Concepts of Object-Oriented Programming
AS 2020

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 8

Multiple Inheritance and Linearization

November 13, 2020

Task 1 (from a previous exam)

A) In this task you will have to model a bakery, which produces and sells different types of `BakedProducts`, such as `Bread` and `SeasonalProducts`. Moreover, for different events (e.g., Saint Nicholas or Easter), the bakery offers `SeasonalBread`.

Fill in the C++ class declarations below, such that all the following subtype relations are fulfilled:

- `Bread <: BakedProduct` and `SeasonalProduct <: BakedProduct`
- `Bread` $\not<:$ `SeasonalProduct` and `SeasonalProduct` $\not<:$ `Bread`
  ($\not<:$ means *is not a subtype of*)
- `SeasonalBread <: SeasonalProduct` and `SeasonalBread <: Bread`

Make sure that your code is accepted by the C++ compiler and when executed prints:

Grittibaenz for 6th of December

Easter bread for 1st of April

```
//Bakery code:
class Product {
  public: string name;
          Product(string n) { name = n; };
};

class BakedProduct : public Product {
  public: BakedProduct(string n): Product(n) {};
};

class Bread : ______________________________________________ {
  public: Bread(string n) :

          _______________(n) {};
};

class SeasonalProduct : ____________________________________ {
   public: SeasonalProduct(string n, string e) :

          _______________(n + " for " + e) {};
};

class SeasonalBread : ______________________________________ {
  public: SeasonalBread(string a, string b):

          ____________________________________________________
```

```
                 ________________________________________________ {};
};

//Client code A:
  Product* prod1 = new SeasonalBread("6th of December", "Grittibaenz");
  Product* prod2 = new SeasonalBread("1st of April", "Easter bread");
  cout << prod1->name << endl;
  cout << prod2->name << endl;
  // prints: Grittibaenz for 6th of December
  //          Easter bread for 1st of April
```

B) C++ supports *private inheritance*, which allows code reuse (access to methods and fields) **without** subtyping. Assume that we change the declaration of the class `BakedProduct` from **Task A** to use *private*, instead of public inheritance. All the other classes remain unchanged.

Fill in the blanks from *Client code B*, such that it compiles and when executed prints the same strings as in **Task A**. You are allowed to add methods (but *not* constructors) to any of the provided classes. For each new method that you add, please explicitly write to which class it belongs. You are *not allowed* to make any other changes.

```
//Client code B:

________________ prod1 = new _________________("6th of December",
                                               "Grittibaenz");
________________ prod2 = new _________________("1st of April",
                                               "Easter bread");
cout << prod1 ___________________________ << endl;

cout << prod2 ___________________________ << endl;
// prints: Grittibaenz for 6th of December
//          Easter bread for 1st of April
```

Task 2

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 3

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 try to 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 does it not 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 is 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.

Task 4

(from a previous exam)

Consider the following Scala code:

```
class A             {          def bar() =              ""   }
trait B extends A { override def bar() = super.bar()+"B" }
trait C extends B { override def bar() = super.bar()+"C" }
trait D extends B { override def bar() = super.bar()+"D" }

object Main {
  def main() { foo(new A with D with C with B) }
  def foo(x: A with D) { println(x.bar()) }
}
```

What would be the output of the call `Main.main()`?

(a) BDB

(b) BBDBC

(c) BBCBD

(d) DB

(e) BDC

(f) BCD

(g) None of the above

Task 5 (from a previous exam)

Consider the following Scala code, which compiles correctly and models some jobs a `Person` may have. To work as a `Lawyer` or as a `TaxiDriver`, one needs to have a valid license. This requirement can be expressed through *self type annotations* added to the traits `Lawyer` and

`TaxiDriver` (as in the given code). These annotations are checked by the compiler and allow the traits `Lawyer` and `TaxiDriver` to be mixed only into subtypes of `PersonWithLicense`. Self type annotations enable code reuse without subtyping, that is, `Lawyer` and `TaxiDriver` $\not<:$ `PersonWithLicense`, but the methods of the class `PersonWithLicense` are available and can be overridden inside these two traits.

```
class Person { def work(): String = { return "working"; }}

class Student { def work(): String = { return "studying"; }}

class PersonWithLicense extends Person {
 def hasValidLicense(): Boolean = { return false; }
}

trait Gardener extends Person {
 override def work(): String = { return super.work() + " in the garden";}
}

trait Lawyer extends Person {
 this: PersonWithLicense => // self type annotation

 override def work(): String = {
   if(this.hasValidLicense()) return super.work() + " in court";
   return "not " + super.work();
 }

 override def hasValidLicense(): Boolean = { return true; }
}

trait TaxiDriver extends Person {
 this: PersonWithLicense => // self type annotation

 override def work(): String = { return super.work() + " in Zurich"; }
}
```

A) For each of the following two code fragments (**A.1** and **A.2**), if they compile, write the output of their execution. Otherwise, briefly explain why they are rejected by the compiler.

A.1

```
val lawyer: Lawyer = new PersonWithLicense with Lawyer with TaxiDriver;
println(lawyer.work());
```

A.2

```
val student: Gardener = new Student with Gardener;
println(student.work());
```

B) Add **one** method to any of the given classes or traits **except** `PersonWithLicense` (explicitly write to which one) and fill in the instantiation from the client code below, such that it compiles and when executed prints `not working in Zurich in the garden`. You are **not allowed** to directly return this string, to use reflection, to define new classes or traits, nor to modify the given code. If this is not possible, briefly explain why.

```
// Client code:
val person = new ________________________________________
println(person.work());
```

The following method should be added to: ____________________