Exercise 5

Multiple Inheritance and Traits

October 29th

In-class Assessment: One or more questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

1) Consider the following C++ code:

```
class Person
{
     Person *spouse;
     string name;

public:
     Person (string n) { name = n; spouse = NULL; }

     bool marry (Person *p)
     {
          if (p == this) return false;
          spouse = p;
          if (p) p->spouse = this;
          return true;
     }

     Person *getSpouse () { return spouse; }
     string getName () { return name; }
 };
```

The method `marry` is supposed to ensure that a person cannot marry itself. Without changing the code above, create a new object that belongs to a subclass of `Person` and marry it with itself. Hint: use multiple inheritance. Explain exactly what happens.

2) Write three classes

- A normal queue class `Queue`
- A subclass of `Queue` that also stores (and allows clients to retrieve) the current sum of all items in the queue, using the `enqueue` and `dequeue` methods
- A subclass of `Queue` that also stores (and allows clients to retrieve) the current product of all items in the queue, using the `enqueue` and `dequeue` methods

We now want a class that supports both functionalities.

- Suppose that we want to use multiple inheritance to do that. We want to override the `enqueue` and `dequeue` methods of the new class, such that the new methods call the `enqueue` and `dequeue` methods of both the old classes. Are there any problems with this approach?

- How do we attack the problem using traits? Does this fix the above-mentioned problems? Are there any new problems with this approach?

3) Consider the following declarations in Scala:

```
class C
trait T extends C
trait U extends C
class D extends C
```

Find all the types that can be created with or without traits, as well as the subtype relations between them.

4) Consider the following Scala code:

```
class Cell
{
  private var x:int = 0
  def get() = { x }
  def set(i:int) = { x=i }
}

trait Doubling extends Cell
{
  override def set(i:int) = { super.set(2*i) }
}

trait Incrementing extends Cell
{
    override def set(i:int) = { super.set(i+1) }
}
```

- What is the difference between the following objects?

```
val a = new Cell
val b = new Cell with Incrementing
val c = new Cell with Incrementing with Doubling
val d = new Cell with Doubling with Incrementing
```

- We use the following code to implement a cell that stores the argument of the set method multiplied by four:

```
val e = new Cell with Doubling with Doubling
```

  Why doesn't it work? What does it do? How can we make it work?
- **(Harder)** Find a modularity problem in the above, or a similar, situation.

  Hint: In Scala, fields of an object may be assigned types. Below we explain how. A class declaration can be as follows:

```
class D
{
      type t<:C
      …
```

```
}
```

where `C` is the name of a known class. Class `D` can be instantiated as follows:

```
val c = new D { override type t = C1 }
```

where `C1` must be a subtype of `C`. One can have access to the field `c.t` as a normal type, i.e.

```
val o = new c.t
```

5) Assume all the definitions of the previous exercise. Assume that `Cell` has the invariant that `x` is always even. Furthermore, consider a Scala method

```
foo(x: Cell with Doubling with Incrementing) {…}
```

- During the execution of `foo`, if we assume that all subclasses of `Cell` respect behavioural subtyping, then are we allowed to conclude that `x.get()` always returns an even number?
- Answer the same question, with an extra assumption that the only traits that extend `Cell` are `Incrementing` and `Doubling`.
- We propose the following solution to support traits together with behavioral subtyping:

  Assume `C` is a class with specification `S`. Each time we create a new trait `T` that extends `C`, we must ensure that `C with T` also satisfies `S`.

  Show a counterexample that demonstrates that this approach does not work