# StdRandom.java

Below is the syntax highlighted version of StdRandom.java from § Standard Libraries.   Here is the Javadoc.

```
/******************************************************************************
 *  Compilation:  javac StdRandom.java
 *  Execution:    java StdRandom
 *  Dependencies: StdOut.java
 *
 *  A library of static methods to generate pseudo-random numbers from
 *  different distributions (bernoulli, uniform, gaussian, discrete,
 *  and exponential). Also includes a method for shuffling an array.
 *
 *
 *  %  java StdRandom 5
 *  seed = 1316600602069
 *  59 16.81826  true 8.83954  0
 *  32 91.32098  true 9.11026  0
 *  35 10.11874  true 8.95396  3
 *  92 32.88401  true 8.87089  0
 *  72 92.55791  true 9.46241  0
 *
 *  % java StdRandom 5
 *  seed = 1316600616575
 *  96 60.17070  true 8.72821  0
 *  79 32.01607  true 8.58159  0
 *  81 59.49065  true 9.10423  1
 *  96 51.65818  true 9.02102  0
 *  99 17.55771  true 8.99762  0
 *
 *  % java StdRandom 5 1316600616575
 *  seed = 1316600616575
 *  96 60.17070  true 8.72821  0
 *  79 32.01607  true 8.58159  0
 *  81 59.49065  true 9.10423  1
 *  96 51.65818  true 9.02102  0
 *  99 17.55771  true 8.99762  0
 *
 *
 *  Remark
 *  ------
 *    - Relies on randomness of nextDouble() method in java.util.Random
 *      to generate pseudorandom numbers in [0, 1).
 *
 *    - This library allows you to set and get the pseudorandom number seed.
 *
 *    - See http://www.honeylocust.com/RngPack/ for an industrial
 *      strength random number generator in Java.
 *
 ******************************************************************************/

import java.util.Random;

/**
 *  The {@code StdRandom} class provides static methods for generating
 *  random number from various discrete and continuous distributions,
 *  including Bernoulli, uniform, Gaussian, exponential, pareto,
 *  Poisson, and Cauchy. It also provides method for shuffling an
 *  array or subarray.
 *  <p>
 *  For additional documentation,
 *  see <a href="http://introcs.cs.princeton.edu/22library">Section 2.2</a> of
 *  <i>Computer Science: An Interdisciplinary Approach</i>
 *  by Robert Sedgewick and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
 */
```

```java
public final class StdRandom {

    private static Random random;    // pseudo-random number generator
    private static long seed;        // pseudo-random number generator seed

    // static initializer
    static {
        // this is how the seed was set in Java 1.4
        seed = System.currentTimeMillis();
        random = new Random(seed);
    }

    // don't instantiate
    private StdRandom() { }

    /**
     * Sets the seed of the pseudorandom number generator.
     * This method enables you to produce the same sequence of "random"
     * number for each execution of the program.
     * Ordinarily, you should call this method at most once per program.
     *
     * @param s the seed
     */
    public static void setSeed(long s) {
        seed   = s;
        random = new Random(seed);
    }

    /**
     * Returns the seed of the pseudorandom number generator.
     *
     * @return the seed
     */
    public static long getSeed() {
        return seed;
    }

    /**
     * Returns a random real number uniformly in [0, 1).
     *
     * @return a random real number uniformly in [0, 1)
     */
    public static double uniform() {
        return random.nextDouble();
    }

    /**
     * Returns a random integer uniformly in [0, n).
     *
     * @param n number of possible integers
     * @return a random integer uniformly between 0 (inclusive) and {@code n} (exclusive)
     * @throws IllegalArgumentException if {@code n <= 0}
     */
    public static int uniform(int n) {
        if (n <= 0) throw new IllegalArgumentException("argument must be positive");
        return random.nextInt(n);
    }

    ///////////////////////////////////////////////////////////////////////////
    //  STATIC METHODS BELOW RELY ON JAVA.UTIL.RANDOM ONLY INDIRECTLY VIA
    //  THE STATIC METHODS ABOVE.
    ///////////////////////////////////////////////////////////////////////////

    /**
     * Returns a random real number uniformly in [0, 1).
     *
     * @return     a random real number uniformly in [0, 1)
     * @deprecated Replaced by {@link #uniform()}.
     */
    @Deprecated
    public static double random() {
        return uniform();
    }
    /**
     * Returns a random integer uniformly in [a, b).
