université de BORDEAUX

4TWE802U – Object Oriented Programming
Master Informatique
Pôle Universitaire Français

Final Exam, April, 7th 2018
No document except a dictionary, length 3h

Any answer difficult to understand will be considered as false.

You may add any code (method, data, class) not requested each time you think it is necessary.

A class or a method requested may be considered available in the follow-up of the problem, even if you have not treated the question. It is recommended to read all the subject before starting your work.

If you want to use a class (or a given method) from the Java API, and you don't remember its name, give to it an explicit name and explain clearly the goal of the class on your copy.

You may use, each time you think it is comprehensible, an abbreviation instead the complete name of a class or a method, and also use '...' to replace an unmodified part of the code.

The instructions `import` and `package` will be not given.

Exercice - Itération

We recall the interface `Iterator` :

```
public interface Iterator<E> {

        /** Returns true if the iteration has more elements. */
        boolean hasNext();

        /** Returns the next element in the iteration.
        Throws NoSuchElementException if the iteration has no more elements*/
        E next();

        /** Removes from the underlying collection the last element returned by
        this iterator (optional operation). This method can be called only once
        per call to next(). It will throw an IllegalStateException if the next method
        has not yet been called, or the remove method has already been called
        after the last call to the next method.
        The behavior of an iterator is unspecified if the
        underlying collection is modified while the iteration is in progress in any
        way other than by calling this method.
        The default implementation throws an instance of
        UnsupportedOperationException and performs no other action. */
        default void remove();

        /** Performs the given action for each remaining element until all
        elements have been processed or the action throws an exception.
        The default implementation behaves as if:
        while (hasNext())
            action.accept(next());
```

```
         */
         default void forEachRemaining(Consumer<? super E> action);
}
```

Question 1 In a class `Iterators`, write a class method
`public static <E> Iterator<E> iteratorArrayWithGap(E [] a)` which returns an iterator *it* which iterates the elements the array a with value different than `null`. The method `remove` will change to `null` in the array the value of the last element returned. The method `forEachRemaining` will execute its default code.

Example : the following code

```
String[] as = { "a", null, "c" };
Iterator<String> it = Iterators.iteratorArrayWithGap(as);
while (it.hasNext())
    System.out.print(it.next() + " ");
System.out.println();
it.remove();
for (String s: as)
    System.out.print(s + " ");
System.out.println();
```

will display on the standard output

```
a c
a null null
```

We want to generalize this method. For that, we will use the interface `Predicate` :

```
public interface Predicate<T> {
    public boolean test(T t); //Evaluates this predicate on the given argument.
}
```

Question 2 Add in the class `Iterators` a class method
`iteratorArrayWithFilter(E [] a, Predicate<E> p)` which take as parameters an array and an instance of `Predicate`. The iterator returned by `iteratorArrayWithFilter` will return only the values of the array `a` for which `p.test` returns `true`. As in the previous question, the method `remove` will set to `null` the last element of the array returned by the iterator.

Question 3 Create in the class `Iterators` a class constant `IS_NOT_NULL` instance of `Predicate<Object>` and such that the method `test` returns `true` if the element given as parameter is not null. If possible, use a lambda-expression to implement `IS_NOT_NULL` .

Question 4 If we write the following code :

```
String[] as = { "a", null, "c" };
Iterator<String> it = Iterators.iteratorArrayWithFilter(as, IS_NOT_NULL);
```

the following error appears during compilation :

```
Type mismatch: cannot convert from Iterator<Object> to Iterator<String>
```

Modify the signature of the method `iteratorArrayWithFilter` to avoid this problem. Give the new code of `iteratorArrayWithGap` using `iteratorArrayWithFilter`.

Problem - Vehicle

We consider the following interface `Vehicle` :

```
public interface Vehicle {
    /** Speed of the vehicle in km/h */
    public double speed();
    /** Direction of the vehicle in radian, relative to the x-axis */
    public double direction();
    /** Position of the vehicle */
    public Point2D position();
    /** Run the vehicle during a time given in hours.
     This method modify the position of the vehicle, following
     its speed and its direction. */
    public void go(double time);
}
```

and its default implementation `VehicleImpl` :

```
public class VehicleImpl implements Vehicle {

    protected double maximumSpeed;
    protected double speed = 0;
    protected double direction = 0;
    protected Point2D position = new Point2D.Double(0., 0.);

    private void checkPositiveSpeed(double speed) {
        if (speed < 0) {
            throw new IllegalArgumentException("negative speed forbidden");
        }
    }

    public VehicleImpl(double maximumSpeed) {
        checkPositiveSpeed(maximumSpeed);
        this.maximumSpeed = maximumSpeed;
    }

    public void setDirection(double direction) {
        this.direction = direction % (2 * Math.PI);
    }
```

```
    public void setSpeed(double speed) {
        checkPositiveSpeed(speed);
        this.speed = Math.min(maximumSpeed, speed);
    }

    public double direction() {
        return direction;
    }

    public double speed() {
        return speed;
    }

    public double maximumSpeed() {
        return maximumSpeed;
    }

    public Point2D position() {
        return position;
    }

    public void go(double time) {
        double distance = time * speed;
        double x = Math.cos(direction) * distance;
        double y = Math.sin(direction) * distance;
        position.setLocation(position.getX() + x, position.getY() + y);
    }

}
```

