

Exercise 5

Behavioral Subtyping and Inheritance

October 28, 2022

Task 1

Consider the following Java code:

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

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?

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 59 and 61?

Task 3

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.

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` (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.

Task 4

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) The code should compile and print 2
- (d) The code should compile and print 4
- (e) None of the above

Task 5

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.

Task 6 Overloading and Overriding

Consider the following class in Java:

```
public class Person {
    protected double salary;

    public Person(double salary) {
        this.salary = salary;
    }

    public boolean haveSameIncome(Person other) {
        return this.salary == other.getIncome();
    }
}
```

```

    public double getIncome() {
        return salary;
    }
}

```

Consider also the following subclass of `Person`, a person with a spouse, which takes the salary of the spouse into account as well:

```

public class MarriedPerson extends Person {
    private double spouseSalary;

    public MarriedPerson(double salary, double spouseSalary) {
        super(salary);
        this.spouseSalary = spouseSalary;
    }

    public boolean haveSameIncome(MarriedPerson other) {
        return this.getIncome() == other.getIncome();
    }

    public double getIncome() {
        return ((salary + spouseSalary) / 2);
    }
}

```

A) Show an example with the variables `p1` and `p2`, such that `p1.haveSameIncome(p2)` returns `false`, but `p1.getIncome() == p2.getIncome()` returns `true`. In other words, fill in the following blank with valid code, such that the assertion below is also valid. Do not use reflection and assume that `Person` has no other subclasses.

```

Person p1;
MarriedPerson p2;

```

```

assert (!p1.haveSameIncome(p2) && p1.getIncome() == p2.getIncome());

```

B) Propose changes to `Person` and `MarriedPerson` such that the assertion will fail.

B.1 Can you change **only** `MarriedPerson.haveSameIncome`, such that the assertion will fail for your solution to subtask **A**? If yes, provide the modified method. Otherwise, explain why this is not possible.

B.2 Can you change **only** `Person.haveSameIncome`, such that the assertion will fail for your solution to subtask **A**? If yes, provide the modified method. Otherwise, explain why this is not possible.