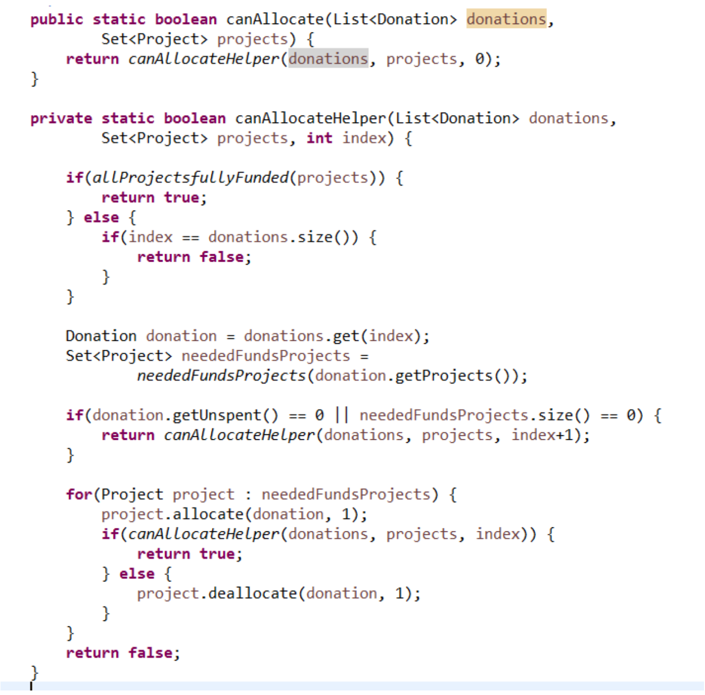
Task2



```
public static boolean canAllocate(List<Donation> donations,
        Set<Project> projects) {
    return canAllocateHelper(donations, projects, 0);
}

private static boolean canAllocateHelper(List<Donation> donations,
        Set<Project> projects, int index) {

    if(allProjectsfullyFunded(projects)) {
        return true;
    } else {
        if(index == donations.size()) {
            return false;
        }
    }

    Donation donation = donations.get(index);
    Set<Project> neededFundsProjects =
            neededFundsProjects(donation.getProjects());

    if(donation.getUnspent() == 0 || neededFundsProjects.size() == 0) {
        return canAllocateHelper(donations, projects, index+1);
    }

    for(Project project : neededFundsProjects) {
        project.allocate(donation, 1);
        if(canAllocateHelper(donations, projects, index)) {
            return true;
        } else {
            project.deallocate(donation, 1);
        }
    }
    return false;
}
```

Donation number: n

Project number: m

Total number of dollars worth of donations available: x

Total number of dollars required to fund all the projects: y

Algorithm analysis:

According to task 1, the recursive method canAllocateHelper(donations,projects,index) will run donations.size() times (because 0<=index<=donations.size()). And then if projects in one donation need to fund, donation will allocate 1 dollars for each project. When all projects in this donation are fully funded, then go to next donation until last donation.

If X <= Y, this recursive method will run n*m*x times.

If X > Y, this recursive method will run n*m*y times.

Example:

If there are n donations and m projects, suppose each donation has m projects, so there is n*m times to donate all projects from first donation to last donation. In addition, all donations total have x dollars, this means all donations have to donate to x times (because allocate one dollar of donation to project each time). Therefore, the worst case behaviour should be Ω(m*n*x).