ENSA de Kenitra
Module : Programmation Java
2020-2021
Série 2 : Héritage & Exception

Exercice 1 :

Partie A :

On dispose de la classe suivante :

```
class Point
{ public void setPoint (int x, int y)  { this.x = x ; this.y = y ; }
  public void deplace (int dx, int dy) { x += dx ; y += dy ; }
  public void affCoord ()
  { System.out.println ("Coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
```

Réaliser une classe *PointNom*, dérivée de *Point* permettant de manipuler des points défini par deux coordonnées (*int*) et un nom (caractère). On y prévoira les méthodes suivantes :

- *setPointNom* pour définir les coordonnées et le nom d'un objet de type *PointNom*,
- *setNom* pour définir seulement le nom d'un tel objet,
- *affCoordNom* pour afficher les coordonnées et le nom d'un objet de type *PointNom*.

Écrire un petit programme utilisant la classe *PointNom*.

Partie B :

On dispose de la classe suivante (disposant cette fois d'un constructeur) :

```
class Point
{ public Point (int x, int y)  { this.x = x ; this.y = y ; }
  public void affCoord()
  { System.out.println ("Coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
```

Réaliser une classe *PointNom*, dérivée de *Point* permettant de manipuler des points définis par leurs coordonnées (entières) et un nom (caractère). On y prévoira les méthodes suivantes :

- constructeur pour définir les coordonnées et le nom d'un objet de type *PointNom*,
- *affCoordNom* pour afficher les coordonnées et le nom d'un objet de type *PointNom*.

Écrire un petit programme utilisant la classe *PointNom*.

Partie C :

On dispose de la classe suivante :

```
class Point
{ public Point (double x, double y) { this.x=x ; this.y=y ; }
  public void deplace (double dx, double dy) { x+=dx ; y+=dy ; }
  public void affiche ()
  { System.out.println ("Point de coordonnees " + x + " " + y) ;
  }
  public double getX() { return x ; }
  public double getY() { return y ; }
  private double x, y ;
}
```

On souhaite réaliser une classe *Cercle* disposant des méthodes suivantes :

- constructeur recevant en argument les coordonnées du centre du cercle et son rayon,
- *deplaceCentre* pour modifier les coordonnées du centre du cercle,
- *changeRayon* pour modifier le rayon du cercle,
- *getCentre* qui fournit en résultat un objet de type *Point* correspondant au centre du cercle,
- *affiche* qui affiche les coordonnées du centre du cercle et son rayon.

1. Définir la classe *Cercle* comme classe dérivée de *Point*.
2. Définir la classe *Cercle* comme possédant un membre de type *Point*.

Dans les deux cas, on écrira un petit programme mettant en jeu les différentes fonctionnalités de la classe *Cercle*.

Exercice 2 :

Partie A :

On souhaite disposer d'une hiérarchie de classes permettant de manipuler des figures géométriques. On veut qu'il soit toujours possible d'étendre la hiérarchie en dérivant de nouvelles classes mais on souhaite pouvoir imposer que ces dernières disposent toujours des méthodes suivantes :

- *void affiche ()*
- *void homothetie (double coeff)*
- *void rotation (double angle)*

Écrire la classe abstraite *Figure* qui pourra servir de classe de base à toutes ces classes.

Partie B :

Compléter la classe abstraite *Figure* de l'exercice précédent, de façon qu'elle implémente :

- une méthode *homoRot (double coef, double angle)* qui applique à la fois une homothétie et une rotation à la figure,
- de méthodes statiques *afficheFigures*, *homothetieFigures* et *rotationFigures* appliquant une même opération (affichage, homothétie ou rotation) à un tableau de figures (objets d'une classe dérivée de *Figure*).

Partie C :

On souhaite disposer de classes permettant de manipuler des figures géométriques. On souhaite pouvoir caractériser celles qui possèdent certaines fonctionnalités en leur demandant d'implémenter des interfaces, à savoir :

- *Affichable* pour celles qui disposeront d'une méthode *void affiche ()*,
- *Tranformable* pour celles qui disposeront des deux méthodes suivantes :

  void homothetie (double coeff)

  void rotation (double angle)

Écrire les deux interfaces *Affichable* et *Transformable*.

Exercice 3 :

Partie A :

Réaliser une classe *EntNat* permettant de manipuler des entiers naturels (positifs ou nuls). Pour l'instant, cette classe disposera simplement :

- d'un constructeur à un argument de type *int* qui générera une exception de type *ErrConst* (type classe à définir) lorsque la valeur reçue ne conviendra pas,
- d'une méthode *getN* fournissant sous forme d'un *int*, la valeur encapsulée dans un objet de type *EntNat*.

Écrire un petit programme d'utilisation qui traite l'exception *ErrConst* en affichant un message et en interrompant l'exécution.

Partie B :

Adapter la classe *EntNat* de l'exercice et le programme d'utilisation de manière à disposer dans le gestionnaire d'exception du type *ErrConst* de la valeur fournie à tort au constructeur.

Exercice 4 :

Soit la classe Etudiant suivante :

```
package etudiant;
public class Etudiant {
}
```

1) Complétez le code de cette classe en lui ajoutant :

•Trois attributs privés de type String nommés nom, prenom et adresse

•Une variable statique pour compter le nombre d'objet Etudiant créé

•Un constructeur servant à initialiser les attributs

•Les mutateurs et les accesseurs

•Une méthode publique nommée toString qui retourne les informations d'un étudiant

•Une méthode publique nommée modifAdresse qui permet de modifier l'adresse

2) Créer une classe TestEtudiant (aussi dans le paquetage etudiant) contenant une méthode main() qui :
•Crée un étudiant (instance de la classe Etudiant) en lui donnant un nom, prenom et adresse

•Invoque la méthode toString de l'étudiant créé ainsi que le nombre d'objet créé.

•Invoque la méthode modifAdresse de l'étudiant créé.

•Modifié l'adresse de l'étudiant crée puis invoque la méthode toString
3) Créer une classe personne dont va hériter la classe étudiant, en lui rattachant les attributs : Nom, prénom, adresse
Puis pour la classe étudiant, on ajoute le code CNE, entier.
Effectuer les modifications nécessaires

4) ajouter les exceptions lors de la saisie des informations de l'étudiant (nom, prénom,, code CNE)

Exercice 5 :

Supposons que nous souhaitions créer une application qui permet de manipuler différents types de comptes bancaires : les comptes simples, les comptes épargnes et les comptes payants.
Tous les types de comptes sont caractériser par :
•Un code et un solde

•Lors de la création d'un compte, son code est défini automatiquement en fonction du nombre de comptes créés (utiliser une variable statique);

Un compte peut subir les opérations de versement et de retrait. Pour ces deux opérations, il faut connaître le montant de l'opération.
Pour consulter un compte on peut faire appel à sa méthode toString() qui permet d'afficher les informations sur le compte.

- Un Compte Simple est un compte qui possède un découvert. Ce qui signifie que ce compte peut être débiteur jusqu'à la valeur du découvert.
- Un Compte Epargne est un compte bancaire qui possède en plus un champ «tauxInterêt» et une méthode calculIntérêt() qui permet de mettre à jour le solde en tenant compte des intérêts.
- Un Compte Payant est un compte bancaire pour lequel chaque opération de retrait et de versement est payante et vaut 5 % du montant de l'opération.

1) Créer la classe mère Compte (de type abstract) et les déférentes classes dérivées que nous considérons de type final

2) Prévoir les exceptions sur les méthodes de lecture et les méthodes de calcul.

2)Créer une application pour tester les différentes classes.