Entiers et manipulation de bits

Pratique de la programmation orientée-objet

Michel Schinz – 2015-04-20

Entiers Java

Types entiers

Java comporte quatre types de base dits **types entiers** (*integral types*) : `byte`, `short`, `int` et `long`.
Malgré leur nom, aucun ne correspond vraiment à la notion mathématique d'entier – l'ensemble $\mathbb{Z}$ – car tous ne peuvent représenter qu'un nombre fini de valeurs. Ainsi, aucun d'entre-eux ne peut représenter l'entier 10^{20}…
En raison de ce nombre limité de valeurs représentables, les opérations arithmétiques sur ces entiers (addition, soustraction, etc.) ont un comportement parfois différent de leur équivalent mathématique.

Entiers/séquences de bits

En mémoire, les entiers Java sont représentés par des séquences de bits de taille fixe : 8 bits pour `byte`, 16 pour `short`, 32 pour `int` et 64 pour `long`.

Ces séquences de bits sont interprétées comme des nombres exprimés en base 2. Par exemple, la séquence de huit bits 00001100 est interprétée comme l'entier douze.

bit n°	7	6	5	4	3	2	1	0
valeur	0	0	0	0	1	1	0	0

En effet :

$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$

$= 2^3 + 2^2$

$= 8 + 4$

$= 12$

Poids des bits

Dans une séquence de bits interprétée comme un entier, tous les bits n'ont pas la même importance :

- le bit le plus à droite vaut 1 s'il est à 1, 0 sinon,
- le second bit depuis la droite vaut 2 s'il est à 1, 0 sinon,
- le troisième bit depuis la droite vaut 4 s'il est à 1, 0 sinon,
- etc.

De manière générale, le bit à la position n vaut 2^n s'il est à 1, 0 sinon.

Pour cette raison, le bit le plus à gauche est dit **bit de poids (le plus) fort** (*most-significant bit*, ou *MSB*) tandis que le bit le plus à droite est dit **bit de poids (le plus) faible** (*least-significant bit* ou *LSB*).

Entiers négatifs

Comment représenter les entiers négatifs sous la forme d'une séquence de bits ?
Plusieurs solutions existent mais celle qui est utilisée par Java – et de manière presque universelle en informatique – est appelée **complément à deux** (*two's complement*).
En complément à deux, la représentation d'un entier négatif s'obtient en inversant les bits de la représentation de sa valeur absolue puis en ajoutant 1.
Par exemple, sur 8 bits, l'entier 12 est représenté par 00001100, donc l'entier –12 est représenté par 11110100 : en inversant les bits de 00001100 on obtient 11110011, puis en ajoutant 1 on obtient 11110100.

Complément à deux

Le complément à deux a plusieurs propriétés importantes :

- les opérations arithmétiques (addition, soustraction, etc.) n'ont pas besoin de traiter les valeurs négatives de manière particulière, tout fonctionne « tout seul »,
- le bit de poids fort donne le signe de la valeur : s'il vaut 0, elle est positive ou nulle, s'il vaut 1, elle est négative,
- une séquence de n bits peut représenter :
  - $2^{n-1}-1$ valeurs strictement positives,
  - 2^{n-1} valeurs strictement négatives, et
  - zéro (qui n'a qu'une représentation).

Constantes entières

En Java, les valeurs entières constantes – appelées aussi entiers littéraux (*integer literals*) – peuvent être écrites simplement en base dix :

```
int earthRadius = 6371;
```

Les chiffres d'un entier littéral peuvent être séparés par des caractères souligné (_) pour faciliter la lecture :

```
int earthRadius = 6_371;
```

Les entiers littéraux ont toujours le type `int`, sauf si on leur ajoute le suffixe `l` ou `L`, auquel cas ils ont le type `long` :

```
long earthPopulation = 7_130_000_000L;
```

(Aucun suffixe similaire n'existe pour `byte` ou `short`, il faut donc recourir à des conversions, implicites ou explicites).

Notation binaire

Les entiers littéraux peuvent être écrites en binaire (base deux), il suffit pour cela de leur ajouter le préfixe `0b` ou `0B` :

```
int twelve = 0b1100; // vaut 12
int maxInt =
  0b01111111_11111111_11111111_11111111;
long twelveAsLong = 0b1100L; // vaut 12
```

Attention, ces entiers littéraux binaires peuvent être négatifs même en l'absence de signe de négation (`-`) :

```
int minusOne = // vaut -1
  0b11111111_11111111_11111111_11111111;
```

et, dès lors, positifs même en présence d'une négation :

```
int one = // vaut 1
  -0b11111111_11111111_11111111_11111111;
```

Notation hexadécimale

Les entiers littéraux peuvent être écrits en hexadécimal (base seize) avec le préfixe `0x` ou `0X`. Les lettres A à F – minuscule ou majuscule – sont utilisées pour représenter les chiffres après 9 :

```
int twelve = 0xC;      // vaut 12
int thirtyTwo = 0x20; // vaut 32
int x = 0xDEAD_BEEF;  // vaut -559038737
long minusOne =       // vaut -1 (long)
  0xFFFF_FFFF_FFFF_FFFFL;
```

L'intérêt de la notation hexadécimale comparé à la notation décimale est que chaque chiffre représente exactement quatre bits.

