

# Exercise 2

## Types and Subtyping

October 8, 2021

**Note:** Please wait until the lecture on the 7<sup>th</sup> 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.

solution

Consider

```
if (x==x) { y = 1; }  
else { y = "Hello"; }  
y = y-1;
```

A usual static type system would reject this program, but the program would not cause typing problems at runtime.

- A program that is rejected by a statically typed language and runs into a type error when executed in a dynamically typed language.

solution

The static type system would reject the following program which would generate a runtime type error:

```
int f(int x) { return x-1 }  
print(f("World"))
```

### 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.

— solution —

$B = C <: A$  and  $D <: E = G <: F$

No other subtyping relations exist, except the reflexive and transitive closure of the above.

### 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.

— solution —

In Java, there are four different types of access modifiers for fields and methods:

- `public`: every class can access the element
- `protected`: only subclasses and classes in the same package can access the element
- *default*: only classes in the same package can access the element
- `private`: only this class can access the element

We can state that

$\text{public} <: \text{protected} <: \text{default} <: \text{private}$

where  $a <: b$  means that the accessibility level  $a$  is weaker than  $b$ .

Class `A` restricts the accessibility of method `get()` (i.e., makes it stronger), since it is `protected` in `B` and `private` in `A`. This means that class `A` allows fewer behaviors than `B`, so it cannot be a subtype of `B`. On the other hand, class `C` relaxes the accessibility level of method `get()`, so it allows more behaviors than `B`, and this is allowed by the Java compiler.

In general, a class can be a subtype of another class if, for all its methods that override methods in the superclass, it assigns weaker accessibility levels than the ones of the superclass. Therefore, if  $a \leq b$  then a subclass can relax the accessibility level  $b$  with  $a$ .

## 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.

— solution —

It is not safe to call methods with the signature of `init` with anything but a pointer to pointer to `X`. A pointer to a pointer can be thought of as an array with one object. As we know, statically safe arrays are invariant.

The code below illustrates

- what can go wrong when `init` is called with a pointer to pointer to a supertype of `X`:

```
class Initializer{  
    public:  
    void init(X** x) {  
        (*x)->a=5; // run time error because SuperX object  
                  // does not have a field "a".  
    }  
};  
  
class Value {  
    private:  
    SuperX* super_x = new SuperX();  
    public:  
    Value(Initializer* i) {  
        i->init(&super_x);  
    }  
};
```

- what can go wrong when `init` is called with a pointer to pointer to a subtype of `X`:

```

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

class Value {
private:
    SubX*   sub_x   = nullptr;
public:
    Value(Initializer* i) {
        i->init(&sub_x);    // sub_x will get a value of type X
        sub_x->b = 5;      // and cause a run time error here
    }
};

```

## 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

— solution —

Note that here `A || B || C`, `A || B`, and `A` can be considered to be identical types, because `C <: B <: A`. Similarly `B || C` can also be considered identical to `B`.

	m1	m2	m3	m4
m1	yes	no	no	yes
m2	no	yes	no	yes
m3	yes	yes	yes	yes
m4	no	no	no	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?

— solution —

No run-time checks are necessary.

(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?

— solution —

We could do an additional static check that all components of the type of `arg` have an accessible method `void bar(int)` and, if it fails, we need a dynamic check that the run-time type of `arg` has `void bar(int)`.

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

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

— solution —

(i) No run-time checks are necessary.

(ii) We can do a static check that  $A$  has `void bar(int)`. If it does, we do not need a run-time check. Otherwise at run-time we need a check that the type of `arg` is  $B$ .

## 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.

— solution —

Yes, contravariant arrays would require run-time type checks when reading values from the array.

By the definition of contravariance, we have that  $S <: T$  implies  $T[] <: S[]$ .  
Therefore  $\text{Object}[] <: \text{String}[]$  since  $\text{String} <: \text{Object}$ . So we can pass an array of type  $\text{Object}[]$  to a method that requires a  $\text{String}[]$  argument.

```
class C {
    String foo(String[] a) {
        return a[0];
    }

    void client() {
        Object[] arr = new Object[1];
        arr[0] = new Object();
        String s = foo(arr);
    }
}
```