Exercice 1 – « Gestion d'exceptions »

Modifier le programme précédent pour qu'il puisse gérer des exceptions. Ajouter les exceptions suivantes :

- `ExpressionException`, si l'expression mathématique postfixée est mal formée. Cette exception devra être levée à chaque fois que les erreurs décrites au précédent TP sont rencontrées. La méthode `eval()` ne lance que des exceptions de type `ExpressionException`.
- `PileVideException`, si on tente d'accéder à la pile et qu'elle est vide. Les méthodes `sommet()` et `depiler()` de l'interface `Pile` sont concernées.
- `MathématiquesException` lorsque qu'une erreur de calcul se produit. Les classes `DivisionZeroException` et `RacineNegativeException` héritent de cette classe d'exception. Les méthodes `calcul(Double a, Double b)` et `calcul(Double a)` de la classe `Operateur` sont susceptibles de lever une telle exception.
- `DivisionZeroException`, si une division par zéro se produit.
- `RacineNegativeException`, si le calcul de la racine d'un nombre négatif se produit.
- `OperateurException`, si à la construction d'une instance de `Operateur` la chaîne de caractères correspondant à l'opérateur n'en est pas un, ou bien si une opération de calcul ne peut s'appliquer à un opérateur (méthodes `calcul()`).

Exercice 2 – « Entrées / sorties »

Modifier la fonction `main()` pour qu'elle ne lise plus une expression envoyée en paramètre, mais une expression entrée dans la console une fois le programme exécuté.

ANNEXES

Mécanisme

Une exception est un objet qui est instancié lors d'un incident : on dit qu'une exception est levée (thrown).
Le traitement du code est interrompu et l'exception est propagée à travers la pile d'exécution de méthode appelée en méthode appelante. Si aucune méthode ne capture l'exception, celle-ci remonte jusqu'à la méthode de fond de pile (la méthode `main()`), et le programme se termine avec une erreur.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Operations.division(Attrape.java:3)
        at Operations.main(Attrape.java:18)
```

Lever une exception

```
class Operations {
    public static double division(double a, double b) {
        return a / b;
    }
    public static void main(String[] args) {
        System.out.println(division(
            Double.parseDouble(args[0]),
            Double.parseDouble(args[1])
        ));
    }
}
```

Dans le code précédent, on aimerait s'assurer que *b* est bien différent de 0, et si c'est le cas, renvoyer une erreur. On ne peut se contenter d'afficher l'erreur dans le console : on aimerait pouvoir savoir s'il y a une erreur et la traiter en cas de problème de manière automatique, de façon à ce que tout soit géré par le programme.
Il faut que la méthode `division()` puisse 'lever' un objet spécial qui indique qu'une erreur a eu lieu. Pour cela, on utilise le mot-clé **throw** suivi de l'objet à lever.

En Java, tout objet levable hérite de la classe `java.lang.Throwable`. L'API Java distingue deux catégories d'objets levables :

- la sous-classe `java.lang.Error` de `java.lang.Throwable` correspond à des erreurs graves (manque de mémoire,…) qui conduisent généralement à l'arrêt du programme ;
- la sous-classe `java.lang.Exception` de `java.lang.Throwable` correspond à des événements anormaux que l'on souhaite traiter (capturer) pour qu'ils ne provoquent pas l'arrêt du programme.

Il faut de plus indiquer que la fonction peut lever un certain type d'objet. On utilise pour cela le mot-clé **throws** au niveau de la signature de la fonction. On peut aussi indiquer que la méthode peut lancer différentes classes d'exceptions, en les séparant par des virgules.
La méthode `division()` précédente devient donc :

```
public static double division(double a, double b)
        throws Exception {
    if (b == 0)
        throw new Exception("Division par zéro");
    return a / b;
}
```

Il existe un grand nombre de classes d'exceptions déjà définies dans l'API. Le programmeur peut créer ses propres classes d'exceptions en étendant la classe `java.lang.Exception`.

