CS 61B Discussion 5: Exam Prep Spring 2020

1 Playing with Puppers

Suppose we have the Dog and Corgi classes which are a defined below with a few methods but no implementation shown. (modified from Spring '16, MT1)

```
1  public class Dog {
2      public Dog(){ /* D1 */ }
3      public void bark(Dog d) { /* Method A */ }
4  }
5
6  public class Corgi extends Dog {
7      public Corgi(){ /* C1 */ }
8      public void bark(Corgi c) { /* Method B */ }
9      @Override
10     public void bark(Dog d) { /* Method C */ }
11     public void play(Dog d) { /* Method D */ }
12     public void play(Corgi c) { /* Method E */ }
13 }
```

For the following main method at each call to play or bark, circle the options corresponding to the methods that will be executed at **runtime**. If there will be a compiler error or runtime error, circle that instead.

```
public static void main(String[] args) {
    Corgi c = new Corgi();
    C1   D1
    Dog d = new Corgi();
    C1   D1
    There is always an implicit call to the superclass's constructor.

    Dog d2 = new Dog();
    D1
    Corgi c2 = new Dog();
    Compiler-Error
    Corgi c3 = (Corgi) new Dog();
    Runtime-Error    D1
    During compile time, we can cast an object along a class's heirarchy with
       no problem. At runtime, java is upset that the Dog instance "is not"
       a Corgi. That is, a Dog does not extend from Corgi. However, the dog
       is instantiated before java attempts to assign it.

    d.play(d);
    Compiler-Error
    d.play(c);
    Compiler-Error
    d's static type Dog does not have a play method.

    c.play(d);
    D
    At compile time, we check c's static type, Corgi, does have a play method
       that takes in a Dog. At runtime, we look at c's dynamic type, Corgi,
```

```
        for a play method. Here we see play is overloaded, so we pick the
        method with the "more specific" parameters relative to our arguments,
        which is method D.
    c.play(c);
    E
    Same as previous.

    c.bark(d);
    C
    c.bark(c);
    B
    d.bark(d);
    C
We notice that bark is overloaded and overriden. As a reminder, dynamic method
selection applies to overriden methods. Method C overrides Method A, and
    method B
overloads C. For c.bark(c), the compiler had bound caller c's static type's
    bark to
argument c's most specific static type, Corgi, thus binding method B.

    d.bark(c);
    C
    d.bark((int) c);
    Compiler-Error
During compile time, the compiler will complain that a Corgi "is not" an int.
You can only cast up or down the heirarchy.

    c.bark((Corgi) d2);
    Runtime-Error
During compile time, we check c's static type, Corgi, for a bark method that
    takes in a Corgi, which exists, so there is no compile time error. At
    runtime, java is upset that d2 "is not" a Corgi. Note that the cast only
    temporarily changes the static type for this SPECIFIC line.

    ((Corgi)d).bark(c);
    B
    ((Dog) c).bark(c);
    C
    c.bark((Dog) c);
    C
}
```

We encourage you to try inheritance problems here: link. Please post on piazza if you have questions!

General flow for one argument methods, suppose we have `a.call(b)` [ST = Static type, DT = dynamic type].

1. During compile time, java only cares about static types. First, check if a's ST, or its superclasses, has a method that takes in the ST of b.
   (a) If not, check a's superclasses for a method that takes in ST of b.
   (b) If not, check if any of the methods take in supertype of ST of b, as we are looking for b's "is-a" relationships. Start from a's ST methods and move up from its superclass.
   (c) If still not, Compiler-Error!
2. Take a snapshot of the method found.
   (a) The method **signature** that is choosen at runtime will try to exactly match with our snapshot. The signature consists of the method name, and the number and type of its paramaters.
3. During runtime, if `call` is an overriden method, then run a's dynamic type's `call` method.
4. Runtime errors can consist of downcasting (as seen in `Corgi c3 = (Corgi) new Dog();`), but also many that are not related to inheritance (NullPointerException, IndexOutOfBoundException, etc).

Notes:

- If a method is overloaded and overriden, as bark is above, the compiler will bind the method first.

- Dynamic method selection has no interaction with assignment.

2 Dynamic Method Selection

Modify the code below so that the max method of DMSList works properly. Assume all numbers inserted into DMSList are positive, and we only insert between `sentinel` and `sentinel.tail`. You may not change anything in the given code. You may only fill in blanks. You may not need all blanks. (Adapted from Spring '17, MT1)

```
1  public class DMSList {
2      private IntList sentinel;
3      public DMSList() {
4          sentinel = new IntList(-1000, new LastIntList());
5      }
6      public class IntList {
7          public int head;
8          public IntList tail;
9          public IntList(int h, IntList t) {
10             head = h;
11             tail = t;
12         }
13         public int max() {
14             return Math.max(head, tail.max());
15         }
16     }
17     public class LastIntList extends IntList  {
18         public LastIntList() {
19             super(0, null);
20         }
21         @Override
22         public int max()   {
23             return 0;
24         }
25     }
26     /* Returns 0 if list is empty. Otherwise, returns the max element. */
27     public int max() {
28         return sentinel.tail.max();
29     }
30 }
```

3 Flirbocon

Consider the declarations below. Assume that `Falcon` extends `Bird`. (Spring '17, MT1)

```
Bird bird = new Falcon();
Falcon falcon = (Falcon) bird;
```

Consider the following possible features for the `Bird` and `Falcon` classes. Assume that all methods are **instance methods** (not static!). The notation `Bird::gulgate(Bird)` specifies a method called `gulgate` with parameter of type `Bird` from the `Bird` class.

F1. The `Bird::gulgate(Bird)` method exists.
F2. The `Bird::gulgate(Falcon)` method exists.
F3. The `Falcon::gulgate(Bird)` method exists.
F4. The `Falcon::gulgate(Falcon)` method exists.

(a) Suppose we make a call to `bird.gulgate(bird);`

Which features are sufficient **ALONE** for this call to compile? For example if feature F3 or feature F4 alone will allow this call to compile, select F3 and F4.

☒ F1 ☐ F2 ☐ F3 ☐ F4 ☐ Impossible

Select a set of features such that this call executes the `Bird::gulgate(Bird)` method. For example, if having features F2 and F4 only (and not F1 and F3) would result in `Bird::gulgate(Bird)` being executed, only select F2 and F4.

☒ F1 ☐ F2 ☐ F3 ☐ F4 ☐ Impossible

Select a set of features such that this call executes the `Falcon::gulgate(Bird)` method.

☒ F1 ☐ F2 ☒ F3 ☐ F4 ☐ Impossible

(b) Suppose we make a call to `falcon.gulgate(falcon);`

Which features are sufficient **ALONE** for this call to compile?

☒ F1 ☒ F2 ☒ F3 ☒ F4 ☐ Impossible

Select a set of features such that this call executes the `Bird::gulgate(Bird)` method.

☒ F1 ☐ F2 ☐ F3 ☐ F4 ☐ Impossible

Select a set of features such that this call executes the `Bird::gulgate(Falcon)` method.

☐ F1 ☒ F2 ☐ F3 ☐ F4 ☐ Impossible

Select a set of features such that this call executes the `Falcon::gulgate(Bird)` method.

☐ F1 ☐ F2 ☒ F3 ☐ F4 ☐ Impossible

Select a set of features such that this call executes the `Falcon::gulgate(Falcon)` method.

☐ F1 ☐ F2 ☐ F3 ☒ F4 ☐ Impossible