Structural Design pattern

=> Decorator design pattern :-

Ice Cream, Coffee, Pizza

HTML, Spring App$^n$

=> Flyweight design pattern :

Push

SWE at Vadilal | Kwality walls

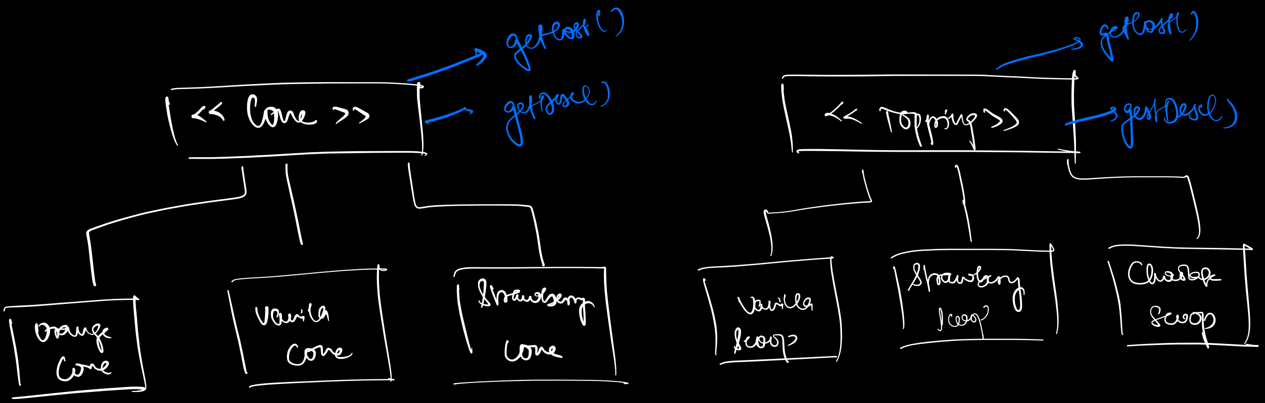
build a ice cream ordering system

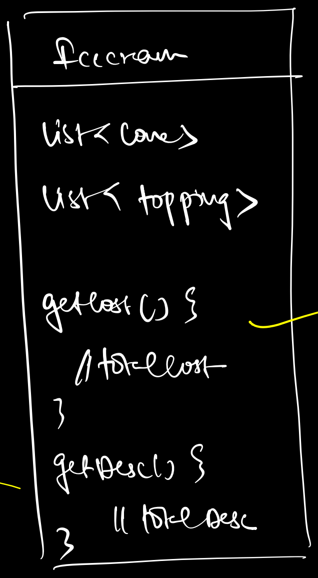
- * Only cone ice-creams
- * App$^n$ takes orders for ice cream cones
- * Ice creams can have custom config
- * Cost of icream
- * Description of icecream

] => needs to printed

ex => Orange Cone + Vanilla Scoop + Chocolate Scoop + Syrup

=> How do you store info. about ice cream ?





will go through all topping & cones and accumulate price

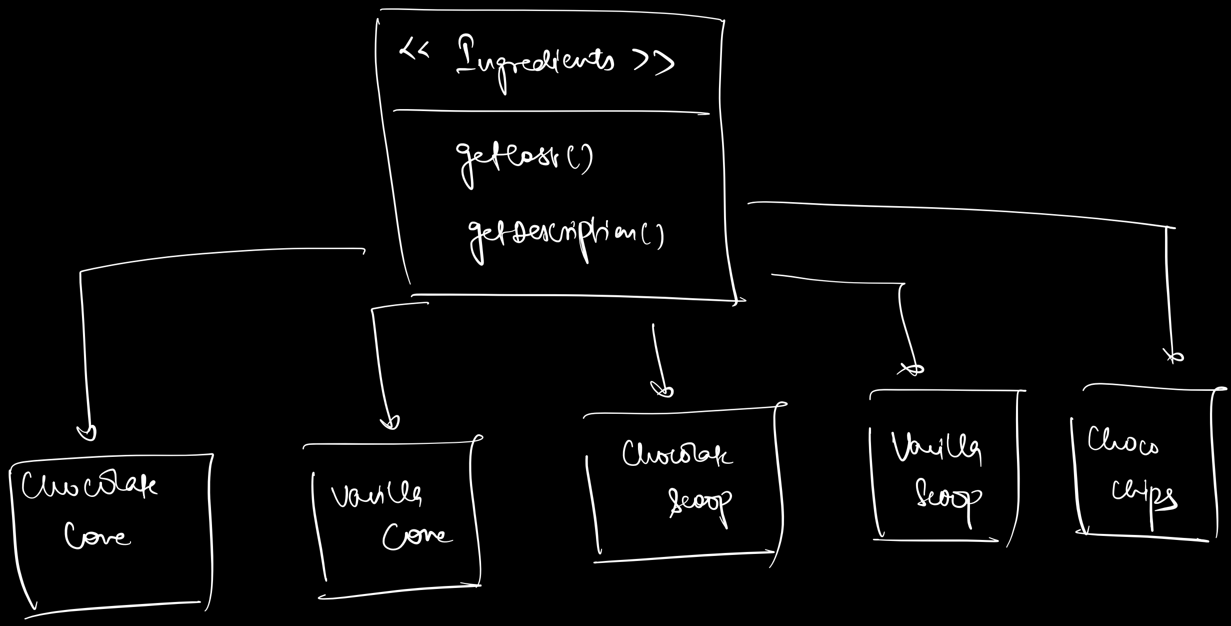
will go through all toppings & cones & accumulate desc.

