1 Challenge: A Puzzle

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

```
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.

```
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.