To What Extent Can The Random Number Generator Math.Random() In Java Be Predicted Before Being Generated?

Candidate number: **xxxx**

Session: **May 2016**

Word count: **3613**

Abstract

This essay looks at the question, “To What Extent Can The Random Number Generator Math.Random() In Java Be Predicted Before Being Generated?” The essay introduces with an overview to what a random number generator is and why this is relevant to be investigate. This allows for the reader to have solid knowledge for the investigation to be relevant and understandable. It then continues to state the background information, distinguishing between ‘true’ and ‘pseudo’ random number generators. It then goes on to explain their differences, stating how `math.random()` is in the latter category. This is as `math.random()` is a linear congruential generator and so produces random numbers using the formula $r_{n+1} = (A \times r + B) \bmod M$. This leads on to the investigation, describing how it is possible to us the formula to predict the outputs of the method. It displays the results of a program, altered to be more specific to this investigation, that accurately predicts the `math.random()` function. This directly leads on to the implications of being able to predict such random number generators, explaining the situation of planetpoker how their shuffling algorithm was bypassed due to someone being able to predict their random number generator. Then the conclusions of the essay are discussed, stating the dangers of using un-secure random number generators and the future possible increase in ease of predicting random number generators due to technological advancements. This then finalises with the conclusions of the investigation, that the random number generator `math.random()` in java can be predicted before being generated.

Contents

2 - Abstract

4 - Introduction

5 - Background Information

7 - Investigation

10 - Implications

13 - Conclusion

14 - Bibliography

Intro

"To what extent can randomly generated numbers in Java be predicted?"

This essay will be investigating the possibility of predicting the numbers that are randomly generated by the Java method `math.random()`. Here a random number is defined as a function object that can be used to generate a random sequence of integers. That is: if *f* is a Random Number Generator and N is a positive integer, then *f*(N) will return an integer less than N and greater than or equal to 0.[1] To predict is defined as say or estimate that (a specific thing) will happen in the future or will be a consequence of something.[2]

The need to investigate this topic is clear as the limitations of random number generators are generally not considered by the programmers that use them. As stated by Donald Kuth *"Many random number generators in use today are not very good. There is a tendency for people to avoid learning anything about such subroutines; quite often we find that some old method that is comparatively unsatisfactory has blindly been passed down from one programmer to another, and today's users have no understanding of its limitations."*[3] This highlights the need to investigate random number generators, in this case the method `math.random()` in order to see its limitations and assess its security.

The reason this may be possible is as numbers generated by a computer can never be truly random, as random is defined as proceeding, made, or occurring without definite aim, reason, or pattern[4], because they have to be generated through some form of equation to output a number. This means by deducing the equation it will be possible to predict which number will be output by the method. However, as there are many differing number generating methods for the purposes of the investigation, it will be looking at specifically the method in Java `math.random()`.Java is defined as a high level, object-orientated computer programming language used especially to create interactive applications running over the internet[5] and `math.random()` is defined as the function that returns a floating point, pseudo-random number in the range from 0 (inclusive) to 1 (exclusive).[6] The reason for this is that I have experience both with Java and the method itself and as this is the commonly

[1] https://www.sgi.com/tech/stl/RandomNumberGenerator.html
[2] http://www.oxforddictionaries.com/definition/english/predict
[3] Donald Knuth; The Art of Computer Programming, Volume 2.
[4] http://dictionary.reference.com/browse/random
[5] http://dictionary.reference.com/browse/java
[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

used method in any Java applications and so this will be relevant for many applications as Java is a very common programming platform.

