

This assignment will be closed on June 20, 2023 (23:59:59).

On June 20, 2023 (15:58:05), we had submissions for the following questions: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 — [Details](#) → (/agns/INF371/TD08/2022/uploads/:my/).

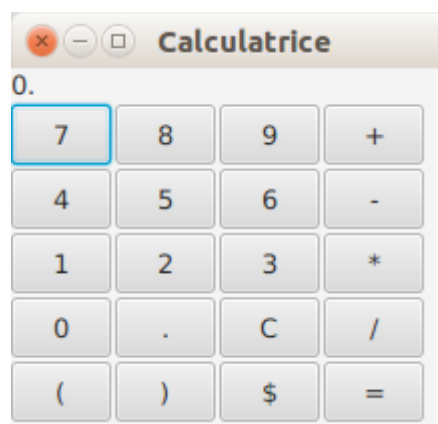
# Calculatrice à mémoire avec interface graphique

Xavier Rival, Jean-Marie Madiot, Julien Signoles, Benjamin Werner

- Le coeur de notre calculatrice : opérations de base
  - Description de l'état de la calculatrice
  - Calcul des opérations de base
  - Commandes pour le contrôle de l'évaluation d'expressions
  - Mémoire
- Une calculatrice textuelle
  - Première version du tokenizer
  - Gestion des opérateurs et parenthèses
  - Nombres décimaux et nombres négatifs
  - Ré-initialisation et mémoire
- Une calculatrice graphique
  - Application et fenêtre graphiques
  - Début
  - Boutons
  - Clavier

Le but de ce TD sur deux séances est double : dans un premier temps, il permettra de programmer le coeur d'une calculatrice capable d'évaluer des opérations de bas niveau tandis que, dans un second temps, il donnera l'occasion de développer deux interfaces, l'une textuelle et l'autre graphique. Il fournit donc l'occasion de s'initier à la programmation graphique et événementielle.

Nous allons ainsi construire progressivement une calculatrice qui ressemblera à ceci :



## Le coeur de notre calculatrice : opérations de base

Cette partie a pour but de concevoir le coeur de la calculatrice, qui en stockera l'état interne et pourra effectuer tous les calculs. Cette calculatrice traitera des expressions structurées qui seront écrites suivant une syntaxe classique que l'on étudiera dans la partie suivante. Pour gérer l'évaluation pas à pas de telles expressions, la calculatrice va devoir maintenir un

état interne décrivant les valeurs et opérations entrées par l'utilisateur. À ce stade, il n'est pas encore question de lire une expression en entrée mais simplement de traiter une suite de "commandes" (comme la lecture d'un nombre ou d'un opérateur) qui seront envoyées à la calculatrice dans l'ordre de lecture de l'expression, de gauche à droite.

Description de l'état de la calculatrice

Pour décrire l'état de la calculatrice, nous allons utiliser deux piles comme représentation de l'état de la calculatrice. La première, `numbers`, sert à enregistrer les nombres décimaux à utiliser dans les prochains calculs : lorsque toutes les opérations ont été calculées, elle contient un seul nombre qui est le résultat final du calcul. La seconde pile, `operators`, contient, sous une forme symbolique, les opérations à effectuer. Par exemple, après la lecture de l'expression `5 + 12.34 * 2`, et avant d'effectuer le moindre calcul, `numbers` est une pile contenant (du bas vers le haut) `5`, `12.34` et `2`, alors qu'`operators` contient `Operator.PLUS` et `Operator.MULT`, représentations symboliques de `+` et `*` respectivement.

Pour représenter les piles, nous allons utiliser celles fournies par la bibliothèque standard de Java, `java.util.Stack<E>` dont la documentation est disponible en ligne (<https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html>). Il s'agit d'une classe générique paramétrée par le type `E` des éléments contenus dans la pile. Dans la suite, la première pile sera de type `java.util.Stack<Double>` pour contenir des doubles, tandis que la seconde sera de type `java.util.Stack<Operator>` pour contenir des opérateurs.

