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

```java
class Person {
    void speakTo(Person other) { System.out.println("kudos"); }
    void watch(SoccerPlayer other) { System.out.println("wow"); }
}

class Athlete extends Person {
    void speakTo(Athlete other) { System.out.println("take notes"); }
    void watch(Athlete other) { System.out.println("game on"); }
}

class SoccerPlayer extends Athlete {
    void speakTo(Athlete other) { System.out.println("respect"); }
    void speakTo(Person other) { System.out.println("hmph"); }
}
```

(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.

```java
Person itai = new Person();
SoccerPlayer shivani = new Person();
Athlete sohum = new SoccerPlayer();
Person jack = new Athlete();
Athlete anjali = new Athlete();
SoccerPlayer chirasree = new SoccerPlayer();
itai.watch(chirasree);
jack.watch(sohum);
itai.speakTo(sohum);
jack.speakTo(anjali);
```
Inheritance

anjali.speakTo(chirasree);

sohum.speakTo(itai);

chirasree.speakTo((SoccerPlayer) sohum);

sohum.watch(itai);

sohum.watch((Athlete) itai);

((Athlete) jack).speakTo(anjali);

((SoccerPlayer) jack).speakTo(chirasree);

((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 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)

```java
public abstract class Container {
    /* Keeps track of the total amount of liquid in the container */
    public int amountFilled;

    public boolean drain() {
        // You may use at most 5 lines of code, i.e. this bracket should be on line 11 or earlier.
        return true;
    }

    abstract int pour(int amount);
}
```

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.

```java
class WaterBottle ______________ Container {
    private static final int DEFAULT_CAPACITY = 16;

    /* The capacity of the container, i.e. the maximum amount of liquid the water bottle can hold */
    private int capacity;

    WaterBottle() {
        this(DEFAULT_CAPACITY);
    }

    WaterBottle(int capacity) {
        this.capacity = capacity;
        this.amountFilled = 0;
    }
}
```
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 re-
tain the functionality in the docstring. You may not modify the drainFirst
method! You may use classes from the previous part assuming they are imple-
mented correctly.

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

```java
class UnknownContainer {
    // TODO
}
```
} // You may use at most 5 lines of code in the Constructor
// i.e. the closing bracket should be on line 18 or earlier

/* Drains the water from the first nonempty container */
void drainFirst() {
    int index = 0;
    while (!containers[index].drain()) {
        index += 1;
    }
}
3 Challenge: A Puzzle

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

```java
public class A {
    public static void main(String[] args) {
        ___ y = new ___();
        ___ z = new ___();
    }

    int fish(A other) {
        return 1;
    }

    int fish(B other) {
        return 2;
    }
}

class B extends A {
    @Override
    int fish(B other) {
        return 3;
    }
}
```

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)`