Background Information

A random number generator is a function object that can be used to generate a random sequence of integers. This includes things as simple as dice, where the dice generates a number at random between 1 and 6, or a flip of a coin, producing a 0 or a 1. However when generating large amounts of random numbers these methods become slow and tedious. It requires the method to be carried out sequentially, over and over in order to produce a sequence of random numbers. Carrying out these methods takes time, the time to roll a dice and wait for the outcome for example, which is not much on its own but when generating large numbers or large amounts of numbers these methods have to be repeated so many times that it takes a significant amount of time. To combat this computer based random number generators are used where large scale random numbers are required. As a computer can process an algorithm much faster than these methods can be undertaken, it is a huge time saving technique. This includes simple things such as lotteries and online bingos, which produce numbers at random as part of the game, to more complex online card games and slot machines where a random draw is needed. Computer based systems work perfectly for these large scale activities due to the fact that a computer system is programmable to produce exactly what is required. With advancements in modern computing this process is now very simple and fast and only going to get faster. Observing Moores Law[7] it is evident that there has been a significant increase in computing ability. As processors, RAM and storage becomes smaller and faster, computational possibilities become bigger. This leads on also to the more complex uses of random number generators such as for creating computer simulations, statistical sampling and cryptography. Here however much more computational power is required and the area is too far afield from the simple method being investigated.

However not all random number generators are the same. There are 2 defining ways that a random number generator can work.
The first is 'true' random number generators, which are non-deterministic .Non-deterministic is defined as Non-predictive, referring to the inability to objectively predict an outcome or result of a process due to lack of knowledge of a cause and effect relationship or the inability to know initial conditions.[8] This means even for the same input it can exhibit different behaviours. These are generators that rely on measuring some form of physical data that is

[7] http://www.mooreslaw.org/
[8] http://www.yourdictionary.com/non-deterministic

inherently random in itself in order to create a random number. For example radioactive decay could be measured, which is the spontaneous transformation of an unstable atomic nucleus into a lighter one[9], and then the computer will use this random data to create its sequence of random numbers. The decay is measured by a Geiger-Muller tube[10] which will produce naturally random decays. These days are used as the input data for the random number generator. This leads to it being impossible to predict the number produces as it is impossible to determine the rate of radioactive decay, an average over time is possible to be calculated but this is based on estimate and does not mean that the decay will occur exactly at that rate. This is advantageous as this creates truly unpredictable numbers. As the numbers generated are based on physical measured data that is random itself, it is impossible to crack this form of number generation. On the other hand, just as before, because this method relies on the measuring of a real time event it takes time to generate a single number. Then when generating large numbers, or large amounts of number, this time adds up and become substantial. This is the main disadvantage when generating truly random numbers.

The second method of generating random numbers is pseudo-random number generators. Pseudo-random is defined as noting or pertaining to random numbers generated by a definite computational process to satisfy a statistical test.[11] These are generators that rely on computational methods to generate a sequence of random numbers. However these numbers are only seemingly random. This is as the numbers created are based on a computer algorithm rather than a physical, random event. This is done by using an initial seed or key and inputting this thorough an algorithm to produce a random number. Then in turn this number is put through the algorithm again to produce a second random number and so on. This is called a linear congruential generator[12]. A linear congruential generator is defined as a class of algorithms that are pseudo-random number generators where the number generated is used to create the next number.[13] This is an algorithm that produces a sequence of pseudo random numbers and is one of the oldest and best known methods of generating pseudo random numbers. Linear congruential generators do this through the equation $r_{n+1} = (A \times r + B) \bmod M$.

[9] http://dictionary.reference.com/browse/radioactive-decay
[10] https://en.wikipedia.org/wiki/Geiger%E2%80%93M%C3%BCller_tube
[11] http://dictionary.reference.com/browse/pseudorandom
[12] https://xlinux.nist.gov/dads//HTML/linearCongruentGen.html
[13] https://xlinux.nist.gov/dads//HTML/linearCongruentGen.html

Investigation

Random number generation is key to all programs that involve any kind of chance. Anything from online games to online casinos will use some random number generation in their applications. As these random numbers have to be pre-programme, to some extent, for the applications to work, they are generated through a pre coded algorithm, then these can be de coded and worked out. This means theoretically if the random number generator can be solved, and known by the users, then they will be able to predict the outcome of such generators. This would in turn allow them to work out some key elements of the game and have an unfair advantage. This all stems from the fact pseudo random number generators can be very easy to solve once the seed is known. As the seed is the key to creating the first number to run through the algorithm once is known so is the rest of the sequence of numbers, but obtaining the seed is the difficult part of solving these algorithms.

