

# Exercise 5

## Behavioral Subtyping and Inheritance

October 29, 2021

### Task 1

Consider the following Java code:

```
1 interface I {};  
2  
3 class C {};  
4  
5 public class Main {  
6  
7     public static C getC() {  
8         return new C();  
9     }  
10  
11     public static void main(String[] argv) {  
12         C c1 = new C();  
13         C c2 = getC();  
14  
15         I i1 = (I) c1;  
16         I i2 = (I) c2;  
17     }  
18 }
```

Try to compile it. If it compiles, try to execute it. What happens? Why? Do you expect to see the same behavior if `I` were a class, instead of an interface?

— solution —

- If `I` is an *interface*: the compiler allows the code to go through, although it cannot prove that `c1` and `c2` implement `I`. The reason is that there might be a subclass `D` of `C` such that `D` implements `I` and `c1` and `c2` might be objects of `D`. Here Java opts for the flexibility of dynamic type checking. When the code executes, a runtime exception is thrown, because `c1` does not implement `I` and this is caught by the runtime check. If we would comment out line 15, a runtime exception would be thrown in line 16, as `c2` does not implement `I` either.
- If `I` were a *class*: the code would not compile, due to type errors in lines 15 and 16. In this case, Java chooses static type checking: as it does not support multiple inheritance, it is not possible to have a subclass `D` of `C`, which also extends `I`.

### Task 2

Consider the following Java classes:

```
class Number {  
    int n;
```

```

    /// requires true
    /// ensures n == p
    void set(int p) {
        n = p;
    }
}

class UndoNaturalNumber extends Number {
    int undo;

    /// requires 0 < q
    /// ensures n == q && undo == old(n)
    void set(int q) {
        undo = n;
        n = q;
    }
}

```

Is UndoNaturalNumber a behavioral subtype of Number, based on the rules from slide 62?

Now consider that we are using specification inheritance. What are the effective pre/post-conditions of the method UndoNaturalNumber.set according to the rules from slides 68 and 72?

— solution —

UndoNaturalNumber is not a behavioral subtype of Number, because it has a stronger precondition for the method set.

The effective precondition is: `true || (0 < q)`, which is equivalent to `true`.

The effective postcondition is: `(old(true) ==> n == p) && (old(0 < q) ==> n == q && undo == old(n))`. Note that `p` and `q` refer to the same parameter, so the effective postcondition actually is: `(old(true) ==> n == q) && (old(0 < q) ==> n == q && undo == old(n))`. Since for parameters we always have `old(q) == q`, the effective postcondition is equivalent to `(n == q) && (0 < q ==> undo == old(n))`.

## Task 3

*From a previous exam.*

Assume the following types in Java:

```

enum Shift {DayShift, NightShift, SpecialShift}

interface PostalWorker {
    boolean sick();

    /// ensures sick()
    void catchDisease();

    /// requires when == SpecialShift || when == DayShift
    /// requires !sick()
    int work(Shift when);
}

interface Bartender {
    boolean sick();

    /// ensures sick()
    void catchDisease();
}

```

```

    /// requires when == SpecialShift || when == NightShift
    /// requires !sick()
    int work(Shift when);
}

```

The `work()` method can be called to request the corresponding person to work the specified shift. The value returned by `work()` is the average hourly wage that was earned during the working shift including tips.

A) Now we introduce another interface:

```

interface HardWorker extends PostalWorker, Bartender {
    /// requires true
    int work(Shift when);
}

```

Assuming the rules for specification inheritance discussed in the course (slides 68 and 72), what is the effective precondition of the `work()` method of the `HardWorker` interface?

— solution —

```

/// requires
    (!sick() && (when == SpecialShift || when == DayShift))
|| (!sick() && (when == SpecialShift || when == NightShift))
|| true

```

which is equivalent to

```

/// requires true

```

B) Now we add postconditions to all `work()` methods. Everything else remains as before.

```

interface PostalWorker {
    ...
    /// ensures result  $\geq$  15 && result  $\leq$  25
    int work(Shift when);
}

interface Bartender {
    ...
    /// ensures result  $\geq$  20 && result  $\leq$  30
    int work(Shift when);
}

interface HardWorker extends PostalWorker, Bartender {
    ...
    /// ensures result  $\geq$  25 && result  $\leq$  50
    int work(Shift when);
}

```

Assuming the rules for specification inheritance discussed in the course (slides 68 and 72), what is the effective postcondition of the `work()` method of the `HardWorker` interface?