Exercice 1

- Créer un fichier `Operator.java` contenant le type énuméré définissant la représentation symbolique des opérateurs :

```
public enum Operator { PLUS, MINUS, MULT, DIV }
```

- Créer une classe `Calculator` contenant les deux piles `numbers` et `operators`. La première est de type `java.util.Stack<Double>` tandis que la seconde est de type `java.util.Stack<Operator>`. Définir un constructeur sans argument pour cette classe les initialisant. Surcharger également la fonction `toString` de `Calculator` de façon à retourner le contenu des deux piles sous forme d'une chaîne de caractères. Si `numbers` (resp. `operators`) contient (de bas en haut) `2`, `12.34` et `5` (resp. `*` et `+`), la chaîne retournée sera :

```
[2.0, 12.34, 5.0]
[MULT, PLUS]
```

Vous pourrez utiliser la méthode `.toString()` de la classe `java.util.Stack<?>`.

- Ajouter deux méthodes `void pushDouble(double d)` et `void pushOperator(Operator o)` à la classe `Calculator`.
- Écrire une méthode publique `getResult` qui retourne le sommet de `numbers` (sans en modifier le contenu), ou bien qui lève une exception (on utilisera `RuntimeException`) dans le cas où celle-ci est vide.

Il serait naturel que les champs `numbers` et `operators` aient une visibilité `private`. Cependant, pour permettre aux tests en ligne de vérifier le bon fonctionnement de vos classes, il est exceptionnellement demandé de les rendre `public`.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (14:58:46)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java

## Calcul des opérations de base

### Exercice 2

Nous allons maintenant nous intéresser aux opérations arithmétiques et à leur évaluation.

- Écrire une méthode publique `void executeBinOperator( Operator op )` qui applique l'opération `op` aux deux arguments placés au sommet de la pile `numbers` après avoir retiré ceux-ci, et qui placera le résultat au sommet de la pile (dans cette méthode on ignore la pile `operators` ).

On suppose qu'on souhaite évaluer l'expression `1+2-3` . Placer `1` , `2` et `3` sur la pile, puis évaluer `-` et enfin évaluer `+` . Que pensez vous du résultat ?

Faire de même pour `1-2+3` . Placer `1` , `2` et `3` sur la pile, puis évaluer `+` et enfin évaluer `-` . Qu'observez vous ?

Déposer ici vos fichiers :

Last submission: June 06, 2023 (15:10:50)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions 

Expected files: Calculator.java, Operator.java

### Exercice 3

Pour prendre en compte correctement la priorité de chaque opérateur, nous allons devoir utiliser intelligemment la pile `operators` , suivant l'algorithme vu en cours. En particulier, lorsque la calculatrice recevra une “commande” correspondant à un opérateur `op` , on commencera par évaluer une partie des opérateurs déjà sur la pile `operators` avant de placer `op` au sommet de cette même pile ([algorithme Shunting-yard](https://fr.wikipedia.org/wiki/Algorithme_Shunting-yard) ([https://fr.wikipedia.org/wiki/Algorithme\\_Shunting-yard](https://fr.wikipedia.org/wiki/Algorithme_Shunting-yard))). Ainsi, nous allons obtenir une calculatrice capable à ce stade de recevoir deux types de “commandes”, qui correspondent respectivement à l'entrée par l'utilisateur d'un nombre et d'un opérateur, et dans la suite nous allons enrichir ce langage de commandes.

- Écrire une méthode statique privée `precedence` prenant un opérateur en argument et retournant un entier correspondant à la priorité de l'opérateur. Par exemple, on doit avoir `precedence(Operator.PLUS)` strictement inférieur à `precedence(Operator.MULT)` .
- Écrire une méthode `void commandOperator(Operator op)` qui met à jour l'état de la calculatrice lorsque celle-ci reçoit la commande qui correspond à cet opérateur. Ajouter également une méthode `void commandDouble(double d)` qui met à jour l'état de la calculatrice lorsque celle-ci reçoit une telle commande.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (16:35:03)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions 

Expected files: Calculator.java, Operator.java

## Commandes pour le contrôle de l’évaluation d’expressions

### Exercice 4

Nous ajoutons maintenant quelques commandes qui vont rendre notre calculatrice vraiment utilisable, avec tout d’abord l’opérateur d’évaluation `=`, puis les parenthèses et enfin la possibilité de réinitialiser l’état de la calculatrice.

