Exercise 4

Inheritance, and more Inheritance

1.

- 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 (see questions 3 and 4). 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!).

2.

- 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).

3.

- i. In the `Matrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof SparseMatrix) {
    // semi-efficient implementation
  } else {
    // old implementation
  }
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof SparseMatrix) {
    // efficient implementation
  } else {
    // semi-efficient implementation
  }
}
```

- ii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  return m.addMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // old implementation
}
Matrix addSparseMatrix(SparseMatrix m) {
  // semi-efficient implementation
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  return m.addSparseMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix addSparseMatrix(SparseMatrix m) {
  // efficient implementation
}
```

- iii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  // old implementation
}
Matrix add(SparseMatrix m) {
  // semi-efficient implementation
}
```

In the `SparseMatrix` class:

```
SparseMatrix add(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix add(SparseMatrix m) {
  // efficient implementation
}
```

- The last approach is probably the simplest and most intuitive.
- For the first and last approaches, all that would be lost is the potential extra efficiency when adding a `SparseMatrix` to a `Matrix`. However, for the second approach (Visitor pattern) it's essential to be able to add the extra methods to the superclass, in order to make the second dispatch possible. Whatever the approach to binary methods, if the `add` method in `Matrix` had been written using direct field accesses on its argument (rather than calls to `get()`) then it will need to be rewritten anyway when the subclass is added.

4.

- The receiver can be immediately returned from such a call. We could overload:

```
Matrix add(ZeroMatrix m) {
  return this;
}
```

However, in a language like Java, which does static dispatch re: argument types, this will not have the desired effect when a `ZeroMatrix` instance has a less specific static type.

-

i. In the `Matrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof ZeroMatrix) {
    return this;
  } else if(m instanceof SparseMatrix) {
    // semi-efficient implementation
  } else {
    // old implementation
  }
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  if(m instanceof ZeroMatrix) {
    return this;
  } else if(m instanceof SparseMatrix) {
    // efficient implementation
  } else {
    // semi-efficient implementation
  }
}
```

In the `ZeroMatrix` class:

```
Matrix add(Matrix m) {
  return m;
}
```

ii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  return m.addMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // old implementation
}
Matrix addSparseMatrix(SparseMatrix m) {
  return this.addMatrix(m);
}
```

In the `SparseMatrix` class:

```
Matrix add(Matrix m) {
  return m.addSparseMatrix(this);
}
Matrix addMatrix(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix addSparseMatrix(SparseMatrix m) {
  // efficient implementation
}
```

In the `ZeroMatrix` class:

```
Matrix add(Matrix m) {
  return m;
}
Matrix addMatrix(Matrix m) {
  return m;
}
SparseMatrix addSparseMatrix(SparseMatrix m) {
  return m;
}
```

iii. In the `Matrix` class:

```
Matrix add(Matrix m) {
  // old implementation
}
Matrix add(SparseMatrix m) {
  // semi-efficient implementation
}
Matrix add(ZeroMatrix m) {
  return this;
}
```

In the `SparseMatrix` class:

```
SparseMatrix add(Matrix m) {
  // semi-efficient implementation
}
SparseMatrix add(SparseMatrix m) {
  // efficient implementation
}
```

```
SparseMatrix add(ZeroMatrix m) {
  return this;
}
```

In the `ZeroMatrix` class:

```
Matrix add(Matrix m) {
  return m;
}
SparseMatrix add(SparseMatrix m) {
  return m;
}
ZeroMatrix add(ZeroMatrix m) {
  return this;
}
```

- We are forced to require specific implementations for many more cases than we originally thought of, in order to ensure that there is always a most-specific fit for any pair of receiver and argument type. The definitions in **bold** above are the extra ones added for this reason.
- The extra requirement seems somewhat annoying for this example, particularly since in all cases where an ambiguity would otherwise arise, the choice of implementation does not intuitively affect the actual result. For example, if we erased the **bold** definitions, then for a `ZeroMatrix` receiver and `ZeroMatrix` argument we would have to choose between the `ZeroMatrix-Matrix` implementation, and the `Matrix-ZeroMatrix` implementation. However, both of these return the non-zero matrix. On the other hand, consider the case when we have a `SparseMatrix` receiver and a `ZeroMatrix` argument. In this case, we have to choose between the `Matrix-ZeroMatrix` implementation and the `SparseMatrix-SparseMatrix` implementation. But it is not completely obvious that the latter would work correctly for a `ZeroMatrix` argument, depending on its implementation (how much it depended on the appropriate fields from `SparseMatrix` being used/initialised as expected).
- In the light of this, there seems to be less to choose between the last two approaches. One further observation though is that in the case of multiple dispatch, although the superclass has been modified, it is only for an improvement in efficiency – if it were essential that the superclass were unchanged then the `Matrix-ZeroMatrix` implementation could be omitted from the code above, and everything would work out fine. The other approaches depend upon being able to modify the superclass, which may not always be acceptable in practice.
- The second approach (Visitor pattern) doesn't require any changes to the existing classes. The other two approaches would have to relinquish the extra efficiency possible when the argument is a zero matrix (but could still be efficient when the receiver was a zero matrix).