Caso práctico I

Simulación y Metaheurísticas

Estudio de generadores de números aleatorios

José Ignacio Escribano

Móstoles, 15 de mayo de 2016

Índice

1. Introducción

En este caso práctico, veremos qué algoritmos utilizan distintos lenguajes de programación para la generación de números aleatorios. En primer lugar veremos R y posteriormente, Java.

2. Generador de números aleatorios de R

La función encargada en R de generar números aleatorios se denomina `RNGkind`[1], que permite seleccionar qué método utilizar para generar números aleatorios. Los métodos disponibles son los siguientes:

- Wichmann-Hill: este generador de números aleatorios toma tres semillas, donde cada una de ellas está en el rango $[1, p_i - 1]$, donde $p = (30269, 30307, 30323)$. Este generador tiene una longitud de ciclo de $6.9536 \cdot 10^{12}$, que se corresponde con $\frac{1}{4}\prod_{i=1}^{3}(p_i - 1)$.
- Marsaglia-Multicarry: este generador tiene un período mayor de 2^{60}. Requiere una semilla dada por dos valores (todos los valores están permitidos).
- Super-Duper: este generador tiene período aproximado de $4.6 \cdot 10^{18}$. Requiere dos enteros para la semilla. El primer enteros permite todos los valores, y el segundo está restringido a ser un número impar.
- Mersenne-Twister: es la opción por defecto de R. Tiene período de $2^{19937} - 1$ y está equidistribuido en 623 dimensiones consecutivas (sobre todo el período completo). La semilla es un vector de 624 enteros de 32 bits, más la posición actual del vector.
- Knuth-TAOCP-2002: este generador está dado por la fórmula

  $$X_n = (X_{n-100} - X_{n-37}) \mod 2^{30}$$

  y la semilla es el conjunto de los últimos 100 números. El período es de aproximadamente de 2^{129}.
- Knuth-TAOCP: es una versión anterior del algoritmo anterior. Data de 1997. La inicialización de este generador puede llevar bastante tiempo.
- L'Ecuyer-CMRG: es un generador recursivo múltiple propuesto por L'Ecuyer en 1999. Su período es de alrededor de 2^{191}.

El código que implementa esta función es el siguiente:

[1] `https://stat.ethz.ch/R-manual/R-devel/library/base/html/Random.html`

```r
function (kind = NULL, normal.kind = NULL)
{
    kinds <- c("Wichmann-Hill", "Marsaglia-Multicarry", "Super-Duper",
        "Mersenne-Twister", "Knuth-TAOCP", "user-supplied", "Knuth-TAOCP-2002",
        "L'Ecuyer-CMRG", "default")
    n.kinds <- c("Buggy Kinderman-Ramage", "Ahrens-Dieter", "Box-Muller",
        "user-supplied", "Inversion", "Kinderman-Ramage", "default")
    do.set <- length(kind) > 0L
    if (do.set) {
        if (!is.character(kind) || length(kind) > 1L)
            stop("'kind' must be a character string of length 1 (RNG to be used).")
        if (is.na(i.knd <- pmatch(kind, kinds) - 1L))
            stop(gettextf("'%s' is not a valid abbreviation of an RNG",
                kind), domain = NA)
        if (i.knd == length(kinds) - 1L)
            i.knd <- -1L
    }
    else i.knd <- NULL
    if (!is.null(normal.kind)) {
        if (!is.character(normal.kind) || length(normal.kind) !=
            1L)
            stop("'normal.kind' must be a character string of length 1")
        normal.kind <- pmatch(normal.kind, n.kinds) - 1L
        if (is.na(normal.kind))
            stop(gettextf("'%s' is not a valid choice", normal.kind),
                domain = NA)
        if (normal.kind == 0L)
            warning("buggy version of Kinderman-Ramage generator used",
                domain = NA)
        if (normal.kind == length(n.kinds) - 1L)
            normal.kind <- -1L
    }
    r <- 1L + .Internal(RNGkind(i.knd, normal.kind))
    r <- c(kinds[r[1L]], n.kinds[r[2L]])
    if (do.set || !is.null(normal.kind))
        invisible(r)
    else r
}
```

Esta función toma dos argumentos: el primero es el algoritmo para generar números aleatorios y el segundo es el algoritmo para generar números aleatorios siguiendo una distribución normal de media 0 y desviación típica 1.

Los algoritmos para generar números aleatorios son los descritos anteriormente, y los algoritmos para generar números normalmente distribuidos son:

- Kinderman-Ramage
- Buggy Kinderman-Ramage
- Ahrens-Dieter
- Box-Muller
- Inversion

La función anterior se encarga de comprobar que los argumentos que se le pasan son correctos, y en caso afirmativo, establece ese algoritmo como generador de números aleatorios.

Notar, que además está función tiene la opción `user-supplied` para establecer nuestras propias funciones que generen números aleatorios. Una descripción completa sobre está opción se puede leer en `https://stat.ethz.ch/R-manual/R-devel/library/base/html/Random-user.html`.

3. Generador de números aleatorios de Java

El generador de números aleatorios de Java se encuentra en la clase `Random`[2]. Esta clase usa una semilla de 48 bits, que se modifica usando una fórmula congruencial lineal, es decir, de la forma

$$X_{n+1} = (aX_n + c) \mod m$$

En el caso de Java, se utilizan los siguientes parámetros:

$$a = 25214903917$$
$$c = 11$$
$$m = 2^{48}$$

Notar que el número 25214903917 se corresponde con $5DEECE66D_{16}$ en base hexadecimal. Este número escrito en base 16 lo encontraremos posteriormente escrito en el código.

La clase Random permite generar números aleatorios uniforme distribuidos en el intervalo $[0, n]$, y en particular en el intervalo $[0, 1]$. También es posible generar números aleatorios con una distribución normal $\mathcal{N}(0, 1)$.

La función encargada de establecer la semilla con la que se iniciará el generador congruencial tiene el nombre `setSeed` y recibe un parámetro `seed`, que establece la semilla a

[2]https://docs.oracle.com/javase/8/docs/api/index.html?java/util/Random.html

```
(seed ^ 0x5DEECE66DL) & ((1L << 48) - 1)
```

El código completo de la función es el siguiente:

```
public synchronized void setSeed(long seed)
{
        this.seed = (seed ^ 0x5DEECE66DL) & ((1L << 48) - 1);
        haveNextNextGaussian = false;
}
```

Notar que, de acuerdo a este código, si se establece la misma semilla, se generarán los mismos números aleatorios.

