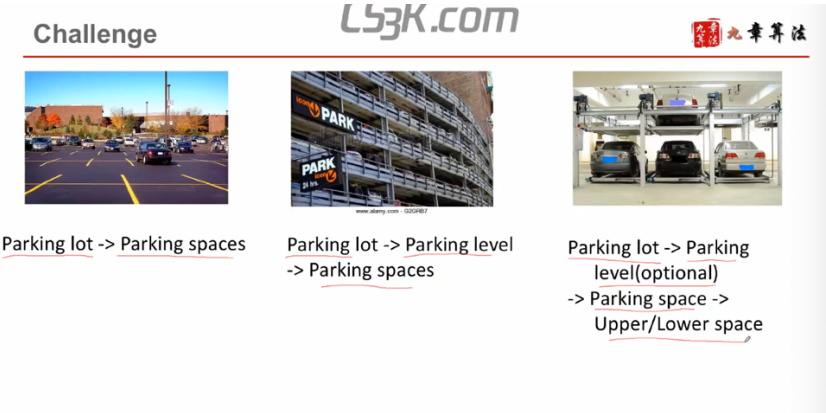
Lecture 2 - 管理类 OOD: Parking Lot & Restaurant - Singleton Pattern

- 管理类 - 题目后面都可以接上三个字: 管理员
  - e.g.: Gym, Parking lot, Restaurant, Library, Super market, Hotel
  - 设计一个模拟/代替管理员日常工作的系统
- 解题方法
  - Clarify: 除了题目中问的名词外，还需要从管理的名词来考
  - Core object: 有进有出
  - Cases: 从管理员角度考虑
    - Reserve: 预定
    - Serve: 服务
    - Checkout: 买单
  - Class: 经常可以使用**收据**的形式，来保管信息
  - Correctness

Clarify

- What
  - 关键词: Parking lot, Vehicle, Parking Spot
  - Parking lot: 考虑多层的 Parking lot, 没有错层



    - Vehicle: 考虑三种大小的车
        - 如何设计停车场支持停不同大小的车？

          两种方案，一种是大车占用两个车位，第二种也可以设计专门的大车停车位，优缺点如下图

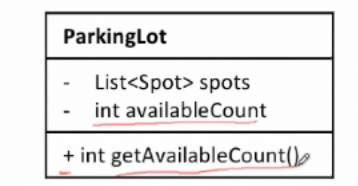
    - 不考虑残疾人停车位/充电车位
- How
    - 规则 1: 如何停车
        - 一定要从**停车场**的角度来考虑,而不是车的角度
            - 停车场: 开进停车场 -> 返回一个能停的地方 ->停进一个位 置
            - 车: 开进停车场 -> 经过每一个位置看看能不能停 -> 停进一个位置
    - 规则 2: 付费
        - 免费还是付费
- Who: Optional

Core Object

- Parking Lot
- Parking spot
- Car, Bus, Motorcycle
- 映射关系
    - Parking has a list of Spot
    - 错误方法: spot 里面有 car,或者 parking lot 有 list of car

Cases - 站在管理员的角度想

- Bus / Car / Motorcycle
  - N/A
- Parking Lot
  - Get available count (reserve)
  - Park vehicle (serve)
  - Clear spot (checkout)
  - Calculate price (checkout)
- Parking Spot
  - N/A



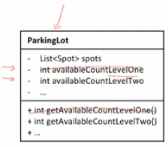
把 availableCount 设成 private 的目的是为了封装，这样外部的类就不能自行修改这个量

