CS2030

Lab 3

Level 1

Develop the following methods for the `Face` class:

- `Face(int[][] grid)`: the constructor that takes in the `3 × 3` array of integers
- `Face right()`: rotates a quarter revolution to the right and returns a new `Face`
- `Face left()`: rotates a quarter revolution to the left and returns a new `Face`
- `Face half()`: rotates half a revolution returns a new `Face`
- `int[][] toIntArray()`: returns the `3 × 3` integer grid associated with the `Face` object
- `String toString()`: returns a String representing the `Face` object

Level 1

Write a `Cloneable` interface that enforces the definition of the `clone` method. This will be useful for a later level.

How should we define the interface Cloneable?

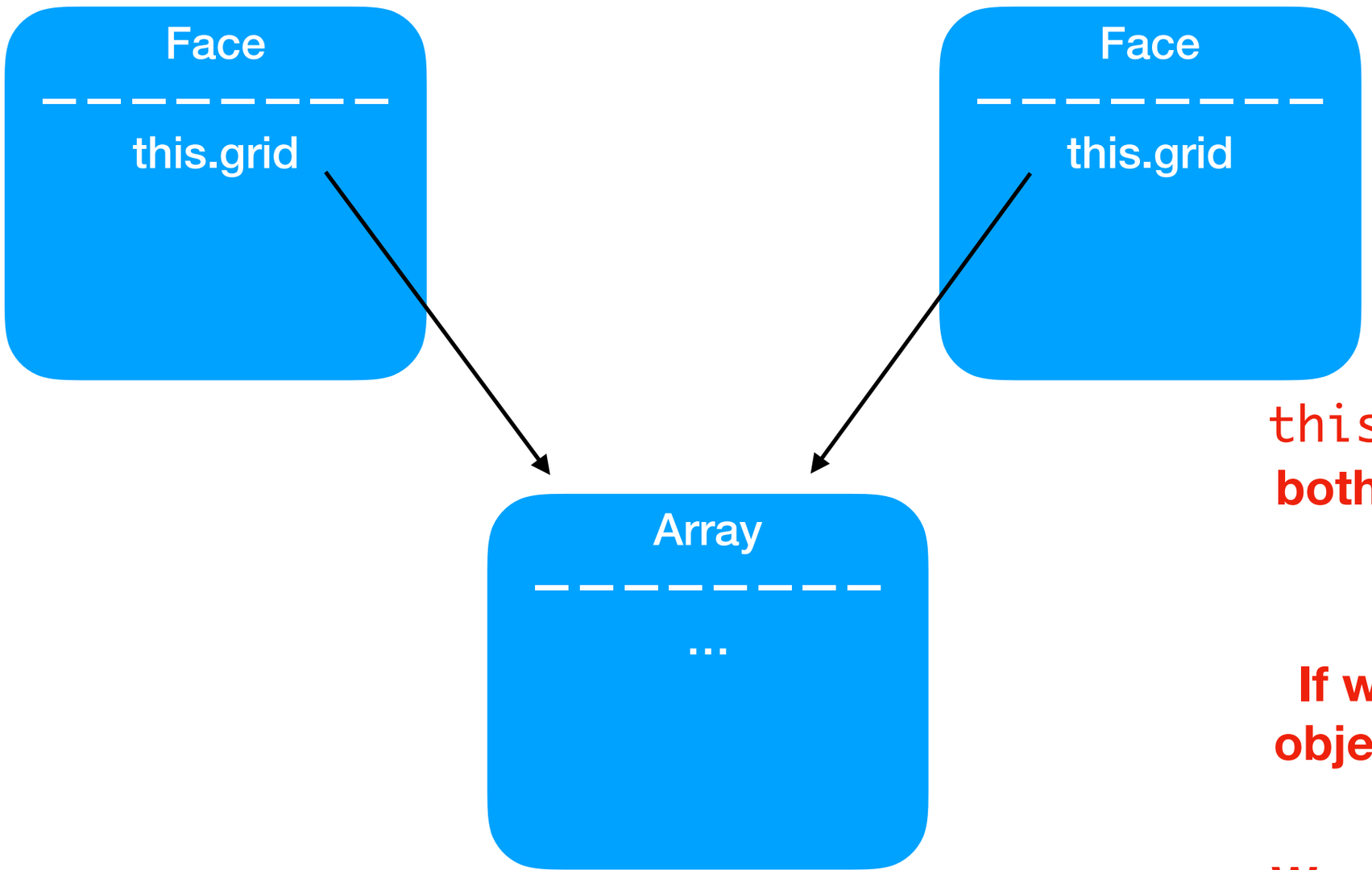
```
interface Cloneable {
  public Cloneable clone();
}

interface Cloneable<A> {
  public A clone();
}

interface Cloneable<A extends Cloneable<A>> {
  public A clone();
}
```

