Saeed bark saeed bin gawhar

C16430

M1 SI

```
public class Rectangle extends Forme {
private int largeur ; private int longueur ;
public Rectangle(int x, int y) { this.largeur = x ; this.longueur = y ;
}
public int getLargeur() { return this.largeur ;
}
public int getLongueur() { return this.longueur ;
}
public int surface() {
return this.longueur * this.largeur ; }
public void affiche() {
System.out.println(”rectangle ” + longueur + ”x” + largeur); }
}
```

Ce code est écrit en Java et définit une classe appelée `Rectangle` qui hérite d'une classe appelée `Forme`. La classe `Rectangle` a deux variables membres privées, `largeur` et `longueur`, qui représentent les dimensions du rectangle.

Le constructeur de la classe prend deux paramètres, `x` et `y`, qui sont utilisés pour initialiser les variables `largeur` et `longueur` du rectangle. Il y a également plusieurs méthodes dans la classe :

1. `getLargeur()`: Cette méthode renvoie la valeur de la variable `largeur`.

2. `getLongueur()`: Cette méthode renvoie la valeur de la variable `longueur`.

3. `surface()`: Cette méthode calcule et renvoie la surface du rectangle en multipliant la `longueur` par la `largeur`.

4. `affiche()`: Cette méthode affiche une représentation textuelle du rectangle en utilisant la méthode `println` de la classe `System.out`. La chaîne affichée inclut la longueur et la largeur du rectangle.

Notez que le code semble contenir une erreur de syntaxe dans la méthode `affiche()`. La chaîne de caractères à afficher doit être encadrée par des guillemets droits (") au lieu des guillemets français (”).

En résumé, cette classe `Rectangle` représente un rectangle en termes de longueur et de largeur, avec des méthodes pour obtenir ces dimensions, calculer la surface du rectangle et afficher une représentation textuelle du rectangle.

```
public class Carre extends Rectangle {
public Carre(int cote) { super(cote, cote);
}
public void affiche() {
System.out.println(”carré ” + this.getLongueur()); }
}
```

Votre classe `Carre` semble bien définie. Elle étend la classe `Rectangle` et a un constructeur spécifique pour les carrés, qui utilise le mot-clé `super` pour appeler le constructeur de la classe `Rectangle` avec les deux côtés égaux.

La méthode `affiche` est également redéfinie pour afficher "carré" suivi de la longueur du côté du carré. Cependant, il y a une petite erreur de syntaxe dans la chaîne de la méthode `affiche`. Les guillemets utilisés autour du mot "carré" sont des guillemets français, cela pourrait provoquer une erreur. Il serait préférable d'utiliser des guillemets droits (") pour corriger cela.

```
Forme[] tableau = new Forme[4];
 tableau[0] = new Rectangle(10,20);
 tableau[1] = new Cercle(15);
 tableau[2] = new Rectangle(5,30);
 tableau[3] = new Carre(10);
```

```
for (int i = 0 ; i < tableau.length ; i++) {
if (tableau[i] instanceof Forme)
System.out.println(”element ” + i + ” est une forme”);
if (tableau[i] instanceof Cercle)
System.out.println(”element ” + i + ” est un cercle”);
if (tableau[i] instanceof Rectangle)
System.out.println(”element ” + i + ” est un rectangle”);
if (tableau[i] instanceof Carre)
System.out.println(”element ” + i + ” est un carré”); }
```

Ce code crée un tableau `tableau` de type `Forme` et y stocke des instances de différentes classes dérivées de `Forme`, telles que `Rectangle`, `Cercle`, et `Carre`. Ensuite, il parcourt le tableau à l'aide d'une boucle for, utilise l'opérateur `instanceof` pour vérifier le type de chaque élément, et affiche un message approprié en fonction du type de la forme.

Des instances de `Rectangle`, `Cercle`, et `Carre` sont créées et stockées dans le tableau.


```
for (int i = 0 ; i < tableau.length ; i++) {
if (tableau[i] instanceof Forme)
System.out.println(”element ” + i + ” est une forme”);
if (tableau[i] instanceof Cercle)
System.out.println(”element ” + i + ” est un cercle”);
if (tableau[i] instanceof Rectangle)
System.out.println(”element ” + i + ” est un rectangle”);
if (tableau[i] instanceof Carre)
System.out.println(”element ” + i + ” est un carré”); }
```


La boucle `for` parcourt chaque élément du tableau. Les blocs `if` utilisent l'opérateur `instanceof` pour vérifier le type de chaque élément, et en fonction du type, un message approprié est affiché.

- Si l'élément est une instance de `Forme`, le message "element i est une forme" est affiché.
- Si l'élément est une instance de `Cercle`, le message "element i est un cercle" est affiché.
- Si l'élément est une instance de `Rectangle`, le message "element i est un rectangle" est affiché.

- Si l'élément est une instance de `Carre`, le message "element i est un carré" est affiché.

L'utilisation de l'opérateur `instanceof` permet de déterminer le type réel de chaque élément dans le tableau, même s'il a été déclaré comme un type plus général (`Forme`).

```
for (int i = 0 ; i < tableau.length ; i++) {
tableau[i].affiche(); }
```

Dans ce fragment de code, une boucle `for` est utilisée pour parcourir chaque élément du tableau `tableau` et appeler la méthode `affiche()` de chaque objet. La méthode `affiche()` est une méthode définie dans les classes `Forme`, `Rectangle`, `Cercle`, et `Carre`, et elle est donc héritée par tous les objets de ces classes. La boucle utilise la polymorphie pour appeler la méthode appropriée en fonction du type réel de chaque objet dans le tableau.

