DMS Comparator

Here is a video walkthrough of the solution.

Implement the Comparator `DMSComparator`, which compares `Animal` instances. An `Animal` instance is greater than another `Animal` instance if its **dynamic type** is more *specific*. See the examples to the right below.

In the second and third blanks in the `compare` method, **you may only use the integer variables predefined** (`first`, `second`, etc), **relational/equality operators** (`==`, `>`, etc), **boolean operators** (`&&` and `||`), **integers**, and **parentheses**.

As a *challenge*, use equality operators (`==` or `!=`) and no relational operators (`>`, `<=`, etc). There may be more than one solution.

```
class Animal {
   int speak(Dog a) { return 1; }
   int speak(Animal a) { return 2; }
}
class Dog extends Animal {
   int speak(Animal a) { return 3; }
}
class Poodle extends Dog {
   int speak(Dog a) { return 4; }
}
```

Examples:

```
Animal animal = new Animal();
Animal dog = new Dog();
Animal poodle = new Poodle();

compare(animal, dog) // negative number
compare(dog, dog) // zero
compare(poodle, dog) // positive number
```

```
1  public class DMSComparator implements ____________________________ {
2
3      @Override
4      public int compare(Animal o1, Animal o2) {
5          int first = o1.speak(new Animal());
6          int second = o2.speak(new Animal());
7          int third = o1.speak(new Dog());
8          int fourth = o2.speak(new Dog());
9
10         if (_____________________________________________________) {
11             return 0;
12
13         } else if (_____________________________________________) {
14             return 1;
15         } else {
16             return -1;
17         }
18     }
19 }
```

Solution:

```
public class DMSComparator implements Comparator<Animal> {

    @Override
    public int compare(Animal o1, Animal o2) {
        int first = o1.speak(new Animal());
        int second = o2.speak(new Animal());
        int third = o1.speak(new Dog());
        int fourth = o2.speak(new Dog());

        if (first == second && third == fourth) {
            return 0;
        } else if (first > second || third > fourth) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

Explanation:

We know that we should return 0 when the dynamic types of `o1` and `o2` are the same. However, just checking `first == second` is insufficient. Consider the case where you have `o1` with dynamic type `Dog` and `o2` with dynamic type `Poodle`. During compilation, both of these will choose the method `speak(Animal a)`. During runtime, `first` will be `3`, since `Dog.speak(Animal a)` overrides `Animal.speak(Animal a)`. `second` will also be 3: `Poodle` does not have a `speak(Animal)` method, so it goes to its superclass `Dog` and finds `Dog.speak(Animal a)`. Thus, we must also check `third == fourth` in the first case.

For the case of returning 1, note that if `o1` is a `Poodle` while `o2` is not, we should return 1. In this case, `fourth = o2.speak(Dog)` will return 4, while `o1.speak(Dog)` will return 1. Thus, we check if `fourth > third`; if it is, `o1` is more specific than `o2`. Then, we consider the case of `o1` being a `Dog` and `o2` being an `Animal`. In this case, `o1.speak(Animal)` will return `3` (since at runtime tye dynamic type `Dog` also has a `speak(Animal)` method) whereas `o2.speak(Animal)` will return 2. This gives us the other condition, `first > second`.

Challenge Solution:

```
public class DMSComparator implements Comparator<Animal> {

    @Override
    public int compare(Animal o1, Animal o2) {
        int first = o1.speak(new Animal());
        int second = o2.speak(new Animal());
        int third = o1.speak(new Dog());
        int fourth = o2.speak(new Dog());
```

```
9
10          if (first == second && third == fourth) {
11              return 0;
12          } else if (third == 4 || (first == 3 && second == 2)) {
13              return 1;
14          } else {
15              return -1;
16          }
17      }
18  }
```

Explanation:

The first if statement is the same as the solution above.

If we reach the second case and `o1` is a `Poodle`, we know `o2` must be a `Dog` or an `Animal` (or we would have returned 0 in the first case). Thus, we can immediately return 1 if `o1` is a `Poodle`. To check if `o1` is a `Poodle`, we can simply check if `third == 4` (since only Poodles can return `4`).

There is one other case in which we should return 1: when `o1` is a `Dog` and `o2` is an `Animal`. If `o1` is a `Dog`, `o2.speak(Animal)` should return 3, so we check if `first == 3`. To check if `o2` is an `Animal`, we ensure that `o2.speak(Animal)` returns `2` (if it had dynamic type `Dog` or `Poodle`, it would use the method in `Dog` which returns 3). Thus, we add the condition `second == 2`.