CS 61B Spring 2022 — Inheritance and Test Review

Exam Prep Discussion 5: February 14, 2022

Note this worksheet is very long and is not expected to be finished in an hour.

1 Athletes

Suppose we have the `Person`, `Athlete`, and `SoccerPlayer` classes defined below.

```
1  class Person {
2      void speakTo(Person other) { System.out.println("kudos"); }
3      void watch(SoccerPlayer other) { System.out.println("wow"); }
4  }
5
6  class Athlete extends Person {
7      void speakTo(Athlete other) { System.out.println("take notes"); }
8      void watch(Athlete other) { System.out.println("game on"); }
9  }
10
11 class SoccerPlayer extends Athlete {
12     void speakTo(Athlete other) { System.out.println("respect"); }
13     void speakTo(Person other) { System.out.println("hmph"); }
14 }
```

(a) For each line below, write what, if anything, is printed after its execution. Write CE if there is a compiler error and RE if there is a runtime error. If a line errors, continue executing the rest of the lines.

```
1  Person itai = new Person();
2
3  SoccerPlayer shivani = new Person();
4
5  Athlete sohum = new SoccerPlayer();
6
7  Person jack = new Athlete();
8
9  Athlete anjali = new Athlete();
10
11 SoccerPlayer chirasree = new SoccerPlayer();
12
13 itai.watch(chirasree);
14
15 jack.watch(sohum);
16
17 itai.speakTo(sohum);
18
```

```
19  jack.speakTo(anjali);
20
21  anjali.speakTo(chirasree);
22
23  sohum.speakTo(itai);
24
25  chirasree.speakTo((SoccerPlayer) sohum);
26
27  sohum.watch(itai);
28
29  sohum.watch((Athlete) itai);
30
31  ((Athlete) jack).speakTo(anjali);
32
33  ((SoccerPlayer) jack).speakTo(chirasree);
34
35  ((Person) chirasree).speakTo(itai);
```

(b) You may have noticed that `jack.watch(sohum)` produces a compile error. Interestingly, we can resolve this error by **adding casting**! List two fixes that would resolve this error. The first fix should print `wow`. The second fix should print `game on`. Each fix may cast either `jack` or `sohum`.

1.

2.

(c) Now let's try resolving as many of the remaining errors from above by **adding or removing casting**! For each error that can be resolved with casting, write the modified function call below. Note that you cannot resolve a compile error by creating a runtime error! Also note that not all, or any, of the errors may be resolved.

2 Hidden Fruits

Suppose we have the `Fruit` and `Persimmon` and classes defined below.

```
class Fruit {
    String flavor = "generic";
    static char start = 'f';

    static int eat(Fruit fruit) {
        return 1;
    }

    char hats() {
        return this.start;
    }
}

class Persimmon extends Fruit {
    String flavor = "superb";
    static char start = 'p';

    static int eat(Fruit fruit) {
        return 2;
    }

    int eat(Persimmon persimmon) {
        return 3;
    }
}
```

For each line below, write what, if anything, is printed after its execution. Write CE if there is a compiler error and RE if there is a runtime error. If a line errors, continue executing the rest of the lines.

```
Fruit shreyas = new Fruit();
Fruit aram = new Persimmon();
Persimmon eric = new Persimmon();

System.out.println(eric.flavor);
System.out.println(aram.flavor);

System.out.println(eric.eat(shreyas));
System.out.println(eric.eat(eric));
System.out.println(aram.eat(eric));

System.out.println(aram.hats());
System.out.println(eric.hats());
```

3 Containers

a) (1 Points). Suppose that we have the `Container` abstract class below, with the abstract method `pour` and the method `drain`. Implement the method `drain` so that all the liquid is drained from the container, i.e. `amountFilled` is set to 0. Return `true` if any liquid was drained, and `false` otherwise. In other words, return true if and only if there is liquid in the container prior to the function being called. You may add a maximum of **5 lines of code**. Note that the staff solution uses 3. You may *only* add code to the `drain` method. (Summer 2021 MT1)

```
1  public abstract class Container {
2      /* Keeps track of the total amount of liquid in the container */
3      public int amountFilled;
4
5      public boolean drain() {
6
7
8
9
10
11     }  // You may use at most 5 lines of code, i.e. this bracket should be on line 11 or earlier.
12
13     abstract int pour(int amount);
14 }
```

b) (1.5 Points). Finish implementing the `WaterBottle` class so that it **is a** `Container`. You should *only* add code to the blanks, i.e. **fill in the pour method and the class signature**.

As stated in the `Container` class, the `pour` method should pour `amount` into the container and return the amount of the excess liquid, or 0 if there is no excess. For instance, suppose we have a `WaterBottle w` with `capacity` **`10`** and `amountFilled` **`5`**. Then, if we execute `w.pour(7)`, `amountFilled` should be set to **`10`** and **`2`** should be returned. Your solution *must* fit within the blanks provided. You may not need all the lines.

```
1  class WaterBottle _______________ Container {
2      private static final int DEFAULT_CAPACITY = 16;
3
4      /* The capacity of the container, i.e. the maximum amount of liquid the water bottle can hold */
5      private int capacity;
6
7      WaterBottle() {
8          this(DEFAULT_CAPACITY);
9      }
10     WaterBottle(int capacity) {
11         this.capacity = capacity;
12         this.amountFilled = 0;
13     }
```

```
14
15      @Override
16      public int pour(int amount) {
17          ____________________________________;
18          if (_________________________________) {
19              _________________________________;
20              _________________________________;
21              _________________________________;
22          }
23          ____________________________________;
24      }
25  }
```

c) (4 Points). Finally, suppose we have the `ContainerList` class, with the `drainFirst` method as implemented below. Unfortunately, the `drainFirst` method *sometimes* errors!

