ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Concepts of Object-Oriented Programming

Midterm Examination

8.11.2019

Prof. Dr. Peter Müller

Last name: ..... First name: .....

Student ID number: .....

Department (if not D-INFK): .....

I confirm with my signature, that I was able to take this exam under regular circumstances and that I have read and understood the directions below.

Signature: .....

Read completely and carefully the following instructions before starting to work on the exam:

1. Write your last and first name on every page that contains parts of your solutions. Use a ballpoint pen or a fountain pen (no pencil). **Do not use a red pen.** Return the instructions, the tasks, and your solutions.
2. It is neither allowed to use your own papers, documents, scripts, etc., nor any electronic equipment (notebook computers, calculators, cell phones, etc.)
3. Write all of your solutions in **English**.
4. **Explain your solutions carefully if a task asks for an explanation.**
5. You have **1 hour** to complete the exam.
6. Place your student ID on the desk.

Good Luck!

Task	1	2	3	4	Total
Max. points	6	6	20	12	44
Achieved					

Task 1 Overloading and Overriding - 6 points

Fill in with types the three blanks from the following Java code, such that it compiles and when executed prints `Moving sofa from the kitchen`:

```
class Furniture {
   public String name() { return "furniture"; }
}

class BigFurniture extends Furniture {
   public String name() { return "big furniture"; }
}

class Sofa extends BigFurniture {
   public String name() { return "sofa"; }
}

class Room {
   public String move(Furniture furniture, Room to) {
      return "Moving " + furniture.name();
   }
}

class SmallRoom extends Room {
   public String move(BigFurniture furniture, SmallRoom to) {
      return super.move(furniture, to) + " from small room";
   }
}

class Kitchen extends SmallRoom {

   public String move(__________________ furniture, __________________ to) {
      return super.move(furniture, to) + " from the kitchen";
   }
}

class Main {
   public static void main(String[] args) {

      __________________ fromRoom = new Kitchen();
      BigFurniture furniture = new Sofa();
      SmallRoom toRoom = new SmallRoom();
      System.out.println(fromRoom.move(furniture, toRoom));
   }
}
```

solution

Furniture, Room, Room

Task 2 Nominal and Structural Subtyping - 6 points

Assume that we have two statically-typed, type-safe languages with Java-like syntax: JStruct has structural subtyping, JNom has nominal subtyping. They support overriding, but not overloading. Both languages have an additional keyword `inherits`; `class C inherits D` expresses that class `C` inherits from class `D` (but makes no statement about subtyping). JNom additionally has a keyword `subtypes`; `class C subtypes D` expresses that class `C` is a subtype of class `D` (without inheritance).

Assume that there are existing classes `X`, `Y`, `Z` such that `Z <: Y <: X`, and that the `subtypes` clauses are **only** present in the JNom version of the following code:

```
class A {
  X[] xs;
  Y foo(Z z, Y y) { ... }
}

class B subtypes A {
  X[] xs;
  Z foo(Y z, Z y) { ... }
}

class C inherits A {
  int bar(int i) { ... }
}
```

```
class D {
  X[] xs;
}

class E inherits D {
  Z foo(X z, X y) { ... }
  int bar(int i) { ... }
}

class Client {
  void main() {
    doWork(new E())
  }
  void doWork(C c) { ... }
}
```

For each of the following four statements, write if they are correct or not, by answering "yes" or "no". If a statement is **not** correct, briefly explain why.

1. JStruct should **accept** class `B`.

solution

Yes

2. JNom should **accept** class `B`.

solution

No, because the type of the second parameter of its `foo` method is more specific than in class `A`.

3. In JStruct, `C` is **not** a subtype of `A`.

solution

No, because it has all the members `A` has and more.