别的类可以通过 public 的 getAvailableCount 来获取这个量的值

- 如何分别显示出每一层的空位个数？

不好的方案：

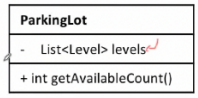
Solution 1: 有几层就保存几个变量

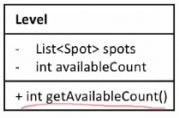


虽然能 work 但是非常低级

- 如何分别显示出每一层的空位个数？

Solution 2: 新建一个Level类



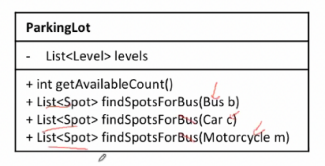


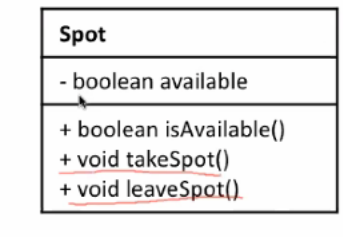
Class

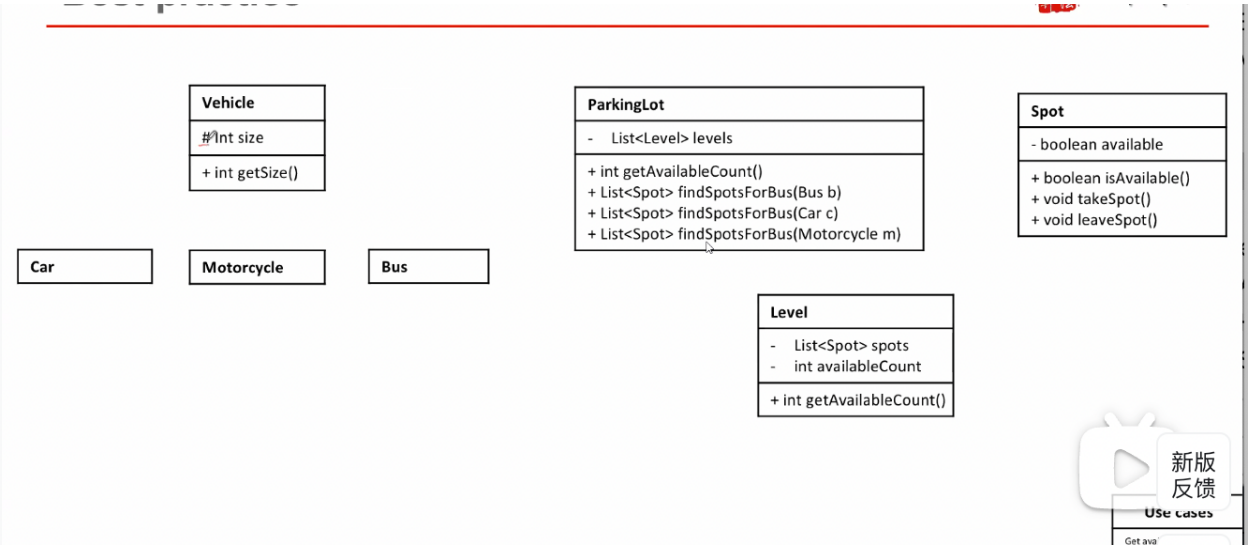
- Use case: Park vehicle

- Parking lot checks the size of vehicle
- Parking lot find an available spot for this vehicle
- Vehicle takes the spot(s)

