1 Athletes

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

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.

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);
Solution: Here is a video walkthrough of the solution.

```java
Person itai = new Person();
SoccerPlayer shivani = new Person(); // CE
Athlete sohum = new SoccerPlayer();
Person jack = new Athlete();
Athlete anjali = new Athlete();
SoccerPlayer chirasree = new SoccerPlayer();
itai.watch(chirasree); // wow
jack.watch(sohum); // CE
itai.speakTo(sohum); // kudos
jack.speakTo(anjali); // kudos
anjali.speakTo(chirasree); // take notes
sohum.speakTo(itai); // hmph
chirasree.speakTo((SoccerPlayer) sohum); // respect
sohum.watch(itai); // CE
sohum.watch((Athlete) itai); // RE
((Athlete) jack).speakTo(anjali); // take notes
((SoccerPlayer) jack).speakTo(chirasree); // RE
((Person) chirasree).speakTo(itai); // hmph
```

Explanation:

Line 3: Person is a superclass of SoccerPlayer, so it can't be assigned to a variable of type SoccerPlayer. (In general, an object can be assigned to a variable that is the same class or a superclass of it).

Line 13: itai has the same static and dynamic type (Person) and Person.watch is allowed to take in a SoccerPlayer argument, so we use that method and print wow.

Line 15: jack has static type Person and dynamic type Athlete. sohum has static type Athlete (we only care about the static type of arguments). During compile time, we choose Person.watch, which can only take in a SoccerPlayer.
**Inheritance**

*Athlete* is a superclass of *SoccerPlayer*, so this method cannot take in an *Athlete* and a compilation error results.

**Line 17**: *sohum* has static type *Athlete*, and *itai* has static and dynamic type *Person*, so we must use *Person.speakTo*. *Person.speakTo* takes in a *Person*, a superclass of *sohum*’s type, so this method works.

**Line 19**: *jack* has static type *Person*, dynamic type *Athlete*. *anjali* has static type *Athlete*. During compile time, we choose the method signature *speakTo(Person other)*. During runtime, we check the class of *jack*’s dynamic type. *Athlete* does not have a method matching our earlier signature, so we use our earlier method and print *kudos*.

**Line 21**: *anjali* has static and dynamic type *Athlete*. *chirasree* has static type *SoccerPlayer*. The only method we can use is *Athlete.speakTo*. This is fine because *SoccerPlayer* is a subclass of *Athlete*, so we print *take notes*.

**Line 23**: *sohum* has static type *Athlete* and dynamic type *SoccerPlayer*. *itai* has static type *Person*. During compilation, we first go to *Athlete*. However, *Athlete.speakTo* cannot take in a *Person*, so we go to it’s parent, *Person*, and choose the signature *speakTo(Person other)*. Then, during runtime, we check *sohum*’s dynamic type, *SoccerPlayer*. *SoccerPlayer.speakTo(Person other)* matches our signature, so we use that method and print *hmph*.

**Line 25**: *chirasree* has static and dynamic type *SoccerPlayer*. We call the *SoccerPlayer.speakTo* method with an argument of type *SoccerPlayer*, which selects the most specific signature possible—*SoccerPlayer.speakTo(Athlete other)*, printing *respect*.

**Line 27**: *sohum* has static type *Athlete* and dynamic type *SoccerPlayer*. *itai* has static type *Person*. During compile time, we go to *Athlete*, but *Athlete.watch(Athlete other)* cannot handle an argument of type *Person*, so we go to its parent. However, *Person.watch(SoccerPlayer other)* also cannot handle an argument of type *Person*, so this results in a compilation error.

**Line 29**: The compiler "trusts" that the cast of *itai* to *Athlete* is correct; however, during runtime, casting a *Person* to an *Athlete* fails, resulting in a runtime exception.

**Line 31**: By casting, we tell the compiler to view *jack*’s static type as *Athlete*. Thus, during compilation, we choose the signature *Athlete.speakTo(Athlete other)*. Then, during runtime, *jack* has dynamic type *Athlete*, so the cast is valid, and we print *take notes*.

