JEP 356: Enhanced Pseudo-Random Number Generators

Author	Guy Steele
Owner	Jim Laskey
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	17
Component	core-libs / java.util
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	M
Duration	M
Reviewed by	Brian Goetz
Endorsed by	Brian Goetz
Created	2017/12/07 19:09
Updated	2023/02/01 20:04
Issue	8193209

Summary

Provide new interface types and implementations for pseudorandom number generators (PRNGs), including jumpable PRNGs and an additional class of splittable PRNG algorithms (LXM).

Goals

- Make it easier to use various PRNG algorithms interchangeably in applications.
- Better support stream-based programming by providing streams of PRNG objects.
- Eliminate code duplication in existing PRNG classes.
- Carefully preserve existing behavior of class `java.util.Random`.

Non-Goals

It is not a goal to provide implementations of numerous other PRNG algorithms, only to provide a framework that can accommodate other PRNG algorithms. However, we have added three common algorithms that have already been widely deployed in other programming language environments.

Success Metrics

The output of the new LXM algorithms passes the existing well-known TestU01 and PractRand test suites.

- Pierre L'Ecuyer and Richard Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Transactions on Mathematical Software* 33, 4 (August 2007), article 22. ISSN 0098-3500. http://doi.acm.org/10.1145/1268776.1268777
- Richard Simard. TestU01 version 1.2.3 (August 2009). http://www.iro.umontreal.ca/~simardr/testu01/tu01.html
- Pierre L'Ecuyer and Richard Simard. *TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators: User's guide, compact version*. Département d'Informatique et de Recherche Opérationnelle, Univerité de Montréal, May 2013. http://www.iro.umontreal.ca/~simardr/testu01/guideshorttestu01.pdf
- Chris Doty-Humphrey. PractRand version 0.90. July 2014. http://pracrand.sourceforge.net [That's not a typo. The name of the software is "PractRand" but the SourceForge project name is "pracrand".]

Jumpable and leapable PRNG algorithms pass tests that verify the commutativity of certain operations.

Motivation

We focus on five areas for improvement in the area of pseudorandom number generators in Java:

- With the legacy PRNG classes `Random`, `ThreadLocalRandom`, and `SplittableRandom`, it is difficult to replace any one of them in an application with some other algorithm, despite the fact that they all support pretty much the same set of methods. For example, if an application uses instances of class `Random`, it will necessarily declare variables of type `Random`, which cannot hold instances of class `SplittableRandom`; changing the application to use `SplittableRandom` would require changing the type of every variable (including method parameters) used to hold a PRNG object. The one exception is that `ThreadLocalRandom` is a subclass of `Random`, purely to allow variables of type `Random` to hold instances of `ThreadLocalRandom`, yet `ThreadLocalRandom` overrides nearly all the methods of `Random`. Interfaces can easily address this.
- Legacy classes `Random`, `ThreadLocalRandom`, and `SplittableRandom` all support such methods as `nextDouble()` and `nextBoolean()` as well as stream-producing methods such as `ints()` and `longs()`, but they have completely independent and nearly copy-and-paste identical

implementations. Refactoring this code would made it easier to maintain and, moreover, documentation would makes it much easier for third parties to create new PRNG classes that also support the same complete suite of methods.

- In 2016, testing revealed two new weaknesses in the algorithm used by class `SplittableRandom`. On the one hand, a relatively minor revision can avoid those weaknesses. On the other hand, a new class of splittable PRNG algorithms (LXM) has also been discovered that are almost as fast, even easier to implement, and appear to completely avoid the three classes of weakness to which `SplittableRandom` is prone.
- Being able to obtain a stream of PRNG objects from a PRNG makes it much easier to express certain sorts of code using streaming methods.
- There are many PRNG algorithms in the literature that are not splittable but are jumpable (and perhaps also leapable, that is, capable of very long jumps as well as ordinary jumps), a property quite different from splitting that nevertheless also lends itself to supporting streams of PRNG objects. In the past, it has been difficult to take advantage of this property in Java. Examples of jumpable PRNG algorithms are Xoshiro256**, and Xoroshiro128+.
    - Xoshiro256** and Xoroshiro128+: http://xoshiro.di.unimi.it

Description

We provide a new interface, `RandomGenerator`, which supplies a uniform API for all existing and new PRNGs. `RandomGenerators` provide methods named `ints`, `longs`, `doubles`, `nextBoolean`, `nextInt`, `nextLong`, `nextDouble`, and `nextFloat`, with all their current parameter variations.