Notation octale

Finalement, les entiers littéraux peuvent être écrits en octal (base huit) grâce au préfixe `0`.
Cette notation, héritée de C, est totalement anachronique mais il faut néanmoins la connaître car elle peut engendrer des surprises étant donné le préfixe utilisé !
En effet, mettre un (ou plusieurs) `0` en tête d'un entier littéral change sa base, donc généralement sa valeur. Par exemple :

```
int thirty    =  30; // vaut 30
int notThirty = 030; // vaut 24 (!)
```

Opérations arithmétiques

Opérations arithmétiques

Les opérations arithmétiques de base sont disponibles sur les entiers, à savoir :

- l'addition (+),
- la soustraction (−),
- la multiplication (*),
- la division entière (/),
- le reste de la division entière (%), souvent appelé modulo.

La plupart de ces opérations peuvent produire des valeurs non représentables dans le type entier concerné. On dit alors qu'il y a **dépassement de capacité** (*overflow*).

Dépassement de capacité

En cas de dépassement de capacité, la valeur résultant de l'opération arithmétique est simplement tronquée au nombre de bits du type du résultat. Dès lors, cette valeur est incorrecte d'un point de vue mathématique.
En particulier, notez qu'en raison de l'utilisation du complément à deux, il est même possible que le résultat n'ait pas le bon signe !
Les dépassements de capacité sont la source de nombreux problèmes de sécurité, particulièrement dans des langages non sûrs comme C et C++.

Exemple : addition

Le comportement en cas de dépassement de capacité peut s'illustrer en calculant la somme de 120 (01111000_2) et 10 (00001010_2) sur des entiers de type `byte` (8 bits) :

retenue	1	1	1	1				
120	0	1	1	1	1	0	0	0
+ 10	0	0	0	0	1	0	1	0
=	1	0	0	0	0	0	1	0

En d'autres termes, si on calcule la somme 120 + 10 avec des valeurs de type `byte`, on obtient –126 (10000010_2) et pas 130 !

Le même genre de problèmes existe avec les autres types entiers, mais pour des valeurs plus grandes.

Dépassement de capacité

Parmi les problèmes moins évidents dus aux dépassement de capacité, on peut en citer deux :

- la valeur absolue d'un entier Java peut être négative – si cet entier est la plus petite valeur entière représentable, p.ex. `Integer.MIN_VALUE` pour le type `int`,
- la négation (−) peut se comporter en Java comme l'identité pour une valeur non nulle – là aussi, cela n'est vrai que pour la plue petite valeur entière représentable.

Il est bon de garder cela à l'esprit, surtout lorsqu'on écrit du code sensible.

Exemple : comparateurs

Un exemple de problème subtile dû aux dépassements de capacité apparaît dans la méthode d'inversion de comparateur suivante :

```
<T> Comparator<T> invComp(Comparator<T> c) {
  return (o1, o2) -> -c.compare(o1, o2);
}
```

Cette méthode fonctionne toujours, sauf dans le cas où le comparateur `c` retourne `Integer.MIN_VALUE`... La manière correcte d'inverser un comparateur est donc :

```
<T> Comparator<T> invComp(Comparator<T> c) {
  return (o1, o2) -> c.compare(o2, o1);
}
```

Opérations bit à bit

Opérations bit à bit

En plus des opérations arithmétiques, Java offre ce que l'on appelle des **opérations bit à bit** (*bitwise operations*), travaillant sur les séquences de bits représentant les entiers.
Ces opérations adoptent un point de vue différent des entiers que les opérations arithmétiques, puisqu'elles les considèrent comme des séquences de bits et pas comme les entiers représentés par celles-ci.
Dès lors, initialement en tout cas, il est préférable de ne pas mélanger ces deux points de vue, et de traiter les entiers Java soit comme des entiers mathématiques – auxquels on applique des opérations arithmétiques – soit comme des séquences de bits – auxquelles on applique des opérations bit à bit.

Opérations bit à bit

En Java, les opérations bit à bit sont au nombre de sept :

- ~x : inversion (ou complément),
- x << y : décalage à gauche,
- x >> y : décalage à droite arithmétique,
- x >>> y : décalage à droite logique,
- x & y : conjonction (et) bit à bit,
- x | y : disjonction (ou) bit à bit,
- x ^ y : disjonction exclusive (ou exclusif) bit à bit.

L'inversion est une opération unaire (à un seul argument), toutes les autres sont binaires (à deux arguments).

