
Génération de variables aléatoires avec l'API Random

Hugo Etiévant

Dernière mise à jour : 20 mai 2004

Qu'est-ce qu'une variable aléatoire ?

Une variable aléatoire est une variable dont la suite des valeurs n'est pas prédictible. Aussi longue que soit la série générée, il n'est pas possible de trouver une équation permettant de prédire le reste de la série à partir de celles déjà générées.

Par exemple X est une variable aléatoire. Et $x_1, x_2, x_3, x_i \dots$ est la suite des valeurs générées successivement.

Toute variable aléatoire suit une loi de probabilité. Cette loi définit la répartition des valeurs au sein d'un intervalle. Par exemple, la loi « uniforme » définit une répartition uniforme des valeurs dans un intervalle, c'est à dire que toutes les valeurs de l'intervalle ont la même probabilité d'apparaître dans une longue série.

Quelle est leur utilité ?

Il est fréquent d'avoir à utiliser des variables aléatoires. Les principaux domaines utilisateurs de ces variables sont les jeux, les simulations et la cryptographie.

Qu'il s'agisse de déterminer la valeur d'un lancé de dés où de positionner un personnage dans un jeu vidéo, les besoins en hasard sont nombreux.

Les logiciels de simulation numérique, permettant par exemple de tester virtuellement la résistance d'une installation nucléaire ont recours de façon massive aux variables aléatoires.

Les logiciels de cryptographie sont eux aussi très gourmands en nombres aléatoires pour la génération de clés secrètes incassables (ou presque).

Comment les générer ?

(1)

Le hasard vrai n'existe que dans la nature. Il est très difficile de le reproduire dans un ordinateur, sauf à relier celui-ci à un dispositif physique de mesure d'un phénomène naturel. Ces dispositifs sont très complexes et coûteux.

En revanche, il existe des algorithmes basés sur l'heure système qui fournissent un pseudo-hasard suffisant pour les applications courantes.

L'heure système possède une précision de l'ordre de la milliseconde, elle change donc très souvent de valeur. Elle est utilisée dans les générateurs aléatoires des langages de programmation les plus courants. Des calculs successifs sur les valeurs précédemment générées et sur cette heure système fournissent donc les valeurs des variables aléatoires.

Comment les générer ?

(2)

Dans certains systèmes, d'autres paramètres entrent en ligne de compte. Par exemple les mouvements de la souris et les frappes au clavier peuvent être mémorisés afin d'influer sur la génération de nombres aléatoires.

Mais ces paramètres ne sont pas parfaits : certaines touches sont beaucoup plus utilisées que les autres et la souris pointe souvent les mêmes zones.

D'autres encore, comme l'activité du réseau, des supports de stockage, des processus du système, etc. peuvent être couplés à l'heure système.

L'API Random de Java

Java fourni dans son JDK depuis la version 1.0 une API standard permettant la génération de nombres aléatoires.

La documentation (en anglais) de cette API se trouve ici :
<http://java.sun.com/j2se/1.4.2/docs/api/java/util/Random.html>

Mécanisme de génération – période

L'algorithme utilisé est à congruence linéaire du type :

$$X_{n+1} = (a \cdot X_n + b) \bmod m$$

Où :

- X_{n+1} est la $n^{\text{ème}} + 1$ valeur générée de la variable X
- X_n est la précédente valeur générée
- a est un coefficient multiplicatif
- b est un coefficient additif
- m est un modulo

Le modulo m implique que les séries générées soient périodiques. La période de cet algorithme est grande, c'est-à-dire que c'est à partir d'un grand nombre de valeurs générées, que la périodicité apparaît.

Mécanisme de génération – germe

La valeur de départ X_0 utilisée par le générateur est l'heure système ou un germe fourni par le programmeur. Si un même germe est utilisé plusieurs fois, alors les mêmes valeurs seront générées, cela provient du fait que l'algorithme est déterministe. Sa force principale réside dans la qualité aléatoire du germe de départ. L'utilisation de l'heure système qui change toutes les milliseconde et qui est unique (équivalent du Timestamp d'Unix : durée écoulée depuis le 1er janvier 1970 UTC) est un bon germe. Cependant, si vous trouvez un meilleur germe, utilisez le.

