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

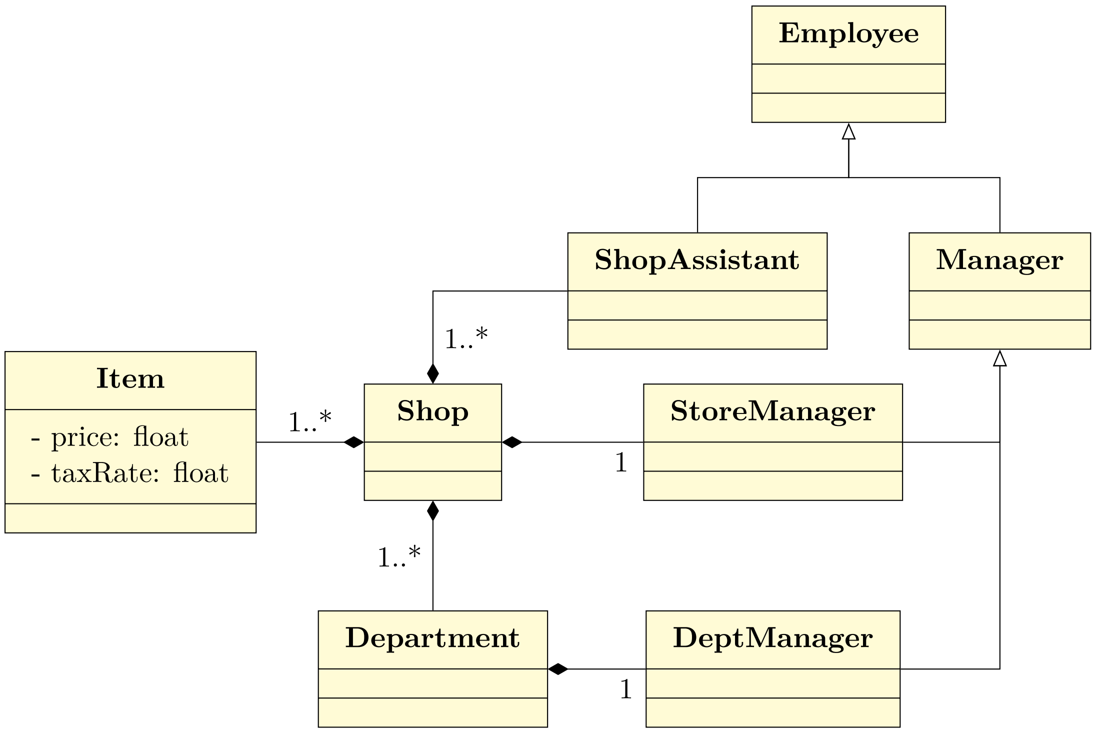
If you don't specify and access modifier, the default is package-private - it can be accessed by anything in its class and anything in the same package as it. See `Q4P2/**` for demo (commented-out code will make the compilation fail if it is uncommented)

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).