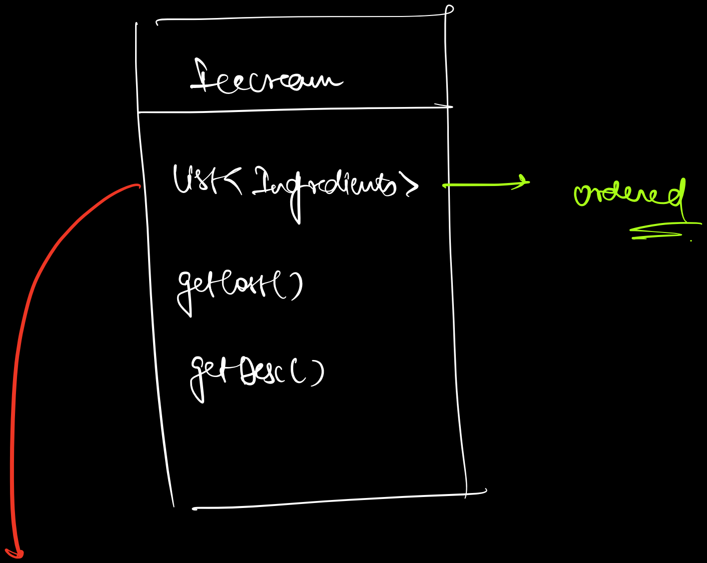
Vanilla Cone + Chocolate Sauce + Chocolate Cone + Vanilla Scoop + Chocolate Scoop

+ Choco Chips.

( Vanilla Cone + Chocochips + Syrup + vanilla sc + choc sc + Cone )