     *
     * @param  a the left endpoint
     * @param  b the right endpoint
     * @return a random integer uniformly in [a, b)
     * @throws IllegalArgumentException if {@code b <= a}
     * @throws IllegalArgumentException if {@code b - a >= Integer.MAX_VALUE}
     */
    public static int uniform(int a, int b) {
        if ((b <= a) || ((long) b - a >= Integer.MAX_VALUE)) {
            throw new IllegalArgumentException("invalid range: [" + a + ", " + b + "]");
        }
        return a + uniform(b - a);
    }

    /**
     * Returns a random real number uniformly in [a, b).
     *
     * @param  a the left endpoint
     * @param  b the right endpoint
     * @return a random real number uniformly in [a, b)
     * @throws IllegalArgumentException unless {@code a < b}
     */
    public static double uniform(double a, double b) {
        if (!(a < b)) {
            throw new IllegalArgumentException("invalid range: [" + a + ", " + b + "]");
        }
        return a + uniform() * (b-a);
    }

    /**
     * Returns a random boolean from a Bernoulli distribution with success
     * probability <em>p</em>.
     *
     * @param  p the probability of returning {@code true}
     * @return {@code true} with probability {@code p} and
     *         {@code false} with probability {@code p}
     * @throws IllegalArgumentException unless {@code p >= 0.0} and {@code p <= 1.0}
     */
    public static boolean bernoulli(double p) {
        if (!(p >= 0.0 && p <= 1.0))
            throw new IllegalArgumentException("probability p must be between 0.0 and 1.0");
        return uniform() < p;
    }

    /**
     * Returns a random boolean from a Bernoulli distribution with success
     * probability 1/2.
     *
     * @return {@code true} with probability 1/2 and
     *         {@code false} with probability 1/2
     */
    public static boolean bernoulli() {
        return bernoulli(0.5);
    }

    /**
     * Returns a random real number from a standard Gaussian distribution.
     *
     * @return a random real number from a standard Gaussian distribution
     *         (mean 0 and standard deviation 1).
     */
    public static double gaussian() {
        // use the polar form of the Box-Muller transform
        double r, x, y;
        do {
            x = uniform(-1.0, 1.0);
            y = uniform(-1.0, 1.0);
            r = x*x + y*y;
        } while (r >= 1 || r == 0);
        return x * Math.sqrt(-2 * Math.log(r) / r);

        // Remark:  y * Math.sqrt(-2 * Math.log(r) / r)
        // is an independent random gaussian
    }

    /**
     * Returns a random real number from a Gaussian distribution with mean &mu;
     * and standard deviation &sigma;.
     *
     * @param  mu the mean
     * @param  sigma the standard deviation
     * @return a real number distributed according to the Gaussian distribution
     *         with mean {@code mu} and standard deviation {@code sigma}
     */
    public static double gaussian(double mu, double sigma) {
        return mu + sigma * gaussian();
    }

    /**
     * Returns a random integer from a geometric distribution with success
     * probability <em>p</em>.
     *
     * @param  p the parameter of the geometric distribution
     * @return a random integer from a geometric distribution with success
     *         probability {@code p}; or {@code Integer.MAX_VALUE} if
     *         {@code p} is (nearly) equal to {@code 1.0}.
     * @throws IllegalArgumentException unless {@code p >= 0.0} and {@code p <= 1.0}
     */
    public static int geometric(double p) {
        if (!(p >= 0.0 && p <= 1.0)) {
            throw new IllegalArgumentException("probability p must be between 0.0 and 1.0");
        }
        // using algorithm given by Knuth
        return (int) Math.ceil(Math.log(uniform()) / Math.log(1.0 - p));
    }

    /**
     * Returns a random integer from a Poisson distribution with mean &lambda;.
     *
     * @param  lambda the mean of the Poisson distribution
     * @return a random integer from a Poisson distribution with mean {@code lambda}
     * @throws IllegalArgumentException unless {@code lambda > 0.0} and not infinite
     */
    public static int poisson(double lambda) {
        if (!(lambda > 0.0))
            throw new IllegalArgumentException("lambda must be positive");
        if (Double.isInfinite(lambda))
            throw new IllegalArgumentException("lambda must not be infinite");
        // using algorithm given by Knuth
        // see http://en.wikipedia.org/wiki/Poisson_distribution
        int k = 0;
        double p = 1.0;
        double expLambda = Math.exp(-lambda);
        do {
            k++;
            p *= uniform();
        } while (p >= expLambda);
        return k-1;
    }

    /**
     * Returns a random real number from the standard Pareto distribution.
     *
     * @return a random real number from the standard Pareto distribution
     */
    public static double pareto() {
        return pareto(1.0);
    }

    /**
     * Returns a random real number from a Pareto distribution with
     * shape parameter &alpha;.