不好的方案：根据不同车型找 spot，这个 extentability 就会很差



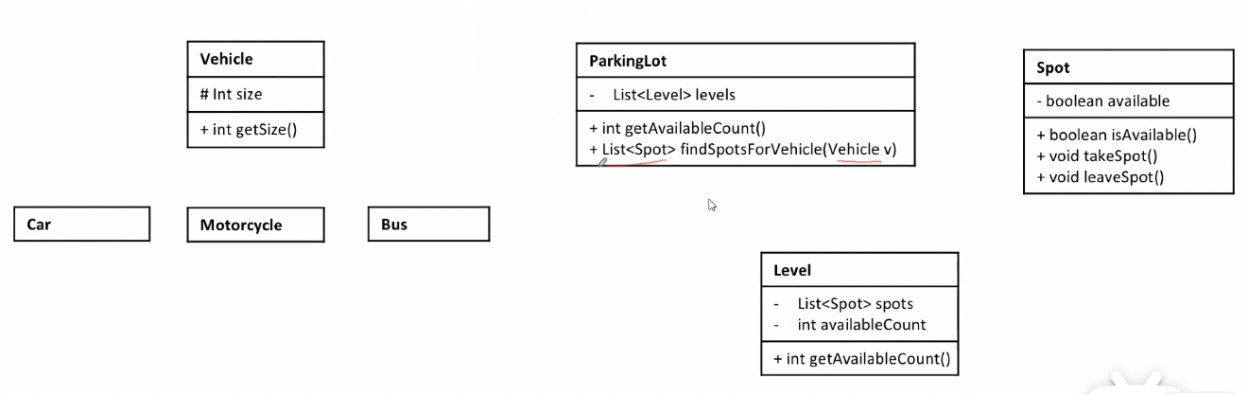




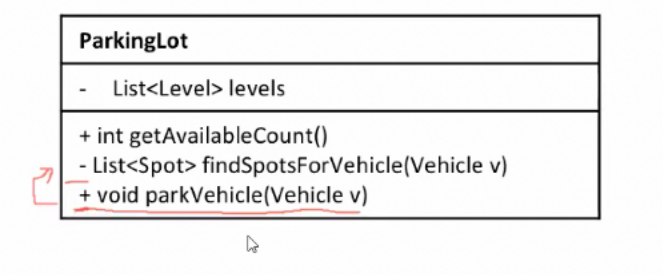
vehicle 的 size 变成#protected

可以看到如果是 private 那么子类里还得再次申明这个变量，但如果是 protect 就不用了

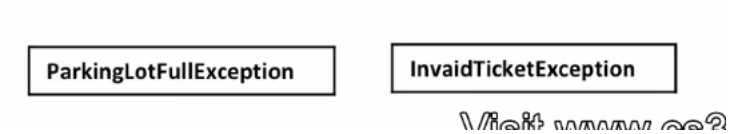
更新版的：



对 parkinglot 进一步的 improve



对于 parkvehicle 怎么知道是不是停成功？可以添加 exception。



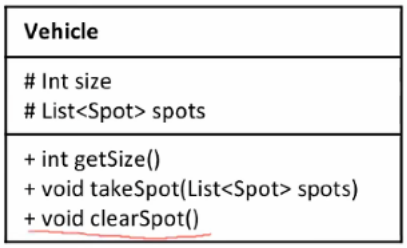
不把它返回值设置成 true false 的原因是因为二元 t f 有不同的解释，并不能代表很多，而且你还得写 doc 来解释，对于产业界不是很好的 practice

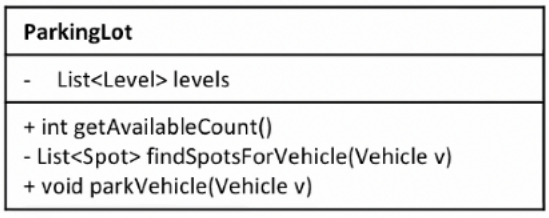
- Use case: Clear spot

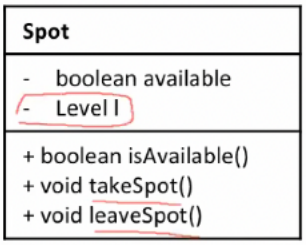
- Parking lot find the spot to clear
- Update spot to be available
- Update available count for each level