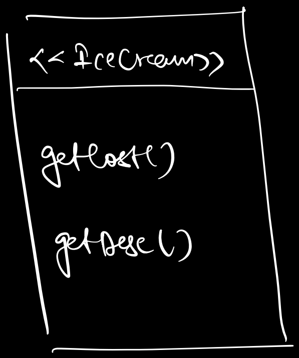
* Order is needed



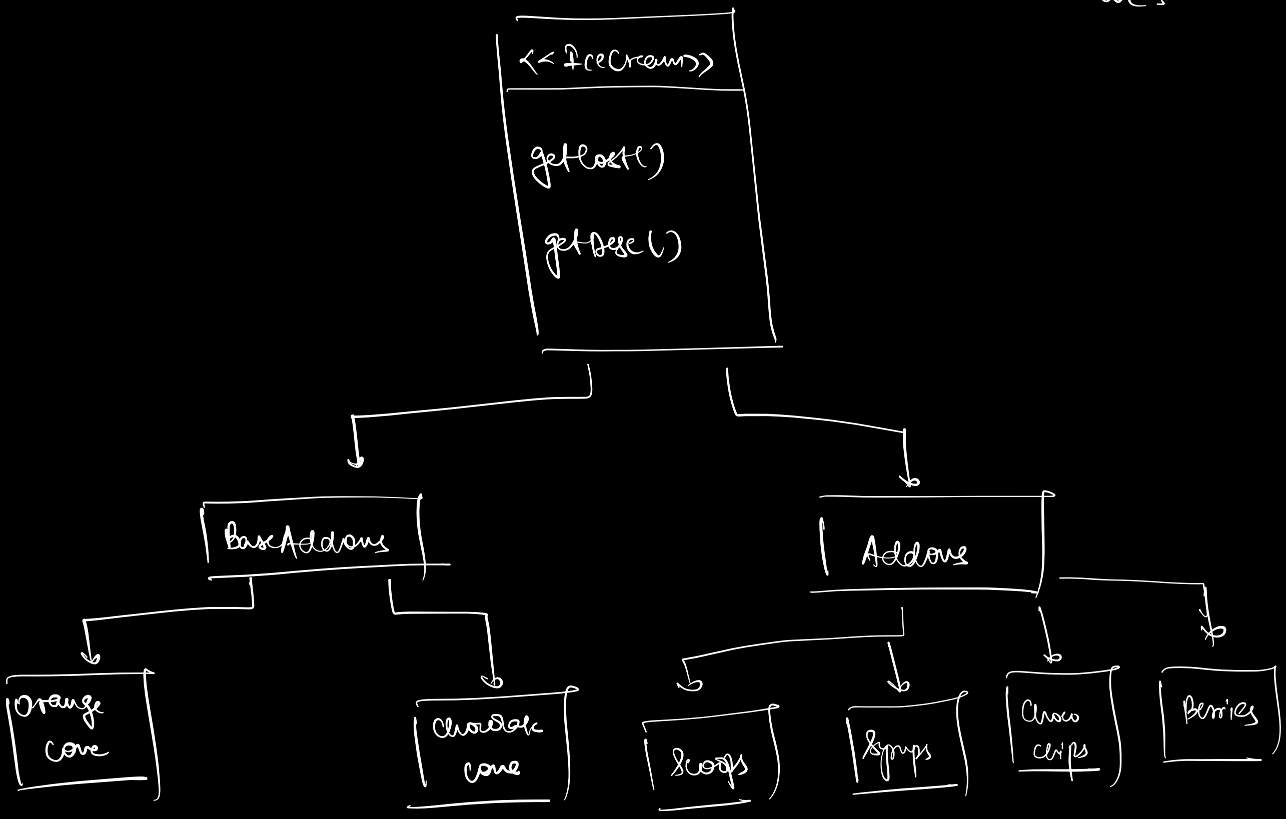


order is preserved as added by user, and since it is a single list, there is no chance of fumbling things up.

S.1 Define an interface/abstract class that represents the thing that we are constructing.



S2. There are 2 types of ingredients that we have:



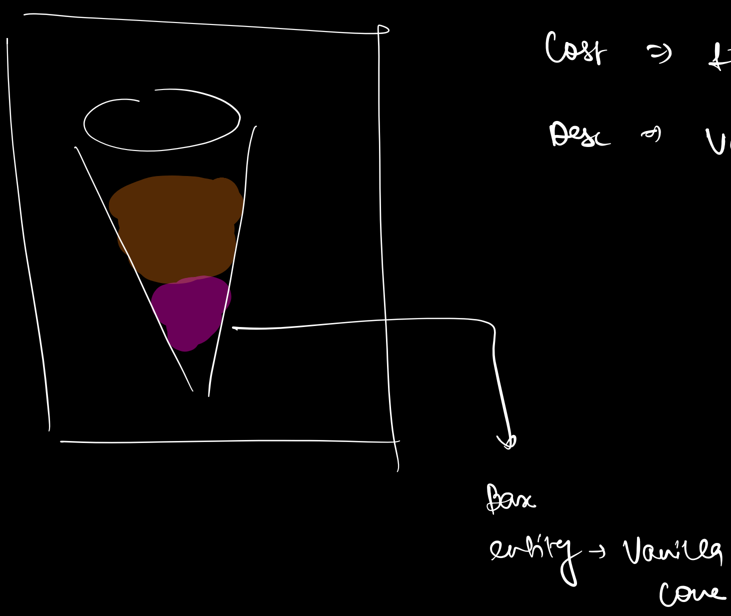
* For each of these individual ingredients, we implement the class