We provide four new specialized RandomGenerator interfaces:

- `SplittableRandomGenerator` extends `RandomGenerator` and also provides methods named `split` and `splits`. Splittability allows the user to spawn a new RandomGenerator from an existing RandomGenerator that will generally produce statistically independent results.
- `JumpableRandomGenerator` extends`RandomGenerator` and also provides methods named `jump` and `jumps`. Jumpability allows a user to jump ahead a moderate number of draws.
- `LeapableRandomGenerator` extends `RandomGenerator` and also provides methods named `leap` and `leaps`. Leapability allows a user to jump ahead a large number of draws.
- `ArbitrarilyJumpableRandomGenerator` extends `LeapableRandomGenerator` and also provides additional variations of `jump` and `jumps` that allow an arbitrary jump distance to be specified.

We provide a new class `RandomGeneratorFactory` which is used to locate and construct instances of `RandomGenerator` implementations. The `RandomGeneratorFactory` uses the `ServiceLoader.Provider` API to register `RandomGenerator` implementations.

We have refactored `Random`, `ThreadLocalRandom`, and `SplittableRandom` so as to share most of their implementation code and, furthermore, make that code reusable by other algorithms as well. This refactoring creates underlying non-public abstract classes `AbstractRandomGenerator`, `AbstractSplittableRandomGenerator`, `AbstractJumpableRandomGenerator`, `AbstractLeapableRandomGenerator`, and `AbstractArbitrarilyJumpableRandomGenerator`, each provide only implementations for methods `nextInt()`, `nextLong()`, and (if relevant) either `split()`, or `jump()`, or `jump()` and `leap()`, or `jump(distance)`. After this refactoring, `Random`, `ThreadLocalRandom`, and `SplittableRandom` inherit the `RandomGenerator` interface. Note that because `SecureRandom` is a subclass of `Random`, all instances of `SecureRandom` also automatically support the `RandomGenerator` interface, with no need to recode the `SecureRandom` class or any of its associated implementation engines.

We also added underlying non-public classes that extend `AbstractSplittableRandomGenerator` (and therefore implement `SplittableRandomGenerator` and `RandomGenerator`) to support six specific members of the LXM family of PRNG algorithms:

- `L32X64MixRandom`
- `L32X64StarStarRandom`
- `L64X128MixRandom`
- `L64X128StarStarRandom`
- `L64X256MixRandom`
- `L64X1024MixRandom`
- `L128X128MixRandom`
- `L128X256MixRandom`
- `L128X1024MixRandom`

The structure of the central nextLong (or nextInt) method of an LXM algorithm follows a suggestion in December 2017 by Sebastiano Vigna that using one LCG subgenerator and one xor-based subgenerator (rather than two LCG subgenerators) would provide a longer period, superior equidistribution, scalability, and better quality. Each of the specific implementations here combines one of the best currently known xor-based generators (xoroshiro or xoshiro, described by

Blackman and Vigna in "Scrambled Linear Pseudorandom Number Generators", ACM Trans. Math. Softw., 2021) with an LCG that uses one of the best currently known multipliers (found by a search for better multipliers in 2019 by Steele and Vigna), and then applies a mixing function identified by Doug Lea. Testing has confirmed that the LXM algorithm is far superior in quality to the SplitMix algorithm (2014) used by SplittableRandom.

We also provide implementations of these widely-used PRNG algorithms:

- `Xoshiro256PlusPlus`
- `Xoroshiro128PlusPlus`

The non-public abstract implementations mentioned above may be supplied as part of a random number implementor SPI in the future.

This suite of algorithms provide Java programmers with a reasonable range of tradeoffs among space, time, quality, and compatibility with other languages.

Alternatives

We considered simply introducing new interfaces while leaving the implementations of `Random`, `ThreadLocalRandom`, and `SplittableRandom` as is. This would help to make PRNG objects more easily interchangeable but would not make it any easier to implement them.

We considered refactoring `Random`, `ThreadLocalRandom`, and `SplittableRandom` without changing their functionality or adding any new interfaces. We believe this would reduce their overall memory footprint, but do nothing to make future PRNG algorithms easier to implement or use.

Testing

- All existing tests for `Random`, `ThreadLocalRandom`, and `SplittableRandom` should continue to be used.
- New test, probably to be applied just once: The output of the refactored versions of `Random`, `ThreadLocalRandom`, and `SplittableRandom` (before repairing the two newly detected weaknesses) should be spot-checked against the existing (JDK 8) implementations to verify that their behavior remains
  unchanged.
- New test, probably to be applied just once: The output of the LXM algorithms
  should be spot-checked against the C-coded versions used for quality verification with TestU01 and PractRand.
- New test, to become a permanent part of the test suite: The `jump()` and `leap()` methods should be tested to verify that they do travel around the state cycle by the claimed distance. For example, starting from any specific initial state, the sequence of operations `nextLong(); jump()` ought to leave a
  generator in the same state as the sequence of operations `jump(); nextLong()`.

Risks and Assumptions

We believe this is a medium project and the risks are minimal. Probably the largest burden has been crafting the specification and the second-largest has been testing.

Care has been give to ensure the behaviour of legacy random number generators has not been affected.