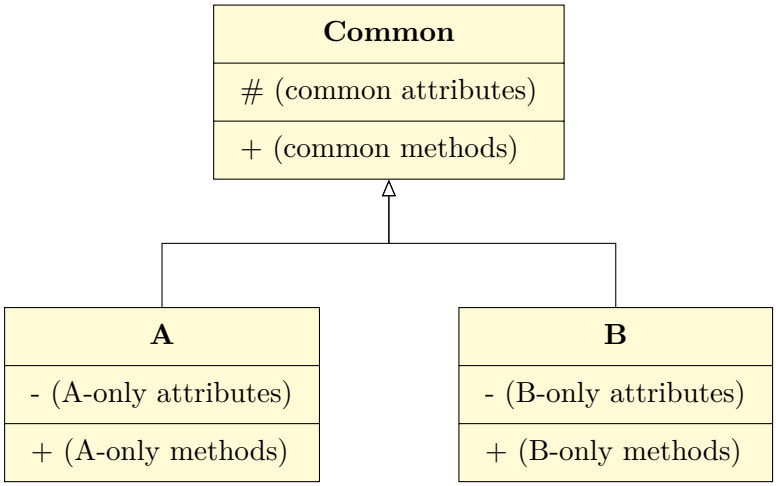
A class can have attributes and methods and can be instantiated, it's simply a blueprint fo an object. An abstract class is the same but can have unimplemented abstract methods that subclasses need to implement (so it can't instantiated directly, it has to be instantiated through a a subclass that implements the required abstract methods). An interface cannot have any attributes, it can only have method signatures that classes implementing that interface must provide (though it can provide default implementations for these methods in more modern versions of Java).

5.2

Dynamic polymorphism is when the method to call is determined by the actual runtime type of an object, rather than te declared type. This means that we can easily call the same method on each item in a list of items of the base class and then those calls will be dispatched to the correct method (potentially on the child class) based on the actual runtime type of each object.
See `Q5P2/*.java` for example.

5.3

This is not a good idea. This is because it means that any classes using this feature will be breaking the Liskov substitution principle as the subclasses cannot be used where the parent class can (as they won't have all of the methods!). Additionally, it could create a problem where a method is inherited but the fields it relies on are not, rendering the method broken on the subclass. And knowing which fields a method depends open completely breaks encapsulation! If the programmer wants a subset o the features of the class, what they should do is create a third class with the common features that they want on both classes and have both of the classes individually inherit the common functionality from that class. For example,



5.4

Before adding `NSStudent`: see `Q5P4OnlyCS/*.java`. After adding `NSStudent`: see `Q5P4WithNS/*.java`. The only thing that needed changing was that `NSStudent` needed adding (and the logic to generate the test list was also changed to add some NST students, but that wouldn't be there in real code as it would fetch them from a database).

5.5

See `QP55/*.java`.