Cet algorithme convient donc pour de petits besoins. Mais les applications nécessitant un grand nombre de valeurs et un hasard de grande qualité doivent se tourner vers d'autres API.

Premier exemple

L'exemple suivant génère et affiche un entier pseudo-aléatoire entre 0 et 9.

```
// packages requis
import java.util.*;
import java.io.*;

// classe de test
public class testRand {

// fonction principale
    public static void main(String args[]) {
        Random rand = new Random(); // constructeur
        int i = rand.nextInt(10);    // génération
        System.out.println(i);      // affichage
    }
}
```

Constructeur

Il existe deux constructeurs. Ils permettent de définir le germe d'initialisation du générateur. Le premier prend pour germe l'heure système en milliseconde, tandis que l'autre prend un germe personnalisé.

public Random()

Il procède ainsi :

```
public Random() {  
    this(System.currentTimeMillis());  
}
```

public Random(long seed)

Il procède ainsi :

```
public Random(long seed) {  
    setSeed(seed);  
}
```

Germe d'initialisation

Le germe d'initialisation qu'il soit celui par défaut (l'heure système) ou personnalisé (entier long) est transformé et stocké.

Il peut également être redéfini après construction du générateur. Le germe doit être un entier long.

public void setSeed(long seed)

Il est transformé bit à bit comme suit :

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

OU exclusif entre le germe et la valeur **0x5DEECE66DL** puis ET bit à bit avec le résultat (moins 1) d'une rotation à gauche de 48 bits du nombre **1L**.

Méthode de base

La méthode de base suivante génère un entier pseudo-aléatoire. Elle ne peut pas être appelée (attribut `protected`), mais elle peut être réutilisées en cas d'héritage de la classe `Random`. Cette méthode est la base de toutes les autres décrites plus tard.

`protected int next(int bits)`

Elle est définie comme suit :

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

On reconnaît là encore l'algorithme à congruence linéaire.

Le paramètre `bits` permet une rotation bit à bit (décale les bits vers la droite, les zéros qui sortent à droite sont perdus, tandis que des zéros sont insérés à gauche).

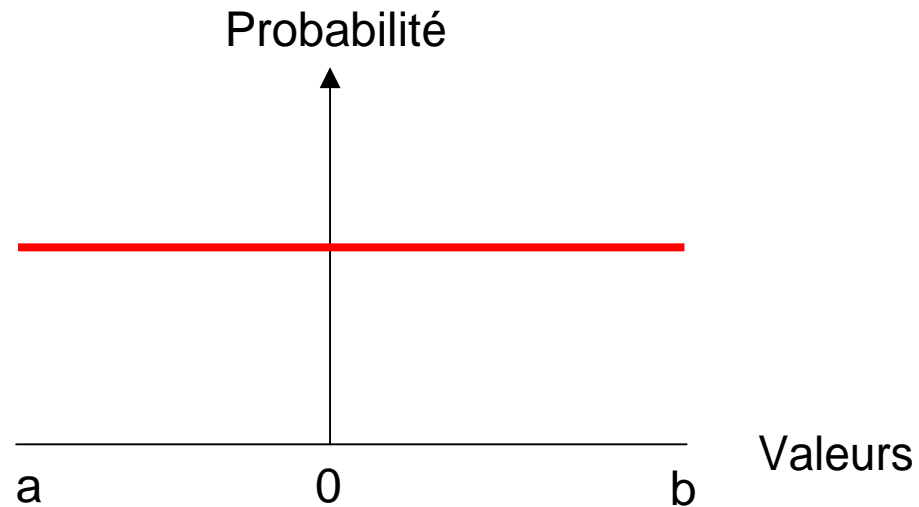
Les types générés

Seules les fonctions génératrices décrites ci-dessous peuvent être appelées depuis un programme Java.

