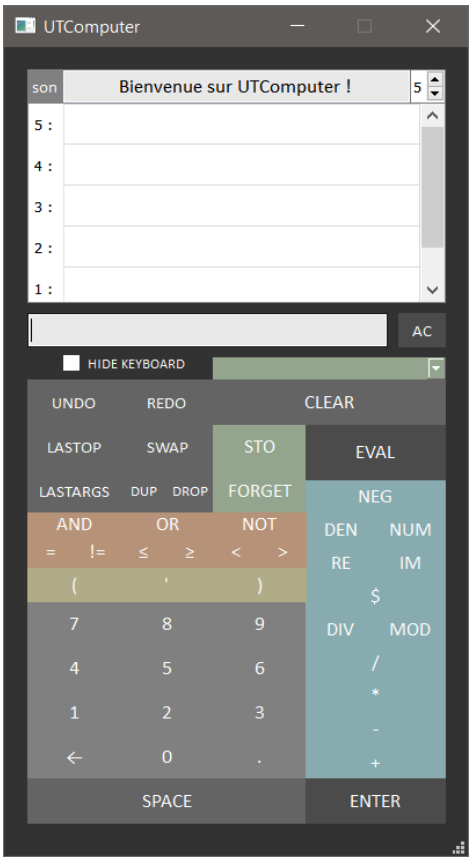
Virgile VANCON --- Nicolas MARCADET

LO21 : UTCOMPUTER



12/06/2016

PROJET LO21 : PRINTEMPS 2016

Table des matières

INTRODUCTION

L'application UTComputer est une calculatrice scientifique à notation Polonaise inversée (RPN) développée en C++ sur l'environnement QtCreator.

Elle a été réalisée dans le cadre de l'UV LO21 : Programmation et conception orientée objet.

DONNEES D'ENTREE :
Sujet mit à disposition.

OBJET DU PROJET :
Définir l'architecture du projet, développer en C++ la calculatrice.

PRODUIT DU PROJET :

- Application fonctionnelle, facile d'utilisation et correspondante aux attentes du porteur (voir sujet)
- Code source de l'application
- Documentation complète en html générée avec Doxygen
- Vidéo de présentation avec commentaire audio
- Rapport final

OBJECTIFS :

- Couts (horaire) : 50 heures par personne
- Délais : à rendre le 12/06/2016

ACTEURS :

- Maître d'ouvrage : Antoine Jouglet (Responsable de l'UV LO21)
- Maîtres d'œuvres : Virgile Vançon et Nicolas Marcadet (étudiants)
- Correcteur : Nicola Zema (Enseignant de LO21)

ARCHITECTURE

Retrouvez l'UML de notre architecture en annexe ici.

1. Traitement des littérales

Les littérales sont au cœur de l'application UTComputer. Ce sont les variables qui vont être traitées par la calculatrice. On peut subdiviser le concept en plusieurs sous catégories. **Littérale** est donc une classe abstraite (composée d'au moins une méthode virtuelle pure).

- Littérales Numériques : les littérales numériques représentent l'ensemble des nombres entiers, réels, rationnels et complexes. La classe **LiNumérique** est donc elle aussi une classe abstraite.
    - Littérales entières représentées par la classe **LiEntiere**. Un seul attribut : un entier.
    - Littérales réelles représentées par la classe **LiRéelle**. Un seul attribut : un double.
    - Littérales rationnelles représentées par la classe **LiRationnelle**. On peut exprimer n'importe quel nombre appartenant à l'ensemble des rationnels comme le quotient de deux entiers. Nous avons donc pour cette classe deux attributs *LiEntiere* représentant le numérateur et le dénominateur.

- Littérales Complexes : les nombres complexes sont eux composés d'une partie réelle et d'une partie imaginaire. Chacune de ces parties peut être réelle, entière ou rationnelle. Nous avons donc choisi de définir **LiComplexe** comme une composition de deux *LiNumerique*. Chaque objet LiComplexe va créer et supprimer dynamiquement des objets de type LiNumerique. Elle hérite directement de *Litterale*.

- Littérales Expressions : les littérales expressions sont des suites de caractères entre guillemets : un attribut *QString*. La classe **LiExpression** possède les mêmes fonctionnalités que les autres littérales (opérations, affichages…), elle hérite donc directement de la classe abstraite *Littérale*.

- Littérales Atomes : les littérales atomes sont des suites de caractères (composées de lettres majuscules et de chiffres et commençant par une lettre majuscule), et peut correspondre à l'identificateur d'une variable ou d'un programme. Nous avons choisi de ne pas créer une classe spéciale pour les atomes, mais plutôt d'utiliser un **map** (conteneur dont chaque élément est formé de deux parties : une clé et une valeur). Ici, les clés seront les identificateurs et les valeurs seront des littérales (LiEntiere, LiRationnelle, etc…). Nous reviendront sur l'implémentation de ce map plus tard dans le rapport.

