OOP supervision 1

See the Github repo at https://github.com/MarcellPerger1/CST-1A-OOP-Supervision1 for the full code.

Section 1: Intro to Java

1.1

See `HelloWorld.java`.

1.2

A typical functional language mainly uses recursion to implement functionality and most functions are pure, have no side effects and result in the same return value each time they're called. You could describe functional languages as declarative (i.e. describing what you want to happen to the data rather than describing the steps to get to the output)

1.3

A primitive is a value type (i.e. it refers to an actual value rather than the memory address of the value). Primitives include `boolean`, `byte`, `short`, `char` (which is 16 bits because Java's designers though UTF-16 is such a great idea), `int`, `long`, `float`, and `double`. No unsigned types apart from `char` (because Java...). For example, `d` and `f` in the code.
A reference is when the variable represents the memory address of the value stored (and not directly the value). This includes arrays and objects. So `i`, `l`, `k`, `t`, and `c` are all references (though `t` and `c` are both `null`).
An object is an instance of a class with its own set of data stored in its attributes. Objects in the code are `l`, `k` (as `Double` is a class wrapping the primitive `double`), `t`, and `c`. A class is a blueprint for objects, defining their attributes (data/state stored) and methods (behaviour/functionality). The classes are `LinkedList`, `Double`, `Tree` and `Computer`.

1.4

Adhering to a language's standard naming conventions is important because it means that programmers unfamiliar with the project can more quickly understand its workings and it reduces the amount of 'surprises' in the code so reduces the likelihood of bugs occurring. It also helps keep the look of all the code in the language similar so when reviewing code, experienced developers can more quickly spot problematic patterns that they've seen before.

1.5

(a) See `MathUtil.java`

(b) Because functions cannot be overloaded on return type only as that would mean that Java is unable to deduce which overload is to be used (well, unless it looks at the expected return type but it's clearly not smart enough to do that).

1.6

See `TailCallOptimisationCheck.java` - it seems to overflow the stack after a depth of 63000 so it doesn't perform tail call optimisation. This is probably because Java has APIs for accessing frames on the stack so it cannot simply replace the current stack frame when the function performs a tail call.

1.7

See `LowestCommonBit.java`.

1.8

See `Question1P8.java`

1.9

`public void Test()` is not a constructor, it's a method with name `Test` so it's never called. Therefore, when a new `Test` object is created, only `x = 0` is executed to `0` will be printed.

Section 2: Class Design and Encapsulation

2.1

Private state with public getters and setters provides encapsulation and this means that the state can only be changed through those getters and setters, so they can perform validation on the data being passed to ensure that the state will always be valid and there aren't any weird bug in unrelated parts of the coe arising from the state being set to an invalid value (this makes debugging easier). Additionally, it allows changing the underlying internal representation of the state in a backwards-compatible way (e.g. you can extract out parts of a class's state into another class - composition) as the attributes can't be accessed by outside code (well, except though reflection but that's a bit of a hack) so if you want to change the attributes, you can simply modify the getters and setters to add extra code to use those new attributes instead of the old ones.

2.2

The advantages of this private-by-convention approach is that it allows easier modification of library functions while still making explicit the disclaimer that "this attribute is intended for internal use".. I have found that this modification is needed quite often as library authors can't think of everything and often leave bugs in their code. It would be a massive pain to fork the project and figure out their build process so possibility of making a one-line change to the internals to fix a bug is good if you your code to 'just work'. It is also nice having some indication of whether an attribute being used in a method is build or private.
However, some people would argue that this slightly lessens encapsulation. I would say that Java also allows accessing private attributes via reflection but this has a much more convoluted syntax (as it requires using the standard library's `java.lang.reflect.*` classes). This means that in cases where accessing private attributes would be useful

(tests, fixing a library), it is a lot more convoluted to do so and the code will be a lot less clear.

2.3

(a) See `Vector2D.java`

(b) The `add`, `divide`, and `normalize` methods need to be changed to return a new vector rather that modifying the current vector. Also, setters need to be removed.

(c)