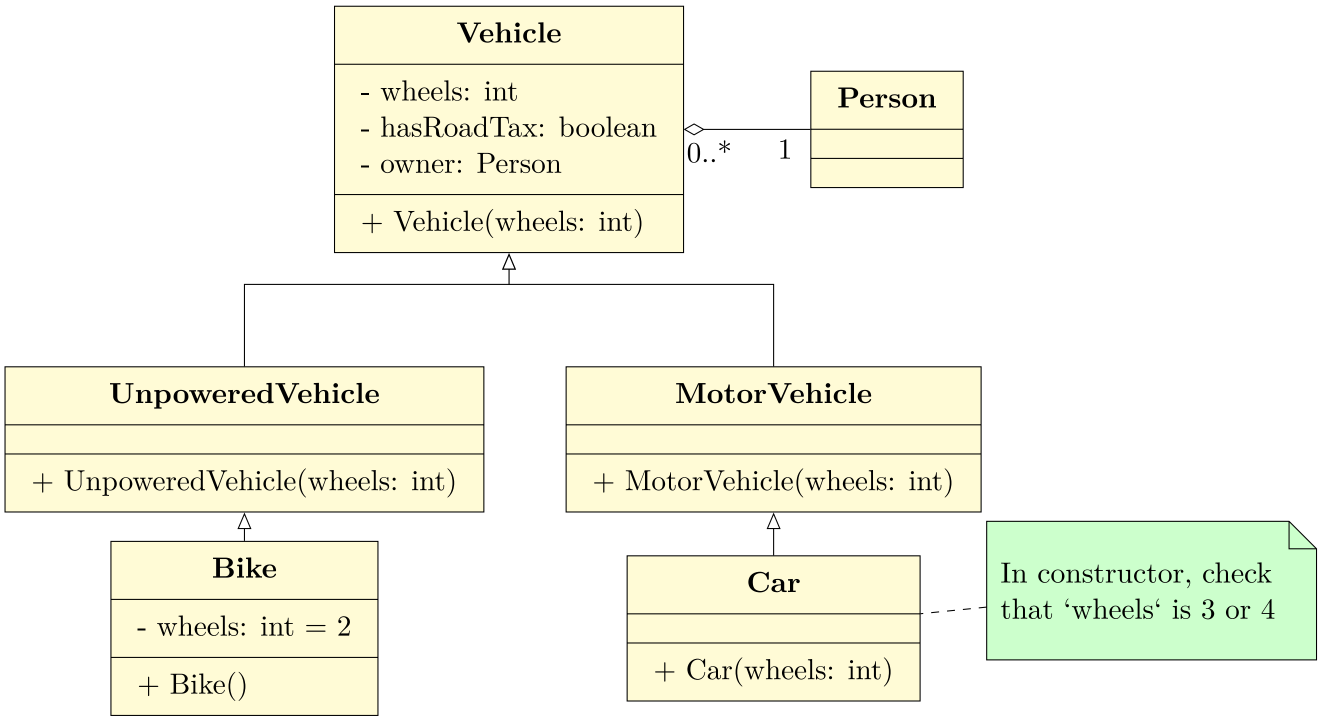
4.4

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

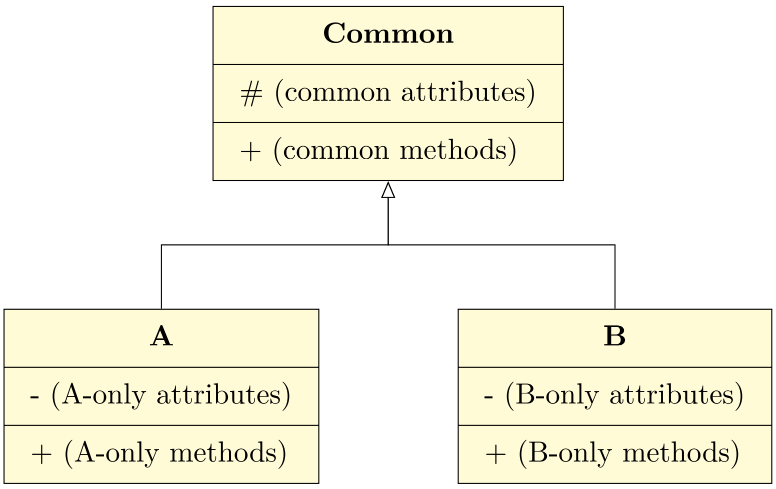
A class can have attributes and methods and can be instantiated, it's simply a blueprint fo an object. An abstract class is the same but can have unimplemented abstract methods that subclasses need to implement (so it can't instantiated directly, it has to be instantiated through a a subclass that implements 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`.

5.6

(a)
Add item: $O(1)$ (amortised),
Get nth item: $O(1)$,
Contains: $O(n)$,
Remove (given the value to remove): $O(n)$

Though there are many possible 'list methods' as the term is very vague and there is no one single definition (e.g. is inserting an item at a specific point a 'list method'? Is replacing an existing item a 'list method'?)

(b) - (c) See `ooplists2/*.java`.

(d) When we have to double the array (an $O(n)$ operation), it will have added $O(n)$ items since the previous expansion so the $O(n)$ expansion cost can be spread ovr the previous $O(n)$ additions to give n additions a time complexity of $O(n)$ so the amortized time complexity is $O(1)$ for each insertion.

5.7

(Skipped due to time constraints.)

5.8

See `Q5P8/*.java`

Section 6: Object lifecycle, Garbage Collection and Copying Objects

6.1

(a) This approach is quite fast and doesn't use much extra memory. However, it has a problem: the memory will get more and more fragmented over time, eventually reaching the point where it's impossible to allocate a large contiguous chunk of memory (e.g. for a large array). Another disadvantage is that it has to iterate over the objects twice: once to mark them and again to add the chunks containing the deleted ones to the free list.