La función `next` es la encargada de generar números aleatorios en el intervalo $[0, 1]$. El código de esta función es el siguiente:

```
protected synchronized int next(int bits)
{
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        return (int) (seed >>> (48 - bits));
}
```

Esta función genera números aleatorios usando la fórmula del generador congruencial lineal:

$$X_{n+1} = (252149039 \cdot X_n + 11) \mod 2^{48}$$

Notar que se han traducido los valores decimales a hexadecimal para operar con ellos con operadores bit a bit (&, $\sim$, $\ll\gg$, $\lll$, $\ggg$, etc.) para trabajar más eficientemente.

La primera línea del código hace el módulo 2^{48} de $(252149039 \cdot X_n + 11)$ usando operaciones bit a bit. Notar que n & $(1 \ll 48) - 1$ es equivalente a hacer $n \mod 2^{48}$, si n es un número binario.

La segunda línea convierte el número en entero, que lo devuelve.

Por otro lado, para generar números aleatorios normalmente distribuidos, se utiliza la función `nextGaussian`, que genera un número con distribución $\mathcal{N}(0, 1)$. El código de esta función es:

```
public double nextGaussian() {
   if (haveNextNextGaussian) {
     haveNextNextGaussian = false;
     return nextNextGaussian;
   } else {
     double v1, v2, s;
     do {
       v1 = 2 * nextDouble() - 1;    // between -1.0 and 1.0
       v2 = 2 * nextDouble() - 1;    // between -1.0 and 1.0
       s = v1 * v1 + v2 * v2;
     } while (s >= 1 || s == 0);
     double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
     nextNextGaussian = v2 * multiplier;
     haveNextNextGaussian = true;
     return v1 * multiplier;
   }
 }
```

Este código se basa en el método polar de Marsaglia[3]. Se generan dos números aleatorios usando la función `nextDouble`[4,5]. Se calcula la norma euclídea de los dos números aleatorios y se repite el proceso hasta que la norma es menor que 1, es decir, si está dentro del círculo unidad. Por último, se devuelve el número con distribución normal dado por:

$$v1 \cdot \sqrt{\frac{-2\ln s}{s}}$$

donde s es la norma euclídea de los dos números aleatorios generados, $v1$ y $v2$.

El código de la clase Random se muestra en el Anexo A.

4. Conclusiones

En este caso práctico, hemos visto que cada lenguaje de programación tiene generadores de números aleatorios distintos. En el caso de R, es posible elegir entre distintos métodos. Sin embargo, en el caso de Java, sólo tiene un generador que es un generador congruencial lineal. Esto es de vital importancia cuando se quiere realizar una simulación: dependiendo de la calidad de los generadores de números aleatorios, obtendremos una mejor o peor simulación, por lo que a la hora de realizar una simulación deberemos documentarnos qué métodos usa cada lenguaje de programación y usar el que mejor convenga a nuestra simulación, teniendo en cuenta el conocimiento del lenguaje o el tipo de simulación. En este sentido, la página

[3] `https://en.wikipedia.org/wiki/Marsaglia_polar_method`

[4] La función `nextDouble` genera un número aleatorio en el intervalo $(0,1)$ usando la función `next`, ya explica anteriormente.

[5] La función `nextDouble` genera un número aleatorio en el intervalo $(0,1)$. Para generarlo en el intervalo $(-1,1)$ buscamos una función lineal $f:(0,1)\to(-1,1)$ de la forma $f(x)=ax+b$, con $a,b\in\mathbb{R}$. Imponiendo las condiciones $f(0)=-1$ y $f(1)=1$, se tiene que la función buscada es $f(x)=2x-1$, que es la usada en el código

web `https://rosettacode.org/wiki/Random_number_generator_(included)` proporciona qué algoritmos se implementan en cada lenguaje de programación y enlaces de utilidad hacia la documentación de cada uno de ellos.

A. Implementación de la clase Random de Java

A continuación, se muestra el código completo de la clase Random de Java[6].

```java
package java.util;
import java.io.*;
import java.util.concurrent.atomic.AtomicLong;
import java.util.function.DoubleConsumer;
import java.util.function.IntConsumer;
import java.util.function.LongConsumer;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;
import java.util.stream.StreamSupport;

import sun.misc.Unsafe;

/**
 * An instance of this class is used to generate a stream of
 * pseudorandom numbers. The class uses a 48-bit seed, which is
 * modified using a linear congruential formula. (See Donald Knuth,
 * <i>The Art of Computer Programming, Volume 2</i>, Section 3.2.1.)
 * <p>
 * If two instances of {@code Random} are created with the same
 * seed, and the same sequence of method calls is made for each, they
 * will generate and return identical sequences of numbers. In order to
 * guarantee this property, particular algorithms are specified for the
 * class {@code Random}. Java implementations must use all the algorithms
 * shown here for the class {@code Random}, for the sake of absolute
 * portability of Java code. However, subclasses of class {@code Random}
 * are permitted to use other algorithms, so long as they adhere to the
 * general contracts for all the methods.
 * <p>
 * The algorithms implemented by class {@code Random} use a
 * {@code protected} utility method that on each invocation can supply
 * up to 32 pseudorandomly generated bits.
 * <p>
 * Many applications will find the method {@link Math#random} simpler to use.
 *
 * <p>Instances of {@code java.util.Random} are threadsafe.
 * However, the concurrent use of the same {@code java.util.Random}
```