In order to fix it, you may add code to the **`ContainerList` constructor and the `UnknownContainer`** class! You may only **use** 5 lines of code in the `ContainerList` constructor and **add** 4 lines of code to the `UnknownContainer` class! If you decide to keep or modify the given line in the `ContainerList` constructor, it counts as one of the 5 lines.

Note that, after making your changes, the `drainFirst` should **never error and retain the functionality in the docstring**. You **may not modify the `drainFirst` method!** You may use classes from the previous part assuming they are implemented correctly.

Hint: Make sure that, with your fix, the `drainFirst` method won't error, even if the `drainFirst` method is called many times.

```
1   class UnknownContainer ____________________ {
2       // TODO
3
4
5
6
7
8   } // You may add at most 4 lines of code to the class above
9   // i.e. the closing bracket should be on line 6 or earlier
10
11  class ContainerList {
12      private Container[] containers;
13
14      ContainerList(Container[] conts) {
15          this.containers = conts; // you may delete, modify, or keep this line
16          // YOUR CODE HERE
17
18
```

```
19
20
21
22      } // You may use at most 5 lines of code in the Constructor
23      // i.e. the closing bracket should be on line 18 or earlier
24
25      /* Drains the water from the first nonempty container */
26      void drainFirst() {
27          int index = 0;
28          while (!containers[index].drain()) {
29              index += 1;
30          }
31      }
32  }
```

The following two problems are very challenging, and we only recommend attempting after finishing the rest of the worksheet.

4 Challenge: Frauds List

(6 Points). Suppose we have the `IntList` and `FraudsList` classes below (Summer 2021, Final)

```
1  public class IntList {
2      public int first;
3      public IntList rest;
4
5      public IntList(int f, IntList r) {
6          first = f;
7          rest = r;
8      }
9
10     public int size() {
11         IntList p = this;
12         int totalSize = 0;
13         while (p != null) {
14             totalSize += 1;
15             p = p.rest;
16         }
17         return totalSize;
18     }
19 }
20
21 class FraudList extends IntList {
22     public FraudList(int f, IntList r) {
23         super(f, r);
24     }
25     public int size() {
26         return -super.size();
27     }
28 }
```

Implement the method `findFrauds` which accepts an array of `IntLists` in which some of the elements are, or may contain, `FraudLists`! That is, the dynamic type of certain `IntList` instances is `FraudList`. As shown above, a `FraudList` is an `IntList` whose size method returns the negative of the correct size. You must report these `FraudLists` by **non-destructively** returning a **new** `FraudList` of all the `FraudList` instances linked together in the order they appear in `arr`.

You may **not** modify the given array `arr` or the `IntLists` inside of `FraudList`. You may **not** use `instanceOf`, `getClass()`, `isInstance()` or any method not explicitly written in the classes above or imported. An instance of the problem is shown below:

```
IntList first = new IntList(1000, new IntList(1002, new FraudList(1, new FraudList(2, null))));
IntList second = new FraudList(3, null);
IntList third = new IntList(3000, null);
IntList fourth = new FraudList(4, new IntList(231, new FraudList(5, null)));
IntList[] arr = new IntList[]{first, second, third, fourth};
FraudList frauds = findFrauds(arr);
```

After executing the lines above, `frauds` should be equal to the `FraudList` with the elements `1, 2, 3, 4, 5` and **`arr`, as well as the contents within `arr`, should be unchanged**. Fill in the skeleton below. You may not delete, modify, or add to any of the provided skeleton code.

```
import static java.lang.System.arraycopy;

public static FraudList findFrauds(IntList[] arr) {
    IntList[] copy = new IntList[arr.length];
    arraycopy(arr, 0, copy, 0, arr.length);
    return helper(____________, ____________);
}

public static FraudList helper(IntList[] copy, int index) {
    if (___________________________________) {
        return null;
    } else if (_____________________________) {
        return ______________________________;
    }
    ___________________________________________;
    ___________________________________________;
    if (___________________________________) {
        return ______________________________;
    } else {
        return ______________________________;
    }
}
```

5 Challenge: A Puzzle

Consider the **partially** filled classes for `A` and `B` as defined below:

```
1  public class A {
2      public static void main(String[] args) {
3          ___ y = new ___();
4          ___ z = new ___();
5      }
6
7      int fish(A other) {
8          return 1;
9      }
10
11     int fish(B other) {
12         return 2;
13     }
14 }
15
16 class B extends A {
17     @Override
18     int fish(B other) {
19         return 3;
20     }
21 }
```

Note that the only missing pieces of the classes above are static/dynamic types! Fill in the **four** blanks with the appropriate static/dynamic type — `A` or `B` — such that the following are true:

1. `y.fish(z)` equals `z.fish(z)`
2. `z.fish(y)` equals `y.fish(y)`
3. `z.fish(z)` does not equal `y.fish(y)`