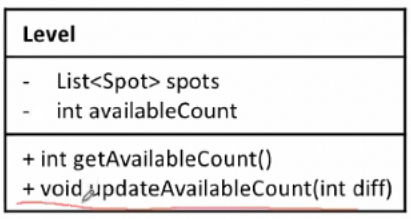
- 如何找到需要被free的spot？

Solution 1: Vehicle保存停的车位



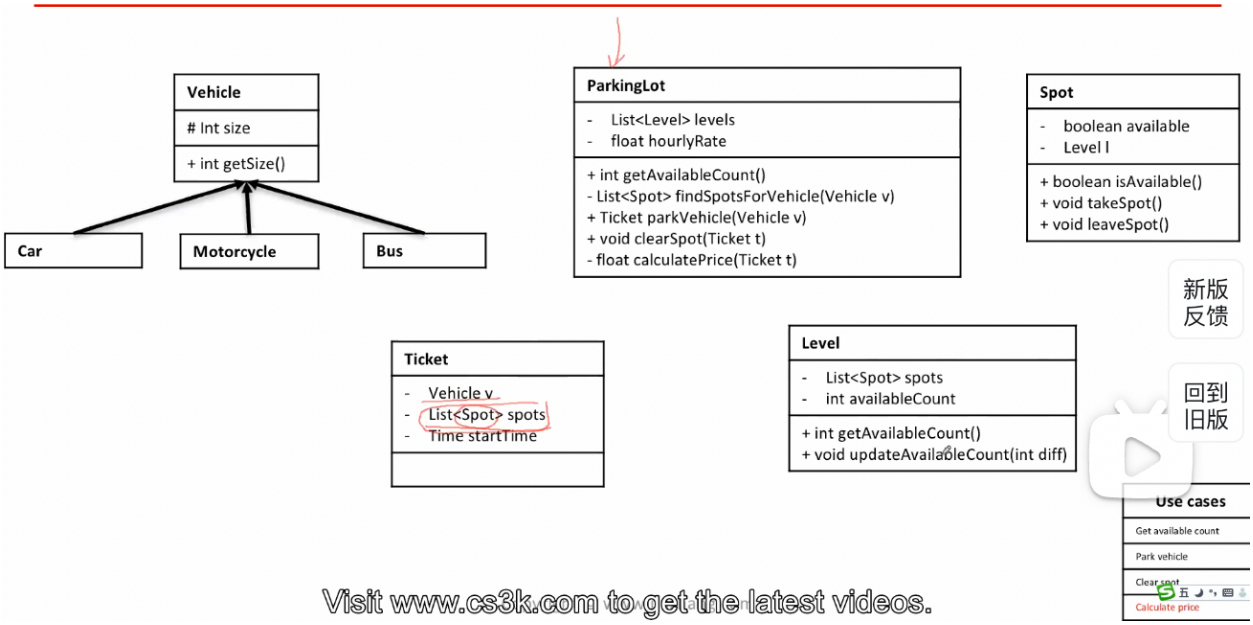






上面的方案虽然可以解决问题，但是缺点是不 elegent

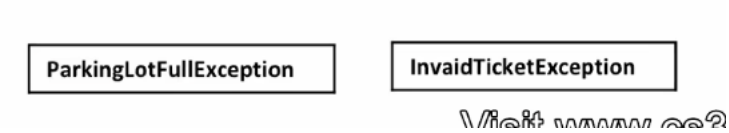
更好的 practice 是用 receipt



Class & Correctness

- Draw UML per Use Case
- 从以下几方面检查:
  - Validate use cases (检查是否支持所有的 use case)
  - Follow good practice (面试当中的加分项，展现一个程序员的经验)
  - S.O.L.I.D
  - Design pattern

加 excption



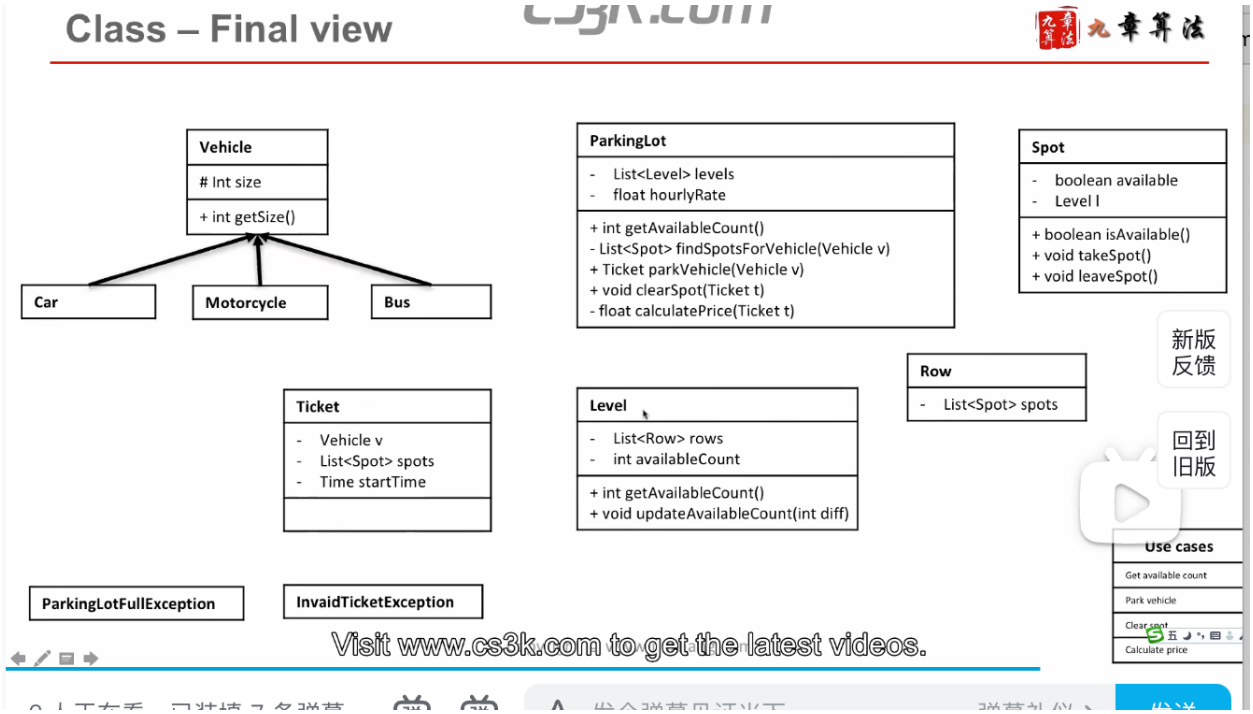
这些 exception 是单独存在的类，不需要和其他类有 connection

