Objets non mutables

Singleton

I Introduction

- Objet mutable : possible de modifier son état après sa création
- **Objet non mutable** : objet dont les attributs sont constants (mot-clé **final**)
- Attributs initialisés lors de l'instanciation (en général dans un constructeur)
- Objet à un seul état
- Nombreux avantages des objets non mutables :
    - garantis **thread-safe**
    - mise en cache possible et sans risque de désynchronisation
    - **constructeur de copie** inutile
    - implémentation de l'interface **Cloneable** inutile (pas de méthode **clone** à définir)
    - **copie défensive** inutile
    - invariants testés uniquement à la création (cf programmation par assertions, par contrat)
    - excellents supports pour les clés des **Map<K,V>**

II Création d'une classe d'objets non mutables

- tous les attributs **final**
- **aucune redéfinition de méthode dans les sous-classes ==> classe final**
- ne jamais exportée la référence à **this** (via un retour d'une méthode)
- tous les attributs **private,** ne jamais exporter un attribut (==> ***getters*** à revisiter)
- aucune méthode de modification de l'état de l'objet (pas de ***setters*** notamment)

1) Exemple correct :

Rappel : la classe **String** est **non mutable**.

classe Author non mutable

- **classe final**
- **attributs constants**
- **passage de paramètres avec des objets non mutables**
  ==> pas d'effet de bord !

```
package book;
import java.util.Objects;

public final class Author {
        private final String firstName;
        private final String lastName;

        public Author(String firstName, String lastName) {
                // Ensure an object is not null.
                // If null, throws a nullPointerException
                Objects.requireNonNull(firstName;
                Objects.requireNonNull(lastName);
                this.firstName = firstName;
                this.lastName = lastName;
        }
        public @Override String toString() {
                return firstName + ' ' + lastName;
        }
}
```

2) L'utilisation d'objets mutables peut casser l'encapsulation

rappel : La classe **StringBuilder** est **mutable**.

```
package mutable;

public final class Cat {
	private final StringBuilder name;

	public Cat(StringBuilder name) {
		this.name = name;
	}

	public StringBuilder getName() {
		return name;
	}

	public static void main(String[] args) {
		StringBuilder name = new StringBuilder("sylvestre");
		Cat cat = new Cat(name);
		name.reverse();
		System.out.println(cat.getName()); // ertsevlys
	}
}
```

3) Faire une copie défensive lors de la création n'est pas suffisant !

```
package mutable;

public final class Cat {
	private final StringBuilder name;

	public Cat(StringBuilder name) {
		// COPIE DEFENSIVE
		this.name = new StringBuilder(name);
	}

	public StringBuilder getName() {
		return name;
	}

	public static void main(String[] args) {
		StringBuilder name = new StringBuilder("sylvestre");
		Cat = new Cat(name);
		cat.getName().reverse();
		System.out.println(cat.getName()); // ertsevlys
	}
}
```

4) La copie défensive doit être fait aussi lors du retour d'un attribut mutable

```
package mutable;
public final class Cat{
	private final StringBuilder name;

	public Cat(StringBuilder name) {
		this.name = new StringBuilder(name);
	}

	public StringBuilder getName() {
		return new StringBuilder(name);
	}

	public static void main(String[] args) {
		StringBuilder name=new StringBuilder("sylvestre");
		Cat cat = new  Cat(name);
		name.reverse();
		System.out.println(cat.getName()); // sylvestre
		cat.getName().reverse();
		System.out.println(cat.getName()); // sylvestre
	}
}
```

5) Le plus simple est de souvent utiliser un objet non mutable quand cela est possible

```
package nonmutable;
public final class Cat {
	private final String name; // classe String non mutable

	public Cat(String name) {
		this.name=name;
	}

	public String getName() {
		return name;
	}

	public static void main(String[] args) {
		String name = "sylvestre";
		Cat cat = new Cat(name);
		// reverse non définie pour String ==> on utilise toUpperCase
		System.out.println(name.toUpperCase()); // SYLVESTRE
		System.out.println(cat.getName()); // sylvestre
		System.out.println(cat.getName().toUpperCase()); // SYLVESTRE
		System.out.println(cat.getName()); // sylvestre
	}
}
```