Ajouter une méthode `void commandEqual()` qui évalue l’ensemble des opérations actuellement sur la pile. Tester avec quelques expressions (dont les deux exemples fournis plus haut).

Déposer ici vos fichiers :

Last submission: June 06, 2023 (16:35:22)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java

### Exercice 5

Ajouter deux méthodes `void commandLPar()` et `void commandRPar()` décrivant l’ouverture et la fermeture de parenthèses. On veillera à évaluer correctement le contenu d’une paire de parenthèses. On recommande pour cela d’ajouter un élément `OPEN` au type énuméré `Operator`, et décrivant la position d’une parenthèse ouvrante sur la pile des opérateurs.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (16:50:06)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java

### Exercice 6

Ajouter une méthode `void commandInit()` qui effectue la réinitialisation de l’état de la calculatrice.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (16:51:08)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java

## Mémoire

### Exercice 7

Ici, nous allons sauvegarder les résultats de nos calculs dans une liste afin de pouvoir y accéder ensuite via des variables entières : la variable `$i` représente le résultat du  $i$ -ème dernier calcul effectué (via la commande `=`). Ainsi, interpréter `"1+2*3=$1+$1+1=$1-$2="` doit produire un état où `8.0` est au sommet de la pile. Les résultats ne doivent pas être effacés par un appel à `commandInit()`.

- Ajouter un attribut `java.util.LinkedList<Double> results` dans la classe `Calculator`. Mettre à jour la méthode `commandEqual` pour stocker le résultat de chaque évaluation.
- Ajouter une méthode `void commandReadMemory(int i)` qui place au sommet de la pile `numbers` la valeur du  $i$ -ème précédent calcul.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (16:57:22)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java

## Une calculatrice textuelle

Dans cette partie, nous allons réaliser une première interface, qui repose sur la lecture de chaînes de caractères passées en argument. C’est un premier pas vers une calculatrice plus facile à utiliser, puisque nous allons bientôt pouvoir lui demander d’évaluer des formules du type `2+3.5*(8-4.2)=`.

Pour cela, nous allons écrire une autre classe appelée `Tokenizer` qui va prendre en charge la lecture de chaînes de caractères dont elle va extraire des commandes de base (souvent appelées “tokens”, d’où son nom). L’extraction de commandes à partir d’une chaîne de caractères n’est pas une opération simple, donc le tokenizer doit maintenir un état interne décrivant le contexte à un instant de la lecture, et mettre à jour cet état à chaque caractère rencontré.

### Première version du tokenizer

Pour commencer, on considère que l’état interne du tokenizer est décrit par la donnée :

- d’un booléen `isStart`, qui est vrai si et seulement si on est au début de la lecture d’une expression (il doit donc être vrai au début, et après évaluation d’une commande `=`) ;
- d’un booléen `isIntNum`, qui est vrai si et seulement si on a commencé à lire un entier ;
- d’un nombre (de type `double`) `num` qui décrit la valeur du nombre en cours de lecture (lorsque le tokenizer est en train de lire un nombre) ; et
- de la donnée de l’état courant `calc` de la calculatrice.

#### Exercice 8

- Définir la classe `Tokenizer` (avec les quatre champs ci-dessus) et un constructeur qui en initialise l’état interne.
- Ajouter une méthode publique `void readChar(char c)` à la classe `Tokenizer`, qui met à jour l’état interne du `Tokenizer` lors de la lecture du caractère `c`, qui ne pourra être qu’un chiffre ou le symbole `=` (nous allons gérer un ensemble de caractères plus important dans les questions suivantes – dans la mesure où le problème est relativement complexe, nous vous guidons pas à pas).