Méthode	Type généré
boolean nextBoolean()	Booléen (true ou false)
int nextInt()	Entier (2^{32} valeurs possibles)
int nextInt(int n)	Entier entre 0 et $n-1$
long nextLong()	Entier long (2^{64} valeurs possibles)
float nextFloat()	Flottant (2^{32} valeurs possibles)
double nextDouble()	Flottant double (2^{53} valeurs possibles)
double nextGaussian()	Flottant double
void nextBytes(byte[] bytes)	Tableau d'entiers byte (valeur entre -2^7 et 2^7-1)

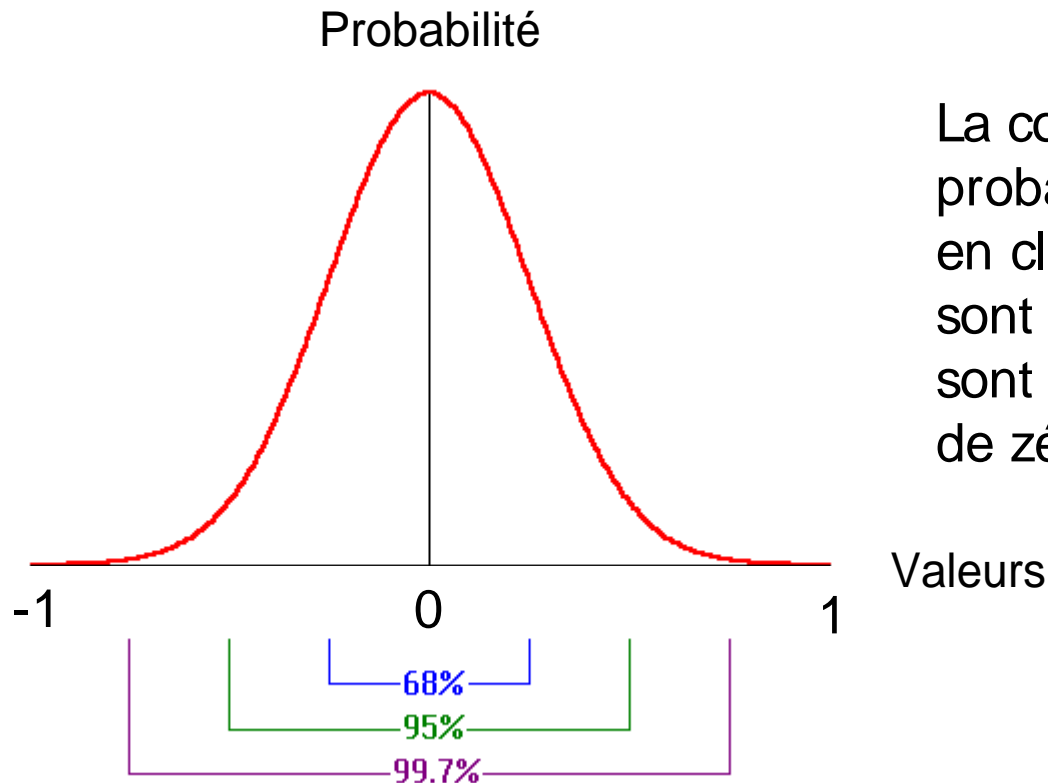
Répartition uniforme

Toutes les fonctions à l'exception de **nextGaussian()** suivent une loi uniforme. C'est-à-dire que les valeurs générées sont uniformément réparties dans l'intervalle du type retourné : les valeurs possibles sont équiprobables (approximativement, la qualité du générateur n'est pas parfaite). Ainsi, un histogramme de la fréquence d'apparition des valeurs d'une longue série sera constant.



Répartition normale

La méthode **nextGaussian()** suit une loi normal (dite aussi gaussienne). La densité de probabilité suit une courbe en cloche (répartition gaussienne).



La courbe de densité de probabilité suit une courbe en cloche : les valeurs ne sont pas équiprobables, elles sont plus fréquentes proche de zéro.

Loi exponentielle

L'algorithme suivant permet de générer une variable aléatoire suivant une loi de Poisson ayant une distribution exponentielle de paramètre `lambda`.

```
public static double Exp(double lambda) {  
    Random rand = new Random();  
    return - (1 / lambda) * log( 1 - rand.nextDouble() );  
}
```

Historique

- ▶ **20 mai 2004** : création du document (17 diapos)

Agissez sur la qualité de ce document en envoyant vos critiques et suggestions à l'auteur.

Pour toute question technique, se reporter au forum Java de Developpez.com

Reproduction autorisée uniquement pour un usage non commercial.

Hugo Etiévant
cyberzoide@yahoo.fr
<http://cyberzoide.developpez.com/>

