Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 6

Inheritance

October 30, 2020

Task 1

From a previous exam

Consider the following Java classes:

```java
public class B {
   public void foo(B obj) {
      System.out.print("B1 ");
   }
   public void foo(C obj) {
      System.out.print("B2 ");
   }
}

class C extends B {
   public void foo(B obj) {
      System.out.print("C1 ");
   }
   public void foo(C obj) {
      System.out.print("C2 ");
   }
   public static void main(String[] args) {
      B c = new C();
      B b = new B();
      b.foo(c);
      c.foo(b);
      c.foo(c);
   }
}
```

What is the output of the execution of method main in class C? Explain your answer.

Task 2

Consider the following Java code:

```java
class A {
    String get(Client a) { return "AC"; }
}

class B extends A {
    String get(SpecialClient a) { return "BS"; }
}

class C extends B {
    String get(Client a) { return "CC"; }
    String get(SpecialClient a) { return "CS"; }
```

```
}

class Client {
    String m(A x, A y) { return "C1" + x.get(this) + y.get(this); }
    String m(C x, A y) { return "C2" + x.get(this) + y.get(this); }
    String m(B x, A y) { return "C3" + x.get(this) + y.get(this); }
    String m(C x, C y) { return "C4" + x.get(this) + y.get(this); }
}

class SpecialClient extends Client {
    String m(A x, A y) { return "S1" + x.get(this) + y.get(this); }
    String m(C x, A y) { return "S2" + x.get(this) + y.get(this); }
    String m(B x, A y) { return "S3" + x.get(this) + y.get(this); }
    String m(B x, C y) { return "S4" + x.get(this) + y.get(this); }
}

public class Main {
    public static void main(String[] args) {
        Client client = new SpecialClient();
        C c = new C();
        B b = c;
        System.out.println(client.m(b, c));
    }
}
```

What is the result of compiling the code and running the `Main.main` method?

(a) The program does not compile due to a type error

(b) The program prints a string starting with "S4"

(c) The program prints a string ending with "CS"

(d) The program prints a string containing "BS"

(e) None of the above

NOTE: The discussion about var/dynamic is presented in the lectures after bytecode verification. Maybe we should move the following exercise.

Task 3

A) Compare dynamic type checking with the `dynamic` keyword to static type inference with `var` in C#:

- Give a correct program which can be realized with `dynamic` but not with `var`.
- Give an incorrect program which will be accepted by the compiler with `dynamic` but not with `var`.

B) C#'s most general type is `object`. Similar to `var` and `dynamic`, you can write `object x = ...` with an expression of any type on the right-hand side.

- Given a compiling program using `var`. Can we replace all `var` keywords by `object` and add explicit casts in the right places so that the program compiles and runs as before?
- Given a compiling program using `dynamic`. Can we replace all `dynamic` keywords by `object` and add explicit casts in the right places so that the program compiles and runs as before?

For both questions, either informally describe how to do the replacement, or give a counter-example where the transformation will always produce a program that does not compile or behaves differently. Note that explicit casts to `dynamic` are not allowed in the transformation.

C) Assume now a language like C#, but with covariant return types and contravariant parameter types. Given four classes `A`, `B`, `C` and `D`:

```
class A { int m (int x); }
class B { void m (dynamic x); }
class C { dynamic m (int x); }
class D { dynamic m (dynamic x); }
```

Develop a subtyping rule for the `dynamic` type annotation and informally explain the reasoning behind it. What are the potential subtypes among the four classes above?

Task 4 Overloading and Overriding

Consider the following class in Java:

```
public class Person {

    protected double salary;

    public Person(double salary) {
        this.salary = salary;
    }

    public boolean haveSameIncome(Person other) {
        return this.salary == other.getIncome();
    }

    public double getIncome() {
        return salary;
    }

}
```

Consider also the following subclass of `Person`, a person with a spouse, which takes the salary of the spouse into account as well:

```
public class MarriedPerson extends Person {

    private double spouseSalary;

    public MarriedPerson(double salary, double spouseSalary) {
        super(salary);
        this.spouseSalary = spouseSalary;
    }

    public boolean haveSameIncome(MarriedPerson other) {
        return this.getIncome() == other.getIncome();
    }

    public double getIncome() {
        return ((salary + spouseSalary) / 2);
    }

}
```

A) Show an example with variables `p1,p2`, such that `p1.haveSameIncome(p2)` returns `false`, but `p1.getIncome() == p2.getIncome()` returns `true`. In other words, fill in the following blank with valid code, such that the assertion below the following box is valid. Do not use reflection and assume that `Person` has no other subclasses.

```
Person p1;
MarriedPerson p2;


assert (!p1.haveSameIncome(p2) && p1.getIncome() == p2.getIncome());
```

B) Propose changes to `Person` and `MarriedPerson` such that the assertion above will fail.