[6]http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/Random.java

```
 * instance across threads may encounter contention and consequent
 * poor performance. Consider instead using
 * {@link java.util.concurrent.ThreadLocalRandom} in multithreaded
 * designs.
 *
 * <p>Instances of {@code java.util.Random} are not cryptographically
 * secure.  Consider instead using {@link java.security.SecureRandom} to
 * get a cryptographically secure pseudo-random number generator for use
 * by security-sensitive applications.
 *
 * @author  Frank Yellin
 * @since   1.0
 */
public
class Random implements java.io.Serializable {
    /** use serialVersionUID from JDK 1.1 for interoperability */
    static final long serialVersionUID = 3905348978240129619L;

    /**
     * The internal state associated with this pseudorandom number generator.
     * (The specs for the methods in this class describe the ongoing
     * computation of this value.)
     */
    private final AtomicLong seed;

    private static final long multiplier = 0x5DEECE66DL;
    private static final long addend = 0xBL;
    private static final long mask = (1L << 48) - 1;

    private static final double DOUBLE_UNIT = 0x1.0p-53; // 1.0 / (1L << 53)

    // IllegalArgumentException messages
    static final String BadBound = "bound must be positive";
    static final String BadRange = "bound must be greater than origin";
    static final String BadSize  = "size must be non-negative";

    /**
     * Creates a new random number generator. This constructor sets
     * the seed of the random number generator to a value very likely
     * to be distinct from any other invocation of this constructor.
     */
    public Random() {
        this(seedUniquifier() ^ System.nanoTime());
    }

    private static long seedUniquifier() {
        // L'Ecuyer, "Tables of Linear Congruential Generators of
        // Different Sizes and Good Lattice Structure", 1999
        for (;;) {
            long current = seedUniquifier.get();
            long next = current * 181783497276652981L;
            if (seedUniquifier.compareAndSet(current, next))
                return next;
        }
    }

    private static final AtomicLong seedUniquifier
        = new AtomicLong(8682522807148012L);

    /**
     * Creates a new random number generator using a single {@code long} seed.
     * The seed is the initial value of the internal state of the pseudorandom
     * number generator which is maintained by method {@link #next}.
     *
     * <p>The invocation {@code new Random(seed)} is equivalent to:
     *  <pre> {@code
     * Random rnd = new Random();
     * rnd.setSeed(seed);}</pre>
     *
     * @param seed the initial seed
     * @see   #setSeed(long)
     */
    public Random(long seed) {
        if (getClass() == Random.class)
            this.seed = new AtomicLong(initialScramble(seed));
        else {
            // subclass might have overriden setSeed
            this.seed = new AtomicLong();
            setSeed(seed);
        }
    }

    private static long initialScramble(long seed) {
        return (seed ^ multiplier) & mask;
    }
```

```java
/**
 * Sets the seed of this random number generator using a single
 * {@code long} seed. The general contract of {@code setSeed} is
 * that it alters the state of this random number generator object
 * so as to be in exactly the same state as if it had just been
 * created with the argument {@code seed} as a seed. The method
 * {@code setSeed} is implemented by class {@code Random} by
 * atomically updating the seed to
 *  <pre>{@code (seed ^ 0x5DEECE66DL) & ((1L << 48) - 1)}</pre>
 * and clearing the {@code haveNextNextGaussian} flag used by {@link
 * #nextGaussian}.
 *
 * <p>The implementation of {@code setSeed} by class {@code Random}
 * happens to use only 48 bits of the given seed. In general, however,
 * an overriding method may use all 64 bits of the {@code long}
 * argument as a seed value.
 *
 * @param seed the initial seed
 */
synchronized public void setSeed(long seed) {
    this.seed.set(initialScramble(seed));
    haveNextNextGaussian = false;
}

/**
 * Generates the next pseudorandom number. Subclasses should
 * override this, as this is used by all other methods.
 *
 * <p>The general contract of {@code next} is that it returns an
 * {@code int} value and if the argument {@code bits} is between
 * {@code 1} and {@code 32} (inclusive), then that many low-order
 * bits of the returned value will be (approximately) independently
 * chosen bit values, each of which is (approximately) equally
 * likely to be {@code 0} or {@code 1}. The method {@code next} is
 * implemented by class {@code Random} by atomically updating the seed to
 *  <pre>{@code (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)}</pre>
 * and returning
 *  <pre>{@code (int)(seed >>> (48 - bits))}.</pre>
 *
 * This is a linear congruential pseudorandom number generator, as
 * defined by D. H. Lehmer and described by Donald E. Knuth in
 * <i>The Art of Computer Programming,</i> Volume 3:
 * <i>Seminumerical Algorithms</i>, section 3.2.1.
```

```
     *
     * @param  bits random bits
     * @return the next pseudorandom value from this random number
     *         generator's sequence
     * @since  1.1
     */
    protected int next(int bits) {
        long oldseed, nextseed;
        AtomicLong seed = this.seed;
        do {
            oldseed = seed.get();
            nextseed = (oldseed * multiplier + addend) & mask;
        } while (!seed.compareAndSet(oldseed, nextseed));
        return (int)(nextseed >>> (48 - bits));
    }

    /**
     * Generates random bytes and places them into a user-supplied
     * byte array.  The number of random bytes produced is equal to
     * the length of the byte array.
     *
     * <p>The method {@code nextBytes} is implemented by class {@code Random}
     * as if by:
     *  <pre> {@code
     * public void nextBytes(byte[] bytes) {
     *   for (int i = 0; i < bytes.length; )
     *     for (int rnd = nextInt(), n = Math.min(bytes.length - i, 4);
     *          n-- > 0; rnd >>= 8)
     *       bytes[i++] = (byte)rnd;
     * }}</pre>
     *
     * @param  bytes the byte array to fill with random bytes
     * @throws NullPointerException if the byte array is null
     * @since  1.1
     */
    public void nextBytes(byte[] bytes) {
        for (int i = 0, len = bytes.length; i < len; )
            for (int rnd = nextInt(),
                     n = Math.min(len - i, Integer.SIZE/Byte.SIZE);
                 n-- > 0; rnd >>= Byte.SIZE)
                bytes[i++] = (byte)rnd;
    }
```

```java
/**
 * The form of nextLong used by LongStream Spliterators.  If
 * origin is greater than bound, acts as unbounded form of
 * nextLong, else as bounded form.
 *
 * @param origin the least value, unless greater than bound
 * @param bound the upper bound (exclusive), must not equal origin
 * @return a pseudorandom value
 */
final long internalNextLong(long origin, long bound) {
    long r = nextLong();
    if (origin < bound) {
        long n = bound - origin, m = n - 1;
        if ((n & m) == 0L)  // power of two
            r = (r & m) + origin;
        else if (n > 0L) {  // reject over-represented candidates
            for (long u = r >>> 1;            // ensure nonnegative
                 u + m - (r = u % n) < 0L;    // rejection check
                 u = nextLong() >>> 1) // retry
                ;
            r += origin;
        }
