IN101: Sujet du TP3

ESIEE Engineering, Denis BUREAU, mars 2012 v2.

1 Les objectifs

Être capable de réaliser sur machine des programmes ne mettant pas en oeuvre des instances. Comprendre l'importance des problèmes de débordement arithmétique et de précision. Savoir utiliser les tests unitaires dans BlueJ. **Lire la version PDF de ce sujet et n'utiliser la version web que pour cliquer sur les liens. Attention !** Ce sujet est 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 un travail personnel avec une auto-évaluation. Donc, aucune raison de ne pas terminer seul ce TP.

2 Factorielle et problèmes avec les entiers

Charger le projet `sujtp3.jar` et écrire les méthodes suivantes dans la classe Factorielle. 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, et respectez les noms indiqués.

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. (Rappel: 0! = 1)
   Essayez 6; ça fait bien 720 ?

2. Dans la classe `FactorielleTest`, **décommentez** *(menu Edit, Uncomment)* toutes les lignes de la méthode `testAfact1()` **sauf la dernière ligne commentée**.
   Compilez; s'il y a des erreurs, corrigez votre code pour qu'il corresponde à ce qu'attend la classe de test. Si le bouton à gauche **Run Tests** n'apparaît pas, menu *Tools/Preferences/Miscellaneous/Show unit testing tools*). Lancer `testAfact1()`. Tout est vert ? Sinon, voir l'encadré ci-dessous.

3. `monTest1()` :
   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` *(par exemple, pour la factorielle de 3)*

   Questions :
   - **Essayez** l'intervalle [-2, 6], puis essayez les bornes supérieures suivantes.
   - Voyons 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. Décommentez le dernier test de `testAfact1()`. Retestez.

4. Dans la classe `FactorielleTest`, **décommentez** la ligne de la méthode `testBmonTest1()`.
   Compilez; s'il y a des erreurs, corrigez votre code pour qu'il corresponde à ce qu'attend la classe de test. Lancez `testBmonTest1()`. Tout est vert ? Sinon, voir l'encadré ci-dessous.

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. Activez également l'affichage des numéros de ligne *(Tools/Preferences/Editor/Display line numbers)*. Il peut aussi être utile d'utiliser le "Code Pad" (menu *View*) 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 supposerons dans un premier temps que le maximum ne peut pas plus que doubler. **Modifiez-le** dans `fact2()`.

7. Dans la classe `FactorielleTest`, **décommentez** toutes les lignes de la méthode `testCfact2()`.
Compilez; s'il y a des erreurs, corrigez votre code pour qu'il corresponde avec ce qu'attend la classe de test. **Run Tests** : Tout est vert ?

8. `monTest2()` :
Recopiez la procédure `monTest1()` et **modifiez**-la 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.

9. Dans la classe `FactorielleTest`, **décommentez** la ligne de la méthode `testDmonTest2()`.
Compilez, Testez. Tout est vert ? Passez directement au **3**.

10. 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.

11. Dans la classe `FactorielleTest`, **décommentez** toutes les lignes de la méthode `testEfact3()`.
Compilez, Testez. Tout est vert ?

12. `monTest3()` :
Recopiez la procédure `monTest2()` et **modifiez**-la pour qu'elle appelle `monTest3()` au lieu de `monTest2()`.
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`).

13. Dans la classe `FactorielleTest`, **décommentez** la ligne de la méthode `testFmonTest3()`.
Compilez, Testez. Tout est vert ?

