Algorithmes simples en Java
Corrigé

Objectifs

— Écrire des algorithmes simples en Java
— Manipuler les structures de contrôle Java

> **Remarque préalable :** Dans les exercices suivants on ne réalisera pas de saisie mais on se contentera d'initialiser des données (théoriquement fournies par l'utilisateur) dans le programme lui-même.

Exercice 1 : Conversion pouce/centimètre
Écrire un programme qui réalise la conversion pouce/centimètre d'une longueur saisie au clavier. Une longueur sera saisie comme un nombre réel suivi d'un caractère précisant l'unité. Les unités possibles sont le pouce (p), le centimètre (c) ou le mètre (m). Le programme affichera la longueur exprimée en pouce et en centimètre.

Voici des exemples d'exécution du programme :

```
Entrez une longueur : 1p
1 p = 2.54 cm

Entrez une longueur : 2m
78.7402 p = 200 cm

Entrez une longueur : 2km
0 p = 0 cm
```

Modifier le programme pour permettre la saisie de l'unité aussi bien en minuscules qu'en majuscules.

Solution :

```
R0 : Afficher une longueur saisie au clavier en pouces et en centimètres
```

Nous reprenons les jeux de test proposés dans le sujet. Nous ajoutons un jeu de test qui consiste à saisir une unité en centimètres. Par exemple, 5.08 cm qui doit donner 1 p = 5.08 p.

Lorsque l'on regarde les résultats d'exécution donnés dans le sujet (et qui servent donc de spécification pour le programme à écrire), on constate que, quelle que soit l'unité utilisée pour saisir la longueur, il faut afficher d'abord la longueur en pouces, puis son équivalent en centimètres. En conséquence, on en déduit qu'il faut calculer les deux longueurs (une dans chaque unité). Ainsi, après saisi de la longueur de l'utilisateur, nous allons la convertir dans les deux unités puis afficher la ligne d'équivalence.

```
R1 : Raffinage De « Afficher une longueur saisie au clavier... »
   | Saisir la longueur          valeur: out Réel ; unité: out Caractère
   | Calculer la longueur en pouces et en centimètres
                                 valeur, unité : in
                                 lg_p  : out Réel -- longueur en pouces
                                 lg_cm : out Réel -- longueur en centimètres
   | Afficher le résultat                lg_p, lg_cm : in
```

La seule étape délicate est la deuxième. Calculer les longueurs en pouces et en centimètres dépend de l'unité saisie par l'utilisateur. Il s'agit donc d'utiliser une conditionnelle et plus précisément de faire un traitement par cas. Puisque l'unité est un caractère, donc un type scalaire, nous pouvons utiliser un `Selon`.

Remarque : Pour traiter le cas des majuscules, il suffit d'ajouter le cas majuscule à côté du cas minuscule correspondant.

On aurait également pu convertir la longueur saisie en minuscule avant de calculer les longueurs en pouces et centimètres.

Remarque : On aurait également pu utiliser des `Si` et `SinonSi`. Cependant, la structure du programme est plus claire en utilisant un `Selon`. Ceci est d'autant plus vrai si l'on traite également les majuscules.

Voici le raffinage correspondant.

```
R2 : Raffinage De « Calculer la longueur en pouces et en centimètres »
   | Selon unité Dans
   |     'p', 'P':       { la longueur a été saisie en pouces }
   |         lg_p <- valeur
   |         lg_cm <- lg_p * UN_POUCE
   |
   |     'c', 'C':       { la longueur a été saisie en centimètres }
   |         lg_cm <- valeur
   |         lg_p <- lg_cm / UN_POUCE
   |
   |     'm', 'M':       { la longueur a été saisie en mètres }
   |         lg_cm <- valeur * 100
   |         lg_p <- lg_cm / UN_POUCE
   |
   | Sinon               { Unité non reconnue }
   |     lg_p <- 0
   |     lg_cm <- 0
   | FinSelon
```