        else {              // range not representable as long
            while (r < origin || r >= bound)
                r = nextLong();
        }
    }
    return r;
}

/**
 * The form of nextInt used by IntStream Spliterators.
 * For the unbounded case: uses nextInt().
 * For the bounded case with representable range: uses nextInt(int bound)
 * For the bounded case with unrepresentable range: uses nextInt()
 *
 * @param origin the least value, unless greater than bound
 * @param bound the upper bound (exclusive), must not equal origin
 * @return a pseudorandom value
 */
final int internalNextInt(int origin, int bound) {
    if (origin < bound) {
        int n = bound - origin;
```

```
            if (n > 0) {
                return nextInt(n) + origin;
            }
            else {  // range not representable as int
                int r;
                do {
                    r = nextInt();
                } while (r < origin || r >= bound);
                return r;
            }
        }
        else {
            return nextInt();
        }
    }

    /**
     * The form of nextDouble used by DoubleStream Spliterators.
     *
     * @param origin the least value, unless greater than bound
     * @param bound the upper bound (exclusive), must not equal origin
     * @return a pseudorandom value
     */
    final double internalNextDouble(double origin, double bound) {
        double r = nextDouble();
        if (origin < bound) {
            r = r * (bound - origin) + origin;
            if (r >= bound) // correct for rounding
                r = Double.longBitsToDouble(Double.doubleToLongBits(bound) - 1);
        }
        return r;
    }

    /**
     * Returns the next pseudorandom, uniformly distributed {@code int}
     * value from this random number generator's sequence. The general
     * contract of {@code nextInt} is that one {@code int} value is
     * pseudorandomly generated and returned. All 2<sup>32</sup> possible
     * {@code int} values are produced with (approximately) equal probability.
     *
     * <p>The method {@code nextInt} is implemented by class {@code Random}
     * as if by:
     *  <pre> {@code
     * public int nextInt() {
     *   return next(32);
     * }}</pre>
     *
     * @return the next pseudorandom, uniformly distributed {@code int}
     *         value from this random number generator's sequence
     */
    public int nextInt() {
        return next(32);
    }

    /**
     * Returns a pseudorandom, uniformly distributed {@code int} value
     * between 0 (inclusive) and the specified value (exclusive), drawn from
     * this random number generator's sequence.  The general contract of
     * {@code nextInt} is that one {@code int} value in the specified range
     * is pseudorandomly generated and returned.  All {@code bound} possible
     * {@code int} values are produced with (approximately) equal
     * probability.  The method {@code nextInt(int bound)} is implemented by
     * class {@code Random} as if by:
     *  <pre> {@code
     * public int nextInt(int bound) {
     *   if (bound <= 0)
     *     throw new IllegalArgumentException("bound must be positive");
     *
     *   if ((bound & -bound) == bound)  // i.e., bound is a power of 2
     *     return (int)((bound * (long)next(31)) >> 31);
     *
     *   int bits, val;
     *   do {
     *       bits = next(31);
     *       val = bits % bound;
     *   } while (bits - val + (bound-1) < 0);
     *   return val;
     * }}</pre>
     *
     * <p>The hedge "approximately" is used in the foregoing description only
     * because the next method is only approximately an unbiased source of
     * independently chosen bits.  If it were a perfect source of randomly
     * chosen bits, then the algorithm shown would choose {@code int}
     * values from the stated range with perfect uniformity.
     * <p>
     * The algorithm is slightly tricky.  It rejects values that would result
     * in an uneven distribution (due to the fact that 2^31 is not divisible
     * by n). The probability of a value being rejected depends on n.  The
     * worst case is n=2^30+1, for which the probability of a reject is 1/2,
     * and the expected number of iterations before the loop terminates is 2.
     * <p>
     * The algorithm treats the case where n is a power of two specially: it
     * returns the correct number of high-order bits from the underlying
     * pseudo-random number generator.  In the absence of special treatment,
     * the correct number of <i>low-order</i> bits would be returned.  Linear
     * congruential pseudo-random number generators such as the one
     * implemented by this class are known to have short periods in the
     * sequence of values of their low-order bits.  Thus, this special case
     * greatly increases the length of the sequence of values returned by
     * successive calls to this method if n is a small power of two.
     *
     * @param bound the upper bound (exclusive).  Must be positive.
     * @return the next pseudorandom, uniformly distributed {@code int}
     *         value between zero (inclusive) and {@code bound} (exclusive)
     *         from this random number generator's sequence
     * @throws IllegalArgumentException if bound is not positive
     * @since 1.2
     */
    public int nextInt(int bound) {
        if (bound <= 0)
            throw new IllegalArgumentException(BadBound);

        int r = next(31);
        int m = bound - 1;
        if ((bound & m) == 0)  // i.e., bound is a power of 2
            r = (int)((bound * (long)r) >> 31);
        else {
            for (int u = r;
                 u - (r = u % bound) + m < 0;
                 u = next(31))
                ;
        }
        return r;
    }

    /**
     * Returns the next pseudorandom, uniformly distributed {@code long}
     * value from this random number generator's sequence. The general
     * contract of {@code nextLong} is that one {@code long} value is
     * pseudorandomly generated and returned.
     *
     * <p>The method {@code nextLong} is implemented by class {@code Random}
     * as if by:
     *  <pre> {@code
     * public long nextLong() {
     *   return ((long)next(32) << 32) + next(32);
     * }}</pre>
     *
     * Because class {@code Random} uses a seed with only 48 bits,
     * this algorithm will not return all possible {@code long} values.
     *
     * @return the next pseudorandom, uniformly distributed {@code long}
     *         value from this random number generator's sequence
     */
    public long nextLong() {
        // it's okay that the bottom word remains signed.
        return ((long)(next(32)) << 32) + next(32);
    }

    /**
     * Returns the next pseudorandom, uniformly distributed
     * {@code boolean} value from this random number generator's
     * sequence. The general contract of {@code nextBoolean} is that one
     * {@code boolean} value is pseudorandomly generated and returned.  The
     * values {@code true} and {@code false} are produced with
     * (approximately) equal probability.
     *
     * <p>The method {@code nextBoolean} is implemented by class {@code Random}
     * as if by:
     *  <pre> {@code
     * public boolean nextBoolean() {
     *   return next(1) != 0;
     * }}</pre>
     *
     * @return the next pseudorandom, uniformly distributed
     *         {@code boolean} value from this random number generator's
     *         sequence
     * @since 1.2