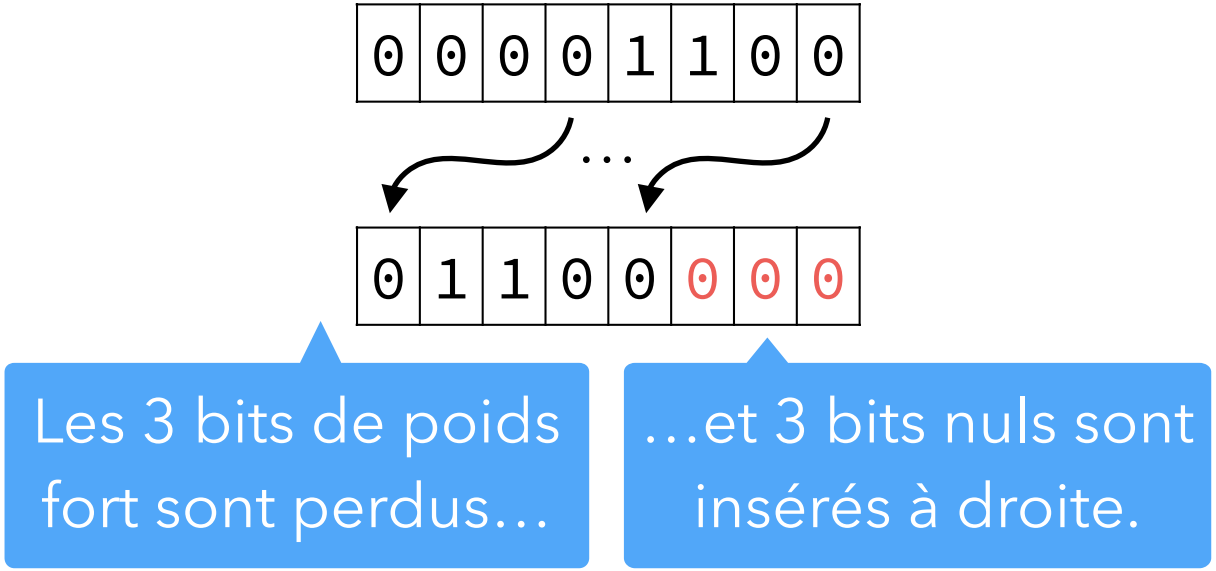
Complément parallèle

L'**inversion** ou **complément parallèle** ou **bit à bit** (*bitwise complement*), noté ~, inverse chaque bit de son opérande.
Par exemple, `~0b11110000` vaut `0b00001111`.
Etant donné que les valeurs négatives sont représentées au moyen du complément à deux, l'inversion et la négation sont liées par l'égalité suivante, valable pour toute valeur x d'un type entier :

```
-x == ~x + 1
```

Décalage à gauche

La valeur de l'expression x << y est obtenue par décalage vers la gauche, de y positions, des bits de x.
Par exemple, `0b00001100 << 3` vaut `0b01100000` :



Attention : la valeur de y est prise modulo 32 lors du décalage d'une valeur de type `int`, et modulo 64 lors du décalage d'une valeur de type `long`.

Décalages à droite

Les deux opérateurs de décalage à droite, >> et >>>, sont similaires à celui de décalage à gauche, mais travaillent dans la direction opposée.

Le décalage dit **arithmétique**, noté >>, copie le bit de poids fort dans toutes les positions laissées libres par le décalage.

Le décalage dit **logique**, noté >>>, insère quant à lui des bits nuls, comme l'opérateur de décalage à gauche.

Par exemple :

`0b11110000 >>` 2 vaut `0b11111100`, mais

`0b11110000 >>>` 2 vaut `0b00111100`.

Les deux opérateurs ne diffèrent que pour les valeurs dont le bit de poids fort vaut 1, c-à-d les valeurs négatives.

Décalage et multiplication

Un décalage à gauche de n bits est équivalent à une multiplication par 2^n :

`x * 2^n == x << n`

(Pour la même raison que, en base 10, un décalage des chiffres d'un nombre de n positions sur la gauche est équivalent à une multiplication par 10^n).

Un décalage à droite de n bits est équivalent à une division *entière* par 2^n pour les valeurs non négatives :

`x / 2^n == x >> n` (si $x \geq 0$)

(Pour les valeurs négatives, les deux expressions diffèrent de une unité dans certains cas.)

Opérateurs parallèles

Les **opérateurs binaires parallèles** ou **bit à bit** (*binary bitwise operators*) &, | et ^ appliquent la même opération à chacun des bits de leurs opérandes. Ces opérations sont définies par les tables ci-dessous.

et (&)