Rappel : la méthode **toUpperCase** retourne un **nouvel objet de type String.**

6) Autre exemple à priori correct :

Rappel : la classe **java.util.Date** est **mutable**.

```
package mutable;
import java.util.Date;

public final class MyDate {
        private final Date date;

        public MyDate(Date date) {
                this.date = date;
        }

        @Override
        public String toString() {
                return date.toString();
        }
}
```

```
package mutable;
import java.util.Date;

public class DateTest {
        @SuppressWarnings("deprecation")
        public static final void main(String []  args){
                Date date = new Date();

                MyDate mydate = new MyDate(date);
                System.out.println(date);
                // deprecated, 1900 + 99
                date.setYear(99);
                System.out.println(mydate);
        }
}
```

```
Sat Mar 05 19:19:42 CET 2016
Fri Mar 05 19:24:58 CET 1999
```

➔ Le **passage de paramètre** d'un objet de type Date ouvre **une faille de sécurité** dans l'objet **mydate** qui se veut non mutable.

➔ **L'attribut date référence la date passée en paramètre.**

➔ Une modification de la date passée en paramètre provoque un **effet de bord.**

➔ **L'objet mydate est modifié!**

Remède : réaliser une **copie défensive** du paramètre

```
public MyDate(Date date) {
                // pas de copie défensive ==> effet de bord possible
                // this.date = date;
                // COPIE DEFENSIVE
                this.date = (Date) date.clone();
        }
```

```
Sat Mar 05 19:38:33 CET 2016
Sat Mar 05 19:38:33 CET 2016
```

Même problématique pour l'exportation d'un attribut en valeur de retour :

exemple d'un getter revisité :

```
public Date getDate(){
                    // pas de copie défensive ==> effet de bord possible
                    // return date;
                    // COPIE DEFENSIVE
                    return (Date) date.clone();
}
```

Pour qu'un objet soit non mutable :

- **les objets qu'il référence ne doivent pas être des objets mutables.**
- **sinon mettre en place des copies défensives**

7) Mutabilité et création

- **Si un objet est non mutable, toute modification entraîne la création d'un nouvel objet.**

Classe Point mutable	Classe Point non mutable

`public class Point {`
`  private int x;`
`  private int y;`
 	
`  public Point(int x,int y) {`
`  this.x = x;`
`  this.y = y;`
`  }`
 	
`  public void translate(int dx,int dy) {`
`  x += dx;`
`  y+ = dy;`
`  }`
`}`
	`public final class Point {`
	`  private final int x;`
	`  private final int y;`
	
	`  public Point(int x,int y) {`
	`  this.x = x;`
	`  this.y = y;`
	`  }`
	`  public Point translate(int dx,int dy) {`
	`  return new Point(x+dx, y+dy);`
	`  }`
	`}`

III Tableau toujours mutable!

- **La copie défensive doit être effectuée lorsque l'on retourne la valeur d'un attribut**

```
package nonmutable;
import java.util.Arrays;

public final class Stack { // Objectif : classe non mutable
	private final int[] array;

	public Stack(int capacity) {
		array = new int[capacity];
	}

	public int[] asArray() {
		return array.clone();
	}

	@Override
	public String toString() {
		return "Stack [array=" + Arrays.toString(array) + "]";
	}

	public static void main(String[] args) {
		Stack s = new Stack(3);
		int [] array = s.asArray();
		array[1] = 30;
		System.out.println("int [] array=" + Arrays.toString(array)); // int [] array=[0, 30, 0]
		System.out.println(s); // Stack [array=[0, 0, 0]]
	}
}
```