In Java `math.random()` random numbers are generated using an equation, $r_{n+1} = (A \times r + B) \bmod M$..[14] This is an example of a linear congruential generator.[15] In this equation *A*, *C* and *M* are constants (the multiplier, addend and the mask) and *r* is both the original seed and the previous number generated by the equation. So once you have the first number generated by the equation you can start to work out the constants that are being used and the original seed that was used. This then, in theory, allows you to recreate the algorithm and know the entire sequence of random numbers that would be generated. The issue is the sheer number of possibilities for the constants and original seed which makes it difficult to sort through and find the correct solution in real time. This means narrowing down the possibilities to as low as possible would result in a higher chance of being able to solve the constants at hand.

For the specific generator that is being investigated, `math.random()` in Java, the constants *A* and *C* are not something that need to be worked out as they are freely known due to the open source nature of Java, however the seed and mask is unknown, and they can also be found in the Java.util.random pathway. This does mean that `math.random()` is not an entirely secure method of random number generation as not only is it only a pseudo random

[14] http://franklinta.com/2014/08/31/predicting-the-next-math-random-in-java/
[15] https://xlinux.nist.gov/dads//HTML/linearCongruentGen.html

generator but some of its constants that are being used are widely known. Using this knowledge it should be possible to obtain the seed and mask, which are different with each iteration of `math.random()`, and use the constants to begin predicting the sequence of numbers that will be produced.

To investigate this, a program was adapted to better suit the needs of the investigation.[16] The program aims to predict the outcome of the `math.random()` generator by exploiting how pseudo random numbers are generated.

It firstly attempts to get the mask that the method would use to generate a new random number. In this methods case this is a 48 bit number. The program uses the method `next.double()`, which is what is called to create a `math.random()` number, in order to return the mask value that was used. By obtaining the random number that the double was generated from it then uses this number in order to attempt to predict the next number output by `math.random()`.

It then uses this mask and the known constants *A* and *C* in order force check for possible seeds that would make the equation work. It uses a brute force method, which is defined as a trial and error method used by application programs to decode encrypted data such as passwords or Data Encryption Standard (DES) keys, through exhaustive effort (using brute force) rather than employing intellectual strategies.[17] Genetic algorithms[18] and simulated annealing[19] are some examples of other techniques possible to use to determine the seed. However the program uses a brute force method as the number of possible seeds should be relatively low due to the amount of known constants in the method. If more than one possible seed is found then the program will not attempt to predict the sequence, instead it will alert the user and display the possible seeds. If no seed is found then the program will not be able to predict the sequence. However, if only a single possible seed is discovered then that seed is used as the expected seed and the expected random number sequence is generated. After this the normal `math.random()` method is called and the results are compared.

[16] https://github.com/fta2012/ReplicatedRandom
[17] http://searchsecurity.techtarget.com/definition/brute-force-cracking
[18] *Mitchell, Melanie (1996). An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press.*
[19] https://en.wikipedia.org/wiki/Simulated_annealing

The program was tested over a hundred times and here are 2 examples of its output:

Fig 1.

