

IN3R11-1: Sujet du TP3

ESIEE Engineering, Denis BUREAU, septembre 2010.

1 Les objectifs

Lire la version PDF de ce sujet et n'utiliser la version web que pour cliquer sur les liens.

Être capable de réaliser sur machine des programmes ne mettant pas en œuvre des instances. Comprendre l'importance des problèmes de débordement arithmétique et de précision. Savoir utiliser les tests unitaires dans BlueJ.

Attention ! Ce sujet peut paraître long, mais les 3 exercices sont faits pour être traités progressivement, ce qui explique les différentes versions, et de nombreux tests sont fournis pour permettre une auto-évaluation.

2 Factorielle et problèmes avec les entiers

Crer un nouveau projet factorielle et une nouvelle classe Factorielle sans attribut. Pour pouvoir tester facilement les méthodes suivantes, déclarez-les toutes en static, afin de ne pas être obligés de créer un objet avant de lancer une méthode.

1. fact1() :

Écrivez en Java (dans la classe `Factorielle`) la fonction factorielle non récursive, mais avec une amélioration : retournez systématiquement -1 si le paramètre est négatif.

Essayez 6; ça fait bien 720 ?

2. monTest() :

Écrivez en Java (dans la classe `Factorielle`) une procédure de test qui affiche la factorielle de tous les nombres compris entre ses 2 paramètres, sous la forme $3! = 6$ (si on veut la factorielle de 3)

Questions :

- **Essayez** l'intervalle $[-2, 6]$, puis essayez les bornes supérieures suivantes.
- Comment trouver le nombre maximum dont on peut calculer la factorielle avec cette fonction ?
- **Essayez** $[0, 17]$; pourquoi, simplement en voyant ce résultat, sait-on immédiatement que le maximum sera inférieur à 17 ?
- **Regardez 13** ; pourquoi, simplement en voyant ce résultat, sait-on immédiatement que le maximum sera inférieur à 13 ?

Ajoutez dans `fact1()` un test pour ne pas accepter un paramètre supérieur au maximum trouvé précédemment. Retournez -1 dans ce cas.

3. Incorporez à votre projet la classe `FactorielleTest` : voir premier encadré ci-dessous.

Compiliez; s'il y a des erreurs, corrigez vous code pour qu'il corresponde avec ce qu'attend la classe de test. **Run Tests** (bouton à gauche). Tout est vert ? Sinon, voir second encadré ci-dessous.

4. Comment intégrer une classe de test ?

- Cliquer dans la page web du sujet sur le lien du fichier de test, sélectionner tout, copier.
- Dans BlueJ, cliquer-droit sur la classe, choisir Create Test Class.
- Sélectionner tout, coller, compiler.
- Ne pas modifier la classe de test.

5. Dans **BlueJ**, en cas d'échec sur un test, cliquez sur la première ligne non "verte", interprétez le message d'erreur, puis cliquez sur le bouton *Show Source* pour savoir ce qui a été testé. Il peut également être utile d'utiliser le "Code Pad" (menu *Voir*) pour essayer interactivement des expressions Java.

6. fact2() :

Pour essayer d'aller plus loin que $12!$, nous allons utiliser un nouveau type primitif entier qui stocke ses nombres non plus sur 4 mais sur 8 octets (alors pourquoi pas le type `double` ?) : le mot java est `long`.

Dupliquez la méthode `fact1()` (dans la classe `Factorielle`) en la changeant de nom, et remplacez uniquement là où il le faut `int` par `long`. Les constantes littérales se terminent par L (*ex*: `-1L`).

Pensez-vous que `fact2()` pourra calculer jusqu'à 100, 1000, plus ?

Soyons raisonnables : ayant doublé le nombre d'octets, nous supposons dans un premier temps que le maximum ne peut pas plus que doubler. **Modifiez-le** dans `fact2()`.

7. `monTest()` :

Modifiez cette procédure pour qu'elle appelle `fact2()` au lieu de `fact1()`.