You will also need to use the following methods from the class `java.awt.geom.Point2D` :

```
double getX();
double getY();
void setLocation(double x, double y);
```

which respectively

- returns the x-coordinate of an instance of `Point2D`,
- returns the y-coordinate of an instance of `Point2D`,
- modify the coordinates (x,y) of an instance of `Point2D`.

`Point2D.Double` is an internal class in `Point2D` which gives an implementation of `Point2D`.

Question 1 Modify the class `VehicleImpl` in such a way that the instruction

```
System.out.print(v);
```

where `v` is an instance of `VehicleImpl` displays on the standard output the following result :
`Vehicle (`x`, `y`), speed = `s`, direction = `d
where x and y are the coordinates of the current position of the vehicle, s its speed and d its direction.

We want now to have a new class `VehicleWithAcceleration` which implements `Vehicle` and contains a new method :

```
/** Modify the speed of the vehicle */
void accelerate(double acceleration) { ... }
```

Question 2 Suggest an implementation of `VehicleWithAcceleration` using the class `VehicleImpl` by inheritance. For the method `accelerate(double acceleration)`, the speed will be changed by adding the value given by `acceleration` which can be positive or negative. Nevertheless, the speed will not become negative, nor exceed the maximum speed of the vehicle.

We consider now the interface `Surface` :

```
public interface Surface {

    /** test if the surface contains the point p. */
    public boolean contains(Point2D p);

    /** test if it is possible to go from the point p1 to the point p2
     * using a straight line and without going outside the surface.
     * Throw an exception IllegalArgumentException if p1 or p2 are not
     * in the surface.
     */
    public boolean areConnected(Point2D p1, Point2D p2);
}
```

Question 3 Write a class `SurfaceRectangle` which implements `Surface`. This surface will be composed by the points of a rectangle, specified by its top left corner, its width and its height, given at the construction. Note that two points inside a rectangle are always connected.

Question 4 Modify the class `SurfaceRectangle` in such a way that two of its instances are equal if and only if they have the same top left corner, the same width and the same height. *Indication : don't forget to redefine all the necessary methods inherited from the class `java.lang.Object`.*

Question 5 Propose a class `SurfaceWithVehicle` which associate a vehicle and a surface. The instances of `SurfaceWithVehicle` will be created from yet existing surface and vehicle, given at the construction. This class will contain a method `void moveVehicle(time)` which will call the method `go(double time)` of the associated vehicle.

If the vehicle go out of the associated surface during a move, an exception `AccidentException` will be thrown.

If the surface does not contain the position of the vehicle at the construction of an instance of `SurfaceWithVehicle`, an exception `java.lang.IllegalArgumentException` will be thrown.

Question 6 Give an implementation of the class `AccidentException`.

Question 7 Show that the current implementation is not correct because it is possible to move a vehicle outside of the associated surface without throwing the exception `AccidentException`.

To avoid this problem, we propose another approach : we want that it will be possible to create observable vehicles. Each time an observable vehicle will be moved, its observers will be notified.

We give below the methods of the class `Observable` you will have to use and also the interface `Observer`.

```
public class Observable {
...
    public void addObserver(Observer o);
    public void notifyObservers(Object arg);
    protected void setChanged();
}

public interface Observer {
   public void update(Observable o, Object arg);
}
```

Question 8 Write a class `VehicleObservable` which inherits from `Observable` and implements `Vehicle`. Use the class `VehicleImpl` by delegation to implement the methods of the interface `Vehicle`.

Question 9 Propose a new version of the class `SurfaceWithVehicle` which implements the interface `Observer` and which will be created using an instance of `Surface` and an instance of `VehicleObservable`. The vehicle will be moved using directly its own methods (using its own reference). The new class `SurfaceWithVehicle` will only implement the method `update` (from `Observer`) and will throw an exception `java.lang.IllegalStateException` which extends the class `java.lang.RuntimeException`, if the vehicle go out of the surface. Justify in two or three lines why it is not possible to re-use the class `AccidentException`.

Question 10 Modify the class `VehicleObservable` in such a way that, if an exception is thrown while we execute the method `go(double time)`, the vehicle will just stay at its original position.