&	0	1
0	0	0
1	0	1

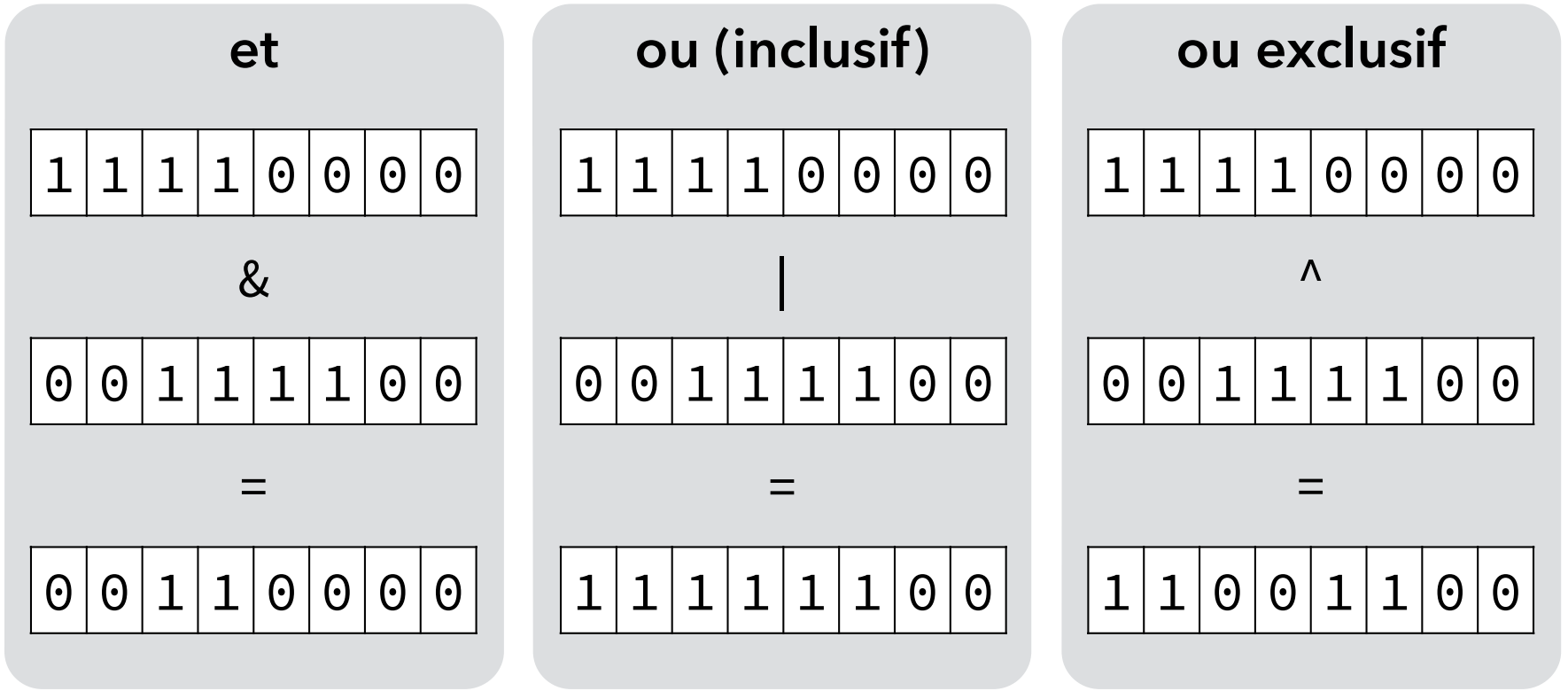
ou (|)

|	0	1
0	0	1
1	1	1

ou exclusif (^)

^	0	1
0	0	1
1	1	0

Opérateurs parallèles



&, | et ^ booléen

Notez au passage que Java permet aussi l'utilisation des opérations &, | et ^ sur les valeurs booléennes. Elles correspondent alors respectivement à la conjonction, la disjonction (inclusive) et la disjonction exclusive en logique.

conjonction

&	false	true
false	false	false
true	false	true

disjonction (incl.)

|	false	true
false	false	true
true	true	true

disjonction (excl.)

^	false	true
false	false	true
true	true	false

Contrairement aux opérateurs && et ||, qui n'évaluent leur second argument que si nécessaire, les opérateurs & et | évaluent toujours leurs deux arguments.

Exemples

Masque

Il est souvent utile de manipuler un ou plusieurs bits d'un entier sans toucher aux autres. Pour ce faire, on construit tout d'abord un entier – appelé le **masque** (*mask*) – dont seuls les bits à manipuler sont à 1. Ensuite, on utilise l'opération bit à bit appropriée (&, | ou ^), appliquée au masque et à la valeur.

Un masque peut soit s'écrire directement sous forme d'entier littéral – généralement en base 2 ou 16 – soit se construire en combinant décalages et disjonctions :

```
int mask13 = 1 << 13; // uniquement bit 13
int mask17 = 1 << 17; // uniquement bit 17
int mask13_17 =
  mask13 | mask17;    // bits 13 et 17
```

Test de bit(s)

Pour tester si un ou plusieurs bits d'un entier sont à 1, on utilise le « et bit à bit » entre le masque et l'entier, puis on regarde si la valeur obtenue est égale au masque.

Par exemple, pour tester si les bits 13 et 17 de l'entier x sont à 1, on écrit :

```
boolean bits13_17Set =
   (x & mask13_17) == mask13_17;
```

Pour tester si tous ces bits sont à 0, on écrit :

```
boolean bits13_17Cleared =
   (x & mask13_17) == 0;
```

Pour tester si au moins l'un de ces bits est à 1, on écrit :

```
boolean bit13OrBit17Set =
   (x & mask13_17) != 0;
```

Mise à 1 de bit(s)

Pour mettre à 1 un ou plusieurs bits d'un entier sans toucher aux autres, on utilise le « ou bit à bit » appliqué au masque et à l'entier.

