Pratical Exercices N° 7 - Debriefing

Expression Tree

1. **Create `Expr`, `Value`, `Add`, `Sub` and `Mul` representation.**

```
public interface Expr {
    public static void main(String[] args) {
        Expr expression = new Add(new Value(2), new Value(3));
        Expr expression2 = new Sub(new Mul(new Value(2), new Value(3)), new
Value(4));
    }
}
```

```
import java.util.Objects;
public record Add(Expr leftExpr, Expr rightExpr) implements Expr {
    public Add {
        Objects.requireNonNull(leftExpr, "leftExpr can't be null");
        Objects.requireNonNull(rightExpr, "rightExpr can't be bull");
    }
}
```

```
import java.util.Objects;
public record Sub(Expr leftExpr, Expr rightExpr) implements Expr {
    public Sub {
        Objects.requireNonNull(leftExpr, "leftExpr can't be null");
        Objects.requireNonNull(rightExpr, "rightExpr can't be bull");
    }
}
```

```
import java.util.Objects;
public record Mul(Expr leftExpr, Expr rightExpr) implements Expr {
    public Mul {
        Objects.requireNonNull(leftExpr, "leftExpr can't be null");
        Objects.requireNonNull(rightExpr, "rightExpr can't be bull");
    }
}
```

```
public record Value(int value) implements Expr {}
```

2. Add `eval()` methode to the `Expr` interface

```
public interface Expr{
    ...
    public int eval();
}
```

So we have to add `eval` method to all classes that implements `Expr`.

```
public record Add(Expr leftExpr, Expr rightExpr) implements Expr {
    ...
    public int eval() {
        return leftExpr.eval() + rightExpr.eval();
    }
}
```

```
public record Sub(Expr leftExpr, Expr rightExpr) implements Expr {
    ...
    public int eval() {
        return leftExpr.eval() - rightExpr.eval();
    }
}
```

```
public record Mul(Expr leftExpr, Expr rightExpr) implements Expr {
    ...
    public int eval() {
        return leftExpr.eval() * rightExpr.eval();
    }
}
```

```
public record Value(int value) implements Expr {
    ...
    public int eval() {
        return value;
    }
}
```

3. **Write `parse()` method which take a given expression and parse it to an expression tree**

```
public interface Expr{
    ...
    static Expr parse(Scanner tokens) {
        var token = tokens.next();
        return switch (token) {
            case "+" -> new Add(parse(tokens), parse(tokens));
            case "-" -> new Sub(parse(tokens), parse(tokens));
            case "*" -> new Mul(parse(tokens), parse(tokens));
            default -> new Value(Integer.parseInt(token));
        };
    }
}
```

4. **How to fix the bug ?**

    To fix the bug, we have to make the `Expr` interface sealed.

```
public sealed interface Expr permits Add, Sub, Mul, Value {}
```

5. **Move the `main()` method to another class.** ✔☑✨✔ Done.

6. **What interface to use to allow passing a list of String ?** To be able to pass a list of String, we have to use the `Iterable` interface. So, the code will look like this:

```
public sealed interface Expr permits Add, Sub, Mul, Value {
    ...
    static Expr parse(Iterator<String> tokens) {
        var token = tokens.next();
        return switch (token) {
            case "+" -> new Add(parse(tokens), parse(tokens));
            case "-" -> new Sub(parse(tokens), parse(tokens));
            case "*" -> new Mul(parse(tokens), parse(tokens));
            default -> new Value(Integer.parseInt(token));
        };
    }
}
```

7. Add a method to be able to print the tree

```
public sealed interface Expr permits Add, Sub, Mul, Value{
    ...
    public StringBuilder stringify();
}
```

So we have to add `stringify` method to all classes that implements `Expr`.

```
public record Add(Expr leftExpr, Expr rightExpr) implements Expr {
    ...
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append("(").append(leftExpr.stringify())
                .append(" + ")
                .append(rightExpr.stringify())
                .append(")");
    }
}
```

```java
public record Sub(Expr leftExpr, Expr rightExpr) implements Expr {
    ...
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append("(").append(leftExpr.stringify())
                .append(" - ")
                .append(rightExpr.stringify())
                .append(")");
    }
}
```

```java
public record Mul(Expr leftExpr, Expr rightExpr) implements Expr {
    ...
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append("(").append(leftExpr.stringify())
                .append(" * ")
                .append(rightExpr.stringify())
                .append(")");
    }
}
```

```java
public record Value(int value) implements Expr {
    ...
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append(value);
    }
}
```