⇒ ONLY A BASE ENTITY :-

getCost() → cost of that base entity

getDesc() → name of that base entity

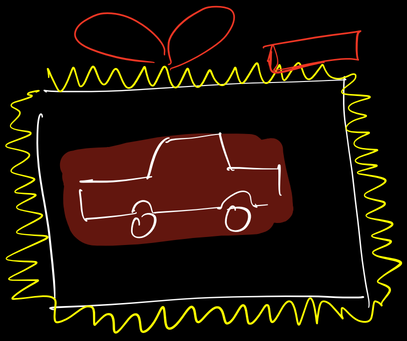
⇒ After an add-on :-

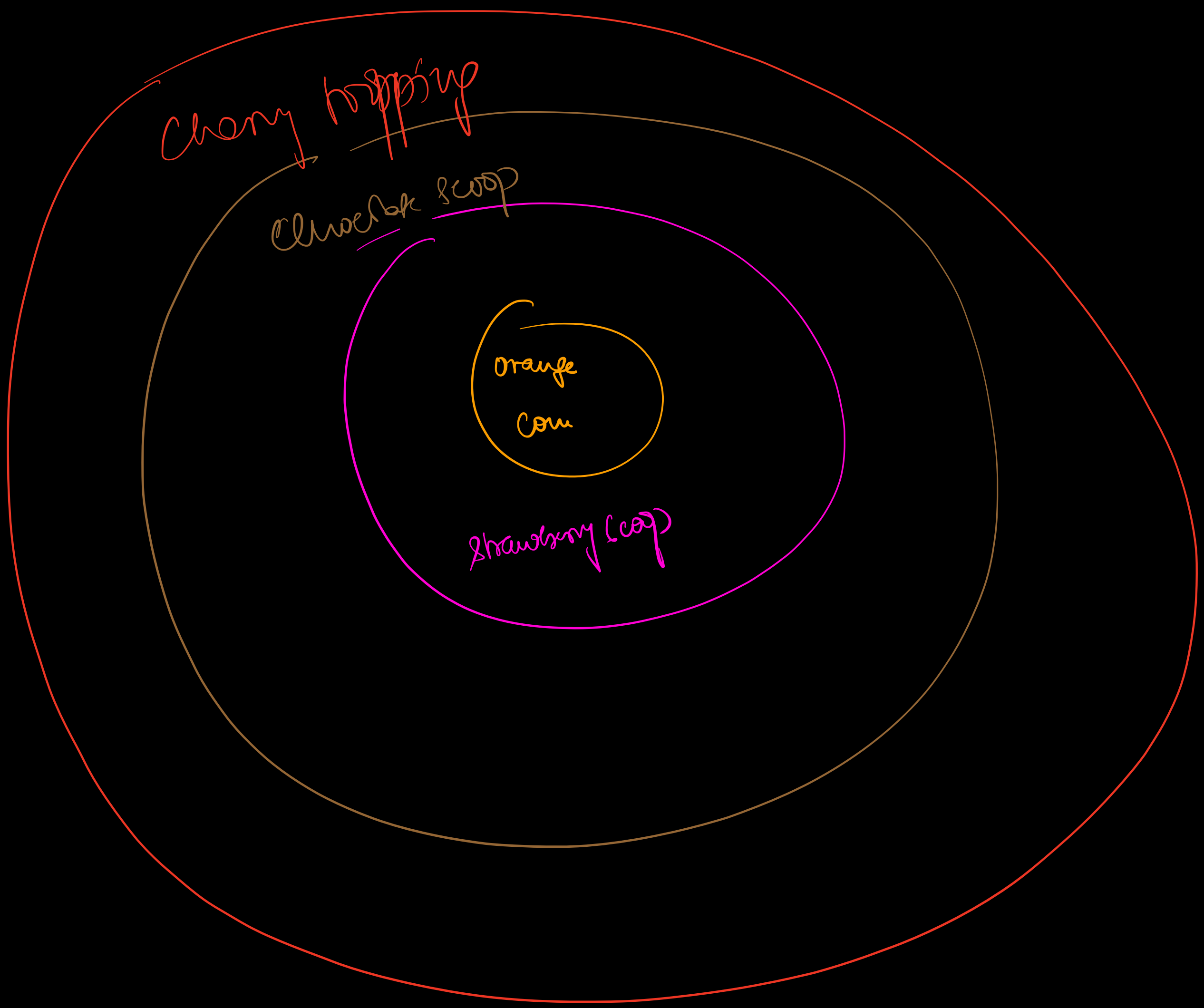


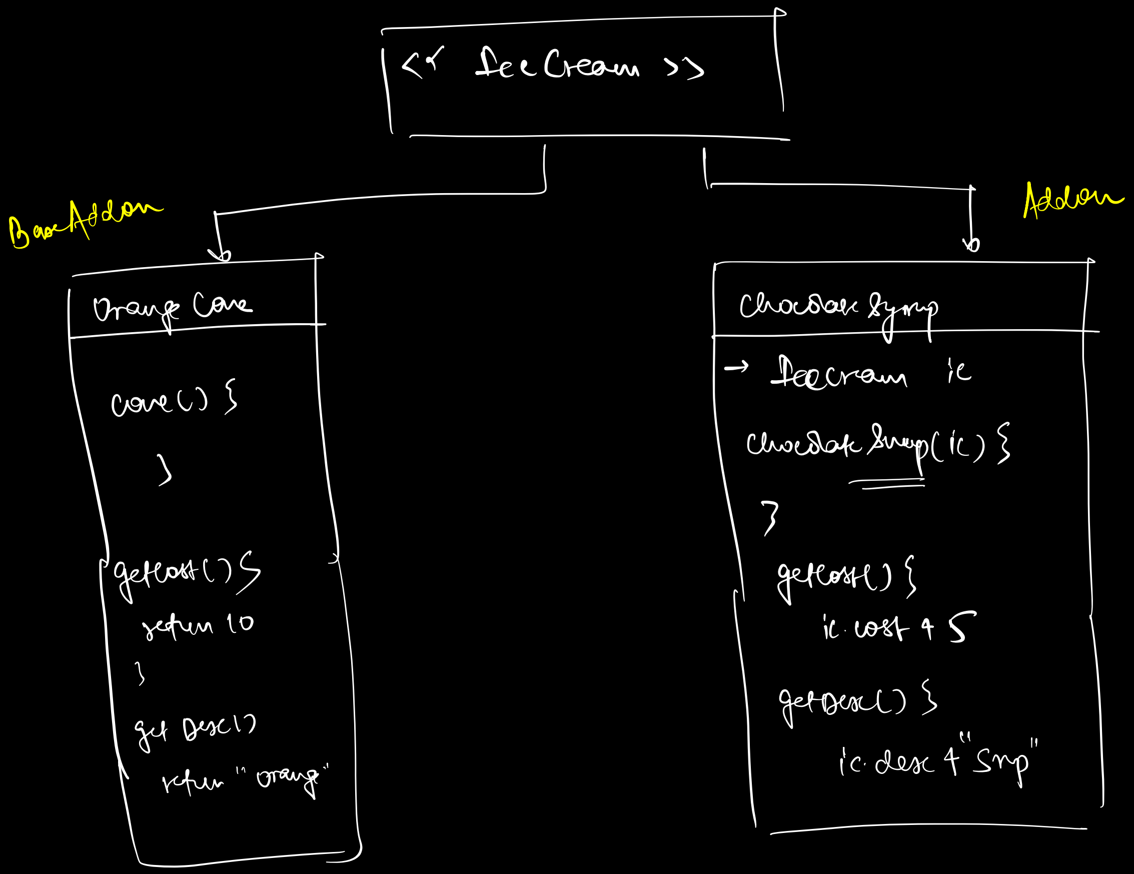
Strawberry   Chocolate
↓   ↓
Cost ⇒ 10 + getCost() + getCost()