Par exemple, pour mettre à 1 les bits 13 et 17 de l'entier x, on écrit :

```
int xWithBits13_17Set = x | mask13_17;
```

Mise à 0 de bit(s)

Pour mettre à 0 un ou plusieurs bits d'un entier sans toucher aux autres, on utilise le « et bit à bit » appliqué au complément du masque et à l'entier.
Par exemple, pour mettre à 0 les bits 13 et 17 de l'entier x, on écrit :

```
int xWithBits13_17Cleared = x & ~mask13_17;
```

Inversion de bit(s)

Pour inverser un ou plusieurs bits d'un entier sans toucher aux autres, on utilise le « ou exclusif bit à bit » appliqué au masque et à l'entier.
Par exemple, pour inverser les bits 13 et 17 de l'entier x, on écrit :

```
int xWithBits13_17Toggled = x ^ mask13_17;
```

Multiplication, division par 2^n

La multiplication par une puissance de deux peut se faire au moyen d'un décalage à gauche :

`x * 2^n == x << n`

Pour les valeurs non négatives uniquement, la division par une puissance de deux peut se faire au moyen d'un décalage à droite :

`x / 2^n == x >> n` (si $x \geq 0$)

Pour ces mêmes valeurs, le reste de la division par une puissance de deux peut se faire par masquage :

`x % 2^n == x & ((1 << n) - 1)` (si $x \geq 0$)

Test de parité

Un cas particulier du reste d'une division par une puissance de deux est le test de parité.
Pour savoir si une valeur est paire (respectivement impaire), il suffit de regarder si son bit de poids faible vaut 0 (respectivement 1). Cela fonctionne également pour les valeurs négatives.
Par exemple, pour tester si l'entier `x` est pair, on écrit :

```
boolean isXEven = (x & 1) == 0;
```

et pour tester s'il est impair :

```
boolean isXOdd = (x & 1) == 1;
```

Entiers
dans l'API Java

Classes d'entiers

La bibliothèque Java contient, dans le paquetage `java.lang`, une classe pour chaque type d'entier :

- `Byte` pour `byte`,
- `Short` pour `short`,
- `Integer` pour `int`,
- `Long` pour `long`.

Ces classes ont deux buts :

1. servir de « classes d'emballage » pour la généricité,
2. offrir, sous forme de méthodes statiques, des opérations sur les valeurs du type qu'elles représentent.

Auto-emballage (rappel)

Pour mémoire, il n'est pas possible d'utiliser un type de base comme paramètre d'un type générique. Par exemple, une liste d'entiers ne peut être déclarée ainsi :

```
List<int> l = Arrays.asList(4); // incorrect !
```

Pour contourner cette limitation, on peut « emballer » les entiers dans des objets d'une classe appropriée, puis faire une liste d'instances de cette classe :

```
List<Integer> = Arrays.asList(new Integer(4));
```

En fait, le compilateur génère automatiquement le code d'emballage (et de déballage, non illustré ici) et on peut donc écrire simplement :

```
List<Integer> l = Arrays.asList(1);
```

Taille des types

Chaque classe d'entiers fournit des attributs statiques donnant des informations quant à la taille du type entier correspondant :

- `MIN_VALUE` et `MAX_VALUE` donnent respectivement la plus petite et la plus grande valeur représentable par le type entier,
- `SIZE` donne la taille, en bits, du type entier,
- `BYTES` donne la taille, en octets, du type entier.

Par exemple, `Byte.MIN_VALUE` vaut –128, `Byte.MAX_VALUE` vaut 127, `Byte.SIZE` vaut 8 et `Byte.BYTES` vaut 1.

Conversion chaîne→entier

Chaque classe d'entiers offre deux variantes d'une méthode statique permettant de transformer une chaîne en l'entier signé correspondant.

Par exemple, `Integer` offre :

- `int parseInt(String s, int b)` : retourne l'entier dont la représentation en base `b` ($2 \leq b \leq 36$) est la chaîne `s`, ou lève une exception si la chaîne est invalide ;
- `int parseInt(String s)` : idem pour la base 10.

Les classes `Byte`, `Short` et `Long` offrent des méthodes similaires, nommées respectivement `parseByte`, `parseShort` et `parseLong`.

Conversion entier→chaîne

Les classes `Integer` et `Long` offrent deux variantes d'une méthode statique permettant de convertir un entier en chaîne de caractères.

Par exemple, `Integer` offre :

- `String toString(int i, int b)` : retourne la chaîne représentant l'entier `i` en base `b` ($2 \leq b \leq 36$),
- `String toString(int i)` : idem pour la base 10.

Les classes `Byte` et `Short` offrent uniquement la seconde variante.

Comptage de bits

`Integer` et `Long` offrent des méthodes statiques permettant de compter certains types de bits.

Par exemple, `Integer` offre :

- `int bitCount(int i)` : retourne le nombre de bits à 1 dans `i` ;
- `int numberOfLeadingZeros(int i)` : retourne le nombre de bits à 0 en tête (à gauche) de `i` ;
- `int numberOfTrailingZeros(int i)` : retourne le nombre de bits à 0 en queue (à droite) de `i`.

Les méthodes de `Long` sont identiques mais prennent un argument de type `long`.

Position des bits

`Integer` et `Long` offrent des méthodes statiques permettant de déterminer la position de certains bits.

Par exemple, `Integer` offre :

- `int lowestOneBit(int i)` : retourne 0 si `i` vaut 0, ou une valeur ayant un seul bit à 1, dont la position est celle du bit à 1 de poids de plus faible de `i`,
- `int highestOneBit(int i)` : idem, mais pour le bit de poids le plus fort.

