calculator\Operator.java

```java
/**
 * @author  Sebastian Diaz & Guillaume Dunant
 * Date   : 16.11.2023
 * Fichier: Operator.java
 */
package calculator;

/**
 * Classe abstraite permettant de représenter
 *  une opération de la caluclatrice à effectuer
 */
abstract class Operator {
    protected State state;
    /**
     * Méthode qui sera appelée pour effectuer l'opération
     */
    abstract void execute();

    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public Operator(State s){
        state = s;
    }
}

/**
 * Opération rajoutant un chiffre à la valeur courante
 */
class Digit extends Operator {
    private String value;
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     * @param val Valeur qui sera ajoutée à la valeur courante
     */
    public Digit(State s, int val){
        super(s);
        value = "" + val;
    }

    public void execute() {
        if(state.noError()){
            if(!state.isUserInput()) {
                state.pushCurrent();
                state.setUserInput(true);
            }
            state.appendToCurrent(value);
        }
    }
}

/**
 * Opération ajouant un point à la valeur courante
 */
```

```java
class Point extends Operator {
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public Point(State s) {
        super(s);
    }

    public void execute() {
        if (state.noError() && state.isUserInput()) {
            state.appendToCurrent(".");
        }
    }
}

/**
 * Opération permettant d'enlevé un caractère de la valeur courante
 */
class Backspace extends Operator {
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public Backspace(State s) {
        super(s);
    }

    public void execute() {
        if(state.noError() && state.isUserInput()){
            state.removeACharFromCurrent();
        }
    }
}

/**
 * Classe abstraite représentant les opérations nécessitant deux opérandes
 */
abstract class DoubleOperation extends Operator {
    /**
     * Constructeur
     * @param s Etat interne de la calculatrice
     */
    public DoubleOperation(State s) {
        super(s);
    }

    /**
     * Opération qui sera effectuée entre les deux valeurs
     * @param d1 Valeur 1
     * @param d2 Valeur 2
     * @return Résultat de l'opération
     */
    abstract protected Double operate(Double d1, Double d2);

    public void execute() {
        if(state.noError()){
            Double d1 = state.getCurrent();
            Double d2 = state.getStackValue();
```

```
116
117             if(d1 != null && d2 != null){
118                 state.setCurrent(operate(d1, d2));
119                 state.setUserInput(false);
120             }
121             else{
122                 state.setError();
123             }
124         }
125     }
126 }
127
128 /**
129  * Opération d'addition
130  */
131 class Addition extends DoubleOperation {
132     /**
133      * Constructeur
134      * @param s Etat interne de la calculatrice
135      */
136     public Addition(State s) {
137         super(s);
138     }
139
140     protected Double operate(Double d1, Double d2) {
141         return d1 + d2;
142     }
143 }
144
145 /**
146  * Opération de soustraction
147  */
148 class Subtraction extends DoubleOperation {
149     /**
150      * Constructeur
151      * @param s Etat interne de la calculatrice
152      */
153     public Subtraction(State s) {
154         super(s);
155     }
156
157     protected Double operate(Double d1, Double d2) {
158         return d1 - d2;
159     }
160
161 }
162
163 /**
164  * Opération de multiplication
165  */
166 class Multiplication extends DoubleOperation {
167     /**
168      * Constructeur
169      * @param s Etat interne de la calculatrice
170      */
171     public Multiplication(State s) {
172         super(s);
173     }
174
```

```
175     protected Double operate(Double d1, Double d2) {
176         return d1 * d2;
177     }
178 }
179 
180 /**
181  * Opération de division
182  */
183 class Division extends DoubleOperation {
184     /**
185      * Constructeur
186      * @param s Etat interne de la calculatrice
187      */
188     public Division(State s) {
189         super(s);
190     }
191 
192     protected Double operate(Double d1, Double d2) {
193         return d2 / d1;
194     }
195 }
196 
197 /**
198  * Classe abstraite représentant les opérations nécessitant une seule opérande
199  */
200 abstract class UnaryOperation extends Operator {
201     /**
202      * Constructeur
203      * @param s Etat interne de la calculatrice
204      */
205     public UnaryOperation(State s) {
206         super(s);
