OOP supervision 1

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.