Ainsi :

```
Integer. lowestOneBit(0b1110) == 0b0010
Integer.highestOneBit(0b1110) == 0b1000
```

Rotation

Les classes `Integer` et `Long` offrent chacune des méthodes statique permettant d'effectuer une rotation des bits.

Par exemple, `Integer` offre :

- `int rotateLeft(int i, int d)` : retourne l'entier obtenu par rotation des bits de `i` de `d` positions vers la gauche ;
- `int rotateRight(int i, int d)` : idem, mais vers la droite.

Une rotation est similaire à un décalage, mais les bits qui sont éjectés d'un côté sont réinjectés de l'autre.

Inversion de bits et octets

Les classes `Integer` et `Long` offrent des méthodes statiques permettant d'inverser l'ordre des bits ou des octets d'un entier.

Par exemple, `Integer` offre :

- `int reverse(int i)` : retourne l'entier obtenu en inversant l'ordre des bits de `i` ;
- `int reverseBytes(int i)` : retourne l'entier obtenu en inversant l'ordre des octets de `i`.

`Short` offre également `reverseBytes`.

Exemple : couleurs empaquetées

Couleurs

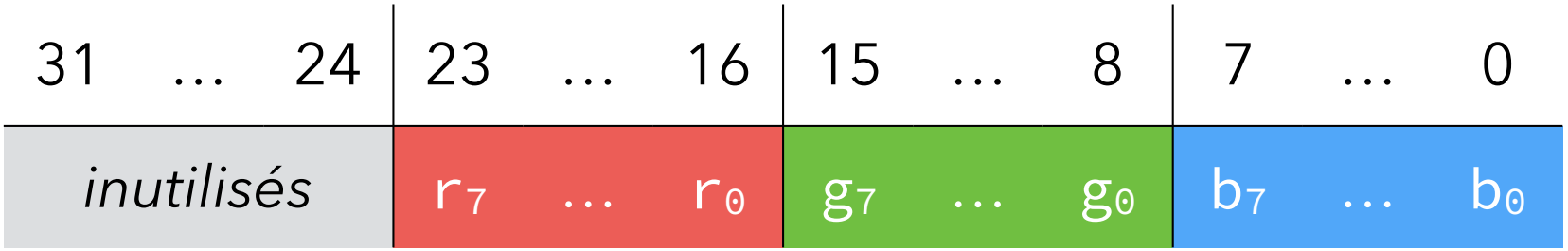
En informatique, une couleur est souvent représentée par ses trois composantes rouge, verte et bleue, chacune étant un nombre réel compris entre 0 et 1 (inclus).
La manière la plus naturelle de représenter une couleur en Java consiste donc à créer une classe dotée de trois champs de type `double` (ou `float`).
Bien que naturelle, cette représentation est coûteuse en espace, chaque couleur occupant au moins 3×8 = 24 octets (192 bits), ou la moitié si le type `float` est utilisé.

Couleurs empaquetées

Une manière plus compacte, mais moins précise, de représenter une couleur consiste à représenter chaque composante par un entier de 8 bits – compris entre 0 et 255 – puis à les « empaqueter » dans un entier de 32 bits.
De la sorte, une couleur occupe exactement 32 bits au lieu de 192 dans la représentation précédente, un gain appréciable.
(Les 8 bits restants sont soit inutilisés, soit utilisés pour stocker l'opacité de la couleur).

Couleurs empaquetées

Les composantes d'une couleur peuvent être empaquetées de nombreuses manières.
Dans ce qui suit, la composante bleue est placée dans les huit bits de poids faible, suivie de la verte puis la rouge.
Cette technique d'empaquetage est souvent désignée par l'acronyme RGB.



La notation hexadécimale convient assez bien pour écrire directement des couleurs empaquetées de la sorte. Par exemple `0xFF_00_00` représente un rouge pur (1,0,0).

Empaquetage

Le passage d'une couleur exprimée sous la forme de trois composantes réelles comprises entre 0 et 1 à une couleur empaquetée RGB se fait en combinant décalages et « ou » parallèle :

```
double r = …, g = …, b = …;
int r0 = (int)(r * 255.9999); // 0 ≤ r0 ≤ 255
int g0 = (int)(g * 255.9999); // 0 ≤ g0 ≤ 255
int b0 = (int)(b * 255.9999); // 0 ≤ b0 ≤ 255
int rgb = (r0 << 16) | (g0 << 8) | b0;
```

Dépaquetage

Une composante quelconque d'une couleur empaquetée peut être extraite en combinant un décalage avec un masquage.

Par exemple, la composante verte peut être obtenue par un décalage à droite de 8 bits suivi d'un masquage pour ne garder que les 8 bits de poids faible :

```
int rgb = …;
int g0 = (rgb >> 8) & 0xFF;
```

Cette valeur entière, comprise entre 0 et 255, peut être ramenée à l'intervalle 0…1 par une simple division :

```
double g = (double)g0 / 255d;
```

Exemple avancé : comptage de bits

Entiers/séquences de bits

On l'a dit, il est généralement préférable d'utiliser les entiers Java de l'une des deux manières suivantes :

- soit comme des entiers mathématiques, auquel cas on leur applique uniquement des opérations arithmétiques,
- soit comme des séquences de bits, auquel cas on leur applique uniquement des opérations bit à bit.