Using generics

(shallow) clone()



Note that `Face` `newFace.grid = this.grid` is not enough, as both will reference the same grid.

If we mutate the grid, both objects will “feel” this effect.

We can’t use a shallow clone to get a copy to do mutation.

shallow copy vs deep copy

from stackoverflow:

1 2 next

696

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated.

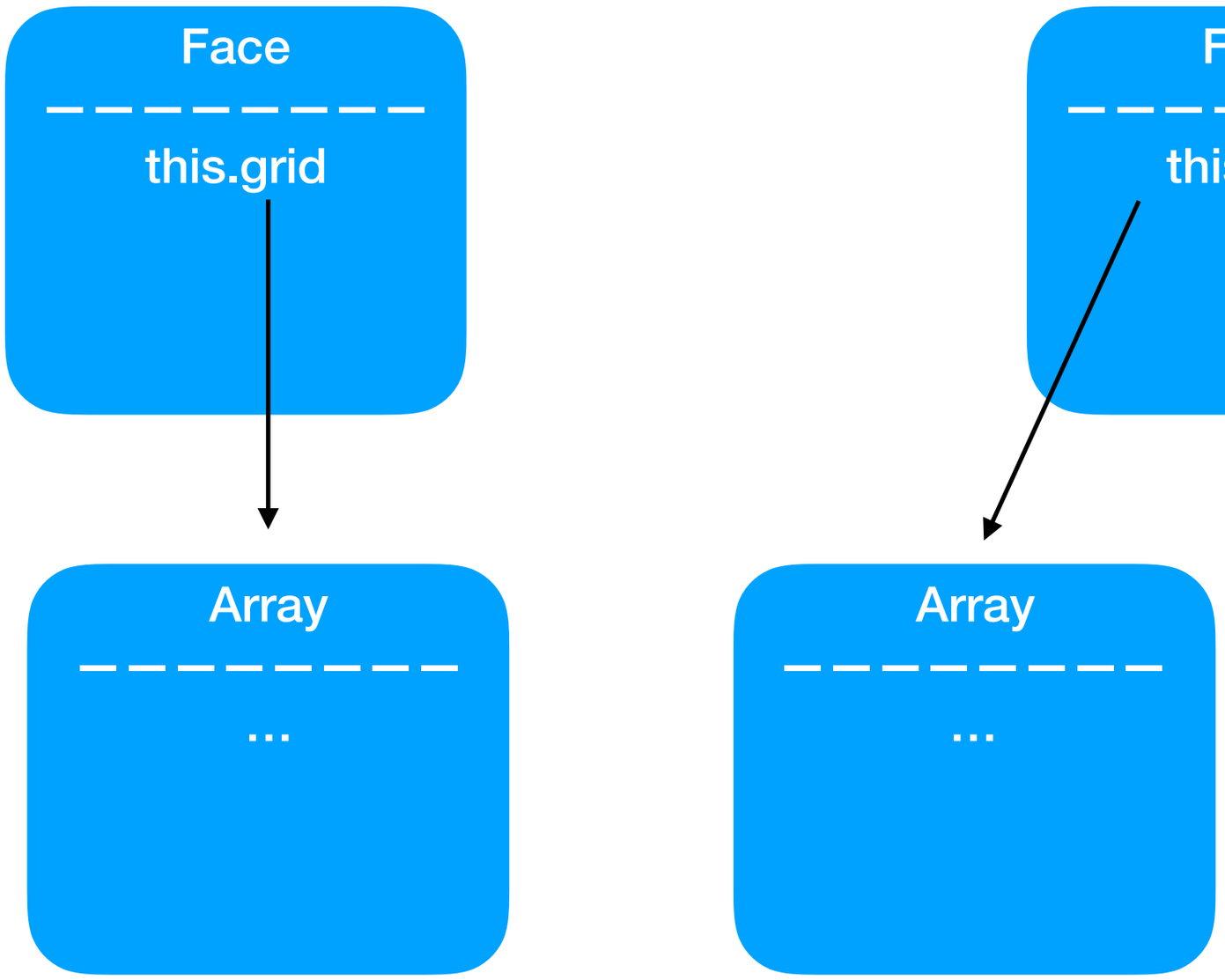
share improve this answer

answered Oct 8 '08 at 20:29

S.Lott

329k ● 71 ● 453 ● 730

(deep) clone()



If our `clone()` worked like this, then we can mutate the clone without affecting the original.

Although we are kind of mutating the cloned object before returning it, in the client's perspective it is still immutable

Level 2

Develop the following methods of the `Rubik` class:

- `Rubik(int[][][] grid)`: the constructor that takes in a three-dimensional (`6 × 3 × 3`) integer array of six face values.
- `String toString()`: returns a String representing the `Rubik` object
- `Rubik right()`: turns the front face clockwise
- `Rubik left()`: turns the front face anti-clockwise
- `Rubik half()`: turns the front face for half a revolution

Implement the `clone()` method for Rubik too!

Level 2

```
jshell> new Rubik(grid)
$.. ==>
......010203......
......040506......
......070809......
101112192021282930
131415222324313233
161718252627343536
......373839......
......404142......
......434445......
......464748......
......495051......
......525354......
```

```
jshell> new Rubik(grid).left()
$.. ==>
......010203......
......040506......
......283134......
101109212427392930
131408202326383233
161707192225373536
......121518......
......404142......