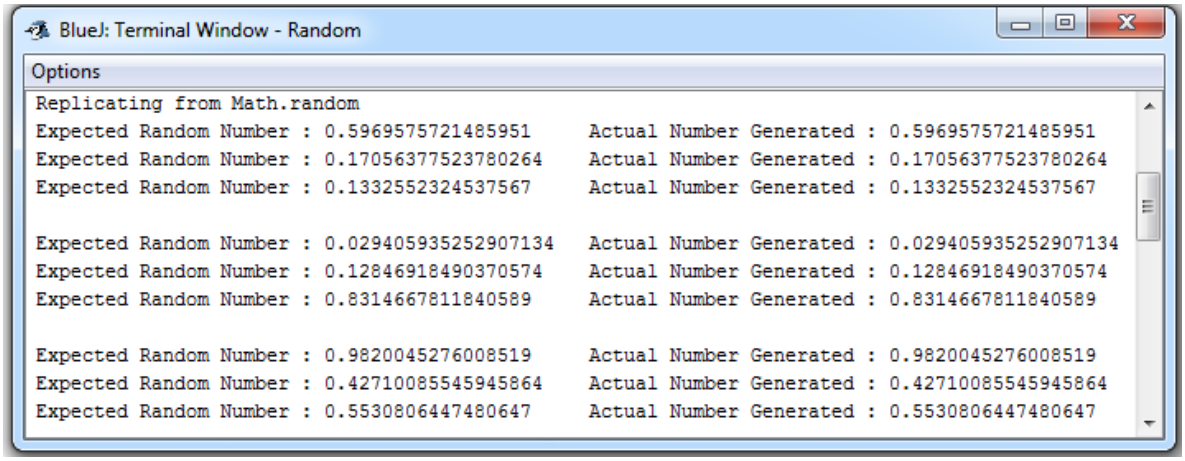
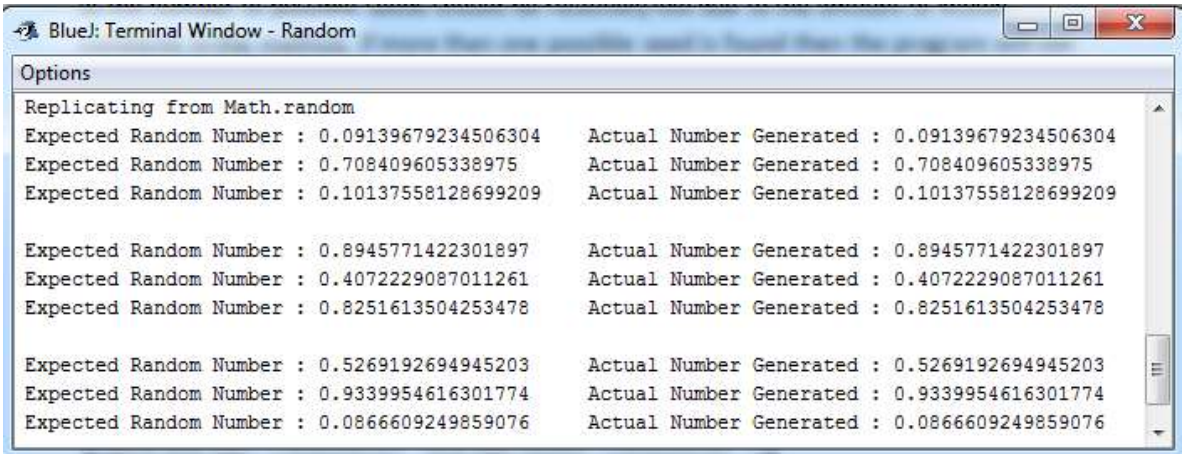


Fig 2.



The code for the adapted program can be seen in the bibliography.

It is clear from the output of the program to see that the program manages to predict the correctly the output that `math.random()` will produce before the method is even called, showing that predicting the outcome of pseudo random number generators is a possibility. Through much testing, over 100 executions of the program, the program has yet to incorrectly predict the outcome of the `math.random()` method when it has managed to use brute force to obtain the seed. The accuracy of the predictions are shocking when thinking about the implications of this possibility to predict random number generators in less controlled environments. For example in online casinos if the player was able to predict the numbers that would come up, which would correspond to some element of the game being played whether that be cards for poker or the numbers on a roulette wheel, then they would have an

unfair advantage. This would be bad for both the casino and the other players in the game as both would be being effectively cheated out of the game.

Implications

Knowing that it is possible to crack these equations and then predict the outcome of random number generators and that they are heavily used in many computer programs, it is only inevitable that some of these programs would be targeted and cracked. If one person, or a group of people, know the equation used to generate random numbers in the program then they have a huge advantage as they can predict what will happen.

This was the case for PlanetPoker[20] in 1999, an online poker game that allowed people to play against each other at poker in real time. The site offered Texas Hold'em poker to players who would place real money bets on these games hoping to win, but this is no longer the case as the website has suspended real money bets as of 2007.[21] However the games system of generating random numbers for the drawing of cards was somewhat flawed. This was discovered by players of PlanetPoker, who are also software security experts, Brad Arkin, Frank Hill, Scott Marks, Matt Schmid and Thomas John Walls.[22] The group started to dig into the frequently asked questions (FAQ) to see what the algorithm was that was used to draw cards.[23] PlanetPoker had the algorithm available to intend to prove that the game was fair and that card draws were truly random, however, when the group saw the algorithm they quickly understood that it was flawed.

```
procedure TDeck.Shuffle;
var
    ctr: Byte;
    tmp: Byte;

    random_number: Byte;
begin
    { Fill the deck with
unique cards }
    for ctr := 1 to 52 do
        Card[ctr] := ctr;

    { Generate a new seed
based on the system clock }
    randomize;

    { Randomly rearrange
each card }
    for ctr := 1 to 52 do
begin
        random_number :=
random(51)+1;
        tmp :=
card[random_number];
        card[random_number]
:= card[ctr];
        card[ctr] := tmp;
    end;

    CurrentCard := 1;
    JustShuffled := True;
end;
```