     *
     * @param  alpha shape parameter
     * @return a random real number from a Pareto distribution with shape
     *         parameter {@code alpha}
     * @throws IllegalArgumentException unless {@code alpha > 0.0}
     */
    public static double pareto(double alpha) {
        if (!(alpha > 0.0))
            throw new IllegalArgumentException("alpha must be positive");
        return Math.pow(1 - uniform(), -1.0/alpha) - 1.0;
    }

    /**
     * Returns a random real number from the Cauchy distribution.
     *
     * @return a random real number from the Cauchy distribution.
     */
    public static double cauchy() {
        return Math.tan(Math.PI * (uniform() - 0.5));
    }

    /**
     * Returns a random integer from the specified discrete distribution.
     *
     * @param  probabilities the probability of occurrence of each integer
     * @return a random integer from a discrete distribution:
     *         {@code i} with probability {@code probabilities[i]}
     * @throws IllegalArgumentException if {@code probabilities} is {@code null}
     * @throws IllegalArgumentException if sum of array entries is not (very nearly) equal to {@code 1.0}
     * @throws IllegalArgumentException unless {@code probabilities[i] >= 0.0} for each index {@code i}
     */
    public static int discrete(double[] probabilities) {
        if (probabilities == null) throw new IllegalArgumentException("argument array is null");
        double EPSILON = 1E-14;
        double sum = 0.0;
        for (int i = 0; i < probabilities.length; i++) {
            if (!(probabilities[i] >= 0.0))
                throw new IllegalArgumentException("array entry " + i + " must be nonnegative: " + probabilities[i]);
            sum += probabilities[i];
        }
        if (sum > 1.0 + EPSILON || sum < 1.0 - EPSILON)
            throw new IllegalArgumentException("sum of array entries does not approximately equal 1.0: " + sum);

        // the for loop may not return a value when both r is (nearly) 1.0 and when the
        // cumulative sum is less than 1.0 (as a result of floating-point roundoff error)
        while (true) {
            double r = uniform();
            sum = 0.0;
            for (int i = 0; i < probabilities.length; i++) {
                sum = sum + probabilities[i];
                if (sum > r) return i;
            }
        }
    }

    /**
     * Returns a random integer from the specified discrete distribution.
     *
     * @param  frequencies the frequency of occurrence of each integer
     * @return a random integer from a discrete distribution:
     *         {@code i} with probability proportional to {@code frequencies[i]}
     * @throws IllegalArgumentException if {@code frequencies} is {@code null}
     * @throws IllegalArgumentException if all array entries are {@code 0}
     * @throws IllegalArgumentException if {@code frequencies[i]} is negative for any index {@code i}
     * @throws IllegalArgumentException if sum of frequencies exceeds {@code Integer.MAX_VALUE} (2<sup>31</sup> - 1)
     */
    public static int discrete(int[] frequencies) {
        if (frequencies == null) throw new IllegalArgumentException("argument array is null");
        long sum = 0;
        for (int i = 0; i < frequencies.length; i++) {
            if (frequencies[i] < 0)
                throw new IllegalArgumentException("array entry " + i + " must be nonnegative: " + frequencies[i]);
            sum += frequencies[i];
        }
        if (sum == 0)
            throw new IllegalArgumentException("at least one array entry must be positive");
        if (sum >= Integer.MAX_VALUE)
            throw new IllegalArgumentException("sum of frequencies overflows an int");
        // pick index i with probabilitity proportional to frequency
        double r = uniform((int) sum);
        sum = 0;
        for (int i = 0; i < frequencies.length; i++) {
            sum += frequencies[i];
            if (sum > r) return i;
        }

        // can't reach here
        assert false;
        return -1;
    }

    /**
     * Returns a random real number from an exponential distribution
     * with rate &lambda;.
     *
     * @param  lambda the rate of the exponential distribution
     * @return a random real number from an exponential distribution with
     *         rate {@code lambda}
     * @throws IllegalArgumentException unless {@code lambda > 0.0}
     */
    public static double exp(double lambda) {
        if (!(lambda > 0.0))
            throw new IllegalArgumentException("lambda must be positive");
        return -Math.log(1 - uniform()) / lambda;
    }

    /**
     * Rearranges the elements of the specified array in uniformly random order.
     *
     * @param  a the array to shuffle
     * @throws IllegalArgumentException if {@code a} is {@code null}
     */
    public static void shuffle(Object[] a) {
        if (a == null) throw new IllegalArgumentException("argument array is null");
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int r = i + uniform(n-i);     // between i and n-1
            Object temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }

    /**
     * Rearranges the elements of the specified array in uniformly random order.