```
class DivisionZeroException extends Exception {
    public DivisionZeroException() {
        super();
    }
    public DivisionZeroException(String message) {
        super(message);
    }
    public DivisionZeroException(Throwable cause) {
        super(cause);
    }
    public DivisionZeroException(String message,Throwable cause){
        super(message,cause);
    }
}
```

La variable *cause* permet de spécifier la raison du déclenchement de cette exception, généralement une autre exception précédemment capturée. Elle n'a donc pas vraiment de sens pour une division par zéro.

Capturer une exception

Une fois levée, une exception va se propager jusqu'à la méthode `main()`. Pour pouvoir traiter l'erreur sans interrompre le programme, il faut capturer l'exception levée. Pour cela, on utilise en Java deux clauses spéciales :

- **try** : bloc dans lequel on désire capturer des exceptions éventuellement levées. Les instructions du bloc sont lues et exécutées jusqu'à la première exception levée. Le contrôle est alors passé à une clause **catch**.
- **catch** : bloc dans lequel on gère les exceptions capturées. Pour cela, on commence par spécifier le nom et le type de l'exception à capturer, puis le bloc d'instructions à exécuter en cas de capture. Il peut y avoir plusieurs blocs **catch** pour un seul bloc **try**.

On capture l'exception levée par la méthode `division` dans la méthode `main()`. On en profite pour gérer la validité des arguments :

- si un argument n'est pas un nombre `Double.parseDouble()` levera une exception de type `NumberFormatException` ;
- si trop peu d'arguments ont été spécifiés, `args[0]` levera une exception de type `IndexOutOfBoundsException`.

Ces deux classes d'exceptions héritent de la classe `Exception` et seront donc traitées par le dernier bloc **catch**.

```
class Operations {
    public static double division(double a, double b)
        throws DivisionZeroException {
        if (b == 0)
            throw new DivisionZeroException("Division par zéro");
        return a / b;
    }
    public static void main(String[] args) {
        try {
            System.out.println(division(
                Double.parseDouble(args[0]),
                Double.parseDouble(args[1])
            ));
        }
        catch (DivisionZeroException dze) {
            System.err.println("Division par zéro.");
        }
        catch (Exception e) {
            System.err.println(
                e.getClass().getName() + ": " + e.getMessage()
            );
        }
    }
}
```

Le programme précédent affichera :

```
$java Operations 3 2
1.5

$java Operations 3 0
Division par zéro.

$java Operations 3 abc
java.lang.NumberFormatException: For input string: "abc"

$java Operations 3
java.lang.ArrayIndexOutOfBoundsException: 1
```

À noter que contrairement à une exception de type `DivisionZeroException`, l'exception précédente n'a pas besoin d'être attrapée pour le programme compile. En effet, l'exception `IndexOutOfBoundsException` hérite d'une classe spéciale définie dans l'API Java, `RuntimeException`. Cette classe désigne les exceptions capturées au moment de l'exécution. Leur particularité est donc qu'il n'est pas nécessaire des les attraper pour que le programme compile. On parle alors d'exceptions non contrôlées.
Attention avec l'utilisation de cette classe d'exceptions : le compilateur ne renverra aucune erreur si une exception non contrôlée n'est pas capturée. Si une exception non contrôlée est levée, elle conduira généralement à l'arrêt du

programme. Les exceptions non contrôlées sont donc généralement des exceptions de débuggage pour indiquer un code mal écrit qui ne prend pas en compte tous les cas de figure.

Lire la console

Comme la plupart des langages de programmation, Java permet de lire des commandes entrées dans la console pendant l'exécution du programme.
Pour lire une chaîne *str*, on utilise le code suivant :

```
Scanner reader = new Scanner(System.in);
String str = reader.readLine();
reader.close();
```

Pour l'adapter au programme précédent, on écrirait alors :

```
import java.io.Console;

class Operations {
    public static double division(double a, double b)
        throws DivisionZeroException {
        ...
    }
    public static void main(String[] args) {
        try {
            Scanner reader = new Scanner(System.in);
            Double a = Double.parseDouble(console.readLine());
            Double b = Double.parseDouble(console.readLine());
            System.out.println(division(a,b));
            reader.close();
        }
        catch (DivisionZeroException dze) {
            System.err.println("Division par zéro.");
        }
        catch (Exception e) {
            System.err.println(
                e.getClass().getName() + ": " + e.getMessage()
            );
        }
    }
}
```

À noter que dans les versions précédentes de Java, lire une chaîne de caractères depuis la console était beaucoup plus complexe. Il est possible de rencontrer du code semblable à celui-ci :

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

class Operations {
    public static double division(double a, double b)
        throws DivisionZeroException {
        ...
    }
    public static void main(String[] args) {
        try {
            InputStreamReader isr =
                        new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            Double a = Double.parseDouble(br.readLine());
            Double b = Double.parseDouble(br.readLine());
            System.out.println(division(a,b));
        }
        catch (DivisionZeroException dze) {
            System.err.println("Division par zéro.");
        }
        catch (IOException e) {
            System.err.println(
                "Erreur lors de la lecture des arguments."
            );
        }
        catch (Exception e) {
            System.err.println(
                e.getClass().getName() + ": " + e.getMessage()
            );
        }
    }
}
```