8. 9, 10 : **Refactor the code and add a second interface `BinOp`**

```java
public sealed interface BinaryOp extends Expr permits Add, Sub, Mul {
    Expr leftExpr();
    Expr rightExpr();
    int operator(int leftValue, int rightValue);
    default int eval() {
        int leftValue = leftExpr().eval();
        int rightValue = rightExpr().eval();
        return operator(leftValue, rightValue);
    }
}
```

```java
public sealed interface Expr permits BinaryOp, Value {
    public int eval();
    public StringBuilder stringify();
    static Expr parse(Iterator<String> tokens) {
        var token = tokens.next();
        return switch (token) {
            case "+" -> new Add(parse(tokens), parse(tokens));
            case "-" -> new Sub(parse(tokens), parse(tokens));
            case "*" -> new Mul(parse(tokens), parse(tokens));
            default -> new Value(Integer.parseInt(token));
        };
    }
}
```

```java
public record Add(Expr leftExpr, Expr rightExpr) implements BinaryOp {
    ...
    @Override
    public int operator(int leftValue, int rightValue) {
        return leftValue + rightValue;
    }
}
```

```java
public record Sub(Expr leftExpr, Expr rightExpr) implements BinaryOp {
    ...
    @Override
    public int operator(int leftValue, int rightValue) {
        return leftValue - rightValue;
    }
}
```

```java
public record Mul(Expr leftExpr, Expr rightExpr) implements BinaryOp {
    ...
    @Override
    public int operator(int leftValue, int rightValue) {
        return leftValue * rightValue;
    }
}
```

Code Source

`Value.java`

```java
public record Value(int value) implements Expr {
    @Override
    public int eval() {
        return value;
    }
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append(value);
    }
}
```

`Add.java`

```java
public record Add(Expr leftExpr, Expr rightExpr) implements BinaryOp {
    public Add {
        Objects.requireNonNull(leftExpr, "leftExpr can't be null");
        Objects.requireNonNull(rightExpr, "rightExpr can't be bull");
    }
    @Override
    public int operator(int leftValue, int rightValue) {
        return leftValue + rightValue;
    }
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append("(").append(leftExpr.stringify())
                .append(" + ")
                .append(rightExpr.stringify())
                .append(")");
    }
}
```

Sub.java

```java
public record Sub(Expr leftExpr, Expr rightExpr) implements BinaryOp {
    public Sub {
        Objects.requireNonNull(leftExpr, "leftExpr can't be null");
        Objects.requireNonNull(rightExpr, "rightExpr can't be bull");
    }
    @Override
    public int operator(int leftValue, int rightValue) {
        return leftValue - rightValue;
    }
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append("(").append(leftExpr.stringify())
                .append(" - ")
                .append(rightExpr.stringify())
                .append(")");
    }
}
```

Mul.java

```java
public record Mul(Expr leftExpr, Expr rightExpr) implements BinaryOp {
    public Mul {
        Objects.requireNonNull(leftExpr, "leftExpr can't be null");
        Objects.requireNonNull(rightExpr, "rightExpr can't be bull");
    }
    @Override
    public int operator(int leftValue, int rightValue) {
        return leftValue * rightValue;
    }
    @Override
    public StringBuilder stringify() {
        return new StringBuilder().append("(").append(leftExpr.stringify())
                .append(" * ")
                .append(rightExpr.stringify())
                .append(")");
    }
}
```

Expr.java

```java
public sealed interface Expr permits BinaryOp, Value {
    public int eval();
    public StringBuilder stringify();
    static Expr parse(Iterator<String> tokens) {
        var token = tokens.next();
        return switch (token) {
            case "+" -> new Add(parse(tokens), parse(tokens));
            case "-" -> new Sub(parse(tokens), parse(tokens));
            case "*" -> new Mul(parse(tokens), parse(tokens));
            default -> new Value(Integer.parseInt(token));
        };
    }
}
```

BinaryOp.java

```
public sealed interface BinaryOp extends Expr permits Add, Sub, Mul {
    Expr leftExpr();
    Expr rightExpr();

    int operator(int leftValue, int rightValue);

    default int eval() {
        int leftValue = leftExpr().eval();
        int rightValue = rightExpr().eval();
        return operator(leftValue, rightValue);
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        //Expr expression = new Add(new Value(2), new Value(3));
        //Expr expression2 = new Sub(new Mul(new Value(2), new Value(3)), new
Value(4));
        System.out.println("Welcome in calculation with tree\n");
        var input = new Scanner(System.in);
        var mainExpr = Expr.parse(input);

        System.out.println(mainExpr.stringify().toString() + " = " + mainExpr.eval());
    }
}
```