java.util

Class Random

java.lang.Object
java.util.Random

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

SecureRandom, ThreadLocalRandom

```
public class Random
extends Object
implements Serializable
```

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, *The Art of Computer Programming, Volume 2*, Section 3.2.1.)

If two instances of `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 `Random`. Java implementations must use all the algorithms shown here for the class `Random`, for the sake of absolute portability of Java code. However, subclasses of class `Random` are permitted to use other algorithms, so long as they adhere to the general contracts for all the methods.

The algorithms implemented by class `Random` use a `protected` utility method that on each invocation can supply up to 32 pseudorandomly generated bits.

Many applications will find the method `Math.random()` simpler to use.

Instances of `java.util.Random` are threadsafe. However, the concurrent use of the same `java.util.Random` instance across threads may encounter contention and consequent poor performance. Consider instead using `ThreadLocalRandom` in multithreaded designs.

Instances of `java.util.Random` are not cryptographically secure. Consider instead using `SecureRandom` to get a cryptographically secure pseudo-random number generator for use by security-sensitive applications.

Since:

1.0

See Also:

Serialized Form

Constructor Summary

Constructors

Constructor and Description
`Random()` Creates a new random number generator.
`Random(long seed)` Creates a new random number generator using a single `long` seed.

Method Summary

Methods

Modifier and Type	Method and Description
`protected int`	**`next`**`(int bits)` Generates the next pseudorandom number.
`boolean`	**`nextBoolean`**`()` Returns the next pseudorandom, uniformly distributed `boolean` value from this random number generator's sequence.
`void`	**`nextBytes`**`(byte[] bytes)` Generates random bytes and places them into a user-supplied byte array.
`double`	**`nextDouble`**`()` Returns the next pseudorandom, uniformly distributed `double` value between `0.0` and `1.0` from this random number generator's sequence.
`float`	**`nextFloat`**`()` Returns the next pseudorandom, uniformly distributed `float` value between `0.0` and `1.0` from this random number generator's sequence.
`double`	**`nextGaussian`**`()` Returns the next pseudorandom, Gaussian ("normally") distributed `double` value with mean `0.0` and standard deviation `1.0` from this random number generator's sequence.
`int`	**`nextInt`**`()` Returns the next pseudorandom, uniformly distributed `int` value from this random number generator's sequence.
`int`	**`nextInt`**`(int n)` Returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
`long`	**`nextLong`**`()` Returns the next pseudorandom, uniformly distributed `long` value from this random number generator's sequence.
`void`	**`setSeed`**`(long seed)` Sets the seed of this random number generator using a single `long` seed.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Random

`public Random()`

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.

Random

`public Random(long seed)`

Creates a new random number generator using a single `long` seed. The seed is the initial value of the internal state of the pseudorandom number generator which is maintained by method `next(int)`.

The invocation `new Random(seed)` is equivalent to:

```
Random rnd = new Random();
rnd.setSeed(seed);
```

Parameters:

`seed` - the initial seed

See Also:

`setSeed(long)`

Method Detail

setSeed

```
public void setSeed(long seed)
```

Sets the seed of this random number generator using a single `long` seed. The general contract of `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 `seed` as a seed. The method `setSeed` is implemented by class `Random` by atomically updating the seed to

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

and clearing the `haveNextNextGaussian` flag used by `nextGaussian()`.

The implementation of `setSeed` by class `Random` happens to use only 48 bits of the given seed. In general, however, an overriding method may use all 64 bits of the `long` argument as a seed value.

Parameters:

`seed` - the initial seed

next

```
protected int next(int bits)
```

Generates the next pseudorandom number. Subclasses should override this, as this is used by all other methods.

The general contract of `next` is that it returns an `int` value and if the argument `bits` is between `1` and `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 `0` or `1`. The method `next` is implemented by class `Random` by atomically updating the seed to

```
(seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)
```

and returning

```
(int)(seed >>> (48 - bits)).
```

This is a linear congruential pseudorandom number generator, as defined by D. H. Lehmer and described by Donald E. Knuth in *The Art of Computer Programming,* Volume 3: *Seminumerical Algorithms*, section 3.2.1.

Parameters:

`bits` - random bits

Returns:

the next pseudorandom value from this random number generator's sequence

Since:

1.1

nextBytes

```
public void nextBytes(byte[] bytes)
```

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.

The method `nextBytes` is implemented by class `Random` as if by:

```
 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;
 }
```

Parameters:

`bytes` - the byte array to fill with random bytes

Throws:

`NullPointerException` - if the byte array is null

Since:

1.1

nextInt

```
public int nextInt()
```

Returns the next pseudorandom, uniformly distributed `int` value from this random number generator's sequence. The general contract of `nextInt` is that one `int` value is pseudorandomly generated and returned. All 2^{32} possible `int` values are produced with (approximately) equal probability.

