Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 6

Multiple Inheritance, Multiple Dispatch and Traits

October 30, 2015

Task 1

Consider the following C++ program:

```
class X {
   public:
      X(int p) : fx(p) {}
      int fx;
};
class Y {
   public:
      Y(int p) : fy(p) {}
      int fy;
};
class B : public virtual X,public Y {
   public:
      B(int p) : X(p-1),Y(p-2){}
};
class C : public virtual X,public Y {
   public:
   C(int p) : X(p+1),Y(p+1){}
};
class D : public B, public C {
   public:
      D(int p) : X(p-1), B(p-2), C(p+1){}
};

int main() {
   D* d = new D(5);
   B* b = d;
   C* c = d;
   std::cout << b->fx << b->fy
             << c->fx << c->fy;
   return 0;
}
```

What is the output of running the program?

(a) `5555`

(b) `2177`

(c) `4147`

(d) `7177`

(e) `7777`

(f) None of the above

Task 2

Consider the following C++ code:

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

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

   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 him-/herself. 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.

Task 3

Consider the following C++ code:

```
class Person
{
   bool likesDiamonds;

public:
   Person (bool l) { likesDiamonds = l; }
};

class Programmer : virtual public Person
{
   public:
      Programmer () : Person (false) {}
      // diamonds are a programmer's worst enemy
};
```

A) It is expected that `!likesDiamonds` is an invariant in class `Programmer`. Use virtual inheritance to break this invariant, without altering the above code.

B) Assuming we have n constructors for class `Person`, where constructor i has the postcondition Q_i (where inputs are existentially quantified), what can we assume at the entry to the body of the `Programmer` constructor?

Task 4

Consider the following C# classes:

```
public class Matrix {
   public virtual Matrix add(Matrix other) {
      Console.WriteLine("Matrix/Matrix");
      return null;
   }
}

public class SparseMatrix : Matrix {
   public virtual SparseMatrix add(SparseMatrix other) {
      Console.WriteLine("SparseMatrix/SparseMatrix");
      return null;
   }
}

public class MainClass {
   public static void Main(string[] args) {
      Matrix m = new Matrix();
      Matrix s = new SparseMatrix();
      add(m,m);
      add(m,s);
      add(s,m);
      add(s,s);
   }

   public static Matrix add(Matrix m1, Matrix m2) {
      return m1.add(m2);
   }
}
```

A) What is the output of this program? Please explain.

B) Without breaking modularity, change only the body of `MainClass.add` to make it possible to always call the most specific `add` method from the matrix hierarchy.

Task 5

Java 8 allows interface methods to have a default implementation directly in the interface.

A) What are some advantages of this feature?

B) What could be some problems with this feature? How can they be resolved?

C) What problems of C++ multiple inheritance are avoided by this new design for Java interfaces?

D) Now suppose that, in addition to method implementations, Java also allowed interfaces to define fields. Interfaces would not have constructors and interface fields would always be initialized with a default value.

- What are some advantages of this feature?
- Given the restrictions above, are there any problems left with such an implementation of multiple inheritance? If so what are they? Propose a solution for each problem you have identified.

Task 6

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.

Task 7

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) }
}
```

A) 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
```

B) 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?

C) Find a modularity problem in the above, or a similar, situation. Hint: a client that gets given a class `C` does not necessarily know if a trait `T` has been mixed in that class.

D) 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.