CodeCrunch

CS2030/S Practical Assessment #2

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

CS2030/S Practical Assessment #2

Java provides a `Random` class with a `nextInt(int bound)` method that returns a random (more precisely pseudorandom), uniformly distributed integer value between `0` (inclusive) and the specified `bound` (exclusive).

Random integer values within the range `0` (inclusive) to `Integer.MAX_VALUE` (exclusive) can be generated as follows:

```
jshell> Random r = new Random(2030)
r ==> java.util.Random@4520ebad

jshell> r.nextInt(Integer.MAX_VALUE)
$.. ==> 1327734301

jshell> r.nextInt(Integer.MAX_VALUE)
$.. ==> 588347554

jshell> r.nextInt(Integer.MAX_VALUE)
$.. ==> 1132554824
```

Notice that each time the statement `r.nextInt(Integer.MAX_VALUE)` is invoked, a different random value is produced. Clearly, some changes in mutable state happens after every invocation of the `nextInt` method.

In order to realize an *immutable* random number generator, we generate a random integer based on a given seed,

```
jshell> new Random(2030).nextInt(Integer.MAX_VALUE)
$.. ==> 1327734301
```

and use this new value as the seed to generate the next random value,

```
jshell> new Random(new Random(2030).nextInt(Integer.MAX_VALUE)).
   ...> nextInt(Integer.MAX_VALUE)
$.. ==> 1503777125
```

Each invocation of the above now returns the same value. Using this technique of seed replacement, the first five random values generated using this method is as follows. We consider the initial seed to be the first random number.

```
jshell> int i = 2030
i ==> 2030

jshell> i = new Random(i).nextInt(Integer.MAX_VALUE)
i ==> 1327734301

jshell> i = new Random(i).nextInt(Integer.MAX_VALUE)
i ==> 1503777125
```

```
jshell> i = new Random(i).nextInt(Integer.MAX_VALUE)
i ==> 1593627480

jshell> i = new Random(i).nextInt(Integer.MAX_VALUE)
i ==> 222432456
```

Task

Your task is to create a context that handles the implementation details of random number generation.

There are altogether five levels. You need to complete ALL levels.

Take Note!

- Write each class/interface/enum in its own file with meaningful names.
- Ensure that ALL object properties and class constants are declared `private final`.
- Ensure that your classes are NOT cyclic dependent.
- You are NOT allowed to use the following java keywords/literals: `null`, `instanceof`
- Java reflection is NOT allowed.
- There should be NO type cast with `SuppressWarnings`
- There is no restriction on Java libraries. However, if you use `java.util.Optional`, do not use the `get`/`isPresent`/`isEmpty` methods.
- Do not use `*` wildcard imports.
- For ease of implementation, you need NOT include bounded/unbounded wildcards.
- Do NOT define anonymous inner classes; define lambdas instead.
- Do NOT use method references `::` (the grader cannot handle it).
- Do NOT use ellipsis, e.g. `String...`

In essence, keep to the constructs and programming discipline that is instilled throughout the module. Moreover, for brevity in presenting test cases, we do not "type-witness" static methods unless necessary, i.e. `Optional.<Integer>of(1)` is more concisely represented as `Optional.of(1)`.

Level 1

Define a `Rand` class to generate random values using the seed replacement technique described above. The `Rand` class comprises of the following:

- a static `of` method that takes in an integer seed and creates a `Rand` object
- a `toString` method that simply outputs `Rand`
- a `get` method that returns the random integer value
- a `next` method that returns the "next" `Rand` object