(b) This approach solves the issue of memory fragmentation but it can get slow as a lot of objects will be moved during the compact stage (a lot of memory read/write operations). It still iterates through the objects twice: once to mark objects and then again to either delete them or to move them to be adjacent to the previous surviving one, at the start of the memory.

(c) This approach also doesn't have the problem of memory fragmentation but once again, a lot of objects must be copied. This one requires quite a lot of extra memory (twice as much as the other ones) but it only has to loop through the objects once, as it can, on marking an object, move it to the other region (though updating *all* the references sounds like it might be a bit cumbersome and/or slow).

6.2

Making classes immutable means that there is a lot of short-lived objects (as we have to create a new one every time we change something) and Java can collect these very quickly as they are still in the Eden generation by the time they're collected so Java can return that memory to the OS as Eden is garbage-collected quite frequently. Having mutable classes, on the other hand, means that when they can finally be freed, it will take Java a long time to find free their memory as they will have moved quite far through the generations.

6.3

Which one? I'll just the original one. See `CloneableOOPLinkedList.java`.

6.4

Marker interfaces have no methods that need to be implemented. They are used to mark that a specific type follow certain constraints or supports a specific thing where this isn't/can't be expressed as a method that needs to be implemented. A non-Java example is that Rust has an `Eq` trait (its version of an interface) that marks that equality is well-defined on that type (e.g. `a == a`). An example where `Eq` would not be implemented is floating point types as `NaN != NaN`.

6.5

If `SomeOtherClass` has some data that must be copied, then that data will not be copied, it will simply create a new instance of with a new set of attributes present on `SomeOtherClass` (won't be the same value). See `Q6P5/*.java` for example.

6.6

(a)-(b) See `MyClass.java`.

(c) Because a copy constructor doesn't do dynamic polymorphism (without reflection) so have some unknown subclass of `MyClass` (e.g. from a list of `MyClass` objects), we can't invoke the correct copy constructor to copy it to be the same class as the original.

(d)

If the class is `final` (i.e. can't have any subclasses), then we don't need to worry about the problem of polymorphically copying arbitrary subclasses (as there are no subclasses). This means that the copy constructor has no downsides there.

6.7 Because after calling `super.clone`, we have a `CloneTest` object with `mData` set as a reference to the same array and we would want to change `mData` to be a clone of itself (this is the deep part the clones all the attributes), but we simply cannot modify `mData` as it is `final`.

Section 7: Collections, Comparison

7.1

`Vector` is just an old (usually considered obsolete) thead-safe version of `ArrayList` that locks around every operation (generally you want to lock around a group of operations so this isn't the most helpful). `ArrayList` and `LinkedList` both implement the `List` interface, `LinkedList` uses a (doubly) linked list in its implementation (so adding/removing from the ends is fast but accessing elements in the middle is slow). On the other hand, `ArrayList` uses an array in its implementation (wow what a surprise!) so accessing items in the middle is fast but removing items from anywhere but the end is slow. `TreeSet` is a completely different thing entirely - it implements a set where the elements are stored in a sorted order and duplicates are not allowed (it uses a binary search tree to provide logarithmic time access to all elements).

7.2

See `Vector3.java`

7.3

See `NamesAndPercentages.java`

7.4

The list is so small that it won't make a very much difference and I would spend the time thinking about this to optimise other parts of the application instead ("Premature optimization is the root of all evil"). However, `LinkedList` would be more performant as it has $O(1)$ insertion and removal on both ends so is suitable to implement the queue structure required, while a `ArrayList` has $O(n)$ insertion/deletion at the front so the implementation will end up being $O(n)$ (well, unless you implement your own circular queue but that would be better done in an array).

7.5

String literals in Java are interned (cached) so the two occurrences of `"Hi"` refer to the same `String` object in memory so the address of `s3` and `s4` is the same (and `==` compares the addresses of the strings so the second `print` is `true`). However, in the first one, each `new String` (by definition) create a new object with its own memory address so `s1` and `s2` point to different memory locations so the first `print` is `true`.

7.6

See `Q7P6/*.java`

7.7

(Skipped due to time constraints.)