Questions :

- **Essayez** l'intervalle $[-2, 6]$, puis essayez les bornes supérieures suivantes.
- Mais quel est réellement le nombre maximum dont on peut calculer la factorielle avec cette fonction ?
- **Essayez** 24; puis regardez 23, puis ... (posez-vous les mêmes questions qu'au 2. ci-dessus)
- **Réajustez** le maximum dans `fact2()` en fonction des tests précédents.

8. Dans la classe `FactorielleTest`, décommentez la 2^{ème} méthode, recompilez, et exécutez les tests.

Tout est vert ? Passez au **3**.

9. A NE PAS COMMENCER EN TP : A TERMINER EN TRAVAIL PERSONNEL `fact3()` :

Pour essayer d'aller plus loin que $20!$, nous allons utiliser un nouveau type entier (objet) qui stocke ses nombres non plus sur 4 ou sur 8 octets, mais dans une `String` de longueur pratiquement illimitée : le nom de cette classe est `BigInteger`.

Dupliquez la méthode `fact2()` (dans la classe `Factorielle`) en la changeant de nom, et **remplacez** les `long` par des `BigInteger`.

Il va falloir faire un certain nombre de modifications car nous remplaçons un type primitif par un type objet :

- repérez le paquetage de `BigInteger` (voir javadoc); cela servira pour la compilation
- utilisez le `BigInteger` de valeur 1 (voir javadoc)
- utilisez la bonne méthode pour prendre l'opposé (voir javadoc)
- utilisez la bonne méthode pour convertir un `int` en `BigInteger` (voir javadoc, sachant qu'un `int` est automatiquement converti en `long` si nécessaire)
- utilisez la bonne méthode pour multiplier (voir javadoc)

Pensez-vous que `fact3()` pourra calculer jusqu'à 100, 1000, plus ?

Soyons déraisonnables : **supprimons** la borne supérieure du test dans `fact3()` et découvrons des factorielles rarement calculées.

10. `monTest()` :

Modifiez cette procédure pour qu'elle appelle `fact3()` au lieu de `fact2()`.

Questions :

- **Essayez** l'intervalle $[-2, 22]$. Les résultats doivent être identiques à ceux de `fact2()` jusqu'à 20. Ensuite, nous découvrons les vraies valeurs de $21!$ et $22!$.
- Maintenant, **modifiez** la boucle pour qu'elle progresse de 100 en 100 sur l'intervalle $[100, 1000]$.
- Maintenant, **modifiez** la boucle pour qu'elle progresse de 1000 en 1000 sur l'intervalle $[1000, 10000]$.
- Maintenant, **modifiez** la boucle pour qu'elle progresse de 10000 en 10000 sur l'intervalle $[10000, 40000]$, mais les nombres devenant trop grands, **affichez** le nombre de chiffres du résultat plutôt que le résultat de la factorielle (voir javadoc de `String`).

11. Dans la classe `FactorielleTest`, décommentez la 3^{ème} méthode, recompilez, et exécutez les tests.

Tout est vert ?

3 Racine carrée et problèmes avec les réels

(D'après un exercice de Jean-Claude GEORGES)

1. `racine1()` :

Écrivez en Java (dans la classe `RacineNewton`) une fonction qui calcule la racine carrée d'un nombre positif (**sans utiliser** `Math.sqrt()`) selon l'algorithme de Newton, c'est-à-dire : si on suppose que r est une valeur approchée (même mauvaise) de \sqrt{x} , alors Newton a montré que $\frac{1}{2}(r + \frac{x}{r})$ est une meilleure valeur approchée. On peut donc en théorie se rapprocher aussi près que l'on veut de la valeur "exacte" de la racine carrée ; essayons de le vérifier à ϵ près ... (on définira ϵ en CONSTANCE égale à 10^{-6} par exemple).

On arrêtera donc la boucle lorsque l'écart entre r^2 et x sera inférieur à ϵ ; pour cela, on définira une fonction booléenne `environ1()` qui comparera ses 2 paramètres réels x et y à ϵ près ($|x - y| < \epsilon$?).

Toute valeur initiale est possible pour r , par exemple $\frac{x}{2}$ ou 1 ou ...