Toutefois, il peut être intéressant de combiner les deux pour résoudre certains problèmes.

Comptage de population

Il est parfois utile de compter le nombre de bits à 1 d'un entier, ce qu'on appelle le **comptage de population** (*population count*). La bibliothèque Java offre la méthode `bitCount` dans ce but.

Admettons que cette méthode n'existe pas et que nous désirions en écrire une version permettant de compter le nombre de bits à 1 d'un entier de type `byte` (8 bits).

Comment faire ?

Comptage de population

L'idée la plus naturelle consiste à écrire une boucle qui teste individuellement chacun des 8 bits et compte le nombre d'entre-eux qui sont à 1 :

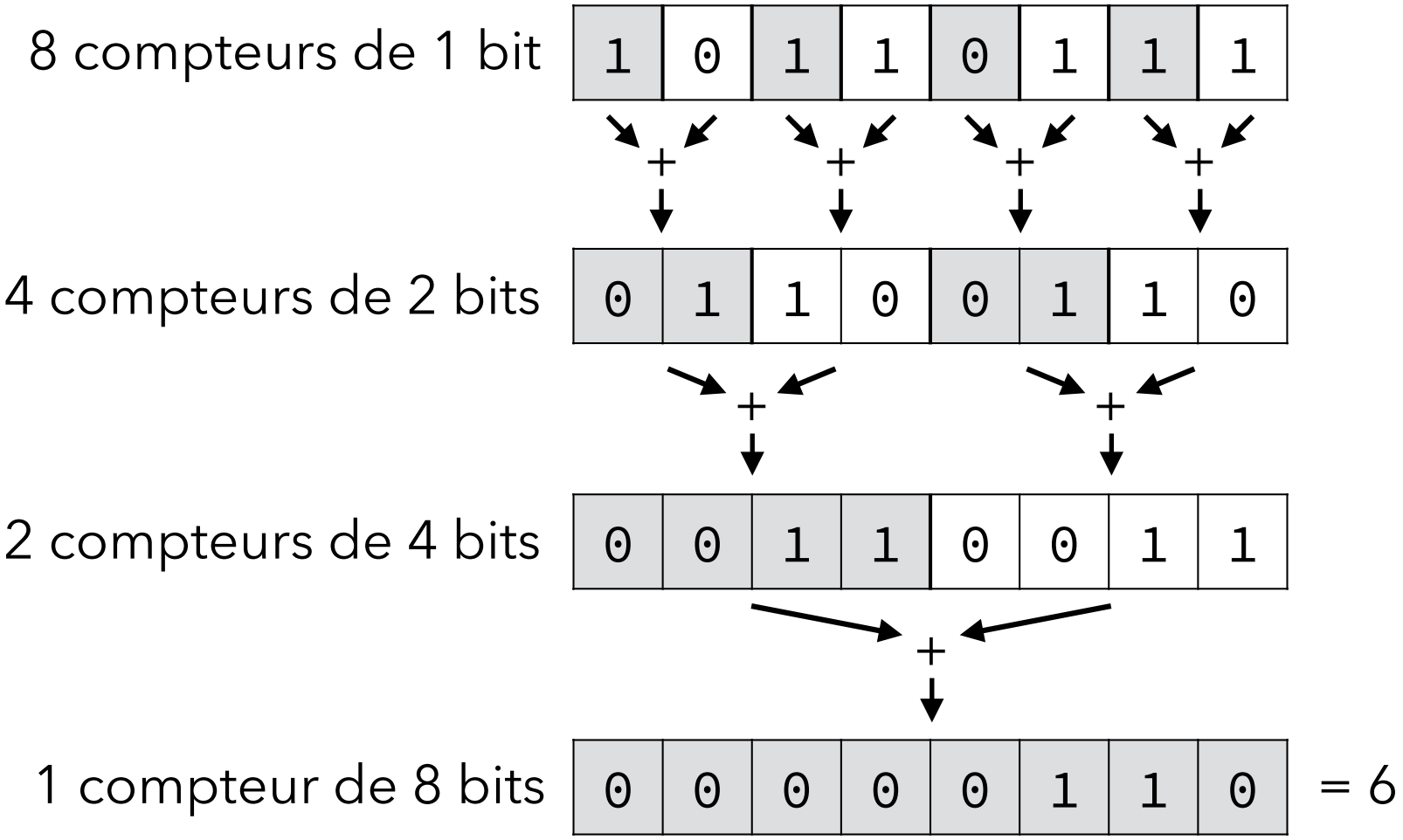
```
public static int bitCount(byte b) {
  int c = 0;
  for (int i = 0; i < 8; ++i) {
    if ((b & (1 << i)) != 0)
      c += 1;
  }
  return c;
}
```

