OBJECT ORIENTED PROGRAMMING

Lab 8: Advanced topics

### Exercise 1:

What results does this program provide? Explain the results.

**class A**

**{**

**public void affiche() {**

**System.out.print ("Je suis un A ") ;**

**}**

**}**

**class B extends A {}**

**class C extends A**

**{**

**public void affiche() {**

**System.out.print ("Je suis un C ") ;**

**}**

**}**

**class D extends C**

**{**

**public void affiche() {**

**System.out.print ("Je suis un D ") ;**

**}**

**}**

**class E extends B {}**

**class F extends C {}**

**public class Poly**

**{**

**public static void main (String arg[]){**

**A a = new A() ; a.affiche() ; System.out.println() ;**

**B b = new B() ; b.affiche() ;a = b ;**

**a.affiche() ; System.out.println() ;**

**C c = new C() ; c.affiche() ;**

**a = c ; a.affiche() ; System.out.println() ;**

**D d = new D() ; d.affiche() ;**

**a = d ; a.affiche() ;**

**c = d ; c.affiche() ; System.out.println() ;**

**E e = new E() ; e.affiche() ;**

**a = e ; a.affiche() ;**

**b = e ; b.affiche() ; System.out.println() ;**

**F f = new F() ; f.affiche() ;**

**a = f ; a.affiche() ;**

**c = f ; c.affiche() ;**

**}**

**}**

Certaines possibilités d’affectation entre objets des types classes A, B, C, D, E et F

ne figurent pas dans le programme ci-dessus. Pourquoi ?

Answers

**In Java, one of the properties of "polymorphism" is that the call of a method is determined at the time of execution, according to the nature of the object actually referenced (and not only according to the type of the reference) . This is why here all poster calls concerning the same object provide the same message, regardless of the type of reference used:**

**Je suis un A**

**Je suis un A Je suis un A**

**Je suis un C Je suis un C**

**Je suis un D Je suis un D Je suis un D**

**Je suis un A Je suis un A Je suis un A**

**Je suis un C Je suis un C Je suis un C**

**However, a T-type reference can only be assigned a T-type or T-derivative reference. This is what actually happened in our program. But (assuming the same statements), these assignments would be incorrect:**

**b = a; e = a; e = b; c = a; d = c; d = a; f = c; f = a;**

**b = c; b = d; b = f; e = c; e = d; e = f; c = b; c = e; d = b; d = e; f = b;**

**f = e;**

### Exercise 2:

We want to have classes allowing the manipulation of geometric figures.

We want to be able to characterize those that have certain functionalities by asking them to implement interfaces, namely:

• *Affichable* for those who will have a *void affiche ()* method,

• *Tranformable* for those who will have the following two methods:

*void* *homothetie (double coeff)*

*void rotation (double angle)*

Write the two interfaces *Affichable* and *Tranformable*.

Answers

**It suffices to apply the rules for defining an interface, which leads us to:**

**Interface Affichable**

**{**

**abstract public void affiche() ;**

**}**

**Interface Transformable**

**{**

**abstract public void homothetie (double coef);**

**abstract public void rotation (double angle);**

**}**

**Here, our interfaces have package access rights. Like classes, they could be declared public. Abstract and public keywords in method headers can be omitted since, in essence, the methods of an interface are public and abstract.**

**A class representing a figure may implement none, one or both of the preceding interfaces. for example**

**class Point implements Affichable**

**{**

**public void affiche() { ..... }**

**}**

**class Rectangle implements Affichable, Transformable**

**{**

**public void affiche() { ..... }**

**public void homothetie (double coef) { ..... }**

**public void rotation (double angle) { .....}**

**}**

### Exercise 3:

We want to have a hierarchy of classes allowing the manipulation of geometric figures. We want it to always be possible to extend the hierarchy by deriving new classes, but we want to be able to impose that the latter always have the following methods:

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

Write the abstract class Figure which can be used as base class for all these classes.

Answers

**It enough to apply the rules of definition of an abstract class. We place the headers of the methods that we want to see redefined in the derived classes, by associating them with the keyword abstract:**

**abstract class Figure**

**{**

**abstract public void affiche() ;**

**abstract public void homothetie (double coef) ;**

**abstract public void rotation (double angle) ;**

**}**

**The abstract keyword appearing before class can be omitted (any class with at least one abstract method is abstract). However, it is advisable to keep it.**

**As for the argument names accompanying the method headers, they are syntactically necessary (although they have no meaning).**

**The classes of the figure hierarchy will then be simply defined as classes derived from Figure and they will have to define the three methods display, scaling and rotation, for example:**

**class Point extends Figure**

**{**

**public void affiche() { ..... }**

**public void homothetie (double coef) { ..... }**

**public void rotation (double angle) { ..... }**

**.....**

**}**

### Exercise 4:

Complete the abstract class Figure from the previous exercise, so that it implements:

* a *homoRot (double coef, double angle)* method which applies both a homothety and a rotation to the figure,
* static methods *afficheFigures*, *homothetieFigures* and *rotationFigures* applying the same operation (display, scaling or rotation) to an array of figures (objects of a class derived from Figure).