207     }
208 
209     /**
210      * Operation qui sera effectuée sur la valeur courante
211      * @param d1 Valeur qui sera modifiée
212      * @return Nouvelle valeur
213      */
214     abstract protected Double operate(Double d1);
215 
216     public void execute() {
217         if(state.noError()){
218             Double d1 = state.getCurrent();
219             if(d1 != null){
220                 state.setCurrent(operate(d1));
221                 state.setUserInput(false);
222             }
223             else{
224                 state.setError();
225             }
226         }
227     }
228 }
229 
230 /**
231  * Opération pour retourner la racine carrée de la valeur courante
232  */
233 class SquareRoot extends UnaryOperation {
```

```
234     /**
235      * Constructeur
236      * @param s Etat interne de la calculatrice
237      */
238     public SquareRoot(State s) {
239         super(s);
240     }
241 
242     protected Double operate(Double d1) {
243         return Math.sqrt(d1);
244     }
245 }
246 
247 /**
248  * Opération pour retourner le carré de la valeur courante
249  */
250 class Power extends UnaryOperation {
251     /**
252      * Constructeur
253      * @param s Etat interne de la calculatrice
254      */
255     public Power(State s) {
256         super(s);
257     }
258 
259     protected Double operate(Double d1) {
260         return d1 * d1;
261     }
262 }
263 
264 /**
265  * Opération pour changer le signe de la valeur courante
266  */
267 class Negate extends Operator{
268     /**
269      * Constructeur
270      * @param s Etat interne de la calculatrice
271      */
272     public Negate(State s) {
273         super(s);
274     }
275 
276     public void execute(){
277         if(state.noError()){
278             if(state.isUserInput()){
279                 state.negateCurrent();
280             }
281             else{
282                 state.setCurrent(-state.getCurrent());
283             }
284         }
285     }
286 }
287 
288 /**
289  * Opération pour obtenir l'inverse de la valeur courante
290  */
291 class Inverse extends UnaryOperation {
292     /**
```

```
293      * Constructeur
294      * @param s Etat interne de la calculatrice
295      */
296     public Inverse(State s) {
297         super(s);
298     }
299 
300     protected Double operate(Double d1) {
301         return 1 / d1;
302     }
303 }
304 
305 /**
306  * Opération pour réinitialiser la valeur courante et la stack
307  */
308 class Clear extends Operator {
309     /**
310      * Constructeur
311      * @param s Etat interne de la calculatrice
312      */
313     public Clear(State s) {
314         super(s);
315     }
316 
317     public void execute() {
318         state.emptyStack();
319         state.rstState();
320     }
321 }
322 
323 /**
324  * Opération pour réinitialiser la valeur courante
325  */
326 class ClearError extends Operator {
327     /**
328      * Constructeur
329      * @param s Etat interne de la calculatrice
330      */
331     public ClearError(State s) {
332         super(s);
333     }
334 
335     public void execute() {
336         state.rstState();
337     }
338 }
339 
340 /**
341  * Opération pour stocker une valeur en mémoire
342  */
343 class MemoryStore extends Operator {
344    /**
345      * Constructeur
346      * @param s Etat interne de la calculatrice
347      */
348     public MemoryStore(State s) {
349         super(s);
350     }
351 
```

```
352     public void execute() {
353         if(state.noError()){
354             state.storeInMemory();
355             state.setUserInput(true);
356         }
357     }
358 }
359 
360 /**
361  * Opération pour récupérer la valeur stockée en mémoire
362  */
363 class MemoryRecall extends Operator {
364     /**
365      * Constructeur
366      * @param s Etat interne de la calculatrice
367      */
368     public MemoryRecall(State s) {
369         super(s);
370     }
371 
372     public void execute() {
373         if(state.noError()){
374             state.getMemory();
375             state.setUserInput(false);
376         }
377 
378     }
379 }
380 
381 /**
382  * Opération pour ajouter la valeur courante sur la stack
383  */
384 class Enter extends Operator {
385     /**
386      * Constructeur
387      * @param s Etat interne de la calculatrice
388      */
389     public Enter(State s) {
390         super(s);
391     }
392 
393     public void execute() {
394         if(state.noError()){
395             state.pushCurrent();
396             state.setUserInput(true);
397         }
398     }
399 }
```