如果用Solution 1, 每行的个数必须要一样

solution1 缺点：

- Solution 2:

- 像添加Level一样，添加一个Row作为新的Class



大致的流程就是一辆车进 parkinglot，然后 parkinglot 这个 class 就去调用 findSpotForVehicle（Vehicle v）来获取有哪些 spot 是可用的，然后通过 parkVehicle 这个 method 去产生 ticket

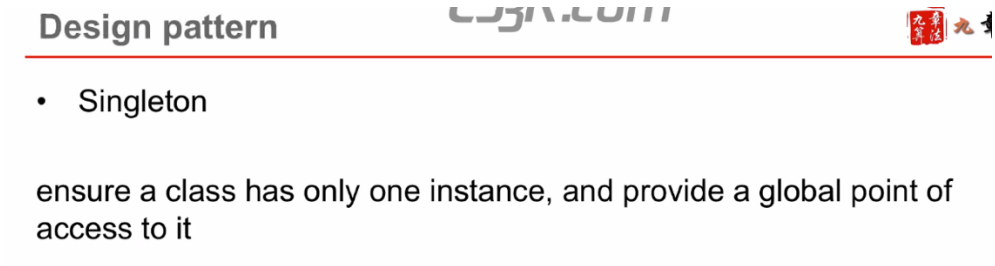
followup：怎么考虑多线程

当你有多个线程的时候，多个线程会同时访问一个 data，这个 data 也就是 critical data，比如这个 parking lot，critical data 就是 parkinglot class 里的 list<Level> levels 就是一个 critical data，比如一辆车正在 3 楼找 spot，这个同时有其他车也正在找车位，那这个时候第一辆车在找 spot 的时候我们就应该把 list<Level> levels 这个 data 锁住，不让其他用户去 access 这个 data，同时只能有一个用户去 access 这个 data

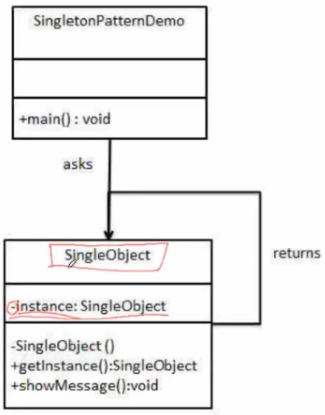
Design Pattern - Singleton

```
public class ParkingLot {
  private static ParkingLot _instance = null;

  private List<Level> levels;

  private ParkingLost() {
    levels = new ArrayList<List>();
  }

  public static synchronized ParkingLot getInstance() {
    if (_instance == null) {
      _instance = new ParkingLot();
    }
    return _instance;
  }
}
```

- `synchronized getInstance()`使得 Singleton 线程安全
- `static _instance` 使得只有一个 instance
- `private ParkingLot()`使得 constructor 只被调用一次



- Singleton



- Singleton – 基本式

```
public class ParkingLot
{
    private static ParkingLot _instance = null;

    private List<Level> levels;

    private ParkingLot()
    {
        levels = new ArrayList<Level>();
    }

    public static ParkingLot getInstance()
    {
        if(_instance == null)
        {
            _instance = new ParkingLot();
        }
        return _instance;
    }
}
```

- Singleton – 线程安全式

```
public class ParkingLot
{
    private static ParkingLot _instance = null;

    private List<Level> levels;

    private ParkingLot()
    {
        levels = new ArrayList<Level>();
    }

    public static synchronized ParkingLot getInstance()
    {
        if(_instance == null)
        {
            _instance = new ParkingLot();
        }
        return _instance;
    }
}
```

以上优点是线程安全，缺点是不够 efficient，因为一个函数调用这个之后就 block 了其他的函数调用

最优解

Design pattern

- Singleton – 静态内部类式

```
public class ParkingLot
{
    private ParkingLot(){}

    private static class LazyParkingLot
    {
        static final ParkingLot _instance = new ParkingLot();
    }

    public static ParkingLot getInstance()
    {
        return LazyParkingLot._instance;
    }
}
```

当一个 method 是 static 时，说明并不需要在 run time 就把它构造出来，在 build time，也就是编译器在构造这个类的时候他就会把这个类里面的东西执行一遍，
final 不会担心别的线程对他进行修改

full code: https://github.com/pulkitent/parking-lot-lld-oop-ood/tree/master/src/main/java