     */
    public boolean nextBoolean() {
        return next(1) != 0;
    }
```

```java
/**
 * Returns the next pseudorandom, uniformly distributed {@code float}
 * value between {@code 0.0} and {@code 1.0} from this random
 * number generator's sequence.
 *
 * <p>The general contract of {@code nextFloat} is that one
 * {@code float} value, chosen (approximately) uniformly from the
 * range {@code 0.0f} (inclusive) to {@code 1.0f} (exclusive), is
 * pseudorandomly generated and returned. All 2<sup>24</sup> possible
 * {@code float} values of the form <i>m&nbsp;x&nbsp;</i>2<sup>-24</sup>,
 * where <i>m</i> is a positive integer less than 2<sup>24</sup>, are
 * produced with (approximately) equal probability.
 *
 * <p>The method {@code nextFloat} is implemented by class {@code Random}
 * as if by:
 *  <pre> {@code
 * public float nextFloat() {
 *   return next(24) / ((float)(1 << 24));
 * }}</pre>
 *
 * <p>The hedge "approximately" is used in the foregoing description only
 * because the next method is only approximately an unbiased source of
 * independently chosen bits. If it were a perfect source of randomly
 * chosen bits, then the algorithm shown would choose {@code float}
 * values from the stated range with perfect uniformity.<p>
 * [In early versions of Java, the result was incorrectly calculated as:
 *  <pre> {@code
 *   return next(30) / ((float)(1 << 30));}</pre>
 * This might seem to be equivalent, if not better, but in fact it
 * introduced a slight nonuniformity because of the bias in the rounding
 * of floating-point numbers: it was slightly more likely that the
 * low-order bit of the significand would be 0 than that it would be 1.]
 *
 * @return the next pseudorandom, uniformly distributed {@code float}
 *         value between {@code 0.0} and {@code 1.0} from this
 *         random number generator's sequence
 */
public float nextFloat() {
    return next(24) / ((float)(1 << 24));
}

/**
```

```
     * Returns the next pseudorandom, uniformly distributed
     * {@code double} value between {@code 0.0} and
     * {@code 1.0} from this random number generator's sequence.
     *
     * <p>The general contract of {@code nextDouble} is that one
     * {@code double} value, chosen (approximately) uniformly from the
     * range {@code 0.0d} (inclusive) to {@code 1.0d} (exclusive), is
     * pseudorandomly generated and returned.
     *
     * <p>The method {@code nextDouble} is implemented by class {@code Random}
     * as if by:
     *  <pre> {@code
     * public double nextDouble() {
     *   return (((long)next(26) << 27) + next(27))
     *     / (double)(1L << 53);
     * }}</pre>
     *
     * <p>The hedge "approximately" is used in the foregoing description only
     * because the {@code next} method is only approximately an unbiased
     * source of independently chosen bits. If it were a perfect source of
     * randomly chosen bits, then the algorithm shown would choose
     * {@code double} values from the stated range with perfect uniformity.
     * <p>[In early versions of Java, the result was incorrectly calculated as:
     *  <pre> {@code
     *   return (((long)next(27) << 27) + next(27))
     *     / (double)(1L << 54);}</pre>
     * This might seem to be equivalent, if not better, but in fact it
     * introduced a large nonuniformity because of the bias in the rounding
     * of floating-point numbers: it was three times as likely that the
     * low-order bit of the significand would be 0 than that it would be 1!
     * This nonuniformity probably doesn't matter much in practice, but we
     * strive for perfection.]
     *
     * @return the next pseudorandom, uniformly distributed {@code double}
     *         value between {@code 0.0} and {@code 1.0} from this
     *         random number generator's sequence
     * @see Math#random
     */
    public double nextDouble() {
        return (((long)(next(26)) << 27) + next(27)) * DOUBLE_UNIT;
    }

    private double nextNextGaussian;
```

```java
private boolean haveNextNextGaussian = false;

/**
 * Returns the next pseudorandom, Gaussian ("normally") distributed
 * {@code double} value with mean {@code 0.0} and standard
 * deviation {@code 1.0} from this random number generator's sequence.
 * <p>
 * The general contract of {@code nextGaussian} is that one
 * {@code double} value, chosen from (approximately) the usual
 * normal distribution with mean {@code 0.0} and standard deviation
 * {@code 1.0}, is pseudorandomly generated and returned.
 *
 * <p>The method {@code nextGaussian} is implemented by class
 * {@code Random} as if by a threadsafe version of the following:
 *  <pre> {@code
 * private double nextNextGaussian;
 * private boolean haveNextNextGaussian = false;
 *
 * public double nextGaussian() {
 *   if (haveNextNextGaussian) {
 *     haveNextNextGaussian = false;
 *     return nextNextGaussian;
 *   } else {
 *     double v1, v2, s;
 *     do {
 *       v1 = 2 * nextDouble() - 1;   // between -1.0 and 1.0
 *       v2 = 2 * nextDouble() - 1;   // between -1.0 and 1.0
 *       s = v1 * v1 + v2 * v2;
 *     } while (s >= 1 || s == 0);
 *     double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
 *     nextNextGaussian = v2 * multiplier;
 *     haveNextNextGaussian = true;
 *     return v1 * multiplier;
 *   }
 * }}</pre>
 * This uses the <i>polar method</i> of G. E. P. Box, M. E. Muller, and
 * G. Marsaglia, as described by Donald E. Knuth in <i>The Art of
 * Computer Programming</i>, Volume 3: <i>Seminumerical Algorithms</i>,
 * section 3.4.1, subsection C, algorithm P. Note that it generates two
 * independent values at the cost of only one call to {@code StrictMath.log}
 * and one call to {@code StrictMath.sqrt}.
 *
 * @return the next pseudorandom, Gaussian ("normally") distributed
 *         {@code double} value with mean {@code 0.0} and
 *         standard deviation {@code 1.0} from this random number
 *         generator's sequence
 */
synchronized public double nextGaussian() {
    // See Knuth, ACP, Section 3.4.1 Algorithm C.
    if (haveNextNextGaussian) {
        haveNextNextGaussian = false;
        return nextNextGaussian;
    } else {
        double v1, v2, s;
        do {
            v1 = 2 * nextDouble() - 1; // between -1 and 1
            v2 = 2 * nextDouble() - 1; // between -1 and 1
            s = v1 * v1 + v2 * v2;
        } while (s >= 1 || s == 0);
        double multiplier = StrictMath.sqrt(-2 * StrictMath.log(s)/s);
        nextNextGaussian = v2 * multiplier;
        haveNextNextGaussian = true;
        return v1 * multiplier;
    }
}

// stream methods, coded in a way intended to better isolate for
// maintenance purposes the small differences across forms.

/**
 * Returns a stream producing the given {@code streamSize} number of
 * pseudorandom {@code int} values.
 *
 * <p>A pseudorandom {@code int} value is generated as if it's the result of
 * calling the method {@link #nextInt()}.
 *
 * @param streamSize the number of values to generate
 * @return a stream of pseudorandom {@code int} values
 * @throws IllegalArgumentException if {@code streamSize} is
 *         less than zero
 * @since 1.8
 */
public IntStream ints(long streamSize) {
    if (streamSize < 0L)
        throw new IllegalArgumentException(BadSize);
    return StreamSupport.intStream
```

```
                (new RandomIntsSpliterator
                         (this, 0L, streamSize, Integer.MAX_VALUE, 0),
                 false);
    }

    /**
