Equals - Exercice (1)

```
public class Point {

    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
    public boolean equals(Point o) {
       return (this.getX() == o.getX() && this.getY() == o.getY());
    }
    // ...
}
```

Equals - Exercice (1)

Tout semble marcher….

```
Point p1 = new Point(1, 2); Point p2 = new Point(1, 2);
Point q = new Point(2, 3);
System.out.println(p1.equals(p2)); // -> true
System.out.println(p1.equals(q)); // -> false
```

Mais…….

```
Object p2a = p2;
System.out.println(p1.equals(p2)); // prints true
System.out.println(p1.equals(p2a)); // prints false
```

Pourquoi?

Equals - Exercice (2)

Soit la méthode equals définie comme telle:

```
// A better definition
public boolean equals(Object o) {
    return o == this ||
           o != null && o.getClass() == getClass() &&
           ((Point) o).getX() == getX() && ((Point) o).getY() == getY();
}
```

Equals - Exercice (2)

Tout semble mieux marcher….

```
Point p1 = new Point(1, 2); Point p2 = new Point(1, 2);
Point q = new Point(2, 3); Object p2a = p2;
System.out.println(p1.equals(p2)); // -> true
System.out.println(p1.equals(q)); // -> false
System.out.println(p1.equals(p2a)); // -> true
```

Mais…

```
HashSet<Point> coll = new HashSet<Point>();
coll.add(p1);
System.out.println(coll.contains(p2)); // -> false
```

Pourquoi?

Equals - Exercice (3)

Imaginons qu'on modifie la class Point pour la rendre mutable (enlever le "final" des attributs x et y), et qu'on y ajoute les méthodes setX et setY….

```
Point p = new Point(1, 2); Set<Point> coll = new HashSet<Point>();  coll.add(p);
System.out.println(coll.contains(p)); // true

Iterator<Point> it = coll.iterator(); boolean containedP = false;
while (it.hasNext()) {
    Point nextP = it.next();
    if (nextP.equals(p)) { containedP = true; break; }
}
System.out.println(containedP); // -> true
```

Mais

```
p.setX(p.getX() + 1);
System.out.println(coll.contains(p)); // false
```

Pourquoi *coll* ne contient pas p, alors que p fait partie des éléments de *coll* ?

Clonable

Pourquoi n'est-il que rarement souhaitable d'utiliser l'interface ***Clonable*** et la méthode clone?

A. L'interface *Cloneable* n'à pas de méthode **clone**. Avoir un object de type Clonable ne dit donc pas ce qu'on peut faire avec.
B. Appeler la méthode ***clone*** sur un object externe de type ***Objet*** et qui implémente *Clonable*, le compilateur se plaindre que vous essayez d'appeler une méthode protégée.
C. clone() crée un objet sans appeler un constructor.
D. clone() fonctionne mal avec les attributs immuables (final)
E. ***CloneNotSupportedException*** n'est pas trivial à prendre en compte correctement.
F. Parce que clone() ne définit pas s'il s'agit d'une shallow-copy ou de deep-copy()

Clonable

Alternative conseillée: Si vous pouvez, utilisez un constructeur de copie.

Dans quel cas vous risquez de quand même devoir implémenter clone()? Par exemple si i la classe dont on hérite implémente Clonable.

Pourquoi Clonable n'a pas de méthode? principalement une erreur de design dans Java, difficile de corriger après coup:
https://bugs.openjdk.java.net/browse/JDK-4098033

https://www.artima.com/articles/josh-bloch-on-design#part13