Pour vous guider, on vous donne quelques fonctions de la librairie standard de Java qui pourraient vous être utiles. La fonction `Character.getNumericValue` permet d’obtenir la valeur numérique d’un caractère (`char`). Ainsi par exemple, `Character.getNumericValue('6')` retourne l’entier 6. La fonction `Character.isDigit` permet de déterminer si un `char` décrit un chiffre. Enfin, `Math.pow(x,y)` calcule  $x$  à la puissance  $y$ .

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:04:59)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Exercice 9

Ajouter une méthode publique `void readString(String s)` qui met à jour l'état du tokenizer après avoir lu les caractères de `s` un à un.

Il pourra également être utile d'ajouter un booléen activant un mode *debug* pour afficher l'état interne du tokenizer à chaque pas.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:06:24)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Gestion des opérateurs et parenthèses

Nous allons maintenant étendre `readChar` afin de gérer les autres opérations de notre calculatrice.

Exercice 10

Modifier `readChar` afin de supporter les opérateurs binaires `+`, `-`, `*`, et `/`. Lors de la lecture d'un opérateur, on commencera par passer à la calculatrice la commande qui correspond au nombre en cours de lecture, puis on lui passera la commande correspondant à l'opérateur courant, et on mettra à jour les autres champs du tokenizer pour en maintenir la cohérence.

On conseille d'éviter de dupliquer du code. Pour faciliter la mise au point du tokenizer, on pourra factoriser certaines opérations, comme la finalisation de la lecture d'un nombre (lors de la lecture d'une opération).

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:08:35)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Exercice 11

Modifier `readChar` pour gérer les parenthèses ouvrantes et fermantes.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:09:20)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Nombres décimaux et nombres négatifs

Jusqu’ici, notre tokenizer n’accepte que les nombres entiers et positifs ce qui n’est bien sûr pas satisfaisant. Nous avons repoussé un peu le traitement des autres valeurs car celui-ci complique l’état du tokenizer...

Exercice 12

Modifier `readChar` pour lire les nombres avec partie décimale (délimitée par le caractère `.`). On pourra ajouter deux champs à l’état interne du tokenizer :

- un booléen `isNonIntNum` vrai si et seulement si on est en train de lire un nombre décimal, dont on a déjà dépassé le point ; et
- un entier `decimalDigits` comptant le nombre de chiffres après ce point.

Il faudra prendre soin aussi de modifier les autres fonctions du tokenizer.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:13:56)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Exercice 13

Modifier `readChar` pour lire les nombres négatifs, comme `-3` ou `-.053` . Il faudra faire attention au caractère `-` qui a maintenant deux sens possibles, soit comme opérateur binaire, soit comme comme négation unaire. On note qu’un `-` observé tout au début de la lecture, ou bien juste après une parenthèse ouvrante, après un caractère `=` ou après un opérateur binaire décrit nécessairement la négation unaire. On pourra ajouter deux champs à l’état interne du tokenizer :

- un boolean `isMinUnary` vrai si et seulement si une occurrence de `-` serait vue comme un opérateur unaire ; et
- un booléen `isNeg` vrai si et seulement si on est en train de lire un nombre négatif.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:36:58)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Ré-initialisation et mémoire

Exercice 14

Notre interface textuelle est presque prête. Nous allons maintenant prendre en compte les caractères gérant les deux dernières commandes.

Modifier `readChar` afin de traiter le caractère `C` comme une commande réinitialisant l'état de la calculatrice.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:38:14)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Exercice 15

Modifier `readChar` afin de traiter la chaîne `$i` comme une lecture dans la mémoire comme défini plus haut (on pourra étendre la définition de l'état interne du tokenizer). Cette question étant plus difficile, on pourra la considérer facultative.

Déposer ici vos fichiers :

Last submission: June 06, 2023 (17:41:22)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions ?