Notons que nous avons utilisé une constante symbolique (`UN_POUCE`) plustôt qu'une constante littéralle (2,54). L'intérêt est d'avoir un programme plus lisible et d'éviter la redondance (si on se trompe sur la valeur d'un pouce en centimètres il suffit de faire un seule changement pour corriger le programme dans le cas de la constante symbolique contre trois sinon).

```
Algorithme pouce2cm

    -- Afficher une longueur saisie au clavier en pouces et en centimètres.

Constantes
    UN_POUCE = 2.54     -- valeur en centimètres d'un pouce

Variables
    valeur: Réel        -- valeur de la longueur lue au clavier
    unité: Caractère    -- unité de la longueur lue au clavier
    lg_cm: Réel         -- longueur exprimée en centimètres
    lg_p: Réel          -- longueur exprimée en pouces

Début
    -- saisir la longueur (valeur + unité)
    Écrire ("Entrer␣une␣longueur␣(valeur␣+␣unité)␣:␣")
    Lire (valeur)               -- saisir la valeur
    Lire (unité)                -- saisir l'unité

    -- calculer la longueur en pouces et en centimètres
    Selon unité Dans
        'p', 'P':       { la longueur a été saisie en pouces }
            lg_p <- valeur
            lg_cm <- lg_p * UN_POUCE

        'c', 'C':       { la longueur a été saisie en centimètres }
            lg_cm <- valeur
            lg_p <- lg_cm / UN_POUCE

        'm', 'M':       { la longueur a été saisie en mètres }
            lg_cm <- valeur * 100
            lg_p <- lg_cm / UN_POUCE

    Sinon               { Unité non reconnue }
        lg_p <- 0
        lg_cm <- 0
    FinSelon

    -- afficher la longueur en pouces et en centimètres
    ÉcrireLn (lg_p, "␣p␣=␣", lg_cm, "␣cm")
Fin
```

```java
/** Afficher une longueur saisie au clavier en pouces et en centimètres.
  * @author     Xavier Crégut
  * @version    1.2
  */
class ConversionsPouceCm {

    public static void main(String[] args) {
        final double UN_POUCE = 2.54;   // centimètres

        // Saisir la longueur
```

```java
        System.out.print("Longueur : ");
        double lg = Clavier.readDouble();
        Clavier.skipWhite();
        char unité = Clavier.readChar();

        // Calculer la longueur en pouces et en centimètres
        double lg_p;    // longueur exprimée en pouces
        double lg_cm;   // longueur exprimée en centimètres
        switch (unité) {
            case 'p':           // la longueur a été saisie en pouces
            case 'P':
                lg_p = lg;
                lg_cm = lg * UN_POUCE;
                break;

            case 'm':           // la longueur a été saisie en mètres
            case 'M':
                lg_cm = lg * 100;
                lg_p = lg_cm / UN_POUCE;
                break;
                // Remarque : On pourrait remplacer les trois lignes
                // précédentes par seulement : lg = lg * 10.
                // Mais ce ne serait vraiment pas joli !

            case 'c':           // la longueur a été saisie en centimètres
            case 'C':
                lg_p = lg / UN_POUCE;
                lg_cm = lg;
                break;

            default:
                lg_p = lg_cm = 0;
        }

        // Afficher les résultats
        System.out.println(lg_p + " p = " + lg_cm + " cm");
    }

}
```

Exercice 2 : Puissance

Afficher la puissance entière d'un réel en utilisant somme et multiplication. On traiter d'abord le cas où l'exposant est positif avant de généraliser aux entiers relatifs.

Solution :

On peut, dans un premier temps, se demander si la puissance entière d'un réel a toujours un sens. En fait, ceci n'a pas de sens si le nombre est nul et si l'exposant est strictement négatif.

```
    2 puissance -3    ->  0.125 -- cas nominal (puissance négative)
```

On choisit de contrôler la saisie de manière à garantir que nous ne sommes pas dans l'un de ces cas. Nous souhaitons donner à l'utilisateur un message le plus clair possible lorsque la saisie est invalide.