— solution —

```

/// ensures
    ( old(!sick() && (when == SpecialShift || when == DayShift))
       $\Rightarrow$  (result  $\geq$  15 && result  $\leq$  25) )
&& ( old(!sick() && (when == SpecialShift || when == NightShift))

```

```

    ⇒ (result ≥ 20 && result ≤ 30) )
&& ( old(true)
    ⇒ (result ≥ 25 && result ≤ 50) )

```

which is equivalent to

```

/// ensures
( old(!sick() && when != NightShift && when != null)
  ⇒ result == 25 )
&& ( old(!sick() && when == NightShift)
  ⇒ (result ≥ 25 && result ≤ 30) )
&& ( old(sick() || when == null)
  ⇒ (result ≥ 25 && result ≤ 50) )

```

C) Consider the following code:

```

/// requires worker != null
/// requires !worker.sick()
int foo(HardWorker worker) {
    return worker.work(Shift.SpecialShift);
}

```

What is the range of possible return values of the method `foo()`?

— solution —

Only 25 is a possible return value.

D) Change the body of the method `foo()` such that it calls the `work()` method of `worker` in a way that makes it possible for this call to return 50.

— solution —

```

int foo(HardWorker worker) {
    worker.catchDisease();
    // DayShift and NightShift also work
    return worker.work(Shift.SpecialShift);
}

```

Another solution is

```

int foo(HardWorker worker) {
    return worker.work(null);
}

```

## Task 4

Suppose that we have a database, for which we would like to add an “automated key generation” feature. This means that each time the user inserts a new tuple, a unique key is automatically generated for the tuple by the system. A way to do this is to write a counter, which increments by 1 the value that it returns each time it is called. The method that generates a new key is called `generate`.

A) Write a Java class `IncCounter` and an accompanying specification for such a counter.

— solution —

```
class IncCounter {
  /// constraint old(key) <= key
  int key;

  IncCounter () { key = 0; }

  /// ensures (key == old(key) + 1) ∧ (result == old(key))
  int generate () { return key++; }
}
```

B) Annotate the following Java class with specifications and show that it is not a behavioural subtype of IncCounter.

```
class DecCounter {
  int key;
  DecCounter () { key = 0; }
  int generate () { return key--; }
}
```

— solution —

The postcondition which precisely describes the behavior of the method DecCounter.generate is  $(key == \text{old}(key) - 1) \wedge (result == \text{old}(key))$ . This postcondition does not refine the postcondition of IncCounter.generate. The history constraint is  $\text{old}(key) \geq key$  and also does not strengthen the one of IncCounter.

C) Write an abstract class GenerateUniqueKey together with a specification, such that both IncCounter and DecCounter (with the specifications from tasks A and B) are behavioural subtypes of GenerateUniqueKey, and such that GenerateUniqueKey.generate generates unique keys. In the specification, you may use helper methods and fields.

— solution —

The abstract parent class GenerateUniqueKey can be declared using a helper *pure* method boolean used(int). Note that only pure (i.e., side-effect free) methods can be used in specifications. Informally, the helper method returns true if x has been used as a key before. Furthermore, the correctness of the class relies on the property that once a number is used, it never becomes unused again. This can be expressed using a history constraint.

The definitions of the classes follow:

```
abstract class GenerateUniqueKey {
  /// constraint  $\forall x:\text{int} \mid (\text{old}(\text{used}(x)) \Rightarrow \text{used}(x))$ 
  abstract boolean used(int);

  /// ensures  $\neg \text{old}(\text{used}(\text{result})) \wedge \text{used}(\text{result})$ 
  abstract int generate ();
}

class IncCounter { // ... and similarly for DecCounter
  /// constraint old(key) <= key
  int key;
  IncCounter () { key = 0; }

  boolean used (int x) { return x < key; }
}
```

```
    /// ensures key == old(key) + 1 ∧ result == old(key)
    int generate () { return key++; }
}
```

## Task 5

*From a previous midterm.*

Imagine extending the syntax of the Java language to support the following keywords:

- `subtypes`: used to declare that a class is a subtype of another class (without inheritance)
- `inherits`: used to declare that a class inherits from another class (without subtyping)

Now consider the following classes:

```
class A {
    public int foo (int n) { return n - 1; }
}

class B {
    public int foo (int n) { return n + 1; }
    public int bar (int n) { return foo(n) - 1; }
}

class C inherits A subtypes B {
    public int bar (int n) { return foo(n); }
}

class Main {
    public static void main(String[] args) {
        B b = new C();
        System.out.println( b.bar(3) );
    }
}
```

What should happen if we tried to compile the code and execute the method `main` from the class `Main`?

- (a) The code should be rejected by the compiler
- (b) The code should compile but the execution should fail
- (c) **CORRECT:** The code should compile and print 2
- (d) The code should compile and print 4
- (e) None of the above

## Task 6

*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)`.