     * Returns an effectively unlimited stream of pseudorandom {@code int}
     * values.
     *
     * <p>A pseudorandom {@code int} value is generated as if it's the result of
     * calling the method {@link #nextInt()}.
     *
     * @implNote This method is implemented to be equivalent to {@code
     * ints(Long.MAX_VALUE)}.
     *
     * @return a stream of pseudorandom {@code int} values
     * @since 1.8
     */
    public IntStream ints() {
        return StreamSupport.intStream
                (new RandomIntsSpliterator
                         (this, 0L, Long.MAX_VALUE, Integer.MAX_VALUE, 0),
                 false);
    }

    /**
     * Returns a stream producing the given {@code streamSize} number
     * of pseudorandom {@code int} values, each conforming to the given
     * origin (inclusive) and bound (exclusive).
     *
     * <p>A pseudorandom {@code int} value is generated as if it's the result of
     * calling the following method with the origin and bound:
     * <pre> {@code
     * int nextInt(int origin, int bound) {
     *   int n = bound - origin;
     *   if (n > 0) {
     *     return nextInt(n) + origin;
     *   }
     *   else {  // range not representable as int
     *     int r;
     *     do {
     *       r = nextInt();
     *     } while (r < origin || r >= bound);
     *         return r;
     *     }
     * }}</pre>
     *
     * @param streamSize the number of values to generate
     * @param randomNumberOrigin the origin (inclusive) of each random value
     * @param randomNumberBound the bound (exclusive) of each random value
     * @return a stream of pseudorandom {@code int} values,
     *         each with the given origin (inclusive) and bound (exclusive)
     * @throws IllegalArgumentException if {@code streamSize} is
     *         less than zero, or {@code randomNumberOrigin}
     *         is greater than or equal to {@code randomNumberBound}
     * @since 1.8
     */
    public IntStream ints(long streamSize, int randomNumberOrigin,
                          int randomNumberBound) {
        if (streamSize < 0L)
            throw new IllegalArgumentException(BadSize);
        if (randomNumberOrigin >= randomNumberBound)
            throw new IllegalArgumentException(BadRange);
        return StreamSupport.intStream
                (new RandomIntsSpliterator
                         (this, 0L, streamSize, randomNumberOrigin, randomNumberBound),
                 false);
    }

    /**
     * Returns an effectively unlimited stream of pseudorandom {@code
     * int} values, each conforming to the given origin (inclusive) and bound
     * (exclusive).
     *
     * <p>A pseudorandom {@code int} value is generated as if it's the result of
     * calling the following method with the origin and bound:
     * <pre> {@code
     * int nextInt(int origin, int bound) {
     *   int n = bound - origin;
     *   if (n > 0) {
     *     return nextInt(n) + origin;
     *   }
     *   else {  // range not representable as int
     *     int r;
     *     do {
     *       r = nextInt();
     *     } while (r < origin || r >= bound);
     *     return r;
     *   }
     * }}</pre>
     *
     * @implNote This method is implemented to be equivalent to {@code
     * ints(Long.MAX_VALUE, randomNumberOrigin, randomNumberBound)}.
     *
     * @param randomNumberOrigin the origin (inclusive) of each random value
     * @param randomNumberBound the bound (exclusive) of each random value
     * @return a stream of pseudorandom {@code int} values,
     *         each with the given origin (inclusive) and bound (exclusive)
     * @throws IllegalArgumentException if {@code randomNumberOrigin}
     *         is greater than or equal to {@code randomNumberBound}
     * @since 1.8
     */
    public IntStream ints(int randomNumberOrigin, int randomNumberBound) {
        if (randomNumberOrigin >= randomNumberBound)
            throw new IllegalArgumentException(BadRange);
        return StreamSupport.intStream
                (new RandomIntsSpliterator
                         (this, 0L, Long.MAX_VALUE, randomNumberOrigin, randomNumbe
                 false);
    }

    /**
     * Returns a stream producing the given {@code streamSize} number of
     * pseudorandom {@code long} values.
     *
     * <p>A pseudorandom {@code long} value is generated as if it's the result
     * of calling the method {@link #nextLong()}.
     *
     * @param streamSize the number of values to generate
     * @return a stream of pseudorandom {@code long} values
     * @throws IllegalArgumentException if {@code streamSize} is
     *         less than zero
     * @since 1.8
     */
    public LongStream longs(long streamSize) {
        if (streamSize < 0L)
            throw new IllegalArgumentException(BadSize);
        return StreamSupport.longStream
                (new RandomLongsSpliterator
                        (this, 0L, streamSize, Long.MAX_VALUE, 0L),
                false);
    }

    /**
     * Returns an effectively unlimited stream of pseudorandom {@code long}
     * values.
     *
     * <p>A pseudorandom {@code long} value is generated as if it's the result
     * of calling the method {@link #nextLong()}.
     *
     * @implNote This method is implemented to be equivalent to {@code
     * longs(Long.MAX_VALUE)}.
     *
     * @return a stream of pseudorandom {@code long} values
     * @since 1.8
     */
    public LongStream longs() {
        return StreamSupport.longStream
                (new RandomLongsSpliterator
                        (this, 0L, Long.MAX_VALUE, Long.MAX_VALUE, 0L),
                 false);
    }

    /**
     * Returns a stream producing the given {@code streamSize} number of
     * pseudorandom {@code long}, each conforming to the given origin
     * (inclusive) and bound (exclusive).
     *
     * <p>A pseudorandom {@code long} value is generated as if it's the result
     * of calling the following method with the origin and bound:
     * <pre> {@code
     * long nextLong(long origin, long bound) {
     *   long r = nextLong();
     *   long n = bound - origin, m = n - 1;
     *   if ((n & m) == 0L)  // power of two
     *     r = (r & m) + origin;
     *   else if (n > 0L) {  // reject over-represented candidates
     *     for (long u = r >>> 1;            // ensure nonnegative
     *          u + m - (r = u % n) < 0L;    // rejection check
     *          u = nextLong() >>> 1) // retry
     *         ;
     *     r += origin;
     *   }
     *   else {              // range not representable as long
     *     while (r < origin || r >= bound)
     *       r = nextLong();
     *   }
     *   return r;
     * }}</pre>
     *
     * @param streamSize the number of values to generate
     * @param randomNumberOrigin the origin (inclusive) of each random value
     * @param randomNumberBound the bound (exclusive) of each random value
     * @return a stream of pseudorandom {@code long} values,
     *         each with the given origin (inclusive) and bound (exclusive)
     * @throws IllegalArgumentException if {@code streamSize} is
     *         less than zero, or {@code randomNumberOrigin}
     *         is greater than or equal to {@code randomNumberBound}
     * @since 1.8
     */
    public LongStream longs(long streamSize, long randomNumberOrigin,
                            long randomNumberBound) {
        if (streamSize < 0L)
            throw new IllegalArgumentException(BadSize);