Expected files: Calculator.java, Operator.java, Tokenizer.java

Une calculatrice graphique

Dans cette dernière partie, nous allons créer une interface graphique (GUI) pour notre calculatrice. Cette seconde interface va transformer des actions de l'utilisateur (actions sur des boutons à l'aide de la souris ou actions sur des touches au clavier) en commandes. Elle repose donc sur un principe similaire à l'interface textuelle, même si la partie visible est bien sûr entièrement différente.

Application et fenêtre graphiques

Nous allons utiliser la bibliothèque `javafx`. Il vous faudra utiliser plusieurs constructions fournies par cette librairie, et parfois rechercher les classes et méthodes adaptées. Pour cela, vous devrez vous référer à la [Documentation en ligne de javafx](https://docs.oracle.com/javase/8/javafx/api/toc.htm) (<https://docs.oracle.com/javase/8/javafx/api/toc.htm>).



Notre classe étendra la classe Application (<https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html>). Une instance de notre classe sera donc une fenêtre graphique possédant des décorations (bordure, titre, ...) et des boutons permettant de la fermer, la miniaturiser ou l'agrandir. Pour lancer une GUI, une méthode `main` est nécessaire (comme pour les applications sans GUI). Cette méthode doit appeler avec ses propres arguments la méthode `launch`, qui est fournie et qu'on n'aura pas besoin de définir. Au travers `launch`, `javafx` préparera alors la fenêtre et appellera la méthode `start(Stage stage)` qu'il nous faut définir. Le code de votre GUI ressemblera donc au code suivant (les parties `TODO` seront complétées par la suite) :

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.input.KeyEvent;

public class GraphicsCalculator extends Application {
    Tokenizer tok;

    @Override
    public void start(Stage stage) {
        stage.show();
        // TODO
        Scene scene = new Scene(new VBox(/* TODO */));
        stage.setScene(scene);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

La première étape est de vous assurer que vous arrivez à utiliser JavaFX, le programme ci-dessus devrait ouvrir une fenêtre blanche.

## VSCode

- Méthode 1 (uniquement sur les ordinateurs de l'école). Quittez VSCode s'il est déjà lancé. Ouvrez un terminal, et relancez VSCode à l'aide de la commande

```
env JAVA_HOME=/usr/java/jdk1.8.0_301-amd64 code &
```

- Méthode 2. Créez un nouveau projet Java. Au lieu de l'usuel `No build tools`, sélectionnez `JavaFX`. Certaines étapes peuvent prendre du temps. Le `group id` est le nom du package (vous ne pourrez pas utiliser le package par défaut par cette méthode). Le `artifact id` n'a pas d'importance. Attention, il va poser des questions dans le terminal, il suffit d'accepter les réponses par défaut avec la touche `entrée`. Il peut falloir cliquer sur `open` pour finir. Vous devriez vous retrouver avec un exemple fonctionnel (vous pouvez le lancer). Il ne reste qu'à copier vos fichiers au même endroit (vous aurez probablement besoin d'ajouter une ligne `package` à chaque fichier java).
- Méthode 3. Suivez les [instructions \(https://openjfx.io/openjfx-docs/\)](https://openjfx.io/openjfx-docs/) (sélectionner `JavaFX and Visual Studio Code`, `Non-modular from IDE`).

## Eclipse

- Méthode 1 (uniquement sur les ordinateurs de l'école). Créez un nouveau projet Java. Dans la page qui demande le nom du projet, cliquez sur `Configure JREs`, `Add`, `Standard VM`, puis comme `JRE_HOME` sélectionner `/usr/java/jdk1.8.0_301-amd64`. Vous devez maintenant avoir une ligne de plus dans la liste des `Installed JREs` (vous pouvez cocher cette ligne). Cliquez à gauche sur `Compiler` et réglez `Compiler compliance level` à 1.8. De retour sur la page de création de projet, assurez-vous que le nouveau JRE est bien sélectionné, soit comme `default JRE` (si vous avez coché la ligne précédemment), soit comme `project specific JRE`.