B.1 Can you change **only** `MarriedPerson.haveSameIncome`, such that the assertion above will fail for your solution to subtask **A**? If yes, provide the modified method. Otherwise, explain why this is not possible.

B.2 Can you change **only** `Person.haveSameIncome`, such that the assertion above will fail for your solution to subtask **A**? If yes, provide the modified method. Otherwise, explain why this is not possible.

Task 5

Some research languages have symmetric multiple dispatch - methods are defined outside classes, and dispatched dynamically on all arguments regardless of order (no overloading at all). There is no designated receiver for a method but rather all arguments are of the same priority - this is intended to handle binary methods better which are often naturally symmetric. Out of all methods that are statically in scope for a given invocation, the runtime selects the most specific method to dispatch according to all arguments, and so there must be a single best implementation for each possible invocation of a method. The return type is not considered in the implementation selection. When compiling a package the compiler analyzes all types used in the package and all methods and makes sure that for each method and argument types combination there is a single best method to be called - or issues an error if that is not the case. Assume the following three classes in such a language:

```
package integer
class Integer
{
   ...
}
Integer add(Integer x,Integer y){...}

package natural
import integer.Integer
class Natural extends Integer
{
   ...
}
Integer add(Natural x,Integer y){...}
Integer add(Integer x,Natural y){...}
Natural add(Natural x,Natural y){...}

package even
import integer.Integer
class Even extends Integer
{
   ...
}

Integer add(Even    x,Integer y){...}
Integer add(Integer x,Even    y){...}
Even    add(Even    x,Even    y){...}
```

The elipsis in each class body represents (possibly) private data but no other methods.

Each package compiles successfully on its own.

A user has now written the following client:

```
package client
import even.*
import natural.*

void f(Integer x,Integer y)
{
   Integer z = add(x,y);
}
```

- What would be the problem in allowing this client to compile in a type safe multiple dispatch language? Show code that would expose the problem.
- Which requirement could we relax so that this call is valid?
- What could we do in the client package, in order to resolve the problem, without modifying other packages and without relaxing the requirement mentioned above?

Task 6 Inheritance

From the midterm 2014.

Consider the following class in Java, which represents a fixed-size sequence of integers:

```
public class Seq {
   public Seq(int size) { a = new int[size]; } // all initialized to 0
   public int getSize(){ return a.length; }
   public int getAt(int i) { return a[i]; }
   public void setAt(int i, int x) { a[i]=x; }
   public void addTo(int i, int x) { a[i]+=x; }
   public void addToAll(int x){
      for (int i=0;i<a.length;i++)
         a[i]+=x;
   }

   private int[] a;
}
```

Consider also the following subclass of `Seq`, which adds a `getSum` method to `Seq` that is implemented efficiently:

```
public class SeqSum extends Seq {
   public SeqSum(int size) { super(size); }
   public int getSum() { return sum; }
   public void setAt(int i, int x) {
      int newSum=sum+x-getAt(i);
      super.setAt(i,x);
      sum = newSum;
   }
   public void addTo(int i, int x) {
      int newSum=sum+x;
      super.addTo(i,x);
      sum = newSum;
   }
   public void addToAll(int x) {
      super.addToAll(x);
      sum += getSize()*x;
   }
```

```
    private int sum=0;
}
```

In this question do not use downcasting or reflection. A "client" refers only to clients instantiating the class, not to subclasses.

A) Change the implementation of `Seq.addToAll` so that class `Seq` behaves exactly the same but `SeqSum.addToAll` calculates the wrong sum. Show a client that produces a different output with the original and modified implementations.

B) Assume the original implementation of both classes. Give an alternative implementation for `Seq.setAt` and separately for `SeqSum.addTo` so that each change alone leaves both classes behaving exactly the same, but putting both changes together would break the behavior of at least one method in class `SeqSum`. Show a client that observes the change in behavior.