Answers

**An abstract class can contain (non-abstract) method definitions which can then be used by derived classes without having to be redefined (but you can always do it!). On the other hand, an abstract class can have static methods, as long as they are not abstract (which would have no meaning).**

**Ultimately, here is the definition of our new Figure class:**

**abstract class Figure**

**{**

**abstract public void affiche() ;**

**abstract public void homothetie (double coef) ;**

**abstract public void rotation (double angle) ;**

**public void HomoRot (double coef, double angle){**

**homothetie (coef) ; rotation (angle) ;**

**}**

**public static void afficheFigures (Figure [] f){**

**for (int i=0 ; i<f.length ; i++) f[i].affiche() ;**

**}**

**public static void homothetieFigures (double coef, Figure[] f){**

**for (int i=0 ; i<f.length ; i++) f[i].homothetie(coef) ;**

**}**

**public static void rotationFigures (double angle, Figure[] f){**

**for (int i=0 ; i<f.length ; i++) f[i].rotation(angle) ;**

**}**

**}**

**Note that, within the homoRot method, it is possible to call the homotethy and rotation methods, and this even though they are abstract. Indeed, according to the rules of polymorphism, the method actually called will be the one corresponding to the actual type of the object that called the homoRot method; thanks to the constraints on the derivatives of abstract classes, we are certain that it will exist.**

**Similar considerations apply to the call of the scaling and rotation methods in the static homothetieFigures and rotationFigures methods.**

### Exercise 5:

Let the Point and PointNom classes thus defined:

**class Point**

**{**

**public Point (int x, int y) {**

**this.x = x ; this.y = y ;**

**}**

**public static boolean identiques (Point a, Point b){**

**return ( (a.x==b.x) && (a.y==b.y) ) ;**

**}**

**public boolean identique (Point a){**

**return ( (a.x==x) && (a.y==y) ) ;**

**}**

**private int x, y ;**

**}**

**class PointNom extends Point**

**{**

**PointNom (int x, int y, char nom){**

**super (x, y) ; this.nom = nom ;**

**}**

**private char nom ;**

**}**

1. What results does this program provide? Explain the conversions involved and the rules used to process the different method calls:

**public class LimPoly**

**{**

**public static void main (String args[]){**

**Point p = new Point (2, 4) ;**

**PointNom pn1 = new PointNom (2, 4, 'A') ;**

**PointNom pn2 = new PointNom (2, 4, 'B') ;**

**System.out.println (pn1.identique(pn2)) ;**

**System.out.println (p.identique(pn1)) ;**

**System.out.println (pn1.identique(p)) ;**

**System.out.println (Point.identiques(pn1, pn2)) ;**

**}**

**}**

2. Provide the *PointNom* class with an identical static method and an identical method both providing the value true when the two points concerned have both the same coordinates and the same name. What results will the previous program provide then? What conversions will be involved and the rules used?

Answers

**Question 1**

***pn1.identique(pn2)***

**During compilation, we look for an identical method in the class of pn1 (PointNom) or its ancestors. There is only one in Point with an argument of type Point, which fixes its signature in the form identique(Point), by imposing an implicit conversion of pn2 into Point. At runtime, we look for such a method first in PointNom (ligature dynamique)and then, as we cannot find any, in Point. Ultimately, the** **identique method of Point is executed.**

***p.identique(pn1)***

**During compilation, we find the identical method in the class of p (Point), which fixes its signature in the form identique(Point), by imposing an implicit conversion of pn1 into Point. At runtime, we look for such a method in the class of p (Point). Ultimately, the** **identique method of Point is executed.**

***pn1.identique(p)***

**During compilation, we look for a method identique in the class of pn1 (PointNom) or its ancestors. There is only one in Point with an argument of type Point, which fixes its signature in the identical form (Point) (this time no argument conversion is planned). During execution, we look for such a method first in PointNom (dynamic ligature) and then in Point. Ultimately, the identique method of Point is executed.**

***Point.identiques(pn1, pn2)***

**Here, the call is resolved upon compilation (static methods cannot be affected by polymorphism). It involves the conversion of pn1 and pn2 into Point.**

**As you would expect, the program provides these results:**

**true**

**true**

**true**

**true**

**Question 2**

**As the x and y fields of Point are not public and since no access method is available, it is necessary, within the desired methods in PointNom, to use the corresponding methods of Point:**

**public static boolean identiques (PointNom a, PointNom b){**

**return (Point.identiques (a, b) && (a.nom==b.nom) ) ;**

**}**

**public boolean identique (PointNom a){**

**return (super.identique(a) && (nom==a.nom) ) ;**

**}**

**Note the super.identique notation which forces the use of the identical method of the ascending class Point.**

***pn1.identique(pn2)***

**During compilation, we look for a method identique in the class of pn1 (PointNom) or its ancestors. This time the PointNom and Point methods are acceptable. But, the first is better, which freezes the signature of the method called in form identique(PointNom). At runtime, we first look for such a method in PointNom and find it. In the end, we execute the method identique of PointNom, contrary to what happened in question 1.**

***p.identique(pn1)***

**During compilation, this time, we look for a method identique in the class of p (Point), which fixes its signature in the form identique (Point), by imposing an implicit conversion of pn1 into Point. At runtime, we look for such a method in the class of p (Point). Ultimately, we execute (as in first question) Point's identique method. Note that the application of the PointNom method would have no meaning anyway, the object p having no name field!**

***pn1.identique(p)***

**During compilation, we look for, as with the first call, an identique method in the class of pn1 (PointNom) or its ancestors. But, this time, only that of Point is acceptable because we cannot implicitly convert the type Point to PointNom (only the reverse is possible). We therefore freeze the signature of the method**

**called in the form identique(Point) (this time no argument conversion is provided). During execution, we look for such a method first in PointNom (dynamic ligature) and then in Point. Ultimately, the identique method of Point is executed.**

***Point.identiques(pn1, pn2)***

**Here, as before, the call is resolved at compilation time and it always involves the conversion of pn1 and pn2 into Point. As you would expect, the program provides these results:**

**false**

**true**

**true**

**true**

**Note that we could force the use of identics of PointNom by writing PointNom.identique (pn1, pn2); in this case, there would be no more conversion and the result would be false.**