

## Goals

This exercise consists of two programs with threads. Use the *6 steps approach to designing concurrent programs*. After this exercise you should be able to:

- synchronize the actions of threads;
- mutual exclusion of threads.

## 1 One Dice for N Players

In this program  $N$  players throw one and the same dice. The class `Dice` is defined as:

```
public class Dice {

    private int pips;
    private final Random rand;
    public static final int MAX_PIPS = 6;

    public Dice() {
        rand = new Random();
    }

    public void throwDice() {
        pips = rand.nextInt(MAX_PIPS) + 1;
    }

    public int getPips() {
        return pips;
    }
}
```

Note that throwing the dice and reading the number of pips are two separate actions.

The players implement the `Runnable` interface. A player throws the dice, reads the number of pips and prints the result. This is repeated the prescribed number of times.

```
public class Player implements Runnable {

    private final Dice dice;
    private final String name;
    private final int numberOfThrows;

    public Player(String name, Dice dice, int numberOfThrows) {
        this.name = name;
        this.dice = dice;
        this.numberOfThrows = numberOfThrows;
    }

    public void run() {
        for (int i = 0; i < numberOfThrows; i += 1) {
            dice.throwDice();
            int pips = dice.getPips();
            System.out.println(name + " throws " + pips + " in turn " + i);
        }
    }
}
```

A naive program to let two players play with one dice is:

```

public class OneDiceNplayers {

    private final Dice dice;
    private static final int PLAYERS = 2;
    private static final int THROWS = 100;

    public OneDiceNplayers() {
        dice = new Dice();
    }

    public void play() {
        for (int i = 0; i < PLAYERS; i += 1) {
            Player p = new Player("p" + i, dice, THROWS);
            Thread t = new Thread(p);
            t.start();
        }
    }

    public static void main(String[] args) {
        OneDiceNplayers game = new OneDiceNplayers();
        game.play();
    }
}

```

First of all note that this program is not thread-safe. One problem is that the players can execute the methods of the dice simultaneously. The other problem is that it is quite unsure whether the number of pips yielded by `getPips` is the result of the `throwDice` of this player, it is very well possible that the other player throws the dice again between the action of a player.

One solution for this problem is to synchronize the methods `throwDice` and `getPips` on the dice object:

```

public void run() {
    for (int i = 0; i < numberOfThrows; i += 1) {
        synchronized (dice) {
            dice.throwDice();
            int pips = dice.getPips();
            System.out.println(name + " throws " + pips + " in turn " + i);
        }
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

```

We have added a small `sleep` to enable the other player to grab the dice after the synchronized block of actions. A serious disadvantage of this solution is that the client class(es), here `Player`, are responsible for the thread-safeness of the server class `Dice`. It is highly desirable that such a server can be used in other contexts without additional measures or limitations.

Follow the six step approach to design and implement a thread-safe version of this 1-dice-N-player game. It can be necessary to add or change some methods of one or more classes.

## 2 Bar

In this program we model an badly equipped bar. This bar has a limited number of glasses (determined during construction). A number of parched man enters the bar. The number of drinkers can be higher than the number of glasses in the bar. Each of this drinkers grabs a glass, fills this glass at the tap, drinks it and digests his drink. This is repeated until the parched man is satisfied.

The class `Glass` is rather simple. It can be filled with some volume and emptied.

```

public class Glass {
    private int volume = 0;

    public void fill(int cc) {
        volume = cc;
    }

    public void empty() {
        volume = 0;
    }

    public int getVolume() {
        return volume;
    }
}

```

The Tap of the bar is also simple. It fills each given glass with fixed amount. The tap sleeps some time to mimic the time it costs to fill the given glass.

```

public class Tap {
    private static final int CC_PER_GLASS = 200;
    private static final int DRAW_TIME_PER_CC = 1;

    public Glass fillUp(Glass glass) {
        try {
            Thread.sleep(CC_PER_GLASS * DRAW_TIME_PER_CC);
        } catch (InterruptedException e) {}
        glass.fill(CC_PER_GLASS);
        return glass;
    }
}

```

The class ParchedMan mimics the drinkers. The method drawAndDrink models the behavior of these drinkers as described above: get a glass, fill it at the bar, drinks it, put it at the bar and digests it until the drinker is satisfied:

```

public class ParchedMan {

    private final int id;
    private static final int DRINK_TIME_PER_CC = 5;
    private static final int NUMBER_OF_GLASSES_TO_DRINK = 5;
    private static final int DIGEST_TIME = 10;
    private int numberOfGlassesDrunk = 0;
    private static Bar bar;
    private static Tap tap;

    public ParchedMan(int id, Bar bar, Tap tap) {
        this.id = id;
        this.bar = bar;
        this.tap = tap;
    }

    public void drawAndDrink() {
        while (!satisfied()) {
            if (bar.areThereGlasses()) {
                Glass glass = bar.getGlass();
                glass = tap.fillUp(glass);
                try {
                    System.out.println("Man " + id + " drinks a glass of beer");
                    Thread.sleep(glass.getVolume() * DRINK_TIME_PER_CC);
                } catch (InterruptedException e) {}
                glass.empty();
                numberOfGlassesDrunk++;
            }
        }
    }
}

```

```

        bar.putGlass(glass);
    }
    try {
        Thread.sleep(DIGEST_TIME);
    } catch (InterruptedException e) {}
}
System.out.println("Man " + id + " is satisfied");
}

private boolean satisfied() {
    return numberOfGlassesDrunk == NUMBER_OF_GLASSES_TO_DRINK;
}
}

```

Finally there is the class `Bar`. Each bar has lists of glasses and drinkers. The number of glasses is determined in the constructor of `Bar`. The number of drinkers is determined by the argument of `letInGuests`. The method `progrstartDrinking` starts the `drawAndDrink` behavior of each parched man.

```

public class Bar {

    private final Tap tap;
    private final List<Glass> glasses;
    private final List<ParchedMan> drinkers;

    public Bar(int numberOfGlasses) {
        tap = new Tap();
        glasses = new ArrayList<>();
        for (int i = 0 ; i < numberOfGlasses ; i++) {
            glasses.add(new Glass());
        }
        drinkers = new ArrayList<>();
    }

    public void letInGuests(int number) {
        for(int i = 0 ; i < number ; i += 1) {
            drinkers.add(new ParchedMan(i, this, tap));
        }
    }

    public void startDrinking() {
        for (ParchedMan man : drinkers) {
            man.drawAndDrink();
        }
    }

    public boolean areThereGlasses() {
        return glasses.size() > 0;
    }

    public Glass getGlass() {
        return glasses.remove(0);
    }

    public void putGlass(Glass glass) {
        glasses.add(glass);
    }

    public static void main(String[] args) {
        Bar bar = new Bar(2);
        bar.letInGuests(3);
        bar.startDrinking();
    }
}

```

}

Although this program runs without errors, its behavior is rather strange for a bar. The given bar is completely sequential. Each guest in turn will drink all his glasses. Each drinker drinks until he is satisfied before the next man starts drinking.

Your task is to design and implement a concurrent version of this program where the guests drink concurrently using the six step approach. Just as in the previous program it can be necessary to add or change some methods of one or more classes.

Ensure that each glass is used by at most one of the parched man at any moment. When there are more drinkers than glasses a drinker can have to wait before there is a glass available for his next drink. The single tap can fill at most one glass at any moment.

### **3 Deadline**

**Sunday May 24, 11:00.**