You may assume that the `of` method is used only at the start of a chain of `Rand` methods.

```
jshell> Rand.of(2030) instanceof Rand
$.. ==> true

jshell> Rand.of(2030).get()
$.. ==> 2030

jshell> Rand.of(2030).next() instanceof Rand
$.. ==> true

jshell> Rand.of(2030).next().get()
$.. ==> 1327734301

jshell> Rand.of(2030).next().next().get()
$.. ==> 1503777125

jshell> Rand r = Rand.of(2030)
r ==> Rand

jshell> r.get()
$.. ==> 2030

jshell> r.next().get()
$.. ==> 1327734301

jshell> r.get()
$.. ==> 2030

jshell> r.next().next().get()
$.. ==> 1503777125
```

```
jshell> r.get()
$.. ==> 2030
```

Level 2

Random numbers are generated repeatedly during simulations. Include a `stream()` method to generate a `Stream<Integer>` of random numbers as shown in the sample run below:

```
jshell> Rand.of(2030).
   ...>     stream().
   ...>     limit(5).
   ...>     forEach(x -> System.out.println(x))
2030
1327734301
1503777125
1593627480
222432456

jshell> Rand.of(2030).
   ...>     next().
   ...>     stream().
   ...>     limit(5).
   ...>     forEach(x -> System.out.println(x))
1327734301
1503777125
1593627480
222432456
1414485811
```

Moreover, generating sequences of random integer values within a specific range is very common. Include a `static` method `randRange` that takes in an integer `seed`, and a `Function` that maps the random generated values to the specified range.

As an example, the first five random values generated from `Rand.of(2030)`, i.e. `2030`, `1327734301`, `1503777125`, `1593627480`, `222432456` can be mapped to the range `0` to `9`, by taking the remainder after dividing by `10`, i.e. `0`, `1`, `5`, `0`, `6`.

```
jshell> Function<Integer,Integer> f = x -> x % 10
f ==> $Lambda$19/0x00000008000aec40@46d56d67

jshell> Rand.randRange(2030, f).
   ...>     limit(20).
   ...>     forEach(x -> System.out.print(x + " "))
0 1 5 0 6 1 0 4 4 2 0 9 1 8 1 5 3 1 3 8
```

Level 3

Include a method `map` that maps the random value in the `Rand` object. Note how the order of method calls to `next` and `map` can be interchanged while still generating the same outcome.

```
jshell> Rand.of(2030).map(x -> x - 1).get()
$.. ==> 2029

jshell> Rand.of(2030).next().map(x -> x - 1).get()
$.. ==> 1327734300

jshell> Rand.of(2030).map(x -> x - 1).next().get()
$.. ==> 1327734300

jshell> Rand.of(2030).next().map(x -> x - 1).next().get()
$.. ==> 1503777124

jshell> Rand.of(2030).map(x -> x - 1).stream().
   ...>     limit(5).
   ...>     forEach(x -> System.out.print(x % 10 + " "))
9 0 4 9 5
jshell> Rand.of(2030).next().map(x -> x - 1).stream().
   ...>     limit(5).
   ...>     forEach(x -> System.out.print(x % 10 + " "))
0 4 9 5 0
jshell> Rand.of(2030).map(x -> x - 1).next().stream().
   ...>     limit(5).
   ...>     forEach(x -> System.out.print(x % 10 + " "))
```

```
0 4 9 5 0
jshell> Rand.of(2030).next().map(x -> x - 1).next().stream().
   ...>     limit(5).
   ...>     forEach(x -> System.out.print(x % 10 + " "))
4 9 5 0 9
jshell> Rand.of(2030).next().map(x -> x % 2 == 0).next().stream().
   ...>     limit(5).
   ...>     forEach(x -> System.out.print(x + " "))
false true true false true
```