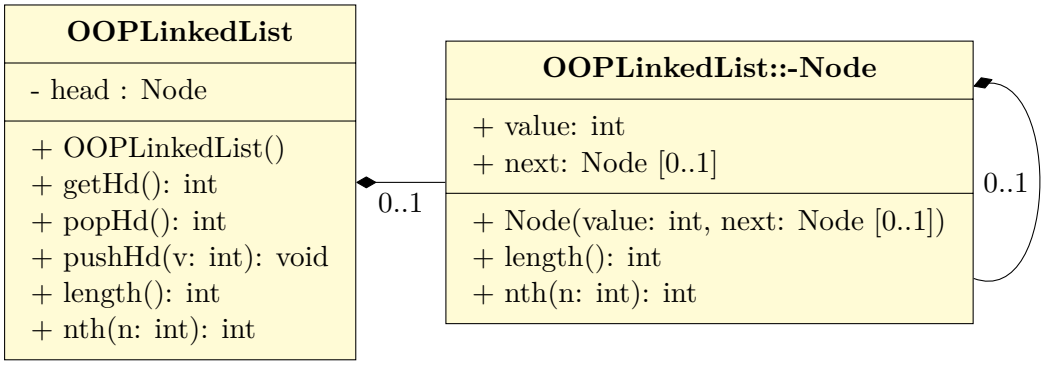
1. The first one is suitable for mutable vectors only as it doesn't return anything (so the method changes the values stored) so it would be useless for the pure immutable vector add as it doesn't return a value.
2. Suitable for immutable vectors as it simply adds `this` and `v` and returns the new value. Also suitable for mutable vectors as it can change `this` in place and then return `this` so that it can be used with method chaining.
3. This version would either add 3 vectors or would ignore one of the vectors (probably `this`). Ignoring one of the passed vectors isn't sensible so I assume it adds 3 vectors and apart from adding an extra vector, it's exactly the same as 2 (though a method to add 3 vectors seems rather useless as it's the same as calling the 2-vector method twice)
4. Suitable for the immutable version as it means that vectors can be added like `Vector2D.add(v1, v2)` and it returns the result. Probably not suitable for the mutable version as it's less clear here which one to modify and the syntax makes it feel like it shouldn't modify any arguments.

(d) Personally, I think the expectation for a `Vector2` class is that it's immutable as it's very close to the mathematical concept of a vector and has most of the operations that you can do on real numbers and `double`s are immutable so I'd expect a `Vector2` to be immutable. So I think a note in it's documentation is enough to convey it to unsure users. On the other hand, I would definitely indicate that if a `Vector2` is mutable (as I would expect them to be immutable, see above). I would probably indicate this by changing the class name to `MutVector2` or `Vector2Mut` so that even users reading unfamiliar code that contains this class will instantly understand that this vector is mutable.

2.4

(a) See `OOPLinkedList.java`

(b)



2.5

It violates SRP because the class has many unrelated methods to do with different parts of the app. For example, fetch the data from the database and parsing the XML data are 2 different parts of the application so should be in 2 different classes (as at the moment this class has 3 responsibilities: fetch data from the database, parsing XML data, and printing to the console). See `Q2P5/*.java`.

2.6

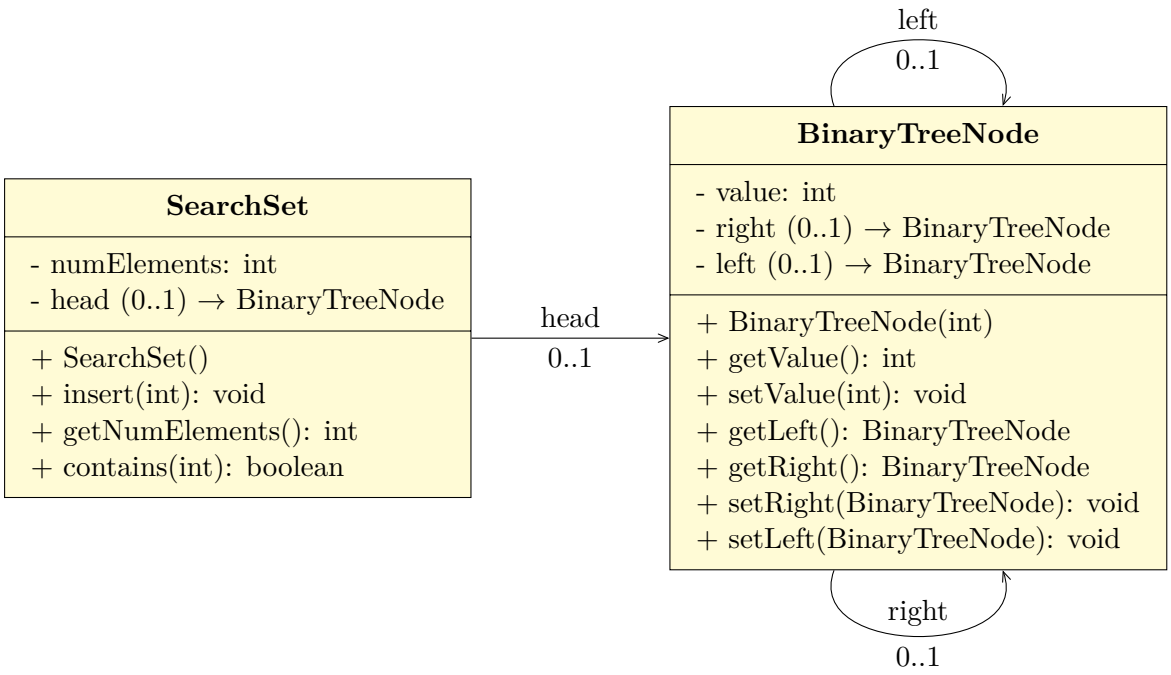
(a) See `Q5P6a/StudentRecord.java`

(b) See `Q5P6b/StudentRecord.java`

(c) If `hashCode()` and `equals()` is simply based on the reference, then seemingly-equivalent objects will not be treated as equal (i.e. `new Student(a, b, c) != new Student(a, b, c)`) and this is very undesirable as it also breaks `HashMap` and `HashSet` lookups as objects will only be equal to each other if they are physically the same reference so the hash tables will become useless for lookups.

