Exercise 4

Inheritance, and more Inheritance

1.

- The intended behaviour is that a `Stack` is first-in-first-out, while a `Queue` is last-in-first-out. Therefore, it is impossible that both the `pop` and `push` methods can have similar behaviours across the two classes, and so neither class can be a behavioural subtype of the other.
- Depending on the internal representation, either the `pop()` or the `push()` method (but not both) could be re-used, from one implementation to the other. For example, if one implements a `Queue` by pushing to the end of a linked list, and popping from the beginning, then a `Stack` could be implemented either by pushing on the beginning of the list and reusing the `pop()` method, or by reusing the `push()` method and popping from the end of the list. Furthermore, it's likely that the `isEmpty()`, `size()` and `reverse()` methods could all be reused.
- Any mechanism which allows code reuse without subtyping, e.g., private inheritance in C++. In principle, aggregation could be employed, but the "common class" would be rather strange (e.g., a list which could only grow, and only at one end). Traits might also provide a solution to this problem, but again, identifying a fragment of the implementation to abstract out might not be natural. One could argue that this kind of code reuse binds the implementations too closely together, when it might be that one or other class wants to evolve independently (e.g., given some other desired methods, we want to change the underlying implementation of one class in a way which isn't helpful for the other). However, the ability to reuse a large number of common methods seems tempting.

2.

- Code reuse is not going to be possible (at least for the primitive operations), since the two classes will use different internal representations of the data.
- So long as the internal representation (fields) cannot be observed, then they should ideally behave as subtypes, since ultimately all of the operations should produce the same answers. In particular, the difference in the implementations cannot be observed by `get()` calls. This seems intuitively to be correct also, since sparse matrices are a special case of matrices.
  However, unless the specifications of the methods are written abstractly, then it will be hard to technically justify behavioural subtyping (e.g., if the specification of `set()` in `Matrix` is written in terms of the array used to store the data, then the specification of `set()` in `SparseMatrix` will not be able to satisfy the requirements of behavioural subtyping).
- If we make them subtypes then we can nicely handle the appropriate implementations of the `add` and `multiply` methods in the various cases. On the other hand, a `SparseMatrix` object will inherit a useless copy of the fields used in `Matrix` – this means an overhead in memory and initialisation time (since by default the superclass constructor will still be called). This can also lead to subtle bugs (see next question).

- An interface (or abstract class) could alternatively be defined, which both classes implement (or subclass). This eliminates the redundant overlap between fields used in the two classes. However, if client code has already been written in terms of the class `Matrix` then adding the interface will not avoid any problems for this client code (this is a good reason to always provide interfaces rather than class definitions, to clients!).

3.

- In the case of the code

```
m.entries[i][j] = 4;
if(m.get(i,j)!= 4) { // crash }
```

  if `m` turns out to reference a `SparseMatrix` object, then because the method call to `get()` will be dynamically dispatched, it will refer to the fields used for the internal representation of `SparseMatrix`, and not the `entries` array. Therefore, there is no reason to expect the if-condition to be true. Making the fields private avoids this problem arising in client code, but it can still occur in other methods of `Matrix` if there is a mixture of direct field accesses and (dynamically dispatched) method calls.
- Similarly to the previous part, if we retain any method implementations from the `Matrix` class then these are likely to refer to the fields used for internal representation of the superclass and not the subclass, which are unlikely to contain meaningful values.
- Any extra methods that we add to `Matrix` will suffer the same difficulty – because they will typically refer to the `entries` array, they will not operate correctly on `SparseMatrix` objects. The only exception is a method which is implemented entirely in terms of previously-defined methods (no field accesses).

4.

- The code will print B1 C1 C1 – the method definition is resolved in terms of the static type of the argument, but the dynamic type of the receiver. Note that this means that it is possible to have two aliases of the same object, and receive different results when passing them as parameter to a method of the same name (note however that, this is not really passing them to the same method – it is better to think of method overloads as definitions of two different methods in the class).