        if (randomNumberOrigin >= randomNumberBound)
            throw new IllegalArgumentException(BadRange);
        return StreamSupport.longStream
                (new RandomLongsSpliterator
                         (this, 0L, streamSize, randomNumberOrigin, randomNumberBou
                 false);
    }

    /**
     * Returns an effectively unlimited stream of pseudorandom {@code
     * long} values, each conforming to the given origin (inclusive) and bound
     * (exclusive).
     *
     * <p>A pseudorandom {@code long} value is generated as if it's the result
     * of calling the following method with the origin and bound:
     * <pre> {@code
     * long nextLong(long origin, long bound) {
     *   long r = nextLong();
     *   long n = bound - origin, m = n - 1;
     *   if ((n & m) == 0L)  // power of two
     *     r = (r & m) + origin;
```

```
     *   else if (n > 0L) {  // reject over-represented candidates
     *     for (long u = r >>> 1;            // ensure nonnegative
     *          u + m - (r = u % n) < 0L;    // rejection check
     *          u = nextLong() >>> 1) // retry
     *         ;
     *     r += origin;
     *   }
     *   else {              // range not representable as long
     *     while (r < origin || r >= bound)
     *       r = nextLong();
     *   }
     *   return r;
     * }}</pre>
     *
     * @implNote This method is implemented to be equivalent to {@code
     * longs(Long.MAX_VALUE, randomNumberOrigin, randomNumberBound)}.
     *
     * @param randomNumberOrigin the origin (inclusive) of each random value
     * @param randomNumberBound the bound (exclusive) of each random value
     * @return a stream of pseudorandom {@code long} values,
     *         each with the given origin (inclusive) and bound (exclusive)
     * @throws IllegalArgumentException if {@code randomNumberOrigin}
     *         is greater than or equal to {@code randomNumberBound}
     * @since 1.8
     */
    public LongStream longs(long randomNumberOrigin, long randomNumberBound) {
        if (randomNumberOrigin >= randomNumberBound)
            throw new IllegalArgumentException(BadRange);
        return StreamSupport.longStream
                (new RandomLongsSpliterator
                         (this, 0L, Long.MAX_VALUE, randomNumberOrigin, randomNumberBoun
                 false);
    }

    /**
     * Returns a stream producing the given {@code streamSize} number of
     * pseudorandom {@code double} values, each between zero
     * (inclusive) and one (exclusive).
     *
     * <p>A pseudorandom {@code double} value is generated as if it's the result
     * of calling the method {@link #nextDouble()}}.
     *
     * @param streamSize the number of values to generate
     * @return a stream of {@code double} values
     * @throws IllegalArgumentException if {@code streamSize} is
     *         less than zero
     * @since 1.8
     */
    public DoubleStream doubles(long streamSize) {
        if (streamSize < 0L)
            throw new IllegalArgumentException(BadSize);
        return StreamSupport.doubleStream
                (new RandomDoublesSpliterator
                         (this, 0L, streamSize, Double.MAX_VALUE, 0.0),
                 false);
    }

    /**
     * Returns an effectively unlimited stream of pseudorandom {@code
     * double} values, each between zero (inclusive) and one
     * (exclusive).
     *
     * <p>A pseudorandom {@code double} value is generated as if it's the result
     * of calling the method {@link #nextDouble()}}.
     *
     * @implNote This method is implemented to be equivalent to {@code
     * doubles(Long.MAX_VALUE)}.
     *
     * @return a stream of pseudorandom {@code double} values
     * @since 1.8
     */
    public DoubleStream doubles() {
        return StreamSupport.doubleStream
                (new RandomDoublesSpliterator
                         (this, 0L, Long.MAX_VALUE, Double.MAX_VALUE, 0.0),
                 false);
    }

    /**
     * Returns a stream producing the given {@code streamSize} number of
     * pseudorandom {@code double} values, each conforming to the given origin
     * (inclusive) and bound (exclusive).
     *
     * <p>A pseudorandom {@code double} value is generated as if it's the result
     * of calling the following method with the origin and bound:
     * <pre> {@code
     * double nextDouble(double origin, double bound) {
     *   double r = nextDouble();
     *   r = r * (bound - origin) + origin;
     *   if (r >= bound) // correct for rounding
     *     r = Math.nextDown(bound);
     *   return r;
     * }}</pre>
     *
     * @param streamSize the number of values to generate
     * @param randomNumberOrigin the origin (inclusive) of each random value
     * @param randomNumberBound the bound (exclusive) of each random value
     * @return a stream of pseudorandom {@code double} values,
     *         each with the given origin (inclusive) and bound (exclusive)
     * @throws IllegalArgumentException if {@code streamSize} is
     *         less than zero
     * @throws IllegalArgumentException if {@code randomNumberOrigin}
     *         is greater than or equal to {@code randomNumberBound}
     * @since 1.8
     */
    public DoubleStream doubles(long streamSize, double randomNumberOrigin,
                                double randomNumberBound) {
        if (streamSize < 0L)
            throw new IllegalArgumentException(BadSize);
        if (!(randomNumberOrigin < randomNumberBound))
            throw new IllegalArgumentException(BadRange);
        return StreamSupport.doubleStream
                (new RandomDoublesSpliterator
                         (this, 0L, streamSize, randomNumberOrigin, randomNumberBound),
                 false);
    }

    /**
     * Returns an effectively unlimited stream of pseudorandom {@code
     * double} values, each conforming to the given origin (inclusive) and bound
     * (exclusive).
     *
     * <p>A pseudorandom {@code double} value is generated as if it's the result
     * of calling the following method with the origin and bound:
     * <pre> {@code
     * double nextDouble(double origin, double bound) {
     *   double r = nextDouble();
     *   r = r * (bound - origin) + origin;
     *   if (r >= bound) // correct for rounding
```

```
 *     r = Math.nextDown(bound);
 *   return r;
 * }}</pre>
 *
 * @implNote This method is implemented to be equivalent to {@code
 * doubles(Long.MAX_VALUE, randomNumberOrigin, randomNumberBound)}.
 *
 * @param randomNumberOrigin the origin (inclusive) of each random value
 * @param randomNumberBound the bound (exclusive) of each random value
 * @return a stream of pseudorandom {@code double} values,
 *         each with the given origin (inclusive) and bound (exclusive)
 * @throws IllegalArgumentException if {@code randomNumberOrigin}
 *         is greater than or equal to {@code randomNumberBound}
 * @since 1.8
 */
public DoubleStream doubles(double randomNumberOrigin, double randomNumberBounc
    if (!(randomNumberOrigin < randomNumberBound))
        throw new IllegalArgumentException(BadRange);
    return StreamSupport.doubleStream
            (new RandomDoublesSpliterator
                     (this, 0L, Long.MAX_VALUE, randomNumberOrigin, randomNumbe
             false);
}

/**
 * Spliterator for int streams.  We multiplex the four int
 * versions into one class by treating a bound less than origin as
 * unbounded, and also by treating "infinite" as equivalent to