4. The body of `Client.main` should be **accepted** by JStruct.

solution

Yes

Task 3 Behavioral Subtyping - 20 points

Consider the following Java code:

```
class Super {
  int k;

  /// requires y < k;
  /// ensures k > y;
  int foo(int x, int y) { ... }
}

class Mid extends Super {
  /// requires y <= k;
  /// ensures result > x && k > old(k);
  int foo(int x, int y) { ... }

  /// requires z != 0;
  /// ensures result > 0;
  int bar(int z) { ... }
}

class Sub extends Mid {
  /// requires ?
  /// ensures ?
  int bar(int z) { ... }
}
```

A (6 points)

Assuming specification inheritance is *not* used, does `Mid.foo` fulfill behavioral subtyping? Write down the condition(s) that need(s) to be checked **without simplifications** as well as "yes" or "no".

solution

Yes.

Condition 1: `y < k ==> y <= k`

Condition 2: `old(y < k) ==> (result > x && k > old(k) ==> k > y)`

Optionally, people can make the universal quantification over parameters and field values explicit, but it is not required. Doing this incorrectly does not result in point deductions.

Throughout the following tasks, we will say that a method is *valid* if and only if:

- Assuming only its precondition and the postconditions of invoked methods
- it does not fail at runtime (e.g., violate an assertion or perform a division by zero) and
- it fulfills the preconditions of all methods it invokes and
- it fulfills its postcondition.

B (7 points)

Give a contract to `Sub.bar` that has the following properties:

- it does **not** fulfill behavioral subtyping if **no** specification inheritance is used.
- if specification inheritance is used, the effective contract of `Sub.bar` is sufficient to prove that the following client code is valid:

```
void client (int x) {
  if (x > 0) {
    int r = new Sub().bar(x);
    assert(r > 2); // the contract of bar must imply that this assertion
                   // succeeds.
  }
}
```

Write down both the **declared contract** of `Sub.bar` and its resulting **effective contract**.

solution

One example:

Declared contract: `requires z > 0; ensures result > 2;`

Effective contract: `requires z != 0 || z > 0; ensures result > 0 && (z > 0 ==> result > 2);`

C (7 points)

Assume that specification inheritance is used and that existing code does not use reflection. Is it possible to add one method to `Super` such that a previously valid implementation of `Mid.bar` becomes invalid? If not, just write "no", otherwise provide the new method in `Super` with contracts and a now-invalid implementation of `Mid.bar`.

solution

We add a method `Super.bar` with contract: `requires true; ensures result > 0;`

Implementation of `Mid.bar` that is no longer valid:

```
class Mid extends Super {
  ...

  /// requires z != 0;
  /// ensures result > 0;
  int bar(int z) { return 1 + Math.abs(5/z); } // div by zero if z==0
}
```

Task 4 Linearization - 12 points

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 (6 points)

For each of the following two code fragments, if they compile, write the output of their execution. Otherwise, briefly explain why they are rejected by the compiler.

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

solution

The code compiles: the traits `Lawyer` and `TaxiDriver` are mixed into a subtype of `PersonWithLicense`, as required by the self type annotation and `new PersonWithLicense with Lawyer with TaxiDriver` $<:$ `Lawyer`.

The linearization of `new PersonWithLicense with Lawyer with TaxiDriver` is `TaxiDriver`, `Lawyer`, `PersonWithLicense`, `Person`. When executed, the code prints: working in court in Zurich

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

solution

The code does not compile, because `Student` is not a subclass of `Person` (the superclass of the trait `Gardener`)

B (6 points)

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:

solution

```
// Client code:
val person = new PersonWithLicense with Lawyer with TaxiDriver with
    Gardener;
println(person.work());

trait TaxiDriver extends Person {
  ...
  // additional method:
  override def hasValidLicense(): Boolean = { return false; }
}
```

Note that if we try to add the method `hasValidLicense` to the trait `Gardener`, the client code does not compile, as `new PersonWithLicense with Lawyer with TaxiDriver with Gardener` inherits two methods with the same signature.

This is not possible because (optional):