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`