 * Long.MAX_VALUE. For splits, it uses the standard divide-by-two
 * approach. The long and double versions of this class are
 * identical except for types.
 */
static final class RandomIntsSpliterator implements Spliterator.OfInt {
    final Random rng;
    long index;
    final long fence;
    final int origin;
    final int bound;
    RandomIntsSpliterator(Random rng, long index, long fence,
                          int origin, int bound) {
        this.rng = rng; this.index = index; this.fence = fence;
        this.origin = origin; this.bound = bound;
    }
    public RandomIntsSpliterator trySplit() {
        long i = index, m = (i + fence) >>> 1;
        return (m <= i) ? null :
                new RandomIntsSpliterator(rng, i, index = m, origin, bound);
    }

    public long estimateSize() {
        return fence - index;
    }

    public int characteristics() {
        return (Spliterator.SIZED | Spliterator.SUBSIZED |
                Spliterator.NONNULL | Spliterator.IMMUTABLE);
    }

    public boolean tryAdvance(IntConsumer consumer) {
        if (consumer == null) throw new NullPointerException();
        long i = index, f = fence;
        if (i < f) {
            consumer.accept(rng.internalNextInt(origin, bound));
            index = i + 1;
            return true;
        }
        return false;
    }

    public void forEachRemaining(IntConsumer consumer) {
        if (consumer == null) throw new NullPointerException();
        long i = index, f = fence;
        if (i < f) {
            index = f;
            Random r = rng;
            int o = origin, b = bound;
            do {
                consumer.accept(r.internalNextInt(o, b));
            } while (++i < f);
        }
    }
}

/**
 * Spliterator for long streams.
 */
static final class RandomLongsSpliterator implements Spliterator.OfLong {
    final Random rng;
    long index;
    final long fence;
    final long origin;
    final long bound;
    RandomLongsSpliterator(Random rng, long index, long fence,
                           long origin, long bound) {
        this.rng = rng; this.index = index; this.fence = fence;
        this.origin = origin; this.bound = bound;
    }

    public RandomLongsSpliterator trySplit() {
        long i = index, m = (i + fence) >>> 1;
        return (m <= i) ? null :
               new RandomLongsSpliterator(rng, i, index = m, origin, bound);
    }

    public long estimateSize() {
        return fence - index;
    }

    public int characteristics() {
        return (Spliterator.SIZED | Spliterator.SUBSIZED |
                Spliterator.NONNULL | Spliterator.IMMUTABLE);
    }

    public boolean tryAdvance(LongConsumer consumer) {
        if (consumer == null) throw new NullPointerException();
        long i = index, f = fence;
        if (i < f) {
            consumer.accept(rng.internalNextLong(origin, bound));
            index = i + 1;
            return true;
        }
        return false;
    }

    public void forEachRemaining(LongConsumer consumer) {
        if (consumer == null) throw new NullPointerException();
        long i = index, f = fence;
        if (i < f) {
```

```java
                index = f;
                Random r = rng;
                long o = origin, b = bound;
                do {
                    consumer.accept(r.internalNextLong(o, b));
                } while (++i < f);
            }
        }

    }

    /**
     * Spliterator for double streams.
     */
    static final class RandomDoublesSpliterator implements Spliterator.OfDouble {
        final Random rng;
        long index;
        final long fence;
        final double origin;
        final double bound;
        RandomDoublesSpliterator(Random rng, long index, long fence,
                                 double origin, double bound) {
            this.rng = rng; this.index = index; this.fence = fence;
            this.origin = origin; this.bound = bound;
        }

        public RandomDoublesSpliterator trySplit() {
            long i = index, m = (i + fence) >>> 1;
            return (m <= i) ? null :
                   new RandomDoublesSpliterator(rng, i, index = m, origin, bound);
        }

        public long estimateSize() {
            return fence - index;
        }

        public int characteristics() {
            return (Spliterator.SIZED | Spliterator.SUBSIZED |
                    Spliterator.NONNULL | Spliterator.IMMUTABLE);
        }

        public boolean tryAdvance(DoubleConsumer consumer) {
            if (consumer == null) throw new NullPointerException();
            long i = index, f = fence;
            if (i < f) {
                consumer.accept(rng.internalNextDouble(origin, bound));
                index = i + 1;
                return true;
            }
            return false;
        }

        public void forEachRemaining(DoubleConsumer consumer) {
            if (consumer == null) throw new NullPointerException();
            long i = index, f = fence;
            if (i < f) {
                index = f;
                Random r = rng;
                double o = origin, b = bound;
                do {
                    consumer.accept(r.internalNextDouble(o, b));
                } while (++i < f);
            }
        }
    }

    /**
     * Serializable fields for Random.
     *
     * @serialField    seed long
     *              seed for random computations
     * @serialField    nextNextGaussian double
     *              next Gaussian to be returned
     * @serialField      haveNextNextGaussian boolean
     *              nextNextGaussian is valid
     */
    private static final ObjectStreamField[] serialPersistentFields = {
        new ObjectStreamField("seed", Long.TYPE),
        new ObjectStreamField("nextNextGaussian", Double.TYPE),
        new ObjectStreamField("haveNextNextGaussian", Boolean.TYPE)
    };

    /**
     * Reconstitute the {@code Random} instance from a stream (that is,
     * deserialize it).
     */
```

```java
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {

    ObjectInputStream.GetField fields = s.readFields();

    // The seed is read in as {@code long} for
    // historical reasons, but it is converted to an AtomicLong.
    long seedVal = fields.get("seed", -1L);
    if (seedVal < 0)
      throw new java.io.StreamCorruptedException(
                              "Random: invalid seed");
    resetSeed(seedVal);
    nextNextGaussian = fields.get("nextNextGaussian", 0.0);
    haveNextNextGaussian = fields.get("haveNextNextGaussian", false);
}

/**
 * Save the {@code Random} instance to a stream.
 */
synchronized private void writeObject(ObjectOutputStream s)
    throws IOException {

    // set the values of the Serializable fields
    ObjectOutputStream.PutField fields = s.putFields();

    // The seed is serialized as a long for historical reasons.
    fields.put("seed", seed.get());
    fields.put("nextNextGaussian", nextNextGaussian);
    fields.put("haveNextNextGaussian", haveNextNextGaussian);

    // save them
    s.writeFields();
}

// Support for resetting seed while deserializing
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long seedOffset;
static {
    try {
        seedOffset = unsafe.objectFieldOffset
            (Random.class.getDeclaredField("seed"));
    } catch (Exception ex) { throw new Error(ex); }
}
```

```
    private void resetSeed(long seedVal) {
        unsafe.putObjectVolatile(this, seedOffset, new AtomicLong(seedVal));
    }
}
```