1. La boucle `for` parcourt chaque élément du tableau `tableau`.

2. Pour chaque élément, la méthode `affiche()` est appelée.

3. En raison de la polymorphie, la méthode `affiche()` appelée dépend du type réel de l'objet dans le tableau. Si l'objet est de type `Rectangle`, `Cercle`, ou `Carre`, la méthode `affiche()` appropriée de la classe correspondante sera exécutée.

Cela permet d'éviter d'avoir à écrire des instructions conditionnelles pour chaque type d'objet, car la méthode correcte est appelée dynamiquement en fonction du type réel de l'objet, ce qui simplifie le code et le rend plus extensible.

```
public interface Forme { public int surface() ;
 public void affiche() ;
}

public class Rectangle implements Forme { ...
}
```

et

```
public class Cercle implements Forme { ...
}
```

Le code fourni définit une interface appelée `Forme` et deux classes qui implémentent cette interface : `Rectangle` et `Cercle`. Les classes `Rectangle` et `Cercle` doivent fournir une implémentation pour les méthodes déclarées dans l'interface `Forme`.

1. L'interface `Forme` définit deux méthodes, `surface()` et `affiche()`, sans fournir d'implémentation. Les classes qui implémentent cette interface doivent fournir une implémentation pour ces méthodes.

2 La classe `Rectangle` implémente l'interface `Forme`. Elle doit fournir une implémentation pour les méthodes `surface()` et `affiche()` déclarées dans l'interface `Forme`.

La classe `Cercle` implémente également l'interface `Forme`. Elle doit fournir une implémentation pour les méthodes `surface()` et `affiche()` déclarées dans l'interface `Forme`.

Avec cette structure, toutes les classes qui implémentent l'interface `Forme` garantissent qu'elles ont une méthode `surface()` qui renvoie un entier et une méthode `affiche()` qui ne renvoie rien. Cela permet une certaine uniformité dans le traitement des différentes formes, ce qui peut être utile dans certaines situations, par exemple, lors de la manipulation d'objets de différentes formes de manière générique.

```
public abstract class Forme { private int origine_x ; private int origine_y ;
public Forme() { this.origine_x = 0 ; this.origine_y = 0 ;
}
public int getOrigineX() { return this.origine_x ;
}
public int getOrigineY() { return this.origine_y ;
}
public void setOrigineX(int x) { this.origine_x = x ;
}
public void setOrigineY(int y) { this.origine_y = y ;
}
public abstract int surface();
public abstract void affiche(); }
```

Ce code définit une classe abstraite `Forme` qui sert de base pour la représentation d'une forme géométrique. Cette classe contient des membres de données privées représentant les coordonnées de l'origine (`origine_x` et `origine_y`). Elle a des méthodes pour obtenir et définir ces coordonnées, ainsi que deux méthodes abstraites, `surface()` et `affiche()`.

Ces deux variables représentent les coordonnées de l'origine de la forme.

Le constructeur par défaut initialise les coordonnées de l'origine à zéro.

Ces méthodes permettent d'accéder et de modifier les coordonnées de l'origine.

Ces méthodes abstraites `surface()` et `affiche()` sont déclarées dans la classe abstraite `Forme`, mais elles n'ont pas d'implémentation ici. Les classes dérivées de `Forme` doivent fournir une implémentation pour ces méthodes.

En utilisant une classe abstraite, on s'assure que toutes les formes dérivées de cette classe doivent implémenter les méthodes abstraites, mais elles peuvent également hériter des membres et des méthodes concrètes de la classe de base. Cela fournit une structure commune tout en permettant une certaine flexibilité pour les classes dérivées.

```
package java.util ;
public class ArrayList<E> extends AbstractList<E> implements List<E>, ...
{
...
public E set(int index, E element) { ...
}
public boolean add(E e) { ...
} ...
}
```

Le code que vous avez fourni est une partie de la source du framework de collection Java, en particulier de la classe `ArrayList` dans le package `java.util`. La classe `ArrayList` est une implémentation de la liste basée sur un tableau dynamique.

La classe `ArrayList` fait partie du package `java.util`, qui est le package Java standard pour les classes utilitaires.

La classe `ArrayList` étend la classe `AbstractList` et implémente l'interface `List`. Elle est paramétrée par le type générique `E`, ce qui signifie qu'elle peut stocker des éléments de n'importe quel type spécifié lors de l'utilisation.

La méthode `set` remplace l'élément à l'indice spécifié dans la liste par l'élément spécifié. Elle renvoie l'ancien élément qui était à cet indice. Cette méthode est héritée de l'interface `List`.

La méthode `add` ajoute l'élément spécifié à la fin de la liste. Elle renvoie `true` si l'ajout réussit. Cette méthode est héritée de l'interface `List`.

Ces méthodes font partie de l'interface `List`, qui définit un ensemble d'opérations de base que toutes les listes doivent fournir. `ArrayList` les implémente pour fournir une liste redimensionnable basée sur un tableau.

Notez que le code réel de ces méthodes n'est pas fourni dans votre extrait, mais elles contiendraient la logique spécifique pour l'implémentation d'une liste basée sur un tableau dynamique.