They discovered that all they needed now is the first few cards generated in order to crack the generator used for the shuffle, just like for predicting the `math.random()` function in Java you require the first number generated to work out the seed for the generator. As predicting the entire deck is a bit more complex, they require the first 5 cards to be known. In a game of poker these 5 cards consist of the 2 that the player is dealt and then the first 3 communal cards. After these cards are known the exact shuffle can be predicted and then it is known exactly what cards are remaining to come up, what cards other players have been dealt as well as who will win the game entirely. It does this by using a brute force method to generate

[20] http://www.planetpoker.com/
[21] http://www.planetpoker.com/about/index.asp
[22] https://www.cigital.com/papers/download/developer_gambling.php
[23] https://www.cigital.com/papers/download/developer_gambling.php

shuffles until the exact shuffle with the cards in order has been found. This is only possible through them narrowing down the possibilities because of the errors in the algorithm used so the possibilities are so few. After doing this they now have the seed used to generate all shuffles for every game until they disconnect from the server.
This is a clear highlight of how dangerous it can be to rely on a poorly coded random number generator. If the seed of the number generator can be cracked then the generator is useless as the numbers are no longer random, they are known and predictable. This can lead to serious issues in security where random number generators are used to provide basic functions of the application. Such as in this case if the application depends on random number generators then it is vital they are as secure as possible to prevent some users from cheating the system.
In the case of PlanetPoker however they were very fortunate as this group of software security engineers didn't use the broken system to their advantage and instead alerted the site of their issue.

Looking at this issue for PlanetPoker from a business point of view it is clear to see how this would present a considerable problem for them. If people are able to bypass the even distribution of wins in the game, then they would effectively be just stealing money from the other players as there is no fair game taking place. If their customers found out about this they could receive significant outcry from their customer base and it could be a serious problem for them to manage to convince there customers to once again put their money into the site. As this whole issue was caused by the company's flawed algorithm, this leaves them with the responsibility of allowing the games security to be compromised. After a significant brand image hit such as this, regaining those regular customers that you will have lost is a difficult task. The move to remove real money bets from the site could have been an effective tactic by the business to seem more ethical by their customers. Removing the real money side of the game allows players to once again trust the site as they won't be losing any real money as a result of a flawed shuffling algorithm. In modern business where customers have freedom of the market it is essential to appear ethical to your customers as there has been a huge paradigm shift of caring more about ethicality of the business' they we use.

Conclusion

As stated by Donald Kuth, the security of random number generators is more crucial than at first thought. This is why it was important to investigate the topic for this essay, "To What Extent Can The Random Number Generator Math.Random() In Java Be Predicted Before Being Generated?". The old methods of generating random numbers are becoming insufficient. The capability of technology is rapidly increasing, meaning it is less times consuming to preform standard brute force attacks and also new methods and exploits are always being developed.

This leads on to how with the development of quantum computing on the rise, due to the advancements in both physics and computing, brute force methods of deciphering code will become almost unlimitedly more effective. This is a huge concern as brute force methods of attack are the most common way for users to obtain the needed information, in this case the seed for the number generator. As Google have shown in there resent test results[24], although these are still to be confirmed by external investigations, quantum computers can be 100,000,000 times faster at solving complex equations. Although this is currently absurdly expensive in the future it will become more affordable. This could lead to a huge increase in the strength of cyber-attacks as it will significantly reduce the time taken to complete a brute force attack. However it may be a long time before we see any practical uses of these machines and sceptics even claim that practical uses for these computers are farfetched.

