10.1

Give an example of how covariant arrays in Java can create runtime errors.

```
public class main {
    public static void main(String[] args) {
        Object[] objects = new String[] {"a", "b", "c", "d"};
        objects[1] = 3;
    }
}
```

Covariant arrays mean that the type assigned to arrays at runtime can be a subtype of the array's type at compilation time. This allows some flexibility with code however can lead to runtime errors such as the one above.

At compilation time: the compiler sees that the Object array objects is being set to a String array. String is a subtype of Object. So this is fine. On the next line, 3 is autoboxed to an Integer. Integer is a subtype of Object, so Integer can be stored in an Object array. And the compiler is happy. So at compilation time, the code is correct and will compile.

However at runtime, the type of the objects array is String. Integer is not a subtype of String and so 3 cannot be stored in the array objects (despite 3 being a perfectly valid thing to store in an Object array). This causes an ArrayStoreException – a runtime error.

11.1

Explain the difference between the State pattern and the Strategy pattern.

The State Pattern allows the behavior of an object to change dependent on the state of the object. This is done by the use of instance classes (which should implement an interface to guarantee that they all have the methods which the outer class will call). Each of the instance classes has the same methods (and return types and parameters) but have different implementations. Typically, there is only one private variable which is an object of an instance inner class which determines the performance of the whole object. Since the inner classes are instance classes, they can access the outer classes scope and hence changing "state" usually only requires a single change to a private variable. This makes changing state very easy – especially if states change in a well-defined order. It keeps control of the state of the object and how methods operate with the computer rather than with the user – the state is hidden from the user. In addition to this, execution time is constant with respect to the number of states and it is often very simple to add more states.

The Strategy Pattern allows you to pass arguments to methods which determine how exactly they execute. A very simple example of this would be to pass an argument to a dfs indicating the maximum depth it should search to. This is another way of making the same method behave differently. However, is different to the State Pattern in that it gives the user full control over how the method should execute (since they pass the parameters). In many situations it is good to allow the user to decide how the method should execute, however this can lead to situations where the user passes invalid arguments or arguments which do not perform as they are intended to (or are expected to) or situations where the user does not necessarily know how something should execute.

11.2

In lectures the examples for the State pattern used academic rank. Explain the problems with the first solution of using direct inheritance of Lecturer and Professor from Academic rather than the State pattern.

Promoting someone from Lecturer to Professor using the subclass method is difficult. To

do this you need to write methods which extract all the data from the lecturer object, then create a new Professor with all the data put into it. However: this is a new object all existing pointers to the old object still exist (for example if there was an array of lecturers on a contacts list it would still contain pointers to the old Lecturer object). There are several possible solutions to this – neither of which are good.

1. Accept that there will be inconsistencies in the data but that they are non-critical. Return the Professor object and assume that the user manually puts it in the appropriate places for any critical operations.
   However, not only will this cause inconsistencies: it will also cause memory leakage – after many promotions the number of unused Lecturer objects will build up. The creation of a new Professor object could also be expensive. If there was some object which you could do more with as a Lecturer than as a Professor, you may have to make a copy of it when creating the Professor object to ensure that it is not changed according to different constraints.

2. When you promote someone, object add a pointer to the new Professor object and change all other fields in the Lecturer object to Null. This deals with the inconsistency issue (since the user is now forced to handle the case that the person has been promoted or face runtime exceptions) and the complexity issue – since all you must do to create the Professor is change pointers. However: this causes runtime exceptions and doesn't resolve the memory leakage problem. Code could be written when accessing Lecturers to check whether they had a pointer a "promoted" Professor object – and if so either replace the Lecturer object with the professor object or remove the lecturer object from the data structure (as appropriate). This solution might work for a small system. However, if each Lecturer had 100 pointers, then they might never all be found. This would mean the garbage collecter could never collect the Lecturer object and so there would still be memory leakage.
   A particularly devoted programmer could perform a bfs of all pointers and use a hash map to change all the obsolete ones to their newer counterparts This would solve the memory leakage problem. However this could not be done in Java and would be **very** difficult to implement and innefficient.

With the state pattern, since the object's behavior changes based on state: a promotion would not need a new object to be created. Rather, the existing object would change a private variable from the LecturerRank to the ProfessorRank. No pointers to objects outside the class would have to be changed. When calling methods in this outer class it calls the inner class's implementation of that method. Since the object itself is the same object, there is no risk of inconsistency, no high overhead to creating the new Professor and no bfs of all pointers.

The Academic class below shows how the promote method can be implemented:

```
public class Academic {

    private AcademicRank rank = new LecturerRank();

    private static interface AcademicRank{
        AcademicRank promote();
    }

    private static class LecturerRank implements AcademicRank{
        @Override
        public AcademicRank promote(){
            return new ProfessorRank();
        }
    }

    private static class ProfessorRank implements AcademicRank{
```

```
        @Override
        public AcademicRank promote(){
            return new LecturerRank();
        }
    }

    public void promote(){
        rank = rank.promote();
    }
}
```