Now, modify the `randRange` method so that it can generate a stream of values of any type.

```
jshell> Rand.randRange(2030, x -> x % 10).
   ...>     limit(5).
   ...>     forEach(System.out::println)
0
1
5
0
6

jshell> Rand.randRange(2030, x -> String.format("[%02d]", x % 100)).
   ...>     limit(5).
   ...>     forEach(System.out::println)
[30]
[01]
[25]
[80]
[56]

jshell> double lo = -1.0
lo ==> -1.0

jshell> double hi = 1.0
hi ==> 1.0

jshell> // random floating point values from lo to hi, both inclusive.

jshell> Rand.
   ...> randRange(2030, x -> (hi - lo) * x / (Integer.MAX_VALUE - 1) + lo).
   ...> limit(10).
   ...> forEach(System.out::println)
-0.9999981094151718
0.23654892876422862
0.4005015850071809
0.4841812490338284
-0.7928436322071064
0.31734256848445397
-0.5744591109216763
-0.4216896737196386
-0.2363805372606782
0.2196226168606641
```

The following are additional sample runs that illustrates the evaluation of a `Rand` object.

```
jshell> Rand.of(2030).
   ...> next().
   ...> map(x -> { System.out.println("ouch!"); return x; }).
   ...> next()
$.. ==> Rand

jshell> Rand.of(2030).next().
   ...> map(x -> { System.out.println("ouch!"); return x; }).
   ...> next().
   ...> get()
ouch!
$.. ==> 1503777125

jshell> Stream<Integer> ints = Rand.randRange(2030, x -> {
   ...>     System.out.println("ouch!"); return x; }).
   ...> limit(5)
ints ==> java.util.stream.SliceOps$1@31ef45e3

jshell> ints.forEach(x -> System.out.println(x))
```

```
ouch!
2030
ouch!
1327734301
ouch!
1503777125
ouch!
1593627480
ouch!
222432456
```

Level 4

So far, we have been generating sequences of individual values. How do we generate say, random lists of two integer elements? We use `flatMap`!

Let's first look at how `flatMap` behaves in the context of individual elements.

```
jshell> Rand.of(2030).map(x -> x / 2).get()
$.. ==> 1015

jshell> Rand.of(2030).map(x -> x / 2).next().get()
$.. ==> 663867150

jshell> Rand.of(2030).flatMap(x -> Rand.of(x).map(y -> y / 2)).get()
$.. ==> 1015

jshell> Rand.of(2030).flatMap(x -> Rand.of(x).map(y -> y / 2)).next().get()
$.. ==> 663867150

jshell> Rand.of(2030).map(x -> x / 2).next().
   ...> stream().limit(5).forEach(x -> System.out.println(x))
663867150
751888562
796813740
111216228
707242905

jshell> Rand.of(2030).next().flatMap(x -> Rand.of(x).map(y -> y / 2)).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
663867150
751888562
796813740
111216228
707242905

jshell> Rand.of(2030).flatMap(x -> Rand.of(x).next().map(y -> y / 2)).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
663867150
751888562
796813740
111216228
707242905

jshell> Rand.of(2030).flatMap(x -> Rand.of(x).map(y -> y / 2).next()).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
663867150
751888562
796813740
111216228
707242905

jshell> Rand.of(2030).flatMap(x -> Rand.of(x).map(y -> y / 2)).next().
   ...> stream().limit(5).forEach(x -> System.out.println(x))
663867150
751888562
796813740
111216228
707242905
```

Now to generate random lists of two elements each, we need to replace the `map` function within `flatMap`.

```
jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.of(x,y))) instanceof Rand
$.. ==> true

jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.of(x,y))).get()
$.. ==> [2030, 2030]

jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.of(x,y))).next().get()
$.. ==> [1327734301, 1327734301]

jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.of(x,y))).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
[2030, 2030]
[1327734301, 1327734301]
[1503777125, 1503777125]
[1593627480, 1593627480]
[222432456, 222432456]
```

The above `flatMap` operations generates a list of two elements which are the same.

Let's include a `next` within the `flatMap`.

```
jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.of(x,y)).next()).
   ...> get()
$.. ==> [2030, 1327734301]

jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.of(x,y)).next()).
   ...> next().
   ...> get()
$.. ==> [1327734301, 1503777125]

jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.of(x,y)).next()).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
[2030, 1327734301]
[1327734301, 1503777125]
[1503777125, 1593627480]
[1593627480, 222432456]
[222432456, 1414485811]
```

The two values within a list are now consecutive random numbers. Also the first element of a list, is the second element of the previous list. This is the expected behaviour.