......434445......
......464748......
......495051......
......525354......
```

Realise that the highlighted parts change too,
in addition to the front face.

Level 3

Interface:

```
Rubik implements Cloneable,
    SideViewable { ..
```

As such our rubik must now be made ***side-viewable***. In particular, consider the following methods:

- `Rubik rightView()`: orientates the cube to view the right-side and returns a new `Rubik` object
- `Rubik leftView`: orientates the cube to view the left-side and returns a new `Rubik` object
- `Rubik upView()`: orientates the cube to view the up-side and returns a new `Rubik` object
- `Rubik downView()`: orientates the cube to view the down-side and returns a new `Rubik` object
- `Rubik backView()`: orientates the cube to view the back-side and returns a new `Rubik` object Although one can view the back of the cube by either orientating right/left or up/down, for ease of correctness checking, we stipulate that you can **only orientate right/left**
- `Rubik frontView()`: no orientation needed

Level 3

```
jshell> new Rubik(grid)
$.. ==>
......010203......
......040506......
......070809......
101112192021282930
131415222324313233
161718252627343536
......373839......
......404142......
......434445......
......464748......
......495051......
......525354......
```

```
jshell> new Rubik(grid).rightView()
$.. ==>
......070401......
......080502......
......090603......
192021282930545352
222324313233515049
252627343536484746
......394245......
......384144......
......374043......
......181716......
......151413......
......121110......
```

Note: it's not just shifting faces! Some of them will be rotated.

