U.B.S. - I.U.T. de Vannes
Département Informatique

TP1
R2.01
Développement Orienté Objets
BUT-Info-1A

Fleurquin R.
Kamp J-F, Tibermacine C.

Janvier 2025

On considère les classes Java `Polynome.java` et `TestPolynome.java` dont vous devez récupérer le code sur Moodle.

Question 1 : compiler

- Compilez dans le bon ordre ces deux classes.
- Générez la javadoc de ces deux classes.
- Ouvrez le fichier d'extension `Polynome.class` avec un éditeur quelconque. Affreux ? Utilisez ensuite la commande `javap` du JDK (avec puis sans l'option `-c` pour « regarder » ce bytecode)

Question 2 : comprendre la structure d'une classe Java

- Observez et comprenez le code de la classe `Polynome` : commentaires, documentation javadoc, attributs, méthodes.
- Indiquez la nature de chaque méthode (constructeur, accesseur/getter, modificateur/setter, autre)

Question 3 : exécuter et comprendre la nature à l'exécution d'un POO

- Exécutez la classe `TestPolynome`.
- Modifiez le code du `main` pour faire apparaitre le contenu des variables locales `p1`, `p2` et `p3`. Que constatez-vous ?
- Avec une trame de diagramme de séquence décrivant l'exécution du `main` initial, identifiez tous les objets en présence et leurs interactions. On ne rentrera pas en particulier dans le détail des codes des méthodes `add` et `isIdentical` mais on « notera » les interactions vers les différents objets qu'elles impliquent.

Question 4 : instancier un objet et faire collaborer des objets

Modifiez le `main` pour obtenir le comportement qui suit.

- Calculez le résultat de la somme du polynôme dont la référence est dans la variable `p3` avec un polynôme pointé par une variable `p4` : $3x^3+2x^2-x-3$.
- Déterminez le degré du nouveau polynôme obtenu via cette addition.
- Modifiez à la valeur -3 le coefficient devant le monôme en x^2 du nouveau polynôme.
- Affichez ce nouveau polynôme sous la forme d'une chaine de caractères.
- Calculez la valeur de ce nouveau polynôme en x=2.
- Comparez ce nouveau polynôme avec `p3`.

Question 5 : réaliser ses premiers pas en test unitaire et en programmation défensive

Le code de la classe `Polynome` bien qu'en apparence « correct » pose en réalité de nombreux problèmes. Nous allons en lister quelques-uns.

Dans cette démarche nous allons commencer à structurer la classe de test selon une approche qui sera reprise et facilitée lors de l'industrialisation des **tests unitaire**s par exemple avec le framework `Junit`.

Développez dorénavant la classe `TestPolynome` pour que le main ne contienne que des appels vers des méthodes `static` qui vont chacune tester unitairement chaque méthode de la classe `Polynome` : par exemple une méthode `testConstructeur()`. Un test unitaire de méthode comportant des paramètres doit tester un cas nominal (les paramètres sont dans leur domaine de valeurs attendus), les cas limites (le paramètres prennent les valeurs en adhérence de domaine), les cas hors limites (les paramètres sortent de la plage des valeurs attendus).

Certaines méthodes ne respectent pas le principe de la **programmation défensive** : « *vérifier systématiquement la validité des paramètres passés, signaler explicitement et clairement à l'appelant les situations anormales et*

assurer si nécessaire un fonctionnement malgré tout correct, même dégradé, de la méthode ». Cette approche vise à rendre le code plus robuste, plus lisible et plus sûr, en prévenant les bugs potentiels et en facilitant le diagnostic des problèmes.

- Testez la méthode constructeur dans la cas ou l'on passe un polynôme non nul (cas nominal), le polynôme nul (cas limite) et aucun polynôme (valeur `null`, cas hors limite). Que constatez-vous ? Corrigez ce problème en acceptant un comportement par défaut assurant une initialisation acceptable garantissant un fonctionnement de l'objet avec les autres méthodes.
- Montrez qu'une instance de la classe Polynôme n'est pas maître de ses propres coefficients et qu'un client peut modifier à loisir ses coefficients sans passer pour autant par le setter dédié. Corrigez ce problème dans le constructeur en appliquant le principe de la **copie défensive** : « *Un objet doit être le seul maître de la modification de son état.* N*e jamais laisser la possibilité à un autre objet de modifier son état interne sans passer par les méthodes ad hoc qu'il propose. Se protéger systématiquement en réalisant en entrée et sortie de chaque méthode des copies défensives de tous les types mutables dont on est propriétaire* ».
- Montrez en testant le getter que l'approche de retour -1 par défaut en cas de non respect de la plage d'indice montre ici les limites du principe « affichage et retour d'une valeur d'erreur ». Vous apprendrez plus tard à gérer proprement ces mauvais usages à l'aide du concept d'**exception**. Ici, nous tolérerons cet inconvénient. Pourquoi ce problème est-il facilement géré dans le setter ?
- Montrez en la testant que la méthode `add` ne respecte pas le principe de programmation défensive. Corrigez son code.
- Montrez en la testant que la méthode `isIdentical()` non seulement ne respecte pas le principe de programmation défensive mais en plus se trompe parfois (saurez vous trouver quand ?!). Corrigez ces deux erreurs.

Question 6 (bonus algorithmique pour les plus rapides)

Ajoutez une méthode `public Polynome multiply(Polynome)` qui permet la multiplication de deux polynômes quelconques en en créant un troisième et testez unitairement cette nouvelle méthode.