The goal for today’s lab is to provide an application programming interface to /dev/urandom, which is a source of bytes derived from the unpredictable behavior of hardware devices built into the computing device or attached to it. The underlying generator of random values has no “seed” and cannot be reset to a known state.

Sequences of random values produced in this way are not reproducible and so are not suitable for applications in which reproducibility on demand is required (e.g., simulations in academic research).

Building a Random Byte Buffer

In principle, we could open the /dev/urandom pseudo-file every time we needed one or more random bytes, extract just the number of bytes we needed, and close it again afterwards. But opening and closing /dev/urandom so frequently is kind of inefficient. It’s better to maintain a large buffer full of random bytes that is held in an array in memory and to parcel out bytes from this buffer as the application needs them, refilling the buffer from /dev/urandom only when we run out.

1. In Eclipse, open up a new project for today’s lab. In that project, create a new class called RandomByteBuffer. It should have two fields: an array of, say, 1048576 bytes and a counter showing how many bytes of that array have been returned as values in method calls since the last time the array was refilled from urandom. It needs at least two methods (and a constructor): One method, refill, should open /dev/urandom and copy bytes from that source into all the positions in the array. The other, getRandomByte, should return the next available byte from the array, and increment the counter. If the byte returned is the last one in the array, getRandomByte should also call refill to replenish the array and reset the counter to 0.

Random Values

2. Create a second class, MyRandom, that has a RandomByteBuffer as its field. Write a zero-argument constructor for MyRandom that initializes this field by constructing a RandomByteBuffer.

3. In the Random class, define a nextByte method that returns a value of the primitive type byte. The values returned in a sequence of calls should be uniformly distributed over all the possible values in the type.

4. In the Random class, define a nextInt method that returns a value of the primitive type int. The values returned in a sequence of calls should be uniformly distributed over all the possible values in the type.

( Hint: Java requires implementations to use the twos-complement representation for integers. Figure out how to calculate a random twos-complement representation from four random byte values using Java’s shift operators.)

5. In the Random class, define a nextDouble method that returns a value of the primitive type double. The values returned in a sequence of calls should be uniformly distributed over the possible double values in the range from 0.0 (inclusive) to 1.0 (exclusive).

( Hint: Java requires implementations to use the IEEE double-precision representation for double values, with a sign bit, and eleven-bit exponent, and a fifty-three-bit mantissa, including one hidden bit. To ensure uniform distribution over the chosen interval, we can concentrate exclusively on the mantissa, building a long value that contains fifty-three random bits using the same
techniques as in the previous exercise and then dividing it by $2^{53}$, expressed as a `double`, in order to map the mantissa into the range from 0.0 to 1.0.)

6. In the `Random` class, define a `nextInRange` method that takes two `int` values, `lower` and `upper`, and returns an `int` value that is greater than or equal to `lower` and less than `upper`. Note that it is a precondition of this method that `lower` is strictly less than `upper`. The values returned in a sequence of calls should be (as nearly as possible) uniformly distributed in the specified interval.

7. In the `Random` class, define a `coinToss` method that returns the string "heads" or the string "tails" with equal probability.

8. In the `Random` class, define a `dieRoll` method that returns an `int` value in the range from 1 to 6, with (as nearly as possible) equal probability for each result.

9. In the `Random` class, define a `randomElement` method that returns a randomly selected element from a given object that implements the `Collection` interface. It is a precondition of this method that the collection is not empty.

10. In the `Random` class, define a `permute` method that randomly permutes the elements of any given object that implements the `List` interface.

Optional Extensions

11. Implement a method `shuffleLines` that reads in lines from an existing text file, randomly permutes those lines, and then writes them to `System.out` in permuted order. (The GNU/Linux utility `shuf` illustrates this capability.)

12. Simulate the process of rolling a die six thousand times, collecting the results in an array. Compute the mean of the simulated die rolls, their variance (that is, the average of the squares of the differences between the die rolls and their mean), and their standard deviation (the square root of the variance).

13. Calculate the expected mean, variance, and standard deviation of six thousand rolls of a theoretically perfect die that shows each of its faces with probability exactly 1/6. How well do these theoretical results correspond to the values obtained in the simulation in the preceding exercise?

14. Add to the `RandomByteBuffer` class a `randomBits` method that takes as argument an integer value `count` in the range from 1 to 64 and returns a `long` in which the `count` least significant bits are random and the other 64 − `count` bits are zero.

15. Unless `count` is a multiple of 8, the `randomBits` method will need to use some, but not all, of the bits from a byte in the `RandomByteBuffer`'s array. Provide a mechanism for storing the unused bits of that byte in an additional field and recovering them as needed to respond to some subsequent invocation of `randomBits`.

16. Revise `nextInt` and `nextDouble` to use `randomBits` instead of calls to `nextByte`.

Copyright © 2018  John David Stone

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit

http://creativecommons.org/licenses/by-sa/4.0/deed.en

or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.