     *
     * @param  a the array to shuffle
     * @throws IllegalArgumentException if {@code a} is {@code null}
     */
    public static void shuffle(double[] a) {
        if (a == null) throw new IllegalArgumentException("argument array is null");
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int r = i + uniform(n-i);     // between i and n-1
            double temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }

    /**
     * Rearranges the elements of the specified array in uniformly random order.
     *
     * @param  a the array to shuffle
     * @throws IllegalArgumentException if {@code a} is {@code null}
     */
    public static void shuffle(int[] a) {
        if (a == null) throw new IllegalArgumentException("argument array is null");
        int n = a.length;
        for (int i = 0; i < n; i++) {
            int r = i + uniform(n-i);     // between i and n-1
            int temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }


    /**
     * Rearranges the elements of the specified subarray in uniformly random order.
     *
     * @param  a the array to shuffle
     * @param  lo the left endpoint (inclusive)
     * @param  hi the right endpoint (inclusive)
     * @throws IllegalArgumentException if {@code a} is {@code null}
     * @throws IndexOutOfBoundsException unless {@code (0 <= lo) && (lo <= hi) && (hi < a.length)}
     *
     */
    public static void shuffle(Object[] a, int lo, int hi) {
        if (a == null) throw new IllegalArgumentException("argument array is null");
        if (lo < 0 || lo > hi || hi >= a.length) {
            throw new IndexOutOfBoundsException("invalid subarray range: [" + lo + ", " + hi + "]");
        }
        for (int i = lo; i <= hi; i++) {
            int r = i + uniform(hi-i+1);     // between i and hi
            Object temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }

    /**
     * Rearranges the elements of the specified subarray in uniformly random order.
     *
     * @param  a the array to shuffle
     * @param  lo the left endpoint (inclusive)
     * @param  hi the right endpoint (inclusive)
     * @throws IllegalArgumentException if {@code a} is {@code null}
     * @throws IndexOutOfBoundsException unless {@code (0 <= lo) && (lo <= hi) && (hi < a.length)}
     */
    public static void shuffle(double[] a, int lo, int hi) {
        if (a == null) throw new IllegalArgumentException("argument array is null");
        if (lo < 0 || lo > hi || hi >= a.length) {
            throw new IndexOutOfBoundsException("invalid subarray range: [" + lo + ", " + hi + "]");
        }
        for (int i = lo; i <= hi; i++) {
            int r = i + uniform(hi-i+1);     // between i and hi
            double temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }

    /**
     * Rearranges the elements of the specified subarray in uniformly random order.
     *
     * @param  a the array to shuffle
     * @param  lo the left endpoint (inclusive)
     * @param  hi the right endpoint (inclusive)
     * @throws IllegalArgumentException if {@code a} is {@code null}
     * @throws IndexOutOfBoundsException unless {@code (0 <= lo) && (lo <= hi) && (hi < a.length)}
     */
    public static void shuffle(int[] a, int lo, int hi) {
        if (a == null) throw new IllegalArgumentException("argument array is null");
        if (lo < 0 || lo > hi || hi >= a.length) {
            throw new IndexOutOfBoundsException("invalid subarray range: [" + lo + ", " + hi + "]");
        }
        for (int i = lo; i <= hi; i++) {
            int r = i + uniform(hi-i+1);     // between i and hi
            int temp = a[i];
            a[i] = a[r];
            a[r] = temp;
        }
    }
    /**
     * Unit test.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        if (args.length == 2) StdRandom.setSeed(Long.parseLong(args[1]));
        double[] probabilities = { 0.5, 0.3, 0.1, 0.1 };
        int[] frequencies = { 5, 3, 1, 1 };
        String[] a = "A B C D E F G".split(" ");

        StdOut.println("seed = " + StdRandom.getSeed());
        for (int i = 0; i < n; i++) {
            StdOut.printf("%2d ",   uniform(100));
            StdOut.printf("%8.5f ", uniform(10.0, 99.0));
            StdOut.printf("%5b ",   bernoulli(0.5));
            StdOut.printf("%7.5f ", gaussian(9.0, 0.2));
            StdOut.printf("%1d ",   discrete(probabilities));
            StdOut.printf("%1d ",   discrete(frequencies));
            StdRandom.shuffle(a);
            for (String s : a)
                StdOut.print(s);
            StdOut.println();
        }
    }

}
```

*Last updated: Wed Nov 2 05:04:34 EDT 2016.*