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:

```
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.

solution

The code will print `B1 C1 C1`.

Overloading is resolved statically, based on the static type of the receiver and the static type of the arguments. Since both `b` and `c` have static type `B`, the compiler will choose for all three calls the method `B.foo(B obj)`.

At runtime, we will execute either the statically chosen method or a method that overrides the statically chosen one, determined based on the dynamic type of the receiver. Note that the dynamic type of the arguments is not relevant.

`b` has dynamic type `B`, so for the first call we will execute the statically chosen method, `B.foo(B obj)`.

`c` has dynamic type `C`, so for the last two calls we will execute the method from class `C` that overrides the statically chosen method, that is, `C.foo(B obj)`.

Task 2

Consider the following Java code:

```
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) **CORRECT:** None of the above

solution

The program compiles and prints "S3CSCC".

`client` has static type `Client`, `b` has static `B`, `c` has static type `C`. The compiler chooses the most specific method from the class `Client`, which accepts a parameter of type `B` and one of type `C`. This is `Client.m(B x, A y)`.

Since client has dynamic type `SpecialClient`, at runtime we will execute the method that overrides the statically chosen method, that is `SpecialClient.m(B x, A y)`. The program will therefore print a string starting with "S3".

We now need to determine the results of the calls `x.get(this)` and `y.get(this)` from the body of the method `SpecialClient.m(B x, A y)`.

`x` has static type `B`, `this` has static type `SpecialClient`. The compiler therefore chooses the method `B.get(SpecialClient a)`. Since `x` has dynamic type `C`, we will actually execute the method from class `C` which overrides the statically chosen method. That is, `B.get(SpecialClient a)`, which returns "CS".

`y` has static type `A`, `this` has static type `SpecialClient`. The compiler therefore chooses the method `A.get(Client a)`. Since `y` has dynamic type `C`, we will actually execute the method from class `C` which overrides the statically chosen method. That is, `C.get(Client a)`, which returns "CC".

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`.

solution

```
static void Main() {
    dynamic x;
    if(condition()) {
        x = 5;
    } else {
        x = "hello";
    }

    Print(x);
}

static void Print(string str) {
    Console.WriteLine(str);
}

static void Print(int value) {
    Console.WriteLine(value);
}
```

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

solution

```
var x = 3;
x.substring(..);
```

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?

solution

This will be possible in all cases where we know what the type of the variable declared with `var` is. In those cases we can just cast the declared variable in all places where it is used to the most general type fulfilling all static type constraints on the corresponding variable. Since the original program compiled, such a type must exist.

In the case of anonymous types however, we do not know the name of the type to cast to. Consider:

```
var x = new { a = 108, b = "Hello" };
Console.WriteLine(x.b);
```

Here, we could change `var` to `object`, but we will not be able to cast `x` in the second line, because we do not know the type name which the compiler generates for this anonymous type.

- 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?

solution

Generally we cannot do this, as shown in the following example:

```
static void Main() {
    dynamic x;
    if(condition()) {
        x = 5;
    } else {
        x = "hello";
    }

    Print(x);
}

static void Print(string str) {
    Console.WriteLine(str);
}

static void Print(int value) {
    Console.WriteLine(value);
}
```

To make this code work with `object`, we would need to add explicit type checks and cast the argument to the proper static type.

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?

solution

Following the Substitution principle, `dynamic` is equivalent to `object`, in that it accepts any type. Therefore, the usual subtyping rules apply, treating `dynamic` as the most general supertype of all other types. The potential subtyping relations are `A <: C` and `D <: C`.

There are two different ways of looking at class `B`. On the one hand, we could just say that `void` is a special keyword that indicates the absence of a return value, and thus the method `B.m` is unrelated to the other methods. Alternatively, we can allow methods with `void` return type to be overwritten by methods with any return type (assuming the parameter variance rules are satisfied): if a client code is written to expect `void` (no return value), then we could instead use a method which returns an arbitrary value and just discard it. In this second interpretation we will additionally have `D <: B`.

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;
```

solution

```
p1 = new MarriedPerson(a,b);
p2 = new MarriedPerson(c,d);
```

for any a, b, c, d such that $a + b = c + d$ but $a \neq (c + d)/2$.

```
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.

solution

Yes, the following solution works.

```
public boolean haveSameIncome(Person other) {
    // changed MarriedPerson to Person in signature
    return this.getIncome() == other.getIncome();
}
```

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.

solution

Yes, the following solution works.

```
public boolean haveSameIncome(Person other) {
    return this.getIncome() == other.getIncome();
    // changed calls to salary to getIncome here
}
```

Another trivial solution would be:

```
public boolean haveSameIncome(Person other) {
    return true;
}
```

Also possible: Type-check with `instanceOf`, then cast both to `MarriedPerson` and call `haveSameIncome` on casted objects.

Also possible: Change parameter type to `MarriedPerson`.

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.

solution

The problem would be that the call `add(x,y)` could be ambiguous between the methods `add(Even,Integer)` and `add(Integer,Natural)` in the call:

```
{
   Even e;
   Natural n;
   f(e,n);
}
```

Both are the most specific implementations but neither is more specific than the other.

- Which requirement could we relax so that this call is valid?

solution

We could allow the runtime to choose any of the viable methods that is not worse than another method - thus we would lose the ability to predict which method gets called, but functionality should conform to at least that of `add(Integer,Integer)`.

- 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?

solution

The client could define a method `add(Even,Natural)` (and any other missing methods) that would resolve the ambiguity.

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.

solution

```
public class Seq {
   ...
   public void addToAll(int x) {
      for (int i=0;i<a.length;i++)
         addTo(i,x);
   }
}
public class Client{
   public void f() {
      SeqSum s = new SeqSum(5);
      s.addToAll(1);
      assert(s.getSum()==5); //getSum() will return 10
   }
}
```

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.

solution

```
public class Seq {
   ...
   public void setAt(int i, int x) { addTo(i,x-getAt(i)); }

}
public class SeqSum extends Seq {
   ...
   public void addTo(int i, int x) { setAt(i,x+getAt(i)); }

}
public class Client{
   public void f() {
      SeqSum s = new SeqSum(5);
      s.setAt(1,1); //this will recurse until stack overflow
   }
}
```