Nous envisageons les jeux de tests suivants :

```
nombre exposant  ->     x^n

   2        3    ->      8 -- cas nominal
   3        2    ->      9 -- cas nominal
   1        1    ->      1 -- cas nominal
   2       -3    ->  0.125 -- cas nominal (exposant négative)
   0        2    ->      0 -- cas nominal (x est nul)
   0        0    ->      1 -- par convention
 1.5        2    ->   2.25 -- x peut être réel
   0       -2    -> ERREUR -- division par zéro
```

Voyons maintenant le principe de la solution. Par exemple, pour calculer 2^3, on peut faire $2 * 2 * 2$. Plus généralement, on multiplie n fois x par lui-même (donc une boucle **Pour**).

Si on essaie ce principe, on constate qu'il ne fonctionne pas dans le cas d'une puissance négative. Prenons le cas de 2^{-3}. On peut le réécrire $(1/2)^3$. On est donc ramené au cas précédent. Le facteur multiplicatif est dans ce cas $1/x$ et le nombre de fois est $-n$.

Nous introduisons donc deux variables intermédiaires qui sont :

```
facteur: Réel   -- facteur multiplicatif pour obtenir les puissances
                -- successives de x
puissance: Réel -- abs(n).  On a : facteur^puissance = x^n
```

On peut maintenant formaliser notre raisonnement sous la forme du raffinage suivant.

```
R0 : Afficher la puissance entière d'un réel

R1 : Raffinage De « R0 »
        | Saisir avec contrôle les valeurs de x et n      x: out Réel; n: out Entier
        | { (x <> 0) Ou (n > 0) }
        | Calculer x à la puisssance n                    x, n: in ; xn: out Réel
        | Afficher le résultat                            xn: in Réel

R2 : Raffinage De « Saisir avec contrôle les valeurs de x et n »
        | Répéter
        |    | Saisir la valeur de x et n            x: out Réel; n: out Entier
        |    | Contrôler x et n                      x, n: in ; valide: out Booléen
        | JusquÀ valide                              valide: in

R1 : Raffinage De « Calculer x à la puisssance n »
        | Si x = 0 Alors
        |    | xn := 0
        | Sinon
        |    | Déterminer le facteur multiplicatif et la puissance
        |    |                      x, n: in ;
        |    |                      facteur out Réel ;
        |    |                      puissance: out Entier
        |    | Calculer xn par itération (accumulation)
        |    |                      n, facteur, puissance: in ; xn : out
        | FinSi

R3 : Raffinage De « Calculer xn par itération (accumulation) »
        | xn <- 1;
```

```
        | Pour i <- 1 JusquÀ i = puissance Faire
        |    | { Invariant : xn = facteur^i }
        |    | xn <- xn * facteur
        | FinPour
```