IV Objet mutable dans un contexte de programmation concurrente

```
package concurrent;

import java.util.Date;


// NO THREAD-SAFE ==> mutable
public final class MyDate {
	private final Date startDate;
	private final Date endDate;

	public MyDate (Date startDate, Date endDate) {
		// PROBLEME :
		// AUCUNE PROTECTION DES ACCES CONCURRENTS :
		// un thread peut modifier les 2 paramètres et le test être vrai
		if (startDate.compareTo(endDate) > 0) {
			throw new IllegalArgumentException("The start date is not <=
								the end date.");
		}

		// POUR EVITER LES EFFETS DE BORD via les paramètres
		// MAIS on peut récupérer des paramètres modifiés via un thread, mais
		// vérifiant toujours le test.

		// De plus, puisqu'il n'est pas possible de prédire le séquencement
		// des opérations multi-thread dans la plupart des environnements, rien
		// ne garantit que le thread principal ne s'arrêtera pas juste après
		// l'exécution du if pour donner la main à un second thread qui
		// modifiera startDate pour que sa valeur soit supérieure a endDate.
		this.startDate = (Date) startDate.clone();
		this.endDate = (Date) endDate.clone();
	}

	@Override
	public String toString() {
		return "MyDate [startDate=" + startDate + ", endDate=" + endDate + "]";
	}
}
```

Remarque :

Pour utiliser des **objets non thread-safe** à partir de **plusieurs threads**, il faut obligatoirement appliquer de la **synchronisation externe**.

Exemples : ArrayList, HashMap

```java
package concurrent;

import java.util.Date;


//THREAD-SAFE et non mutable
public final class MyDate {
	private final Date startDate;
	private final Date endDate;

	public MyDate (Date startDate, Date endDate) {
		// POUR EVITER LES EFFETS DE BORD via les paramètres
		// ET SE PROTEGER DES ACCES CONCURRENTS
		startDate = (Date) startDate.clone();
		endDate = (Date) endDate.clone();

		if (startDate.compareTo(endDate) > 0) {
			throw new IllegalArgumentException("The start date is not <=
								the end date.");
		}

		// opérations atomiques pas besoin de synchronisation
		this.startDate = startDate;
		this.endDate = endDate;
	}

	@Override
	public String toString() {
		return "MyDate [startDate=" + startDate + ", endDate=" + endDate + "]";
	}
}
```

Remarque :

Un objet peut être thread-safe mais néanmoins mutable.

Sa classe utilise le **mécanisme** de **synchronisation** dans son implémentation **interne**, et garantit dès **lors** que son état sera toujours correct lorsque cet objet est partagé entre plusieurs threads.

Avec cette **garantie**, on peut donc utiliser de tels objets sans aucune **précaution** particulière.

Exemples : Random, ConcurrentHashMap.

V Singleton

```java
package singleton;

/**
 *  Les méthodes statiques sont souvent utilisés pour assurer l'unicité d'un objet
 *  Design Pattern : singleton
 */

public class Platform {
	private static final String NAME_PER_DEFAULT = "Java 8";

	// initialisation statique au chargement de la classe
	// ==> une seule instance
	private static Platform platform = new Platform(NAME_PER_DEFAULT);

	private String name;

	// Constructeur privé afin qu'un code client n'instancie pas plusieurs fois
	private Platform(String name){
		this.name = name;
	}

	// retourne l'unique instance par défaut
	public static Platform getDefaultPlatform() {
		return platform;
	}

	@Override
	public String toString() {
		return "Platform [name=" + name + "]";
	}
}
```

```java
package singleton;

public class PlatformTest {

	public static final void main(String [] args){
		Platform platform1 = Platform.getDefaultPlatform();
		System.out.println(platform1);

		// NE COMPILE PAS : constructeur privé
		// Platform platform2 = new Platform("Java 5");
	}
}
```