EMP5117. Foundations of Software Engineering
Winter 2019

At-Home Exercise
(Last modified on March 12, 2019)

For: February 15, 2019, 4:00 pm

1 A class Die

You need to create a class **Die**. An instance of a that class behaves like a real-life die: you can roll it and look at its value.

The class has the following methods:

- **void roll()**: rolls the die. It sets the current value randomly, between 1 and 6.
- **int getCurrentValue()**: returns the result of the last roll.
- **Die()**: the constructor of the class. When a die is created, it must be rolled automatically, to ensure that it will always have a valid, random value.

Here is an example of code that uses Die:

```java
Die die;
die = new Die();
System.out.println("Inital value: " + dice.getCurrentValue());
for(int i = 0 ; i < 10; i++) {
        dice.roll();
        System.out.println("Next value: " + dice.getCurrentValue());
}
```

This code produces the following output (note that another run will provide a different output)

```
Inital value: 6
Next value: 4
Next value: 4
Next value: 3
Next value: 6
Next value: 2
Next value: 1
Next value: 2
Next value: 6
Next value: 5
Next value: 6
```

2 Class OneDiePlayer

The class **OneDiePlayer** is used to run "experiments" with a single Die. It simply rolls the die a given number of times, and records the results of each roll. At the end, it prints how often each value was thrown. The number of times the die will be thrown is passed as a runtime parameter to the application. If no parameter is give, then the die is rolled 500,000 times.

Here are two sample runs. The first one uses the default value, and the second one uses 10,000:

```
> java OneDiePlayer

 Results: [83581, 82969, 83207, 83747, 83305, 83191]

> java OneDiePlayer 10000

 Results: [1713, 1631, 1651, 1709, 1614, 1682]
```

3 Class TwoDicePlayer

The class **TwoDicePlayer** is similar to the previous one, but uses 2 dices instead of one. Make sure to use 2 objects Die in your implementation.

Here are two sample runs. The first one uses the default value, and the second one uses 10,000:

```
> java TwoDicePlayer

 Results: [13784, 27949, 41772, 55550, 69160, 83231, 69810, 55699, 41525, 27509, 14011]

> java TwoDicePlayer 10000

 Results: [267, 544, 818, 1155, 1356, 1684, 1396, 1124, 834, 540, 282]
```

4 Class SetDicePlayer

The class **SetDicePlayer** is also similar to the previous one, but now the number of dices to use can also be specified as a runtime parameter. If no value are specified, 2 dices are used, and they are rolled 500,000 times. If 1 value n is specified, then n dices are used, and they are rolled 500,000 times. If 2 values n and m are specified, then then n dices are used, and they are rolled m times.

Here are three sample runs. The first one uses the default values (2 dices, 500,000 rolls). The second one uses 3 dices and the default 500,000 rolls. The third one uses 3 dices and 10,000 throws.

```
> java SetDicePlayer

 Results: [13926, 27825, 41571, 55016, 69310, 83752, 69660, 55656, 41640, 27796, 13848]

> java SetDicePlayer 3

 Results: [2323, 6916, 13890, 23119, 34729, 48649, 57650, 62401, 62459, 57908, 48935, 34907, 23067, 13837,

> java SetDicePlayer 3 10000

 Results: [41, 142, 267, 498, 702, 992, 1158, 1205, 1232, 1162, 958, 712, 466, 292, 129, 44]
```

5 Sorting set of dice

We want to sort an array of Die by increasing values. For this, we first modify our implementation of Die, adding the following method:

- **int compareTo(Die other)**: returns -1 if the current value of the die is less than the current value of the die **other**, 0 if the current values are the same, and 1 if the current value of the die is greater than the current value of the die **other**.

We then create a class **DiceSorted**. it has the following methods:

- **DiceSorted(int numberOfDice)**: the constructor receives the number of dices to use as parameter.
- **void rollAndSort()**: rolls all the dices, then sorts them out. It prints the values of the die after rolling but before sorting, and then again after sorting.

You will probably want to add more methods to **DiceSorted**. When sorting the dices, make sure to use the method **compareTo** of Die.

Here is an example of code that uses DiceSorted with 10 dices. It calls **sorter.rollAndSort** 5 times.

```java
DiceSorted sorter = new DiceSorted(10);
sorter.rollAndSort();
sorter.rollAndSort();
sorter.rollAndSort();
sorter.rollAndSort();
sorter.rollAndSort();
```

This code produces the following output (note that another run will provide a different output)

```
Before Sorting:
[2, 2, 4, 4, 1, 1, 5, 5, 3, 3]
After Sorting:
[1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
Before Sorting:
[6, 2, 5, 1, 4, 6, 3, 6, 1, 4]
After Sorting:
[1, 1, 2, 3, 4, 4, 5, 6, 6, 6]
Before Sorting:
[4, 6, 6, 3, 2, 3, 5, 1, 6, 3]
After Sorting:
[1, 2, 3, 3, 3, 4, 5, 6, 6, 6]
Before Sorting:
[6, 3, 1, 6, 1, 6, 5, 4, 3, 4]
After Sorting:
[1, 1, 3, 3, 4, 4, 5, 6, 6, 6]
Before Sorting:
[1, 2, 3, 6, 4, 1, 2, 5, 4, 5]
After Sorting:
[1, 1, 2, 2, 3, 4, 4, 5, 5, 6]
```