**Line 33**: *Jack* has dynamic type *Athlete*, which cannot be downcast to a subclass *SoccerPlayer*.

**Line 35**: During compilation, we treat *chirasree* as a *Person* and choose the method signature *speakTo(Person other)*. Then, during runtime, we see that *chirasree* has dynamic type *SoccerPlayer*, so we choose the *SoccerPlayer.speakTo(Person other)* method that matches our earlier signature, and print *hmph*.

(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. To print **wow**, we can cast `sohum` as a `SoccerPlayer`, resulting in the function call `jack.watch((SoccerPlayer) sohum);`

2. To print **game on**, we can cast `jack` as an `Athlete`, resulting in the function call `((Athlete) jack).watch(sohum);`

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.

**Solution:**

```java
jack.speakTo(chirasree);
```

**Explanation:** This resolves the casting error on line 33. `jack` has static type `person`, and `Person.speakTo(Person other)` can handle an argument of type `SoccerPlayer`, so no compilation errors are produced either.
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 amountFilled > 0;
        amountFilled = 0;
    }

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

Solution: Here is a video walkthrough of the solution.

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

    public boolean drain() {
        boolean answer = amountFilled > 0;
        amountFilled = 0;
        return answer;
    }

    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
Inheritance

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;
    }

    @Override
    public int pour(int amount) {
        // Solution: Here is a video walkthrough of the solution.
        filled += amount;
        if (filled > capacity) {
            // Solution: Here is a video walkthrough of the solution.
        }
    }
}
```

```java
class WaterBottle extends 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;
    }

    @Override
    public int pour(int amount) {
        filled += amount;
        if (filled > capacity) {
```
```java
    int excesss = filled - capacity;
    filled = capacity;
    return excesss;
  }
}

return 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 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.
```
```
// 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;
    }
}

Solution: Here is a video walkthrough of the solution.

class UnknownContainer extends WaterBottle {
    @Override
    public boolean drain() {
        return true;
    }
}

class ContainerList {
    private Container[] containers;

    ContainerList(Container[] conts) {
        containers = new Container[conts.length + 1];
        for (int i = 0; i < conts.length; i += 1) {
            containers[i] = conts[i];
        }
        containers[conts.length] = new UnknownContainer();
    }

    /* Drains the first nonempty container */
    void drainFirst() {
        int index = 0;
        while (!containers[index].drain()) {
            index += 1;
        }
    }
}

Explanation: drainFirst cannot handle a case with all empty containers—it keeps incrementing index until it's out of bounds. The solution is to add a Container which can always be drained, the UnknownContainer. Thus, we write an UnknownContainer.drain which always returns true.

However, we can't just override Container, since this will require you to implement both drain and pour (which requires more than 4 lines). Instead, we have to extend WaterBottle.
Then, in ContainerList, we add an extra UnknownContainer to the end of the containers list. However, an array's size cannot be changed, so we have to copy conts, then add the last UnknownContainer.
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)

**Solution:** Here is a video walkthrough of the solutions.

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

**Explanation:** To get to this solution, it’s helpful to write a matrix of possible static/dynamic types, and eliminate ones that don’t work. First note that because of (3), y and z cannot both be static type B; otherwise only B.fish(B other) would ever get called. Also, they cannot both have static type A: method arguments only check static types, so only A.fish(A other) would ever get called, violating (3). Since we know A and B must have different static types, let’s try assigning static
type A to y and static type B to z. (z must also have dynamic type B, since an object’s dynamic type either the same as or a subclass of it’s static type). Checking the result of \( y.fish(z) \), we see that this will choose the method signature \( fish(B\ other) \) inside \( A \) at compile time. However, for \( z.fish(z) \); the compiler goes to \( B \) and chooses \( B.fish(B\ other) \). In order for these two method calls to be equal, the dynamic type of y must be B.

This gives us our final answer: y has static type A, dynamic type B; and z has static and dynamic type B. We check (2) to make sure this works. \( z.fish(y) \) will go to B first, but since B only has a method for \( fish(B\ other) \), we must go to it’s superclass and choose \( fish(A\ other) \) in A at compile time. \( y.fish(y) \) choose the same method, \( A.fish(A\ other) \). During runtime, we check the dynamic type of z, B, which does not have a matching signature, so both these calls return 2 as desired.