2. Testez votre fonction avec au moins les valeurs suivantes :
 - 6.25 : ça fait bien 2.500000 ?
 - 1E-6 : ça fait bien 0.001000 ? Pourquoi ?
 - 1E-20 : ça fait bien 1E-10 ? Pourquoi ?
 - 1E11 : ça fait bien 316227.766017 ? Pourquoi ? (*aide: bouton droit sur la barre rouge et blanche*)
3. `racine2()` :

Compte tenu des problèmes précédents, écrivez une nouvelle fonction qui appellera une nouvelle fonction booléenne `environ2()` qui tiendra compte de l'ordre de grandeur de x ou y pour comparer ses 2 paramètres réels x et y à ϵ près ($|\frac{x-y}{x}| < \epsilon$? si x n'est pas trop proche de 0).

On considérera que x est assimilable à 0 si $|x| < 10^{-300}$; dans ce cas, $\sqrt{x} = 0.0$.
4. **Retestez au moins les 4 valeurs** ci-dessus.

4 Projet "Lines" (A COMMENCER EN TP ET A TERMINER EN TRAVAIL PERSONNEL)

Cet exercice va consister à créer une nouvelle classe, indépendante, permettant de représenter une droite dans le plan, puis à comparer 2 droites.

Rappels mathématiques:

- 2 droites sont orthogonales si le produit scalaire des vecteurs directeurs est nul ($aa' + bb' = 0$)
- 2 droites sont colinéaires si elles ont le même coefficient directeur ($-\frac{b}{a} = -\frac{b'}{a'}$ si $a \neq 0$ et $a' \neq 0$)
- 2 droites peuvent être confondues sans avoir nécessairement $a = a'$ et $b = b'$ et $c = c'$
- $\frac{x_1}{y_1} = \frac{x_2}{y_2} \iff x_1y_2 = x_2y_1$

4.1 Créer un nouveau projet

Créer dans le répertoire `tp3`, préalablement créé, un nouveau projet BlueJ de nom Lines.

4.2 Créer une nouvelle classe

Définir une classe `Line`, en la documentant au fur et à mesure, répondant aux spécifications suivantes :

- finalité : représenter une droite du plan
- attributs : trois réels a , b et c définissant les coefficients de la droite $ax + by + c = 0$
- constructeur : un constructeur pour définir une droite à partir de ses coefficients a , b et c
- méthodes :
 - 1) une fonction d'accès pour chaque attribut (`getA()` etc...)
 - 2) une fonction `toString()` qui renverra une chaîne de caractères de la forme: "`(1.0x +0.5y -3.0 = 0)`", si $a=1$, $b=0.5$, et $c=-3$.

Aide: construire la chaîne étape par étape en incorporant les tests nécessaires.

4.2.1 Version 1

- - 3) une fonction `estSOL()` déterminant si la droite courante est **S**écante non orthogonale, **O**rthogonale, ou co-**L**inéaire (c'est-à-dire confondue ou parallèle) par rapport à une droite donnée en paramètre et retournant la chaîne de 1 caractère "S", "O", ou "L".

On supposera que les coefficients stockés dans chacune des 2 droites définissent toujours une équation de droite, qui de plus n'est ni horizontale ni verticale.

Contrainte: un seul return!

Tester cette version. Pour cela, il suffit d'intégrer la classe `LineTest` au projet, et d'"exécuter les tests".

Tout est vert? (*interdiction de modifier LineTest*)

Exemples:

- les droites $[1., 2., -3.]$ et $[3., 6., -9.]$ sont co-linéaires ; retourner "L"
- les droites $[1., 2., -3.]$ et $[3., 6., -3.]$ sont co-linéaires ; retourner "L"
- les droites $[1., -1., 1.]$ et $[2., 2., -10.]$ sont orthogonales ; retourner "O"
- les droites $[1., -2., -2.]$ et $[1., 2., -6.]$ sont sécantes non orthogonales; retourner "S"

4.2.2 Version 2

- – 4) On souhaite maintenant qu'au lieu de répondre "L", la fonction réponde "C" quand les droites sont confondues et "P" lorsqu'elles sont strictement parallèles. Pour cela, **modifier** la fonction ci-dessus et la **nommer désormais estSOPC()**. On suppose encore que les coefficients stockés dans chacune des 2 droites définissent toujours une équation de droite, de plus ni horizontale ni verticale. **Tester cette version.** Pour cela, il suffit de copier/coller le contenu du fichier `LineTest2` en écrasant la classe `LineTest` de la version 1, et d'"exécuter les tests".

