Pratical Exercices N° 3 - Debriefing

Nom	Prenom
BADAROU	Mikdaam

Exercice 1 - Book

1. Declare a record `Book` with `title` and `author` components.

```java
public record Book(String title, String author) {}
```

2. Adding main method to the `Book` record

```java
 public record Book(String title, String author) {
     public static void main(String[] args) {
         var book = new Book("Da Vinci Code", "Dan Brown");
         System.out.println(book.title + ' ' + book.author);
     }
 }
```

3. Creates a class `Main` and move the `main()` method in it.

```java
public class Main {
    public static void main(String[] args) {
        var book = new Book("Da Vinci Code", "Dan Brown");
        System.out.println(book.title + ' ' + book.author);
    }
}
```

   - In the `main()` method of the class, we can not access directly to the components of the records, so the above code is not valid. A valid code wiil be :

```diff
public class Main {
    public static void main(String[] args) {
-       var book = new Book("Da Vinci Code", "Dan Brown");
+       System.out.println(book.title() + ' ' + book.author());
    }
}
```

4. Avoid the record to create object with `null` components To avoid the creation of object with `null` components, we can use the static method `requireNonNull` in the constructor of the record.

```java
import java.util.Objects;
public record Book(String title, String author) {
    public Book(String title, String author) {
        Objects.requireNonNull(title, "title must not be null");
        Objects.requireNonNull(author, "author must not be null");
    }
}
```

5. Use a compact constructor for the previous questions

```java
import java.util.Objects;
public record Book(String title, String author) {
    public Book {
        Objects.requireNonNull(title, "title must not be null");
        Objects.requireNonNull(author, "author must not be null");
    }
}
```

6. Create a second constructor who takes just the `title` components

```diff
import java.util.Objects;
public record Book(String title, String author) {
    public Book {
        Objects.requireNonNull(title, "title must not be null");
        Objects.requireNonNull(author, "author must not be null");
    }

+   public Book(String title) {
+       this(title, "<no author>");
+   }
}
```

7. How does the compiler know what constructor call ?

   The compiler looks the signature of each constructor to decide what constructor call.

8. Change the `title` component with `withTitle` method. The following code

```java
public void withTitle(String title) {
    this.title = title;
}
```

doesn't works because the `title` component is `final` . It means that any component of the record can't be change in order to preserve integity of data.

A better way to perform this operation will be this :

```java
public Book withTitle(String title) {
    return new Book(title, this.author);
}
```

We can test this method in the `main()` :

```diff
public class Main {
    public static void main(String[] args) {
        var book = new Book("Da Vinci Code", "Dan Brown");
        System.out.println(book.title() + ' ' + book.author());

+       book = book.withTitle("The Da Vinci Code");
+       System.out.println(book.title() + ' ' + book.author());
    }
}
```

Exercice 2 - Liberty, Equality, toString

1. The above code

   ```java
   var b1 = new Book("Da Java Code", "Duke Brown");
   var b2 = b1;
   var b3 = new Book("Da Java Code", "Duke Brown");

   System.out.println(b1 == b2);
   System.out.println(b1 == b3);
   ```

   will display

   ```
    true
    false
   ```

   Explanation

   The `==` operator on object always compares the reference of the object.

   - The first comparison, it compares the reference of `b1` and `b2` objects which is the same
   - The second comparison return false because the reference of `b1` and `b3` is different.

2. To compare the contains of two objects, we have to use the `equals()` method.

- A valid version of the previous code will be :

```java
var b1 = new Book("Da Java Code", "Duke Brown");
var b2 = b1;
var b3 = new Book("Da Java Code", "Duke Brown");

System.out.println(b1.equals(b2));
System.out.println(b1.equals(b3));
```

3. Create a method `isFromTheSameAuthor()` which returns `true` if two books has the same author.

```java
import java.util.Objects;
public record Book(String title, String author) {
    ...
    public boolean isFromTheSameAuthor(Book other) {
        return author.equals(other.author);
    }
}
```

4. Rewrite the `toString()` method

```java
import java.util.Objects;
public record Book(String title, String author) {
    ...
    public String toString() {
        return title + " by " + author;
    }
}
```

5. Adding the `@Override` annotation on the `tostring()` method.

```diff
import java.util.Objects;
public record Book(String title, String author) {
    ...
+   @Override
    public String toString() {
        return title + " by " + author;
    }
}
```

6. What the `@Override` annotation is used for ?

The `@Override` annotation ensure that the signature of a method is correct. It to avoid miswriting the name of a basic method

Exercice 3

Consider the following code :

```
public class Book2 {
    private final String title;
    private final String author;

    public Book2(String title, String author) {
        this.title = title;
        this.author = author;
    }

    public static void main(String[] args) {
        var book1 = new Book2("Da Vinci Code", "Dan Brown");
        var book2 = new Book2("Da Vinci Code", "Dan Brown");
        System.out.println(book1.equals(book2));
    }
}
```

1. The above code has an unexpected behavior because the class `Book2` doesn't have an `equals` method, so the compiler use the `equals` method of `Object` class.

2. To fix this issue, we have to add an `equals` method.

```
public class Book2 {
    ...
    @Override
    public boolean equals(Object o) {
        return o instanceof Book book
            && Objects.equals(title, book.title)
            && Objects.equals(author, book.author);
    }
    ...
}
```

Exercice 4 - Bubble Sort

1. Write a method named `swap()` which change the values of two cases of an array

```java
public static void swap(int[] array, int index1, int index2) {
    int tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
}
```

2. Write a method `indexOfMin()` which return the index of minimum value of an array

```java
public static int indexOfMin(int array[]) {
    int min = array[0];
    int index = 0;
    for (int i = 1; i < array.length; i++) {
        if (array[i] < min) {
            min = array[i];
            index = i;
        }
    }
    return index;
}
```

3. Edit the previous method

```diff
-public static int indexOfMin(int array[]) {
-    int min = array[0];
-    int index = 0;
-    for (int i = 1; i < array.length; i++) {
-        if (array[i] < min) {
-            min = array[i];
-            index = i;
-        }
-    }
-    return index;
-}

+public static int indexOfMin(int array[], int start, int end) {
+    int min = array[start];
+    int index = start;
+    for (int i = start + 1; i < end; i++) {
+        if (array[i] < min) {
+            min = array[i];
+            index = i;
+        }
+    }
+    return index;
+}
```

4. Write the `sort()` method usibg the two previous methods

```java
public static void selectionSort(int[] array) {
    for (int i = 0; i < array.length; i++) {
        int minIndex = indexOfMin(array, i, array.length);
        swap(array, i, minIndex);
    }
}
```