On peut alors en déduire l'algorithme suivant :

```
Algorithme puissance

    -- Afficher la puissance entière d'un réel

Variables
    x: Réel                -- valeur réelle lue au clavier
    n: Entier              -- valeur entière lue au clavier
    valide: Booléen        -- x^n peut-elle être calculée
    xn: Réel               -- x à la puissance n (en fait x^(i-1))
    facteur: Réel          -- facteur multiplicatif pour obtenir
                           -- les puissances successives
    exposant: Entier       -- abs(n).  On a facteur^exposant = x^n
    i: Entier              -- variable de boucle

Début
    -- Saisir avec contrôle les valeurs de x et n
    Répéter
        -- saisir la valeur de x et n
        Écrire("x = ")
        Lire(x)
        Écrire("n = ")
        Lire(n)

        -- contrôler x et n
        valide <- VRAI
        Si x = 0 Alors
            Si n = 0 Alors
                ÉcrireLn("x et n sont nuls. x^n est indéterminée.")
                valide <- FAUX
            SinonSi n < 0 Alors
                ÉcrireLn("x nul et n négatif. x^n n'a pas de sens.")
                valide <- FAUX
            FinSi
        FinSi

        Si Non valide Alors
            ÉcrireLn("  Recommencez !")
        FinSi
    JusquÀ valide
    { x <> 0 Ou n >= 0 }

    -- Calculer x à la puisssance n
    --   Déterminer le facteur multiplicatif et l'exposant
    Si n >= 0 Alors
        facteur <- x
        exposant <- n
    Sinon
```

```
        facteur <- 1/x
        exposant <- -n
    FinSi

    --   Calculer xn par itération (accumulation)
    xn <- 1;
    Pour i <- 1 JusquÀ i = exposant Faire
        { Invariant : xn = facteur^i}
        xn <- xn * facteur
    FinPour

    -- Afficher le résultat
    ÉcrireLn(x, "^", n, " = ", xn)
Fin.
```

```
/** Calculer la puissance entière d'un réel
  * @author     Xavier Crégut
  * @version    1.2
  */
class Puissance {

    final static double EPSILON = 1.0e-5;

    /** Calculer la puissance entière d'un réel.  On suppose x != 0 || n >= 0.
      * @param x le nombre réel
      * @param n la puissance entière
      * @return x à la puissance n
      */
    public static double puissance(double x, int n) {
        assert x != 0 || n >= 0;
        double resultat = 0;
        if (x == 0) {   // cas trivial
            resultat = 0;
        } else {
            // déterminer le facteur multiplicatif et la puissance
            double facteur = 0; // facteur multiplicatif
            int puissance = 0;  // abs(n)
                                // On a x^n == facteur^puissance
            if (n >= 0) {
                facteur = x;
                puissance = n;
            } else {
                facteur = 1.0 / x;
                puissance = -n;
            }

            // Calculer xn par itération (accumulation)
            resultat = 1;
            for (int i = 1; i <= puissance; i++) {
                // Invariant : resultat == facteur^i
                resultat = resultat * facteur;
            }
        }
```

```java
        return resultat;
    }

    public static void main(String[] args) {
        assert puissance(1, 1) == 1;
        assert puissance(2, 1) == 2;
        assert puissance(2, 2) == 4;
        assert puissance(3, 2) == 9;
        assert puissance(2, 3) == 8;
        assert puissance(2, -1) ==  0.5;        // Dangereux !
        assert puissance(2, -3) ==  0.125;      // Dangereux !
        assert Math.abs(puissance(2, -1) - 1.0 / 2.0) < EPSILON;
        // assert puissance(3, -1) ==  1.0 / 3.0;       // FAUSSE !!!
        assert Math.abs(puissance(3, -1) - 1.0 / 3.0) < EPSILON;
    }
}
```

Exercice 3 : Division entière

Étant donnés deux entiers positifs, calculer le quotient et le reste de la division euclidienne du premier nombre par le deuxième. On utilisera uniquement l'addition et la soustraction sur les entiers.

Solution :

