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) The following code breaks the invariant:

```
class Girl : virtual public Person
{
public: Girl () : Person (true) {}
     // diamonds are a girl's best friend
};

class GirlProgrammer : public Girl, public Programmer
{
public: GirlProgrammer () :
     Person (true), Girl (), Programmer () {}
};

void oops ()
{
      GirlProgrammer gp;
}
```

Following the rules of C++ virtual inheritance, the call of the constructor `Person (true)` in class `GirlProgrammer` bypasses the corresponding call `Person (false)` in class `Programmer`, breaking the invariant.

3) A plausible reason is that we cannot resolve ambiguities, such as the following:

```
C c;
c.D::method ();  // which copy of D::method is called?
```

4) Here are the three requested classes:

```
class Queue
{
    int[] contents;

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

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 SizeQueue : virtual public Queue {…};

class BothFunctionalityQueues : public SizeQueue, SumQueue
{
public:
    BothFunctionalityQueues ()
          : public Queue (), SizeQueue (), SumQueue () {}
```

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

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

The 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 (the capacity is half of the capacity of the original), but in terms of efficiency it is not good.

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; return x }
}

trait Count extends Queue
{
    var count:int = 0
    override def enqueue (x:int) =
          { count+=1; super.enqueue (x) }
    override def dequeue ():int =
          { count-=1; return super.dequeue }
}
```

Now, an object of `Queue with Sum with Count` has both functionalities, but calls each underlying enqueue/dequeue method only once.

5) Here is an adaptation of Q.2 in Scala:

```
abstract class Person
{
      def likesDiamonds : boolean
}

trait Programmer extends Person
{
      override def likesDiamonds = false
}

trait Girl extends Person
{
      override def likesDiamonds = true
}
```

Here is what Scala thinks of our definitions now:

```
scala> var x = new Person with Programmer with Girl
x: gp.Person with gp.Programmer with gp.Girl = $anon$1@1d2bb9f

scala> x.likesDiamonds
res1: Boolean = true

scala> var x = new Person with Girl with Programmer
x: gp.Person with gp.Girl with gp.Programmer = $anon$1@b53b32

scala> x.likesDiamonds
res2: Boolean = false
```

6) 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 $A \supseteq B$.

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!

7) Yes. Scala adapts this convention too.

8) Consider the following toy example:

```
// square matrices only

abstract class Matrix
{
  def size : int = 0
  def get (i:int, j:int) : int = 0
  def add (m:Matrix) : Matrix = null
  def mul (m:Matrix) : Matrix = null
}
```

```
class OneOneMatrix (val contents:int) extends Matrix
{
  override def size = 1
  override def get (i:int, j:int) = contents
  override def add (m:Matrix) =
    new OneOneMatrix (contents + m.get(0,0))
  override def mul (m:Matrix) =
    new OneOneMatrix (contents * m.get(0,0))
}

class Unary (override val size:int) extends Matrix
{
  override def get (i:int, j:int) = if (i==j) 1 else 0
  override def add (m:Matrix) = m.add (this)
  override def mul (m:Matrix) = m
}

trait RichMatrix extends Matrix
{
  def power (n: int) : Matrix =
  {
    if (n==0) return new Unary (super.size)
    else super.mul (power (n-1))
  }

  def pprint : String =
  {
    var s : String = ""
    for (i <- 0 until super.size)
    {
      for (j <- 0 until super.size)
        s += super.get(i,j) + " "
      s += "\n"
    }
    s
  }
}
```

The idea behind this design is the creation of a minimal interface (`get/add/mul`) that must be implemented by all subclasses of `Matrix` and a rich interface (here it is only `power` and `pprint`, but conceptually there could be many more methods) which is implemented only once but can extend all subclasses of `Matrix`.

Without traits, multiple inheritance together with dynamic dispatch would be needed to achieve the same effect. The trait `RichMatrix` can be created after several subclasses of `Matrix` are created, something that is not possible in multiple inheritance.

One more merit of this design is the possibility of creating several different traits which encompass various different functionalities. The client is then free to mix in exactly the functionalities that the client requires for a given matrix.