Level 4

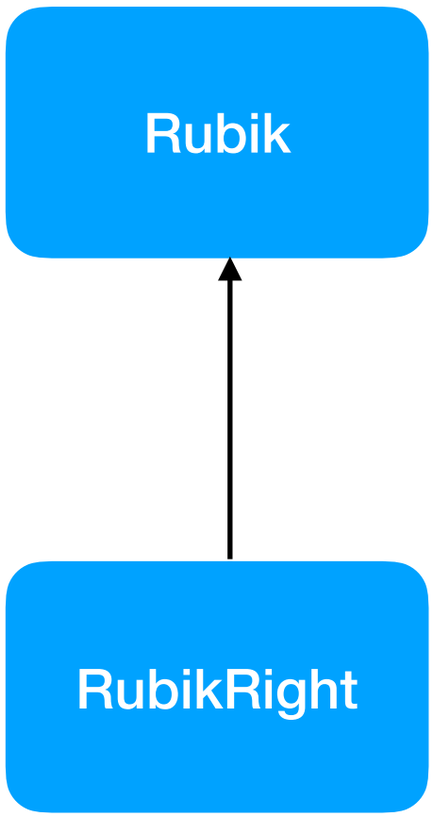
We now extend our implementation to include the classes `RubikRight`, `RubikLeft`, `RubikUp`, `RubikDown`, `RubikBack` s o that they are each responsible for the turns on their respective sides.

What is the relationship between `RubikRight`, `RubikLeft`, etc and `Rubik`?

Tip: use the methods that we have implemented already!

Inheritance

- **is-a** relationship
  - `RubikRight` **is a** `Rubik`
  - `RubikLeft` **is a** `Rubik`
- We model this kind of relationship with inheritance
  - `A extends B`
  - `RubikRight extends Rubik`



- Since `RubikRight` is a `Rubik`, it has all the methods `Rubik` should have
  - `RubikRight` will get all methods for free from `Rubik`
  - `RubikRight` can override methods from `Rubik` too, to define new behaviour

LSP

- Keep in mind to not violate LSP!
- “To use LSP, one must first define the property $phi(S)$ that S satisfies. Take Object::equals for instance, the API specifies some properties, such as reflective, symmetry, etc. As long as the subclass’s equals() satisfies the same set of properties, LSP is maintained.

  The “art” here is to define $phi(S)$ such that it is as general (less restrictive) as possible so that the code that we write makes as few assumptions as possible about the behavior of S. Then the code is easier to extend. (ref. Open-Closed Principle)”

- e.g left() rotates Rubik left. If you override left(), it should still rotate Rubik right (keep in mind RubikRight is a Rubik)

This is a trivial example; it can be sometimes difficult to tell!

Level 4

```
jshell> new Rubik(grid)
$.. ==>
......010203......
......040506......
......070809......
101112192021282930
131415222324313233
161718252627343536
......373839......
......404142......
......434445......
......464748......
......495051......
......525354......
```

```
jshell> new RubikUp(rubik).left()
$.. ==>
......030609......
......020508......
......010407......
545352101112192021
131415222324313233
161718252627343536
......373839......
......404142......
......434445......
......464748......
......495051......
......302928......
```

one example: we do a left rotate for the rubric face on top

Level 5

- Input/Output, just like the previous labs!

Level 6

Implement a class `Dice` that represents a dice object with six sides labeled `U`, `F`, `R`, `B`, `L`, and `D`.

The dice object can be instantiated with a constructor with no arguments, and it is represented by the following output

```
U
FRBL
    D
```

To visualise, if you "fold" along the edges where two characters meet you'll get a dice

A dice is side-viewable!
It should *implement* `SideViewable`, which implies that it has methods declared in the `SideViewable` interface

That’s all!

Wai Lun - wailun@u.nus.edu
Si Mun - simun@u.nus.edu