Desc ⇒ Vanilla Cone + getDesc() + getDesc
↓   ↓
Strawberry   Choc

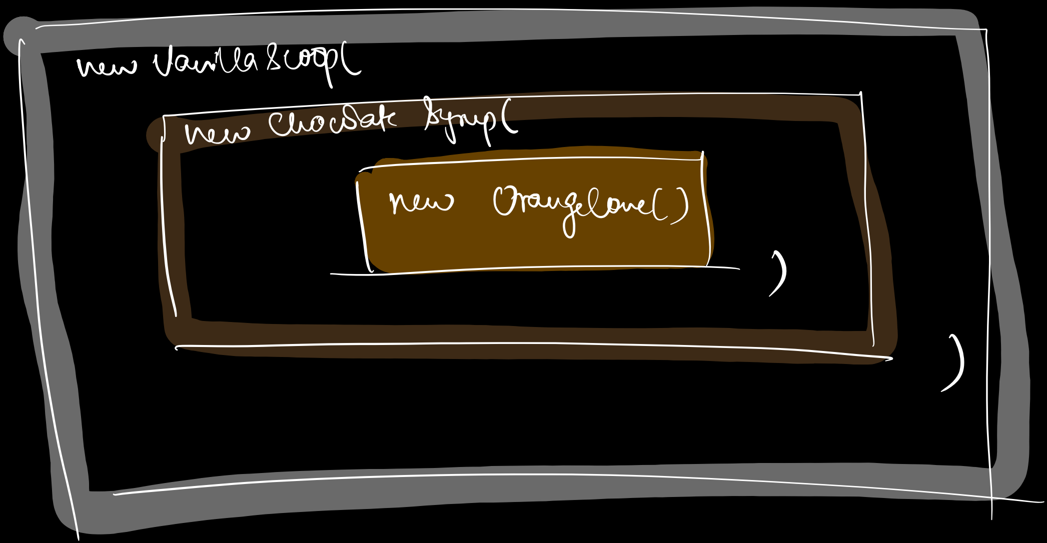
⇒ If we have a scenario where we add properties / features to a base entity at runtime, where the final o/p depends on the o/p of base, consider using decorator design pattern




Cherry topping
chocolate scoop
orange
cone
strawberry scoop



IceCream ic =



```
new VanillaScoop(
    new ChocolateSyrup(
        new OrangeCone()
    )
)
```

```
IceCream ic = new OrangeCone();
  ↓
  (ic.getCost())

  ic = new ChocolateSyrup (new OrangeCone())
  ↓
  ic.getCost()
        ↓
        10 + 5
```

⇒ break : 8:25 PM

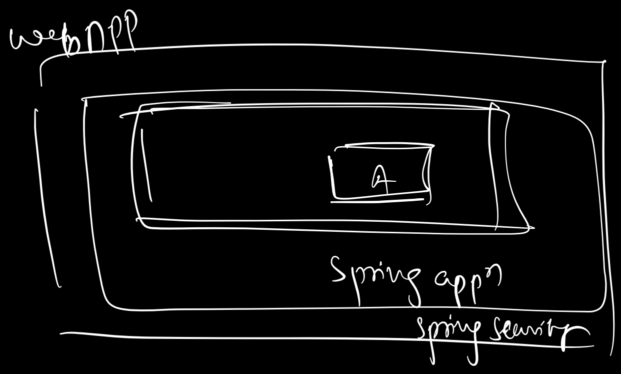
```
<html>
    <body>
        <div>
            <p> </p>

        </div>
    </body>
<</html>
```

```
{
    Paragraph p = new Paragraph("          ");
    Div d = new Div(p);
    Body b = new Body(d)
    HTML h = new HTML(b, h);
    return h.
}
```