Même si est correct, il est possible de faire mieux…

Comptage de population

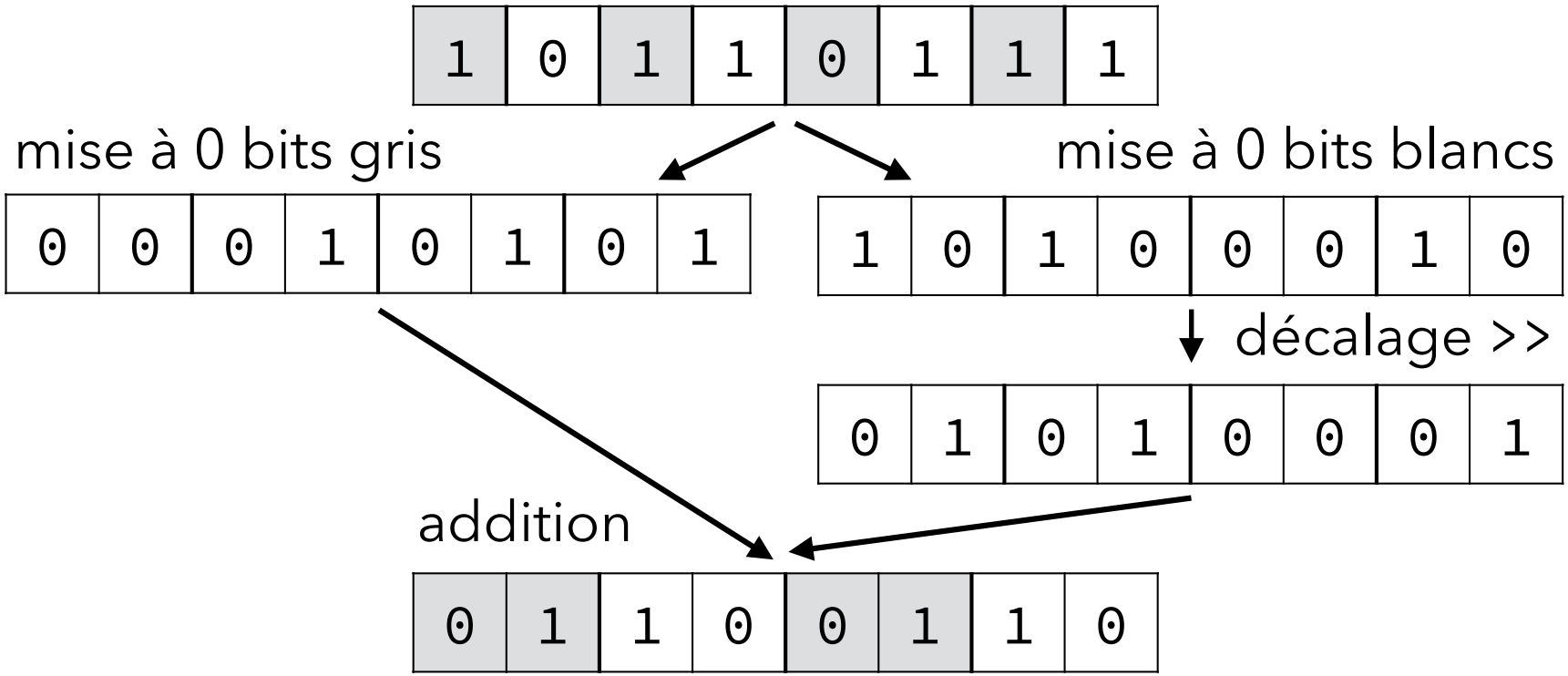
L'idée de la version optimisée du comptage de population consiste à considérer l'entier comme une séquence de huit compteurs de 1 bit chacun.
Ces huit compteurs peuvent être additionnés deux par deux en combinant un masquage, un décalage et une addition, pour produire quatre compteurs de 2 bits chacun.
Ensuite, ces quatre compteurs peuvent être additionnés deux par deux, pour produire deux compteurs de 4 bits chacun.
Finalement, ces deux compteurs peuvent être additionnés pour obtenir un seul compteur de 8 bits, qui contient le résultat.

Comptage de population



Comptage de population

L'addition parallèle de petits compteurs se fait en combinant masquage, décalage de 1 position et addition. Par exemple, pour additionner huit compteurs à 1 bit et obtenir quatre compteurs à 2 bits :



Comptage de bits à 1

Cette seconde version du comptage de population s'écrit facilement en Java :

```
public static int bitCount(byte b) {
  int s1 = ((b  & 0b10101010) >>> 1)
    + (b  & 0b01010101);
  int s2 = ((s1 & 0b11001100) >>> 2)
    + (s1 & 0b00110011);
  int s3 = ((s2 & 0b11110000) >>> 4)
    + (s2 & 0b00001111);
  return s3;
}
```

Comparaison

Les deux versions peuvent être comparées en comptant le nombre d'opérations qu'elles effectuent :

- décalages : 8 pour la première, 3 pour la seconde,
- « et bit à bit » : 8 pour la première, 3 pour la seconde,
- additions : 8 à 16 pour la première, 3 pour la seconde,
- tests : 16 pour la première, 0 pour la seconde.

La seconde version, qui peut d'ailleurs encore être améliorée, est donc nettement meilleure que la première. Cela dit, la différence entre les deux ne sera mesurable que dans un programme utilisant massivement le comptage de population…

Pour en savoir plus...

Cet exemple est tiré du livre *Hacker's Delight* de Henry Warren, qui en contient beaucoup d'autres.
(Voir `http://www.hackersdelight.org/`)