PlushMarket

Plush market is a trading place for all the fluffy plush toys.

In the market you can either **sell your plush directly at the current market price** or **trade it for another one**. Each trade can only be done once.

Your goal is to implement functionality, which calculates the **optimal plush toy sales strategy**: - Maximizes the sales price - Minimizes the number of trades

Things to remember (Not entirely relevant for a verbal discussion)

- Work on the assignment as if it was production code worth 10M EUR
- Your answer will be graded on the basis of correctness (it works) and code quality (others can see how it works)
- The solution must be submitted by opening a pull request on the GitHub project repository
- Include tests in your answer
- You are free to use any library from Maven Central
- Pay attention to the aesthetics and maintainability of your code

The Assignment

You need to implement the `PlushMarket` interface to return the optimal strategy.

```
public interface PlushMarket {
    String calculateStrategy(String offerJSON, String marketJSON);
}
```

The interface uses `JSON` format for both input and response. For the exact format of the input and response `JSON` check the examples below.

Examples

Example 1

If the `offerJSON` is

```
{"plush": "Stella"}
```

and `marketJSON` is

```
{"plushes": [
    {"plush": "RedBird", "price": 80},
    {"plush": "Stella", "price": 90},
    {"plush": "Pig", "price": 75}
],
"trades": [
    {"take": "RedBird", "give": "Pig"},
    {"take": "Pig", "give": "Stella"}
]}
```

then the response providing the optimal strategy is

```
{"actions":[
    {"action":"sell","plush":"Stella","price":90}
]}
```

since the sales price of the `Stella` is highest, you cannot improve the price by trading.

Example 2

If the `offerJSON` is

```
{"plush": "RedBird"}
```

and the `marketJSON` is the same as above, then the optimal strategy is

```
{"actions":[
    {"action":"trade","give":"RedBird","take":"Pig"},
    {"action":"trade","give":"Pig","take":"Stella"},
    {"action":"sell","plush":"Stella","price":90}
]}
```

since you can trade your `RedBird` to `Pig` and then trade the `Pig` to `Stella` with sales price of 90 instead of selling it directly at the market price of 80.

Example 3

If the `offerJSON` is

```
{"plush": "BlueBird"}
```

and the `marketJSON` is the same as above, then the optimal strategy is

```
{"actions":[]}
```

since there is no market price nor trades for the `BlueBird` plush.

Example 4

```
{
    "plushes": [
        { "plush": "Chuck", "price": 100 },
        { "plush": "RedBird", "price": 80  },
        { "plush": "Gaia", "price": 80 },
        { "plush": "Stella", "price": 90 },
        { "plush": "Bomb", "price": 85 },
        { "plush": "Pig", "price": 75 }
    ],
        "trades": [
        { "take": "RedBird", "give": "Pig" },
        { "take": "Pig", "give": "Stella" },
        { "take": "Chuck", "give": "Stella" },
        { "take": "Stella", "give": "Bomb" },
        { "take": "Gaia", "give": "RedBird" },
        { "take": "Bomb", "give": "RedBird" }
    ]
}
```

If the `offerJSON` is

```
{"plush": "Stella"}
```

then the optimal strategy would be

```
{
  "actions": [
    { "action": "sell", "plush": "Stella", "price": 90 }
  ]
}
```

While the above are simple indicative examples, any client of such a system can provide wider ranging, more complex markets. Efforts should be made to generalize the solution provided to handle such non-trivial inputs.

Other critical requirements and conventions (not entirely applicable for a verbal solution)

Immutability

A strong recommendation is to keep things immutable; this applies especially to anything returned from any public API, but also internally having mutable state should not be the default behaviour.

We accept mutability if really needed.

Immutables is used for immutable classes.

No nulls

This rule applies everywhere in the code.

Methods must not return or accept nulls. The only exception are 3rd party classes and the usage of those classes must not leak outside the using class.

Internal class communication must always use `Optional.empty()` instead of `null`. This makes it clear when a variable can be missing and the caller can prepare for it. There won't be unnecessary null checks in the code with this policy.

Note, Immutables doesn't allow nulls as variable values. **Lists/sets/maps are empty, not null**.

If the class can be serialized/deserialized to JSON, possibly missing values must be handled as Optional.

There are several good reasons for this policy:

```
APIs should make it clear if something can or cannot be missing. Nullable vari
Optional provides cleaner code with its functional API than null does.
Multiple developers and long running project tend to degenerate into tangle of
This policy works only if it is applied everywhere within the project. There m
```

Add tests

And then test some more

Soft requirements

Apply SOLID principles

Please try the best you can to stick to those, we are obsessed with clean code and we would like to find more people on the same page

Checkout object calisthenics

This is a recent subject also in our team, we think we can get some benefit out of those rules. Do you too?