Remarque : En utilisant seulement l'addition et la soustraction, on ne peut pas utiliser une boucle **`Pour`**. Il faut donc choisir entre **`TantQue`** et **`Répéter`**.

```
Algorithme div_mod

    -- Calculer le quotient (div) et le reste (mod) de la division entière de
    -- deux entiers lus au clavier

Variables
    dividende: Entier   -- dividende lu au clavier, positif
    diviseur: Entier    -- diviseur lu au clavier, strictement positif
    reste: Entier       -- reste de la division entière
    quotient: Entier    -- quotient de la division entière

Début
    -- saisir le dividende et le diviseur avec contrôle
    ...

    -- calculer le quotient et le reste
    reste <- dividende
    quotient <- 0
    TantQue reste >= diviseur Faire
        { Variant : reste }
        { Invariant : diviseur * quotient + reste = dividende }
        quotient <- quotient + 1
        reste <- reste - diviseur
    FinTQ

    -- afficher le résultat
    ÉcrireLn(dividende, "␣/␣", diviseur, "␣=␣",
            quotient, "␣*␣", diviseur, "␣+␣", reste)
```

```
Fin.

--      5       2       -->     2 * 2 + 1
--      10      2       -->     5 * 2 + 0
--      5       0       -->     diviseur nul !
```

```java
/** Afficher le reste et le produit de la division entière.
  * @author     Xavier Crégut
  * @version    1.1
  */
class DivMod {
    /** Afficher le quotient (div) et le reste (mod) de la division entière de
     * deux entiers.
     * @param dividende le dividende (strictement positif)
     * @param diviseur le diviseur (positif)
     */
    public static void divMod(int dividende, int diviseur) {
        int reste = dividende;  // reste de la division entière
        int quotient = 0;       // quotient de la division entière
        while (reste >= diviseur) {
            // Variant : reste;
            // Invariant : diviseur * quotient + reste = dividende;
            quotient++;
            reste = reste - diviseur;
        }
        System.out.println("Division␣de␣" + dividende + "␣par␣" + diviseur);
        System.out.println("quotient␣=␣" + quotient);
        System.out.println("reste␣=␣" + reste);
    }

    public static void main(String[] args) {
        divMod(5, 2);   // Resultat : 2 et 1
        divMod(2, 5);   // Resultat : 0 et 2
        divMod(0, 5);   // Resultat : 0 et 0
    }
}
```

Exercice 4 : Nombres premiers

Écrire un programme qui permet à son utilisateur de saisir une valeur entière et qui, en retour lui indique si c'est un nombre premier ou pas.

Solution : Il s'agit d'afficher si un nombre saisi par l'utilisateur est premier ou non.

```
R0 : Afficher si un nombre saisi est premier ou non
```

Pour ce qui concerne les jeux de test, on peut vérifier si le programme fonctionne sur les premiers nombres, par exemple de 1 à 20 ou de 1 à 100.

Le premier niveau de raffinage est classique. Il permet de clairement séparer la partie calcul de la partie interaction avec l'utilisateur.

```
R1 : Raffinage De « Afficher si un nombre saisi est premier ou non »
   | Saisir un nombre                             n: out Entier
   | Déterminer si le nombre est premier          n: in ; premier: out Booléen
   | Afficher le résultat                         n, premier: in
```

Seule la deuxième étape mérite d'être détaillée. Un nombre premier est un nombre qui n'admet pas de diviseurs autres que 1 et lui-même. Nous allons en proposer plusieurs raffinements que nous allons comparer d'un point de vu performance (en nombre d'opérations arithmétiques).

Ainsi, étant donné un entier n, on peut regarder s'il est divisible par les entiers compris entre 2 et $n-1$. L'idée est d'essayer chaque entier. Si on trouve un diviseur le n n'est pas premier. Si on ne trouve pas de diviseur, n est premier. On se sert donc de la variable booléenne que l'on initialise à $VRAI$ et qui sera mise à faux dès que l'on trouve un diviseur. On pourrait donc l'écrire avec un **Pour**.

```
premier <- VRAI
Pour diviseur <- 1 JusquÀ diviseur = n-1 Faire
    Si diviseur est un diviseur de n Alors
        premier <- FAUX
    FinSi
FinPour
```

Ce qui s'écrit de manière équivalente :

```
premier <- VRAI
Pour diviseur <- 1 JusquÀ diviseur = n-1 Faire
    premier <- premier Et (diviseur est un diviseur de n)
FinPour
```

Si l'utilisation d'un **Pour** est possible, elle est cependant peu judicieuse. En effet, dès qu'on a trouvé un diviseur, on sait que le nombre n'est pas premier et il est inutile d'essayer d'autres diviseurs. Nous devons donc utiliser un **TantQue** ou un **Répéter**. Nous optons pour un **TantQue**.