2.7

(a)



(b) See `Q2P7/{BinaryTreeNode,SearchSet}.java`.

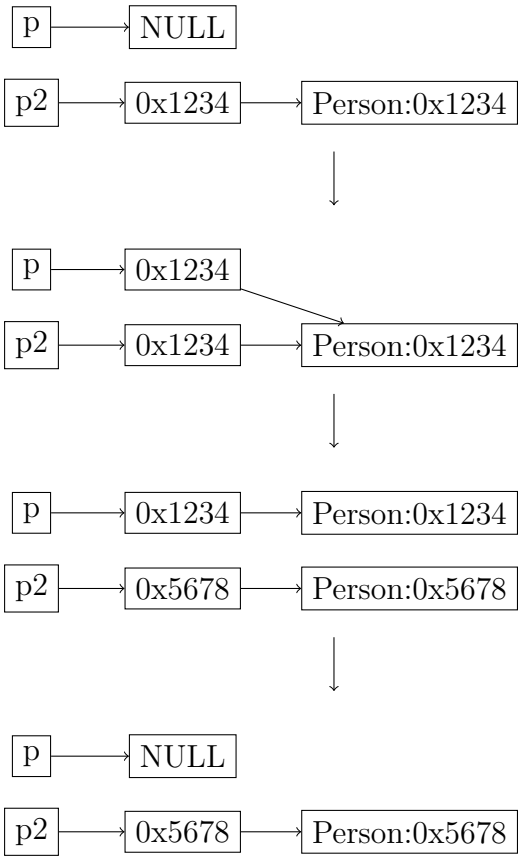
(c) See `Q2P7/FunctionalArray.java`.

Section 3: Points, References and Memory

3.1

The advantage of using references oover pointers is that a reference is either null or a useful value and this means that the code can detect if it's a null value and raise a `NullPointerException` that can be caught and (relatively) elegantly handled. However, pointers can point to null, a useful value, or just some other block of memory. This means that it is impossible to distinguish between pointers to a valid object and pointers to another block of memory (potentially owned by another process or maybe already freed and being used for something else). This means that the code can't elegantly raise an exception if it receives an invalid pointer so the OS will just kill the process (via `SIGSEGV`) so the process will shut down without doing any required cleanup. Or, even worse, that pointer could be pointing to some memory that has been freed and now contains sensitive information like passwords and this could cause security vulnerabilities if a pointer is accidentally used after the memory is freed.

3.2



3.3

For the first `println`, the `add(int, int, int, int)` is called. All of the arguments are passed by value and as they are primitives, this means that the integers themselves are passed in. This means that the modifications in the function only affect the locals copies of the variables and don't propagate outside as the entire value has been copied. Therefore, `1 1` is printed.
For the second `println`, the `add(int[], int, int)` is called. A reference to the array is passed by value (as well as the two `int`s), i.e. the memory address of the array is passed by value so the `xy` in the function still refers to the same bit of memory as `xypair` so when `xy`'s elements are updated, `xy`'s elements will also change (as they're literally the same array). Therefore, `2 2` will now be printed.

3.4

See `ValueRefDemo.java`.

3.5

Advantages:

- Consistent handling of primitives and non-primitives makes it so that the programmer doesn't need to think about whether something is a primitive or not.
- Lower memory consumption than just pass-by-value as no values are copied.

- Faster than pass-by-value for non-primitives as it doesn't have to copy possibly-huge data structures.
- Inner functions can propagate any value changes out, so multiple returns cld be implemented by passing in a variable that will be reassigned by the function (I think C# might have something like this?).
- Passing all values by reference allows something that Java doesn't have: passing primitive types directly into generics (i.e. `ArrayList<int>` works, you don't need to do `ArrayList<Integer>`)

Disadvantages:

- Higher memory usage than Java's hybrid method as passing primitives takes up at least 8 bytes (the size of an address) and all primitives have size at most 8 bytes (most are less 8 bytes though)
- Can be slower than the hybrid approach for primitives as an extra level of indirection is added (which might be quite slow if it isn't in the CPU's cache)
- Sometimes, you accidentally reassign a variable in a function and often, you don't want those changes propagating to the users of the function (it's actually quite rare that the arguments are completely reassigned)