ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 5
Inheritance
October 27, 2017

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` - the method definition is resolved in terms of the static type of the argument, but the dynamic type of the receiver. Note that this means that it is possible to have two aliases of the same object, and receive different results when passing them as parameter to a method of the same name (note however that, this is not really passing them to the same method - it is better to think of method overloads as definitions of two different methods in the class).

Task 2 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 {
   ...
   public void addTo(int i, int x) { setAt(i,x+getAt(i)); }

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

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

```
dynamic x;
if (condition()) {
   x = 4;
} else {
   x = "5";
}

dynamic y = x + x;
```

- 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:

```
dynamic x;
if (condition()) {
   x = 4;
} else {
   x = "5";
}

dynamic y = x + x;
```

To make this code work with `object`, we would need to add explicit type checks and duplicate the addition on the last line.

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

Assume we are working with a Java-like language in which method dispatch is dynamic for the type of the receiver and static for the type of the arguments. Consider a class `Matrix` to implement matrices with integer values. A simple implementation would be to store a (private) 2-dimensional array of integers, and provide methods such as:

```
void set(int i, int j, int value);
int get(int i, int j);
Matrix add(Matrix m);
Matrix multiply(Matrix m);
```

A sparse matrix is a matrix which contains mainly zeros. When such matrices are large it can be that an alternative representation of the matrix, which only stores the locations and values of non-zero entries, can provide much more efficient implementations for common expensive operations such as addition and multiplication with other sparse matrices. If a sparse matrix is to be added or multiplied with a standard matrix, it also is possible to define an implementation which is more efficient that the standard one (but not as good as for two sparse matrices).

Consider writing a new class `SparseMatrix` to implement sparse matrices, with the similar methods available to those for `Matrix`.

- Is it likely that there will be scope for reusing code from the class `Matrix`?

solution

Code reuse is not going to be possible (at least for the primitive operations), since the two classes will use different internal representations of the data.

- Does it seem that `SparseMatrix` can (and should?) be a behavioural subtype of `Matrix` ?

solution

As long as all fields are private, the classes should be indistinguishable in terms of behaviour (except for operation complexity). However, in order to state that formally, we would have to write an abstract specification for the matrix class, which does not refer to any fields.

- What would be the implications of making `SparseMatrix` a subclass of `Matrix`?

solution

`SparseMatrix` would be a subtype of `Matrix`, so an instance of the former could be used anywhere where a `Matrix` is expected. On the other hand, a `SparseMatrix` object will inherit a useless copy of the fields used in `Matrix` - this means an overhead in memory and initialization time (since by default the superclass constructor will still be called). This can also lead to subtle bugs.

- What alternative ways are there of expressing the relationship between the classes?

solution

An interface (or abstract class) could alternatively be defined, which both classes implement (or subclass). This eliminates the redundant overlap between fields used in the two classes. However, if client code has already been written in terms of the class `Matrix` then adding the interface will not avoid any problems for this client code (this is a good reason to always provide interfaces (or abstract classes in C++) rather than class definitions, to clients!).

Task 5

Suppose from now on that `SparseMatrix` is to be implemented as a subclass of `Matrix`. Assume (reasonably!) that the two classes will use different internal representations (fields). If you sketch a possible implementation, it might help.

- What would happen if client code could access the fields? E.g., suppose `entries` is the 2-d array field of `Matrix`, and `m` is a local `Matrix` variable, and consider:

  ```
  m.entries[i][j] = 4;
  if(m.get(i,j)!= 4) { // crash }
  ```

  What can go wrong here? To what extent are these problems avoided by making the fields private?

solution

In the case of the code

```
m.entries[i][j] = 4;
if(m.get(i,j)!= 4) { // crash }
```

if `m` turns out to reference a `SparseMatrix` object, then because the method call to `get()` will be dynamically dispatched, it will refer to the fields used for the internal representation of `SparseMatrix`, and not the entries array. Therefore, there is no reason to expect the if-condition to be false. Making the fields private avoids this problem arising in client code, but it can still occur in other methods of `Matrix` if there is a mixture of direct field accesses and (dynamically dispatched) method calls.

- What might go wrong (or at least give unexpected behavior) if we do not override all of the methods of `Matrix` when writing `SparseMatrix`?

solution

Similarly to the previous part, if we retain any method implementations from the `Matrix` class then these are likely to refer to the fields used for internal representation of the superclass and not the subclass, which are unlikely to contain meaningful values.

- What difficulties might occur if we wanted to add extra methods to `Matrix` later?

solution

Any extra methods that we add to `Matrix` will suffer the same difficulty - because they will typically refer to the entries array, they will not operate correctly on `SparseMatrix` objects. The only exception is a method which is implemented entirely in terms of previously-defined methods (no field accesses).

Task 6

Some research languages have symmetric multiple dispatch - methods are defined outside classes, and dispatch 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? Dispatch must remain completely symmetric.

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.