Remarquons que nous avons en fait initialisé la variable `premier` avec `n <> 1` pour tenir compte du fait que 1 n'est pas un nombre premier.

Voici le raffinage correspondant.

```
R2 : Raffinage De « Déterminer si un nombre est premier »
   | premier <-  n > 1
   | diviseur <- 2
   | TantQue premier Et (diviseur < n - 1) Faire
   |     premier <- Non diviseur est un diviseur de n
   |     diviseur <- diviseur + 1
   | FinTQ

R3 : Raffinage De « diviseur est un diviseur de n »
   | Résultat <- n Mod diviseur = 0
```

En fait, on sait qu'il n'est pas nécessaire de regarder les diviseurs au delà de $n/2$ car il ne peut pas y en avoir (trivial). On peut donc reformuler la condition du **TantQue**

```
   | TantQue premier Et (diviseur <= n Div 2) Faire
```

Cette optimisation permet de réduire par 2 le nombre d'opérations. Cependant, on peut faire beaucoup mieux. En effet, si un nombre n'est pas premier il admet un diviseur et en fait au moins deux dont l'un au moins est inférieur ou égal à sa racine carrée.

```
   | TantQue premier Et (diviseur <= racine carrée de n) Faire
```

Cette optimisation est bien meilleure que la précédente. En effet, si on considère un nombre premier supérieur à 10000, dans la version initiale, on doit considérer 10000 diviseurs, dans la première optimisation 5000 et seulement 100 dans la seconde.

Remarque : Pour éviter d'utiliser les réels et donc les problèmes d'arrondis (et éventuellement une perte de performance), on peut transformer l'expression

```
diviseur <= racine carrée de n
```

par (en élevant au carré)

```
diviseur * diviseur <= n
```

et, pour éviter les débordements (`diviseur * diviseur` peut dépasser la capacité des entiers), il est préférrable de réorganiser les calculs :

```
diviseur <= n Div diviseur
```

Une fois cette fransformation réalisée, on obtient donc :

```
| TantQue premier Et (diviseur <= n Div diviseur) Faire
```

On peut encore améliorer l'algorithme. En effet, si on sait que le nombre n'est pas divisible par 2, il est inutile d'essayer les diviseurs pairs (4, 6, 9, etc.). On peut se limiter aux diviseurs impairs (3, 5, 7, 9, 11...). Voici le raffinage correspondant.

```
R2 : Raffinage De « Déterminer si un nombre est premier »
  | Si n = 2 Alors
  |   premier <- VRAI
  | Sinon
  |  | premier <-  (n >= 2) Et Non (2 divise n)
  |  | diviseur <- 3
  |  | TantQue premier Et (diviseur <= n Div diviseur) Faire
  |  |     premier <- Non diviseur est un diviseur de n
  |  |     diviseur <- diviseur + 2
  |  | FinTQ
  | FinSi
```

Remarque : On pourrait se passer du `Si` en initialisant `premier` de la manière suivante :

```
premier = (n == 2) Ou ((n >= 3) Et (n Mod 2 <> 0))
```

Modifier le programme pour qu'il propose à l'utilisateur d'entrer une autre valeur à traiter ou d'arrêter le programme.

Solution : Le raffinage est le suivant :

```
R0 :  Indiquer si des entiers saisis au clavier sont premiers ou non

R1 :  Raffinage De « R0 »
        | Répéter
        |  | Saisir un entier
        |  | Indiquer le caractère premier de l'entier
        |  | Demander à l'utilisateur s'il veut continuer
        | JusquÀ réponse = 'n'
```