3 Racine carrée et problèmes avec les réels

(D'après un exercice de Jean-Claude GEORGES) **Affichez le sujet en PDF (version imprimable)**

Écrivez les méthodes suivantes dans la classe `RacineNewton` :

1. `environ1()` :
   Pour pouvoir comparer des nombres réels, comme nous ne pouvons pas utiliser l'opérateur `==`, on définira une fonction booléenne `environ1()` qui comparera ses 2 paramètres réels x et y à ϵ près ($|x-y| < \epsilon$?) et on utilisera `environ1(`x, y`)` au lieu de x `==` y.
   On définira ϵ en CONSTANTE de classe (qu'on choisira égale à 10^{-6}).
2. Dans la classe `RacineTest`, **décommentez** toutes les lignes de la méthode `testAenviron1()`.
   Compilez, Testez. Tout est vert ?
3. `racine1()` :
   Cette fonction doit calculer 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 très mauvaise) de $\sqrt{a}$, alors Newton a montré que $\frac{1}{2}(r + \frac{a}{r})$ 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 arrêtera donc la boucle lorsque l'écart entre r^2 et a sera inférieur à ϵ.
   AIDE : Toute valeur initiale est possible pour r, par exemple $\frac{a}{2}$ ou 1 ou ...
4. Dans la classe `RacineTest`, **décommentez** toutes les lignes de la méthode `testBracine1()`.
   Compilez, Testez. Tout est vert ?
   Testez encore votre fonction avec au moins les valeurs suivantes :
   1E-6 : ça fait bien 0.001000 ? Pourquoi ?
   1E-20 : ça fait bien 1E-10 ? Pourquoi ?
   1E11 : ça fait bien 316227.766017 ? Pourquoi ? *(oscillations ?)*
5. `environ2()` :
   Compte tenu des problèmes précédents, écrivez une nouvelle fonction booléenne qui comparera ses 2 paramètres réels x et y en tenant compte de l'ordre de grandeur de x ou y ($|\frac{x-y}{x}| < \epsilon$?) Cette formule n'est correcte que si x n'est pas trop proche de 0). Dans le cas contraire, il faut donc diviser par y. Si les deux sont très proches de 0, on peut directement retourner vrai.
   On considérera que x est assimilable à 0 si $|x| < 10^{-300}$. **Pour cela**, définir cette valeur particulière en CONSTANTE de classe et définir une fonction booléenne `zero()` qui retournera vrai si son paramètre réel est assimilable à 0.
6. Dans la classe `RacineTest`, **décommentez** toutes les lignes de la méthode `testCenviron2()`.
   Compilez, Testez. Tout est vert ?
7. `racine2()` :
   Recopiez la fonction `racine1()` en changeant son nom et **modifiez**-la pour qu'elle appelle `environ2()` au lieu de `environ1()`. D'autre part, si x est assimilable à 0, $\sqrt{x} = 0.0$.
8. Dans la classe `RacineTest`, **décommentez** toutes les lignes de la méthode `testDracine2()`.
   Compilez, Testez. Tout est vert ?
   Est-ce correct pour les 3 valeurs problématiques du 3.4 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 (d'équations $a_1x + b_1y + c_1 = 0$ et $a_2x + b_2y + c_2 = 0$).

Rappels mathématiques:
- 2 droites sont orthogonales si le produit scalaire des vecteurs directeurs est nul ($a_1a_2 + b_1b_2 = 0$)
- 2 droites sont colinéaires si elles ont le même coefficient directeur ($-\frac{b_1}{a_1} = -\frac{b_2}{a_2}$ si $a_1 \neq 0$ et $a_2 \neq 0$) ou mieux, si le déterminant principal est nul.
- 2 droites peuvent être confondues sans avoir nécessairement $a_1 = a_2$ et $b_1 = b_2$ et $c_1 = c_2$, il suffit que les deux déterminants secondaires soient nuls.
- $x_1y_2 = x_2y_1$ remplace avantageusement $\frac{x_1}{y_1} = \frac{x_2}{y_2}$

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"`

- 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.

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"`

- Cette version résoud déjà les exemples de la version 3 si elle repose sur le test du déterminant principal.

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 demploi tout au début du fichier.
  1. **intégrer** la classe précédente dans le projet Lines.
  2. **compiler**. En cas derreur, 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**.