@Service          @Component

Decorators

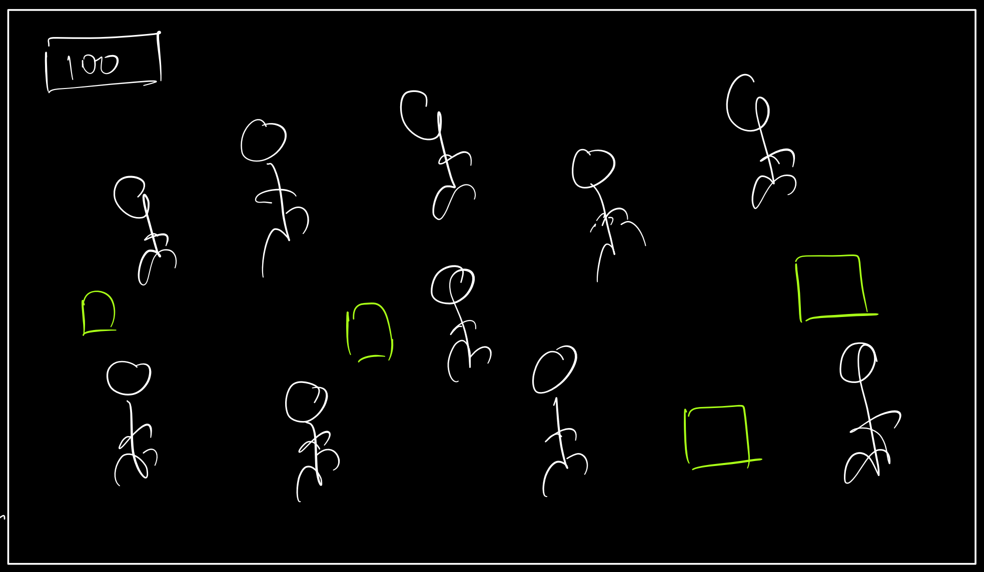
```
@Component
class A {
}
```



lru-cache → python decorator

## : Flyweight Design Pattern:

ex => building an online multiplayer game, ex: Pubg



100 players compete, 1 player remains alive

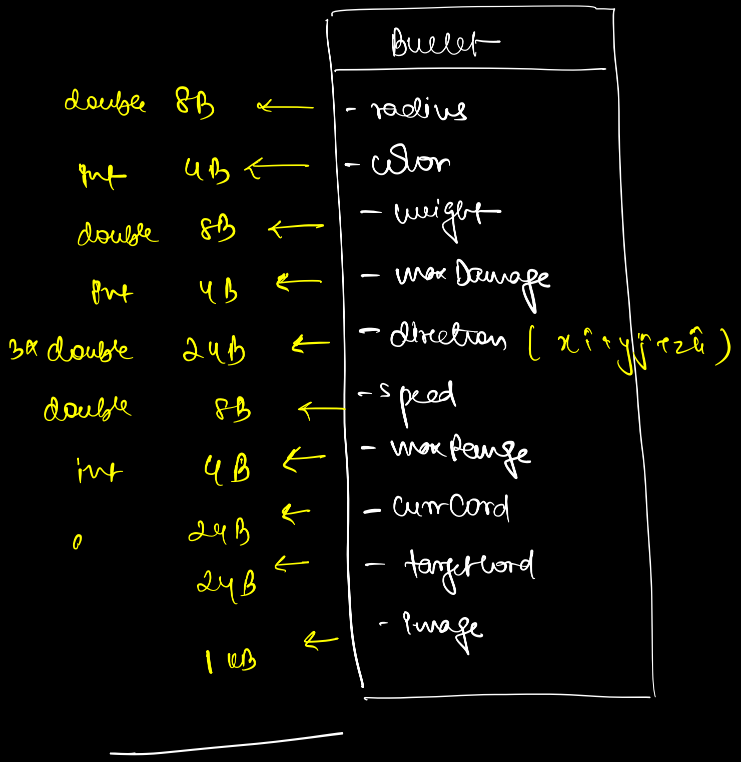
↳ winner

2 guns / player
300 bullets

↳ at start

i) Complete state of the game is downloaded to each player's machine

ii) Changes of the game are transferred to every player



Bullet

| | | |
|---|---|---|
| double | 8B | - radius |
| int | 4B | - color |
| double | 8B | - weight |
| int | 4B | - max Damage |
| 3* double | 24B | - direction ( $x\hat{i} + y\hat{j} + z\hat{k}$ ) |
| double | 8B | - speed |
| int | 4B | - max Range |
| 0 | 24B | - currCord |
| | 24B | - targetCord |
| | 1 kB | - image |