On peut en déduire le raffinage suivant :

```
Algorithme nb_premier

    -- Indiquer si un nombre est premier où non
    -- Possibilité de recommencer

Variables
    n: Entier             -- parcourir les entiers de 1 à max
    i: Entier             -- parcourir les diviseurs potentiels de n
    premier: Booléen      -- n est-il premier
    réponse: Caractère    -- réponse de l'utilisateur (o/n)

Début
    Répéter               -- analyser un nombre de l'utilisateur
        -- saisir le nombre
        ÉcrireLn ("J'indique␣si␣un␣nombre␣est␣premier")
        Écrire ("Le␣nombre␣:␣")
        Lire (n)

        -- Déterminer si n est premier
        Si n <= 3 Alors
            premier <- n > 0            -- 0 n'est pas premier
        Sinon
            premier <- ((n mod 2) <> 0) -- n Non divisible par 2
                    Et  ((n mod 3) <> 0)        -- n Non divisible par 3
            i <- 3
            TantQue premier Et (i < n Div i) Faire
                    -- n peut encore être premier
                    -- et il reste des diviseurs potentiels
                i <- i + 2
                premier <- (n mod i) <> 0       --  n Non divisible par i
            FinTQ
        FinSi

        -- afficher le résultat
        Si premier Alors
            ÉcrireLn ("OUI,␣", n, "␣est␣premier")
        Sinon
            ÉcrireLn ("NON,␣", n, "␣n'est␣pas␣premier")
        FinSi

        -- Demander à l'utilisateur si il veut continuer
        Écrire ("Encore␣un␣nombre␣(o/n)␣?␣")
        Lire (réponse)
    JusquÀ réponse = 'n'
Fin.
```

```java
/** Indique si un nombre est premier.
  * @author     Xavier Crégut
  * @version    1.1
  */
class EstPremier {

    public static boolean estPremierSimple(int n) {
```

```
        boolean premier = n >= 2;
        int diviseur = 2;
        while (premier && diviseur <= n / diviseur) {
            premier = n % diviseur != 0;
            diviseur++;
        }
        return premier;
    }

    public static boolean estPremierOptimise(int n) {
        boolean premier = (n == 2) || (n >= 3  &&  n % 2 != 0);
            // n vaut 2 ou est supérieur à 3 et non divisible par 2

        int diviseur = 3;
        while (premier && (diviseur <= n / diviseur)) {
                // n peut encore être premier
                // et il reste des diviseurs à essayer
            premier =  n % diviseur != 0;
            diviseur += 2;      // candidat suivant
        }
        return premier;
    }

    public static void main(String[] args) {
        assert ! estPremierSimple(0);
        assert ! estPremierSimple(1);
        assert estPremierSimple(2);
        assert estPremierSimple(3);
        assert ! estPremierSimple(4);
        assert estPremierSimple(5);
        assert ! estPremierSimple(6);
        assert estPremierSimple(7);
        assert ! estPremierSimple(8);
        assert ! estPremierSimple(9);
        assert ! estPremierSimple(10);
        assert estPremierSimple(11);
        assert ! estPremierSimple(12);
        assert estPremierSimple(13);
        assert estPremierSimple(97);

        assert ! estPremierOptimise(0);
        assert ! estPremierOptimise(1);
        assert estPremierOptimise(2);
        assert estPremierOptimise(3);
        assert ! estPremierOptimise(4);
        assert estPremierOptimise(5);
        assert ! estPremierOptimise(6);
        assert estPremierOptimise(7);
        assert ! estPremierOptimise(8);
        assert ! estPremierOptimise(9);
        assert ! estPremierOptimise(10);
        assert estPremierOptimise(11);
        assert ! estPremierOptimise(12);
        assert estPremierOptimise(13);
```

```
            assert estPremierOptimise(97);
        }

}
```

