Athletes

Here is a video walkthrough of the solution.

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);

anjali.speakTo(chirasree);

sohum.speakTo(itai);

```

```
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);
```

Solution:

```
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`. `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 dyanmic 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 treet `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.

2.

Solution: Here is a video walkthrough of the solutions for this part and the next.

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:

`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.