- Méthode 2. Suivez les [instructions \(https://openjfx.io/openjfx-docs/\)](https://openjfx.io/openjfx-docs/) (sélectionner JavaFX and Eclipse , Non-modular from IDE ).

Si vous avez une erreur de compilation de la forme

The type 'Application' is not API (restriction on required library rt.jar)

exécutez les actions suivantes :

- Allez dans les propriétés de votre projet (clic droit sur le nom de votre projet, puis `Properties` ) ;
- Allez dans `Java Build Path` , puis dans le tab `Libraries` ;
- Dépliez `JRE System Library` et sélectionnez `Access rules` ;
- Cliquez sur `Edit...` , puis sur `Add...` . Positionnez `Resolution` sur `Accessible` et entrez `javafx/**` dans `Rule Pattern` ;
- Validez le tout.

Début

On vous demande de :

1. Modifier `GraphicsCalculator` de manière à changer le titre de votre GUI en utilisant une méthode de `stage` .

2. Définissez la largeur et la longueur de la fenêtre (par exemple 200x200) en utilisant des méthodes de `stage` .
- La structure de la fenêtre va comme suit : `stage` contient une `Scene scene` , qui contient une `VBox` , boîte d’alignement verticale, au constructeur de laquelle on pourra fournir des `HBox` s, respectivement horizontales, auxquelles on pourra fournir un `Label` pour le résultat et des `Button` s pour les entrées.
3. Ajouter à votre classe `GraphicsCalculator` un champ de type `Label` correspondant à la zone de résultat votre calculatrice et ajouter ce dernier dans une `HBox` , qui sera la première ligne donnée au constructeur de `VBox` .
- Boutons
4. Ajouter une méthode `Button b(char c)` (non statique), qui renvoie un nouveau bouton de taille constante, par exemple 30x30, en utilisant `setMinSize` et `setMaxSize` . Utilisez-la une fois par caractère nécessaire à votre calculatrice, en répartissant les boutons sur plusieurs lignes `HBox` en ajoutant ces dernières à `VBox` .

5. Placer les boutons et la zone de texte de votre calculatrice de manière similaire à [l’image du début de l’énoncé](#), l’opérateur `'$'` étant optionnel.

6. Définir une méthode `update(char c)` qui interprète le caractère `c` et affiche le résultat dans la zone textuelle.

7. Si `b` est un bouton, alors `b.setOnAction(value -> /*TODO*/);` exécutera `/*TODO*/` à chaque clic du bouton (on appelle cette notation “lambda-notation” et cela correspond à la définition d’une fonction prenant en argument `value` et qui évalue l’expression qui remplacera `/*TODO*/` ). Faites en sorte qu’appuyer sur un bouton appelle la méthode `update` avec le caractère correspondant, et testez votre calculatrice.
- Clavier
- Les événements sont les interactions avec l’utilisateur. Ils rendent par exemple un programme réactif aux clics de souris ou au fait qu’une touche du clavier soit pressée ou relâchée.
8. Implémenter un gestionnaire d’événements de façon à mettre correctement à jour votre calculatrice lorsqu’une touche du clavier est pressée. Pour cela, on ajoutera l’écouteur d’événements clavier à la scène en cours :  
`scene.setOnKeyTyped(e -> handlekey(e));` . Implémentez la méthode `void handlekey(KeyEvent e)` et testez votre calculatrice. Certaines touches spéciales ( `Entrée` , `Échap` ) sont plus compliquées à gérer et on pourra les ignorer, au moins dans un premier temps.


Déposer ici vos fichiers (le test va échouer, ignorez-le, le but est seulement de fournir les fichiers) :

Last submission: June 13, 2023 (17:47:01)

Choose

Choose one or more files...

Submit

☒ Reuse files from previous submissions 

Expected files: Calculator.java, GraphicsCalculator.java, Operator.java, Tokenizer.java