

Exercise 4

Behavioral Subtyping and Inheritance

October 20, 2017

Task 1

Investigate the behavior of the following Java code:

```
interface I {};  
  
class C {};  
  
public class E2_1  
{  
    public static void main(String [] argv)  
    {  
        C c = new C();  
        I i = (I) c;  
    }  
}
```

Try to compile it. If it compiles, try to execute it. What happens? Why?

Task 2

Consider the example in Slide 58 of the Lecture 2:

```
class Number {  
    int n;  
  
    /// requires true  
    /// ensures n == p  
    void set(int p) { n = p; }  
}  
  
class UndoNaturalNumber extends Number {  
    int undo;  
  
    /// requires 0 < p  
    /// ensures n == p && undo == old(n)  
    void set(int p) { undo = n; n = p; }  
  
    /// requires true  
    /// ensures n == undo && undo == old(undo)  
    void reset() { n = undo; }  
}
```

The invariants have been removed. Class `UndoNaturalNumber` is not a behavioral subtype of `Number`. One solution is to use specification inheritance. What are the effective pre/post-conditions of method `UndoNaturalNumber.set` according to the rules of Slides 67 and 71?

Task 3 Behavioral Subtyping

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 in order to request that the corresponding person work the requested 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 improved rule for specification inheritance discussed in the course, what is the effective precondition of the `work()` method of the `HardWorker` interface?

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

```
interface PostalWorker {
    ...
    ///ensures result ≥ 15 && result ≤ 25
    int work(Shift when);
}

interface Bartender {
    ...
    ///ensures result ≥ 20 && result ≤ 30
    int work(Shift when);
}

interface HardWorker extends PostalWorker, Bartender {
    ...
    ///ensures result ≥ 25 && result ≤ 50
    int work(Shift when);
}
```

Assuming the improved rules for specification inheritance, what is the effective postcondition of the `work()` method of `HardWorker`?

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 `foo()` method?

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

Task 4

From 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 would happen if we tried to compile the code and execute method `main` in class `Main`?

- The code will be rejected by the compiler
- The code will compile but the execution will fail
- The code will compile and print 2
- The code will compile and print 4
- None of the above

Task 5

Consider two classes `Stack` and `Queue`, implementing the standard LIFO/FIFO data structures, both of which have methods with the following signatures:

```
void push(Object o);
Object pop();
bool isEmpty();
int size();
void reverse();
```

- A) Despite having identical signatures, these two classes cannot be behavioral subtypes of one another. Why not?
- B) When implementing these two classes, is there any possibility of code reuse? If so, give details.
- C) Describe at least one way of reusing the code in one class by the other - which programming language features are needed for this to work?

Task 6

Suppose that we have a database, for which we want 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. An obvious way to do that 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.
- 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--; }
}
```

- C) Write an abstract class `GenerateUniqueKey` together with a specification, such that both `IncCounter` and `DecCounter` are behavioural subtypes of `GenerateUniqueKey`. In the specification, you may use helper methods and fields.