Exercice 5 : Racine carrée

Déterminer la racine carrée d'un réel en utilisant la suite ci-après qui converge vers $\sqrt{a}$:

$$\begin{cases} u_{n+1} = (u_n + a/u_n)/2 \\ u_0 = a \end{cases}$$

Solution : Voici un raffinage possible de ce problème :

```
R0 :  Calculer la racine carrée d'un entier saisi au clavier

R1 : Raffinage De « Calculer la racine carrée d'un entier saisi au clavier »
      | Saisir un réel a                          a: out Réel
      | Si a >= 0 Alors
      |   | Calculer la racine carrée de a        a: in Réel ; racine: out Réel
      |   | Afficher la racine carrée             racine: in Réel
      | Sinon
      |   | Signaler racine carrée inexistante dans IR
      | FinSi

R2 : Raffinage De « Calculer la racine carrée de a »
      | Initialiser la suite (U_n = a = U_0)      U_n: out Réel
      | Répéter
      |    | Conserver la valeur de U_n           U_n: in Réel ; précédent: out Réel
      |    | Calculer U_n <- U_{n+1}              U_n: in out Réel
      | JusquÀ précision atteinte                 précédent, U_n: in ; précision: out Boo
      | racine <- U_n                             U_n: in Réel ; racine: out Réel

R3 : Raffinage De « précision atteinte »
      | Résultat <- abs (U_n - Up) < EPSILON
```

La racine carrée est approximée en calculant le terme d'une suite. On peut montrer que cette suite converge effectivement vers la racine carrée cherchée. On dot donc calculer un certains nombre de termes de cette suite. On ne sait pas combien à priori. Notre condition d'arrêt consiste à comparer deux termes consécutifs de la suite et à s'arrêter lorsque leur distance est inférieure à un EPSILON donné. Il faut montrer ici aussi que ce critère est valide. En conséquence, on utilise un **Répéter** puisqu'on ne connait pas le nombre de répétition et que l'on doit déterminer la distance entre deux termes pour la condition d'arrêt.

On peut alors en déduire l'algorithme suivant :

```
Algorithme racine_carrée

    -- Calculer la racine carrée d'un entier saisi au clavier

Constantes
    EPSILON = 1.0e-10   -- précision souhaitée

Variables
```

```
    a: Réel               -- valeur saisie au clavier
    racine: Réel          -- la racine carrée de a
    précédent: Réel       -- élément u_{n-1}
    Un: Réel              -- élément u_n
    précision_atteinte: Booléen -- l'erreur sur U_n est-elle acceptable ?

Début
    -- Saisir un réel a
    Écrire ("Valeur réelle : ")
    Lire (a)

    -- Déterminer la racine carrée
    Si a >= 0 Alors
        -- Calculer la racine carrée de a
        U_n = a  -- Initialiser la suite (U_n = U_0 = a)
        Répéter
            -- Conserver la valeur de U_n
            précédent <- U_n

            -- Calculer la nouvelle valeur de U_n (U_{n+1})
            U_n = (U_n + a / U_n) / 2;

            -- Calculer la précision
            précision_atteinte <- abs (U_n - précédent) < EPSILON
        JusquÀ précision_atteinte
        racine <- U_n

        -- Afficher la racine carrée
        ÉcrireLn (racine)
    Sinon
        -- Signaler racine carrée inexistante dans IR
        ÉcrireLn ("Pas de racine carrée dans IR")
    FinSi
Fin.
```

```java
/** Calculer la racine carrée d'un nombre réel.
  * @author     Xavier Crégut
  * @version    1.1
  */
class RacineCarrée {
    final static double EPSILON = 1.0e-5;

    /** Obtenir la racine carrée d'un réel positif.
     * @param x le réel (>= 0)
     * @return la racine carrée de x
     */
    public static double racineCarrée(double x) {
        double resultat = 0;
        if (x != 0) {
            double un = x;      // un terme de la suite
                                // Initialisation : U_n = U_0
            double precedent = un;
            do {
                precedent = un; // terme précédent
```

```java
                un = (un + x / un) / 2;
            } while (Math.abs(un - precedent) >= EPSILON);
            resultat = un;
        }
        return resultat;
    }

    public static void main(String[] args) {
        assert Math.abs(racineCarrée(4) - 2) < EPSILON;
        assert Math.abs(racineCarrée(9) - 3) < EPSILON;
        assert Math.abs(racineCarrée(5) - Math.sqrt(5)) < EPSILON;
    }

}
```