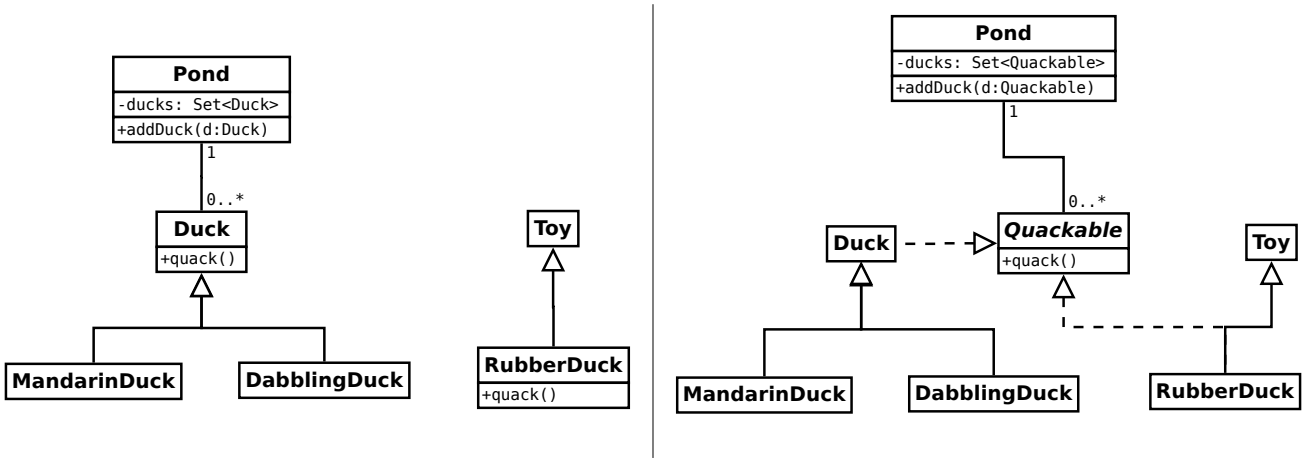
1 Polymorphism

In traditional OOP, polymorphism relies on the inheritance tree to guarantee certain functionality for a given object. Consider the following example, on the left we have inheritance-based polymorphism and on the right we have duck-typing.



With inheritance, a RubberDuck cannot be added to the Pond even though it could quack like a Duck because it doesn't inherit from the Duck class. However, with duck-typing, such functionality guarantees can be delegated to mixins (AKA: interfaces, traits). Now anything that behaves like a duck can be added to the Pond (i.e. can quack). In Ruby and other languages with reflection, you can go one step further by checking for a given functionality at runtime.

2 Ruby vs Java

Duck-typing support in Java is weaker than Ruby because mixins (interface with default methods) does not have access to fields and methods in the implementing class. Compare the FoodBuyer mixin in Ruby and Java below:

```ruby
module FoodBuyer
    def buyFood(foodName, cost)
        puts "Buy #{foodName} and pay $#{cost}."
        pay(cost)
    end
end
```

```java
interface FoodBuyer {
    public Consumer<Integer> pay();
    default public void buyFood(String foodName, int cost, Consumer<Integer> pay) {
        out.printf("Buy %s and pay $%d.\n", foodName, cost);
        pay.accept(cost);
    }
}
```

In the Java mixin, the buyFood method needs an extra 'pay' closure to deligate the pay operation to the implementing class because it does not have direct access unlike Ruby.