Below are some other sample tests.

```
jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(1010).map(y -> List.of(x,y))).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
[2030, 1010]
[1327734301, 1010]
[1503777125, 1010]
[1593627480, 1010]
[222432456, 1010]

jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(1010).map(y -> List.of(x,y)).next()).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
[2030, 1530112223]
[1327734301, 1530112223]
[1503777125, 1530112223]
[1593627480, 1530112223]
[222432456, 1530112223]

jshell> Rand.of(2030).
   ...> flatMap(x -> Rand.of(x).map(y -> List.<Number>of(x, (y % 10) / 2.0))).
   ...> stream().limit(5).forEach(x -> System.out.println(x))
[2030, 0.0]
[1327734301, 0.5]
[1503777125, 2.5]
```

```
[1593627480, 0.0]
[222432456, 3.0]

jshell> Stream<Integer> ints = Rand.of(2030).
   ...>     flatMap(x -> {
   ...>         System.out.println("ouch!");
   ...>         return Rand.of(x); }).
   ...>     stream().limit(5)
ints ==> java.util.stream.SliceOps$1@5bcab519

jshell> ints.forEach(x -> {})
ouch!
ouch!
ouch!
ouch!
ouch!

jshell> ints = Rand.of(2030).
   ...>     flatMap(x -> {
   ...>         System.out.println("ouch!");
   ...>         return Rand.of(x).map(y -> {
   ...>             System.out.println("aiyo!");
   ...>             return y + 1; }); }).
   ...>     stream().limit(5)
ints ==> java.util.stream.SliceOps$1@1e397ed7

jshell> ints.forEach(x -> System.out.println(x))
ouch!
aiyo!
2031
ouch!
aiyo!
1327734302
ouch!
aiyo!
1503777126
ouch!
aiyo!
1593627481
ouch!
aiyo!
222432457
```

Level 5

Finally, let us use our random number generator for a classic Monte-Carlo simulation — that of estimating the value of `pi`.

Briefly, we can generate random points within the square region spanning bottom-left `(-1,-1)` to top-right `(1,1)`, and determine how many of these points fall into the circle centred at `(0, 0)` with radius `1.0`. The proportion of points that falls within the circle will be approximately equals to `pi/4`.

You are given the `Circle` and `Point` classes. Define a `Main` class with a `static` method `double simulate(int seed, int n)` that takes in the `seed` and the number of simulation trials `n`. The method generates `n` number of points within the region `(-1,-1)` and `(1,1)`, determines how many of these fall within the circle, and returns the estimated value of `pi` as a double value.

Take note that unlike the levels above, we wish to avoid duplicate random values. Here is a peek on what the first ten points should look like using `2030` as the seed. You may compare this with the random floating point values generated from one of the test cases in level 3.

```
(-0.9999981094151718, 0.23654892876422862)
(0.4005015850071809, 0.4841812490338284)
(-0.7928436322071064, 0.31734256848445397)
(-0.5744591109216763, -0.4216896737196386)
(-0.2363805372606782, 0.2196226168606641)
(-0.0677647097667351, 0.05293416423065045)
(-0.87413697771182, 0.7369958010846709)
(-0.563276542875242, -0.03873627450199446)
(-0.2820637172852305, -0.46053203051959357)
(0.07041811018289823, -0.8967840819589645)
```

Below are simulations over an increasing number of trials with the same seed `2030`. Notice the convergence to the value of `pi`.

```
jshell> Stream.iterate(10, x -> x * 10).
   ...>     limit(6).
   ...>     map(x -> Main.simulate(2030, x)).
   ...>     forEach(x -> System.out.println(x))
3.2
3.0
3.224
3.1788
3.151
3.1474
```

You do not need to submit `Point.java` and `Circle.java`.