total memory ≈ 1.1kB → per bullet Object

1 game ⇒ 100000 bullets

for every bullet, we created an unique Object

total RAM ⇒ $1.1 \times 10^3 \times 10^5$

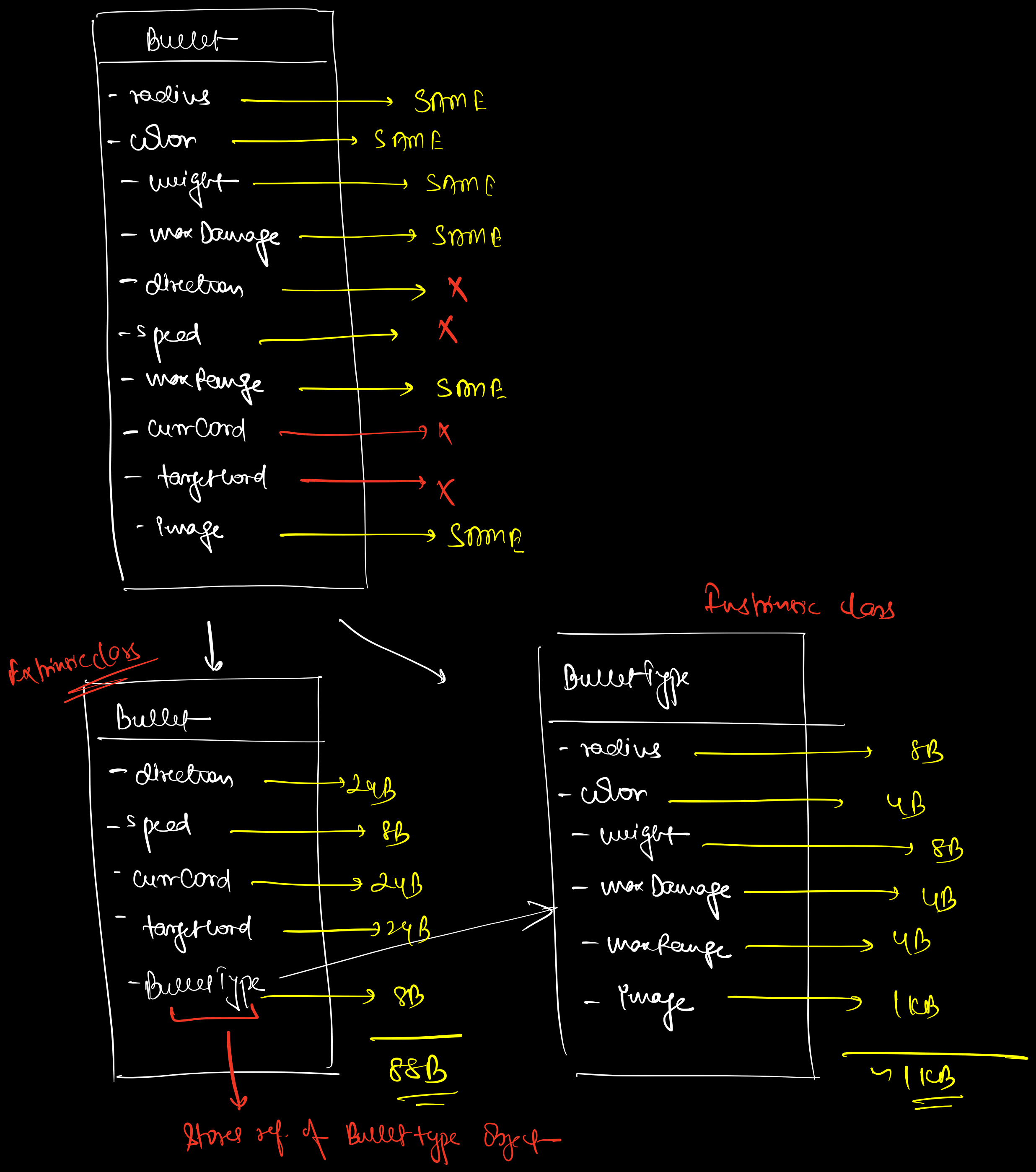
$= 1.1 \times 10^8$

$=$ 0.11 GB

≈ 100 MB

obs even though we have 100k bullets, not all bullets are unique,

PUBG :- has 10 types of bullets -



| Bullet | |
|---|---|
| - radius | SAME |
| - color | SAME |
| - weight | SAME |
| - max Damage | SAME |
| - direction | X |
| - speed | X |
| - max Range | SAME |
| - currCord | X |
| - targetCord | X |
| - image | SAME |

Extrinsic class

| Bullet | |
|---|---|
| - direction | 24B |
| - speed | 8B |
| - currCord | 24B |
| - targetCord | 24B |
| - BulletType | 8B |
| | 88B |

stores ref. of BulletType Object

Intrinsic class

| BulletType | |
|---|---|
| - radius | 8B |
| - color | 4B |
| - weight | 8B |
| - max Damage | 4B |
| - maxRange | 4B |
| - image | 1KB |
| | ~1KB |

=> If you have a class, for which, we might need to create a huge no. of objects, then check if it has a few properties, which are same for a group of objects. In that scenario, we divide the class into 2 parts

i) Extrinsic — Values which are unique / change with time.

ii) Intrinsic — Values which are same / dont change with time

=> PUBG → 10 types of bullet => 10 x 1 KB = 10 KB.

=> total bullets used => 100000 => 88 x $10^5$ B

=> 8 8 000 00

=> 8.8 MB

total space = 8.8 MB + 10 KB

≈ 9 MB

100 MB → 9MB

```
Class Bullet {

        public Bullet( ________, BulletType ) {

        }
}
```