The method `nextInt` is implemented by class `Random` as if by:

```
 public int nextInt() {
   return next(32);
 }
```

Returns:

the next pseudorandom, uniformly distributed `int` value from this random number generator's sequence

nextInt

```
public int nextInt(int n)
```

Returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence. The general contract of `nextInt` is that one `int` value in the specified range is pseudorandomly generated and returned. All `n` possible `int` values are produced with (approximately) equal probability. The method `nextInt(int n)` is implemented by class `Random` as if by:

```
 public int nextInt(int n) {
   if (n <= 0)
```

```
        throw new IllegalArgumentException("n must be positive");

    if ((n & -n) == n)  // i.e., n is a power of 2
        return (int)((n * (long)next(31)) >> 31);

    int bits, val;
    do {
        bits = next(31);
        val = bits % n;
    } while (bits - val + (n-1) < 0);
    return val;
}
```

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 `int` values from the stated range with perfect uniformity.

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.

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 *low-order* 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.

Parameters:

`n` - the bound on the random number to be returned. Must be positive.

Returns:

the next pseudorandom, uniformly distributed `int` value between `0` (inclusive) and `n` (exclusive) from this random number generator's sequence

Throws:

`IllegalArgumentException` - if n is not positive

Since:

1.2

nextLong

```
public long nextLong()
```

Returns the next pseudorandom, uniformly distributed `long` value from this random number generator's sequence. The general contract of `nextLong` is that one `long` value is pseudorandomly generated and returned.

The method `nextLong` is implemented by class `Random` as if by:

```
public long nextLong() {
    return ((long)next(32) << 32) + next(32);
}
```

Because class `Random` uses a seed with only 48 bits, this algorithm will not return all possible `long` values.

Returns:

the next pseudorandom, uniformly distributed `long` value from this random number generator's sequence

nextBoolean

`public boolean nextBoolean()`

Returns the next pseudorandom, uniformly distributed `boolean` value from this random number generator's sequence. The general contract of `nextBoolean` is that one `boolean` value is pseudorandomly generated and returned. The values `true` and `false` are produced with (approximately) equal probability.

The method `nextBoolean` is implemented by class `Random` as if by:

```
public boolean nextBoolean() {
   return next(1) != 0;
 }
```

Returns:

the next pseudorandom, uniformly distributed `boolean` value from this random number generator's sequence

Since:

1.2

nextFloat

`public float nextFloat()`

Returns the next pseudorandom, uniformly distributed `float` value between `0.0` and `1.0` from this random number generator's sequence.

The general contract of `nextFloat` is that one `float` value, chosen (approximately) uniformly from the range `0.0f` (inclusive) to `1.0f` (exclusive), is pseudorandomly generated and returned. All 2^{24} possible `float` values of the form $m \times 2^{-24}$, where m is a positive integer less than 2^{24} , are produced with (approximately) equal probability.

The method `nextFloat` is implemented by class `Random` as if by:

```
public float nextFloat() {
   return next(24) / ((float)(1 << 24));
 }
```

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 `float` values from the stated range with perfect uniformity.

[In early versions of Java, the result was incorrectly calculated as:

```
return next(30) / ((float)(1 << 30));
```

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.]

Returns:

the next pseudorandom, uniformly distributed `float` value between `0.0` and `1.0` from this random number generator's sequence

nextDouble

`public double nextDouble()`

Returns the next pseudorandom, uniformly distributed `double` value between `0.0` and `1.0` from this random number generator's sequence.

The general contract of `nextDouble` is that one `double` value, chosen (approximately) uniformly from the range `0.0d` (inclusive) to `1.0d` (exclusive), is pseudorandomly generated and returned.

The method `nextDouble` is implemented by class `Random` as if by:

```
public double nextDouble() {
  return (((long)next(26) << 27) + next(27))
    / (double)(1L << 53);
}
```

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 `double` values from the stated range with perfect uniformity.

[In early versions of Java, the result was incorrectly calculated as:

```
return (((long)next(27) << 27) + next(27))
    / (double)(1L << 54);
```

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.]

Returns:

the next pseudorandom, uniformly distributed `double` value between `0.0` and `1.0` from this random number generator's sequence

See Also:

`Math.random()`

nextGaussian

`public double nextGaussian()`

Returns the next pseudorandom, Gaussian ("normally") distributed `double` value with mean `0.0` and standard deviation `1.0` from this random number generator's sequence.

The general contract of `nextGaussian` is that one `double` value, chosen from (approximately) the usual normal distribution with mean `0.0` and standard deviation `1.0`, is pseudorandomly generated and returned.

The method `nextGaussian` is implemented by class `Random` as if by a threadsafe version of the following:

```
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;
  }
}
```

This uses the *polar method* of G. E. P. Box, M. E. Muller, and G. Marsaglia, as described by Donald E. Knuth in

The Art of Computer Programming, Volume 3: *Seminumerical Algorithms*, section 3.4.1, subsection C, algorithm P. Note that it generates two independent values at the cost of only one call to `StrictMath.log` and one call to `StrictMath.sqrt`.

Returns:

the next pseudorandom, Gaussian ("normally") distributed `double` value with mean `0.0` and standard deviation `1.0` from this random number generator's sequence

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.