When investigating random number generators, more specifically pseudo random number generators, it was quickly evident that they had the possibility to be predicted. This wasn't as easy as it first seemed due to the fact that the seed was required to be known (or the constants used in the Rnd = $A * X + C \bmod M$ equation). However, due to the open source nature of Java the constants were available for the method that was being investigated. This allowed for the manipulation of some Java methods in order to make it possible to discover the seed. After this it was simple enough to generate the string of random numbers that the method would produce, thus predicting the methods outcome. Looking at the results of the program there is conclusive evidence that in fact the method `math.random()` is possible to predict. After the first few executions of the program, it seemed the program was perfect at predicting the method. After 100 attempts it was assumed it had a 100% success rate, never predicting

[24] http://www.popsci.com/googles-quantum-computer-is-100-million-times-faster-than-yours

wrong or failing to provide a seed. However, in the final few times executing the program there was one instance where the program failed to identify a seed. After trying countless further times there were no other repeats of this event and there was no way to replicate the example. This highlights the ease of predicting the outcomes of the number generators. The prediction accuracy was also perfect, with no incorrect predictions over the 100 times the program was run. So it is clear to see that it is in fact possible to predict the random number generating method `math.random()` and to predict it to a high degree of accuracy.

Bibliography

Websites

Pseudo random number generators
Available from: https://www.khanacademy.org/computing/computer-science/cryptography/crypt/v/random-vs-pseudorandom-number-generators
(Accessed 03/12/15)

Genuine random numbers, generated by radioactive decay
Available from: http://www.fourmilab.ch/hotbits/
(Accessed 12/11/15)

How we learned to cheat at online poker
Available from: https://www.cigital.com/papers/download/developer_gambling.php
(Accessed 01/11/15)

PlanetPoker
Available from: http://www.planetpoker.com/
(Accessed 01/11/15)

Predicting the next math.random() in java
Available from: https://news.ycombinator.com/item?id=8253048
(Accessed 01/11/15)

Original Program
Available from: https://github.com/fta2012/ReplicatedRandom
(Accessed 12/11/15)

Predicting the next math.random() in java
Available from: http://franklinta.com/2014/08/31/predicting-the-next-math-random-in-java/
(Accessed 26/10/15)
How does java.util.random work and how good is it?
Available from:
http://www.javamex.com/tutorials/random_numbers/java_util_random_algorithm.shtml#.Vjy

CTbfhDIV
(Accessed 26/10/15)
Is multi-millionfold speedup proof that google is really quantum computing?
Available from: http://www.popsci.com/googles-quantum-computer-is-100-million-times-faster-than-yours
(Accessed 12/01/16)

Brute force definition
Available from: http://searchsecurity.techtarget.com/definition/brute-force-cracking
(Accessed 12/01/16)

Simulated annealing
Available from: https://en.wikipedia.org/wiki/Simulated_annealing
(Accessed 12/01/16)

Linear congruential generator.
Available from: https://xlinux.nist.gov/dads//HTML/linearCongruentGen.html/
(Accessed 12/01/16)

Radioactive decay definition
Available from: http://dictionary.reference.com/browse/radioactive-decay
(Accessed 12/01/16)

Geiger Muller tube
Available from: https://en.wikipedia.org/wiki/Geiger%E2%80%93M%C3%BCller_tube
(Accessed 12/01/16)

Pseudo random definition
Available from: http://dictionary.reference.com/browse/pseudorandom
(Accessed 12/01/16)

Moores law
Available from: http://www.mooreslaw.org/

(Accessed 12/01/16)

Math.random()
Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random
(Accessed 12/01/16)

Java definition
Available from: http://dictionary.reference.com/browse/java
(Accessed 12/01/16)

Random definition
Available from: http://dictionary.reference.com/browse/random
(Accessed 12/01/16)

Predict definition
Available from: http://www.oxforddictionaries.com/definition/english/predict
(Accessed 12/01/16)

Random number generator
Available from: https://www.sgi.com/tech/stl/RandomNumberGenerator.html
(Accessed 12/01/16)

Non-deterministic definition
Available from: http://www.yourdictionary.com/non-deterministic
(Accessed 12/01/16)

Literature

An Introduction to Genetic Algorithms
Available from: *Mitchell, Melanie (1996). An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press.*
(Accessed 12/01/16)

The Art of Computer Programming, Volume 2.

