Concepts of Object-Oriented Programming
AS 2022

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 2

Types and Subtyping

October 7, 2022

Note: Please wait until the lecture on the 6th of October before attempting tasks from 2 to 6, because they require subtyping rules not yet covered in the lecture. **This is an exception for which we apologize. For all the other exercises, you will have one week to solve the tasks; they are based on the material presented in the lecture the week *before* the exercise is discussed in class.**

Task 1

Show:

- A program that is rejected by a statically typed language but is executed without typing errors in a dynamically typed language.
- A program that is rejected by a statically typed language and runs into a type error when executed in a dynamically typed language.

Task 2

Suppose that we have a language with structural subtyping, contravariant parameter types and covariant return types. Consider the following types:

```
class A { int m(int x){...}; }
class B { int m(int x){...};  int n(int x) {...}; }
class C { int n(int y){...};  int m(int x) {...}; }
class D { C m(A a) {...}; }
class E { C m(B b) {...}; }
class F { A m(B e) {...}; }
class G { B m(C e) {...}; }
class H { G m(D d, E e) {...}; }
class I { F m(E e, D d) {...}; }
class J { A a; }
class K { B b; }
class L { B a; }
```

Find all the subtyping relations among them. Assume that `int` has no subtype other than itself.

Task 3

Consider the following Java program:

```
class B {
   protected int get() {...}
}

class A extends B {
   private int get() {...}
```

```
}

class C extends B {
   public int get() {...}
}
```

When we compile it, we obtain the following error:

```
get() in A cannot override get() in B; attempting to
assign weaker access privileges; was protected
        private int get() {...}
                    ^
```

Explain why this is the behavior of the Java compiler.

Task 4

In C++ object aliasing is achieved using pointers and it is possible to have a pointer to a pointer. Here is an example:

```
class SuperX {};
class X : public SuperX { public: int a; };
class SubX : public X { public: int b; };

class Initializer {
   public:
      void init(X** x) {
         *x = new X();
      }
};

class Value {
   private:
      X* x = nullptr;
   public:
      Value(Initializer* i) {
         i->init(&x); // The initializer object will set the value of x
      }
};
```

How does the substitution principle apply to values of type pointer to pointer? Is it safe to call methods that have the signature of `init` with a value of type pointer to pointer to a subtype/supertype of X? If yes, explain why. Otherwise, modify and extend the code above to show how calling `init` with such a value can lead to an error at run time.

Task 5 Union Types

Assume a language with nominal subtyping, covariant return types and contravariant parameter types that allows types to be defined as a disjunction of other types, as in the following declarations:

```
String || Number get();
void set(String || Number newValue);
```

Such a type is called a *union type* and the different types that form the disjunction are its *components*. Classes can be thought of as union types with just one component.

A type `Sub` is a subtype of another type `Super`, i.e. `Sub <: Super`, if for each component C_{sub} of `Sub` there exists a component C_{sup} of `Super` such that $C_{sub} <: C_{sup}$. The usual nominal subtyping rules apply for classes.

A) Consider the signatures of the four methods below, assuming that `C <: B <: A` (`A`, `B`, and `C` are regular class types)

```
m1: B            foo (B b)
m2: A            foo (A || B ab)
m3: B || C       foo (A a)
m4: A || B || C  foo (C c)
```

Your task is to complete the table below. For each row and column, write 'yes', if the method at the left of the row could override the method at the top of the column. Otherwise write 'no'.

	m1	m2	m3	m4
m1	yes			
m2		yes		
m3			yes	
m4				yes

B) Assume that `A`, `B`, and `Q` are classes such that `B <: A` and `Q` is unrelated to `A` and `B`. Consider this code fragment:

```
void foo(A || Q arg) { arg.bar(42); }
```

(i) Assume that the type checker admits method `foo` only if all components of `arg`'s static type have a method `void bar(int)` which is accessible from `foo`. Do we need any run-time checks in order to avoid run-time errors? If so, what are they? Under what conditions could they be omitted?

(ii) Assume that the type checker admits method `foo` if at least one component of `arg`'s static type has an accessible method `void bar(int)`. Do we need any run-time checks in order to avoid run-time errors? If so, what are they? Under what conditions could they be omitted?

(iii) Answer the questions from (i) and (ii) for the code fragment below.

```
void foo(A || B arg) { arg.bar(42); }
```

Task 6

As you will see in the lectures, arrays are covariant in Java and C#. Because of this, each array update requires a run-time type check. Another approach would have been to adopt contravariant arrays. Does this solution require run-time type checks? If this is the case, explain in which cases you need these run-time type checks and provide an example in which a check would fail.