Exercise 5

Multiple Inheritance and Traits

1) The following C++ code breaks the invariant:

```
class B : public Person
{ public: B (string n) : Person (n) {} };
class C : public Person
{ public: C (string n) : Person (n) {} };
class D : public B, public C
{ public: D (string n) : B(n), C(n) {} };


void marryMyself ()
{
     D me ("Me");
     B *b = &me;
     C *c = &me;
     b->marry (c);
     if (b->getSpouse ()) cout << b->getSpouse ()->getName ();
}
```

The object `me` contains an object of class `B` and an object of class `C`. The addresses of these objects are different and they are obtained using the assignments to `b` and `c` respectively. During the call `b->marry (c)`, the condition `p == this` compares these two addresses and finds them not equal.

2) Here are the three requested classes:

```
class Queue
{
    int[] contents;
    int size;

public:
    Queue() { contents = new int[100]; size = 0; }
    void enqueue(int x) {…}
    int dequeue() {…}
    int getSize() { return size; }
};

class SumQueue : virtual public Queue
{
    int sum;

public:
    SumQueue() : Queue() { sum = 0; }
```

```
    void enqueue(int x)
    {
          sum+=x;
          Queue::enqueue(x);
    }

    int dequeue()
    {
          int r = Queue::dequeue();
          sum-=r;
          return r;
    }

    int getSum() { return sum; }
};

class ProductQueue : virtual public Queue {…};

class SuperQueue : public ProductQueue, SumQueue
{
public:
    SuperQueue()
          : public Queue(), ProductQueue(), SumQueue() {}

    void enqueue(int x)
    {
          SizeQueue::enqueue(x);
          SumQueue::enqueue(x);
    }

    int dequeue()
    {
          int r = SizeQueue::dequeue();
          SumQueue::dequeue();
          return r;
    }
};
```

One obvious problem is that the `enqueue` and `dequeue` methods of the superclass are called twice. An item is enqueued and dequeued twice. Interestingly, this behaves exactly like a queue, but the capacity is half of the capacity of the original and the `getSize` method reports the correct size multiplied by 2.

We can use traits and linearization to ensure that the enqueue/dequeue methods are called only once. Here is the relevant Scala code:

```
class Queue
{
    …
    def enqueue(x:int) = {…}
    def dequeue():int = {…}
}
```

```
trait Sum extends Queue
{
    var sum:int = 0
    override def enqueue(x:int) =
          { sum+=x; super.enqueue(x) }
    override def dequeue():int =
          { var x = super.dequeue; sum=sum-x; x }
}

trait Prod extends Queue
{
    var count:int = 1
    override def enqueue(x:int) =
          { prod*=x; super.enqueue(x) }
    override def dequeue():int =
          { var x = super.dequeue; prod=prod/x; x }
          // side remark: this assumes no zeros in the queue!
}
```

Now, an object of `Queue with Sum with Prod` has both functionalities, but calls each underlying enqueue/dequeue method only once. The problems of the multiple inheritance solution do not appear here.

3) Let X',Y' be the two base classes from which we derive X and Y by mixing in traits. Let A be the set of all traits mixed in to the first class and B the set of all traits mixed in to the second class. The rule is as follows:

X<:Y if and only if X'<:Y' and A⊇B.

Note: The above rule applies in our example, but it is not a general rule for subtyping in the presence of traits.

Notice that `D with T with U` and `D with U with T` are equivalent types (subtypes of each other)! Since, as we saw, they can describe different behaviour, this causes a subtle problem for behavioral subtyping!

4)

- Object `a` behaves like a normal cell. Object `b` is also a cell, but it increases the stored value by 1. The interesting difference is between `c` and `d`. They are both cells. They have mixed in exactly the same traits. However, calling `set(i)` has a different effect on them: it stores `2i+1` to the first one and `2(i+1)` to the second one.
- Trait `Doubling` will not get mixed in twice, as perhaps the programmer would expect. Scala rejects this statically.

  The problem can be bypassed in an ugly way, by creating a new trait `Doubling2` that behaves exactly like `Doubling` and then introducing `e = new Cell with Doubling with Doubling2`. Here is our first try:

```
trait Doubling2 extends Doubling
val e = new Cell with Doubling with Doubling2
```

The code passes through, but dynamically `e` behaves as if it were a `Cell with Doubling`. Scala lets the code go through, because `Doubling2` may introduce new functionality, but refuses to include `Doubling` twice in the linearization.

Our last try, the ugliest of all, but the one which will finally work, is to create a whole new trait from scratch, reusing nothing:

```
trait Doubling3 extends Cell
{
  override def set(i:int) = { super.set(2*i) }
}
val e = new Cell with Doubling with Doubling3
```

And now `e.set` quadruples its argument as expected.

- We can produce the problem using the traits provided, but here is a more interesting case. Consider the following code:

```
class C
{
    def m() = { println("m executing") }
}

trait Logging extends C
{
    val logFileName: String
    override def m() =
    {
          println("Logging to: " + logFileName)
          super.m()
    }
}

class C1 extends
    C with Logging
    { override val logFileName = "A" }
          // this class logs all calls to m
          // to a file named "A"
```

Suppose now that we give the client the classes `C`, `C1` and the trait `Logging`. The client has no knowledge that `C1` was created using `Logging`. The client wishes to log calls to `m` to a file called `"B"`. The client does this for both classes `C`, `C1`.

```
class C2 extends
    C with Logging
    { override val logFileName = "B" }

class C3 extends
    C1 with Logging
    { override val logFileName = "B" }

object LogEx1
{
    def main (args:Array[String]) =
    {
```

```
            val a = new C2
            val b = new C3
            a.m
            b.m
        }
    }
```

The call `a.m` works as it should: the method call is logged to file `"B"` only.

However, the call `b.m`, does not behave as it should: it logs the call to `m` only to file `"B"`, even though it is an instance of `C1`, which is supposed to log calls to `m` to file `"A"` too.

The problem is that, unbeknownst to the client, the trait `Logging` has been mixed in twice. This overrode its initial behaviour, interrupting the logging to `"A"`.

5)

- No. The dynamic type of `x` can be mixing in traits that break the invariant
- No! A parameter of type `Cell with Incrementing with Doubling` is allowed by Scala
- Consider the following example:

```
class C
{
        var x:int;
        def foo() = {} //ensures true
}
trait T1 extends C
{
        override foo() = { x=x+1 } //ensures x>old(x)
}
trait T2 extends C
{
        override foo() = { x=x-1 } //ensures x<old(x)
}
```

Both `C with T1` and `C with T2` are behavioral subtypes of `C`. But `C with T1 with T2` is not a subtype of `C with T1`.