Tout est vert? (*interdiction de modifier LineTest*)

Exemples:

- les droites [1., 2., -3.] et [3., 6., -9.] sont confondues ; retourner "C"
- les droites [1., 2., -3.] et [3., 6., -3.] sont strictement parallèles ; retourner "P"

4.2.3 Version 3

- – On souhaite maintenant que la fonction marche aussi pour des droites horizontales ou verticales. On suppose encore que les coefficients stockés dans chacune des 2 droites définissent toujours une équation de droite. **Ajouter** ce qu'il faut dans `estSOPC()`. **Tester cette version.** Pour cela, il suffit de copier/coller le contenu du fichier `LineTest3` en écrasant la classe `LineTest` de la version 2, et d'"exécuter les tests".

Tout est vert? (*interdiction de modifier LineTest*)

Exemples:

- les droites [1., 0., -3.] et [3., 0., -9.] sont confondues ; retourner "C"
- les droites [0., 2., -3.] et [0., 9., -5.] sont strictement parallèles ; retourner "P"
- les droites [2., 0., 3.] et [0., 9., 5.] sont orthogonales ; retourner "O"

4.2.4 Version 4

- – On veut aussi que la fonction réponde "?" lorsque les coefficients ne définissent pas 2 équations de droites. **Ajouter** les tests nécessaires.

Tester cette version. Pour cela, il suffit de copier/coller le contenu du fichier `LineTest4` en écrasant la classe `LineTest` de la version 3, et d'"exécuter les tests".

Tout est vert? (*interdiction de modifier LineTest*)

Exemples:

- les "droites" [0., 0., 1.] et [1., 2., -3.]: la première n'en est pas une ; retourner "?"
- les "droites" [1., 2., -3.] et [0., 0., 0.]: la seconde n'en est pas une ; retourner "?"
- les droites [1., 0., -3.] et [3., 0., -9.] sont confondues ; retourner "C"
- les droites [0., 2., -3.] et [0., 9., -5.] sont strictement parallèles ; retourner "P"
- les droites [2., 2., 0.] et [1., -1., 1.] sont orthogonales ; retourner "O"
- les droites [0., 1., -2.] et [1., 2., -6.] sont sécantes non orthogonales; retourner "S"

4.2.5 Version 5

- – On veut enfin que les comparaisons soient effectuées à 10^{-6} près \Rightarrow **définir** une constante `EPSILON`, et même une fonction `environ()` qui compare 2 réels à `EPSILON` près).

Tester cette version. Pour cela, il suffit de copier/coller le contenu du fichier `LineTest5` en écrasant la classe `LineTest` de la version 4, et d'"exécuter les tests".

Tout est vert? (*interdiction de modifier LineTest*)

Exemples:

- les droites [1.23456789, 9.87654321, -8.] et [9.8765432, -1.2345678, -1.] sont orthogonales; retourner "O"

4.2.6 Version graphique optionnelle

- – On souhaite pouvoir visualiser graphiquement les droites et leur statut (SOPC). Pour cela on utilisera la classe graphique: `LineDrawer` Lire le mode d'emploi tout au début du fichier.

1. **intégrer** la classe précédente dans le projet Lines.
2. **compiler.** En cas de erreur, modifier votre classe `Line`, pas `LineDrawer`!
3. **tester** plusieurs exemples en créant des lignes, en appelant la méthode `draw2Lines()`, en cliquant sur les 2 objets `Line` à afficher, et **en fermant la fenêtre graphique entre chaque appel.**