Available from: Donald Knuth; The Art of Computer Programming, Volume 2.

(Accessed 12/01/16)

Here is the Code that was adapted and used in the investigation

Original Program

Available from: https://github.com/fta2012/ReplicatedRandom

(Accessed 12/11/15)

Code used (addendum A)

```
import java.util.Random;


public class ReplicatedRandomTest {

    public static void main(String args[]) {

        System.out.println("Replicating from Math.random");

        ReplicatedRandom rr = new ReplicatedRandom();

        for (int i = 0; i < 2; i++) {

            if (rr.replicateState(Math.random())) {

                for (int j = 0; j < 2; j++)

                       System.out.println("Expected Random Number : "+rr.nextDouble() + "\t" +
                       "Actual Number Generated : "+Math.random());

                System.out.println();

            }

        }

    }

}


import java.util.ArrayList;

import java.util.Random;


public class ReplicatedRandom extends Random {

    // Replicate the state of a Random using a single value from its nextDouble

    public boolean replicateState(double nextDouble) {

        // nextDouble() is generated from ((next(26) << 27) + next(27)) / (1L << 53)

        // Inverting those operations will get us the values of next(26) and next(27)

        long numerator = (long)(nextDouble * (1L << 53));

        int first26 = (int)(numerator >>> 27);

        int last27 = (int)(numerator & ((1L << 27) - 1));

        return replicateState(first26, 26, last27, 27);

    }



    // Replicate the state of a Random using a single value from its nextLong

    public boolean replicateState(long nextLong) {

        int last32 = (int)(nextLong & ((1L << 32) - 1));

        int first32 = (int)((nextLong - last32) >> 32);

        return replicateState(first32, 32, last32, 32);

    }
```

```
    // Replicate the state of a Random using two consecutive values from its nextInt
    public boolean replicateState(int firstNextInt, int secondNextInt) {
        return replicateState(firstNextInt, 32, secondNextInt, 32);
    }

    // Replicate the state of a Random using two consecutive values from its nextFloat
    public boolean replicateState(float firstNextFloat, float secondNextFloat) {
        return replicateState((int)(firstNextFloat * (1 << 24)), 24, (int)(secondNextFloat *
(1 << 24)), 24);
    }

    public boolean replicateState(int nextN, int n, int nextM, int m) {
        // Constants copied from java.util.Random
        final long multiplier = 0x5DEECE66DL;
        final long addend = 0xBL;
        final long mask = (1L << 48) - 1;
        long upperMOf48Mask = ((1L << m) - 1) << (48 - m);
        // next(x) is generated by taking the upper x bits of 48 bits of (oldSeed * multiplier
+ addend) mod (mask + 1)
        // So now we have the upper n and m bits of two consecutive calls of next(n) and
next(m)
        long oldSeedUpperN = ((long)nextN << (48 - n)) & mask;
        long newSeedUpperM = ((long)nextM << (48 - m)) & mask;
        // Bruteforce the lower (48 - n) bits of the oldSeed that was truncated.
        // Calculate the next seed for each guess of oldSeed and check if it has the same top
m bits as our newSeed.
        // If it does then the guess is right and we can add that to our candidate seeds.
        ArrayList<Long> possibleSeeds = new ArrayList<Long>();
        for (long oldSeed = oldSeedUpperN; oldSeed <= (oldSeedUpperN | ((1L << (48 - n)) -
1)); oldSeed++) {
            long newSeed = (oldSeed * multiplier + addend) & mask;
            if ((newSeed & upperMOf48Mask) == newSeedUpperM) {
                possibleSeeds.add(newSeed);
            }
        }

        if (possibleSeeds.size() == 1) {
            // If there's only one candidate seed, then we found it!
            setSeed(possibleSeeds.get(0) ^ multiplier); // setSeed(x) sets seed to `(x ^
multiplier) & mask`, so we need another `^ multiplier` to cancel it out
            return true;
```

```
        }

        if (possibleSeeds.size() >= 1) {

            System.out.println("Didn't find a unique seed. Possible seeds were: " +
possibleSeeds);

        } else {

            System.out.println("Failed to find seed!");

        }

        return false;

    }

}
```