- Littérales programmes : les littérales programmes sont des suites d'opérandes commençant par "[" et terminant par "]". Malheureusement, nous n'avons pas eu le temps d'implémenter cette classe.

Dans chacune des classes concrètes héritant de Littérale, les opérateurs usuels sont surchargés.

2. Manager

Les littérales sont ensuite gérées par deux classes différentes : **Pile** et **Calculatrice**. Une troisième classe **Memento** existe pour sauvegarder le contexte de la pile principale à un moment T.

Une **Pile** possède plusieurs attributs, dont un tableau (une pile) de pointeurs de *Litterale* (car Litterale est une classe abstraite). Des méthodes sont définies pour ajouter ou supprimer si besoin est des éléments de la pile. La classe Pile hérite de *QObject*, elle a donc pour particularité de pouvoir émettre des signaux. Deux signaux ont été définis : *modificationEtat()* est émis lorsque qu'un attribut de la classe **Pile** est modifié (cela va permettre de rafraîchir l'affichage de la pile dans la partie graphique), newMessage() est émis lorsque seulement l'attribut *message* est modifié (cela va permettre de rafraîchir l'affichage du message dans la partie graphique ainsi que d'émettre un son d'avertissement).

La classe **Memento** possède les mêmes attributs que la classe *Pile* (à l'exception de message car il n'est pas nécessaire de les sauvegarder).

Afin de sauvegarder le contexte de la **Pile** et de permettre à l'utilisateur d'effectuer des Undo/Redo, la classe possède deux méthodes :

- *saveStatetoMemento()* retourne un pointeur vers un objet de la classe Memento. La taille de la pile, la taille max, le nombre d'éléments sont copiés dans les attributs correspondant dans l'objet. On fait également une deep copie de chaque *Littérale* de la pile : les éléments du tableau dans *Memento* ne pointeront pas vers les mêmes *Littérales* que la pile, on effectue donc une copie de chaque *Litterale* (deep copie).
- *getStateFromMemento(Memento* m)* fonctionne de la même façon que la première méthode mais dans l'autre sens, elle permet donc de restaurer le contexte de la pile (pour les Redos).

La classe calculatrice est la classe qui nous permettra de gérer toutes les opérations commandées par l'utilisateur : elle est entre autre composée d'un objet pile, dans lequel elle ajoutera des littérales qu'elle dépilera ensuite pour réaliser les opérations souhaitées. Cette classe possède d'autres attributs (en plus du pointeur vers un objet pile) : deux pointeurs vers littérales et un attribut de type QString pour enregistrer les opérandes utilisées lors de chaque opérations et ainsi pouvoir utiliser LASTARGS et LASTOP (ces attributs sont mis à jour à chaque opération).

La classe possède également un attribut de type map<QString, Litterale*> afin de gérer les atomes de notre calculatrice. Chaque fois qu'un utilisateur utilise l'opérateur STO pour ajouter enregistrer une nouvelle variable, on ajoute cette dernière au map. La clé de cet élément est alors l'identificateur de la variable, et sa valeur est le pointeur de littérale associé. Nous avons donc implémenté certaines fonctions pour gérer les atomes tels que *AddAtome()* (permettant d'ajouter un nouvel atome à la liste), *removeAtome()* (permettant de supprimer un atome de la liste) *et alreadyExists()* (pour savoir si un atome existe déjà).

Dans notre architecture, les atomes sont donc représenté par une classe association entre la classe Litterales et la classe Calculatrice. Cette classe association possède un attribut de type QString correspondant à l'identificateur de la variable.

La classe possède également toutes les méthodes nécessaire à la réalisation des calculs : une méthode commande prend en paramètre un QString et en fonction de sont type (opérateur binaire/unaire, entier, double, expression, etc…) appel les méthodes correspondantes (*operateur1()* pour les opérateur unaire, *operateur2()* pour les opérateurs binaire, etc..)

Pour évaluer un expression, la classe Calculatrice possède une méthode *infixePostfixe()*, permettant de transformer l'expression infixée que l'utilisateur a rentré en une expression écrite en RPN (Reverse Polish Notation). Cette expression sera alors interprétée comme toutes les autres par la méthode *commande()*.

3. Fenêtre principale et partie graphique

C'est la classe **MainWindow** qui gère la fenêtre principale de l'application. Elle possède différents attributs comme un pointeur vers la ***Pile*** principale, deux pointeurs vers des ***Mementos***, et un pointeur vers un objet de la classe ***Calculatrice***. Ces attributs sont initialisés dans le constructeur de MainWindow. L'attribut *cal* (*Calculatrice*) pointe vers la même pile que l'objet MainWindow instancié.

La partie graphique de l'application a été créée sur l'interface **QtDesigner**. Cette partie est directement liée à la MainWindow car c'est dans la fenêtre principale que s'affiche les différents Widgets. Nous avons utilisé différents Widgets répondre aux demandes du projet :

- Des **QPushButton** pour le clavier de la calculatrice.
- Deux **QLineEdit** pour la ligne de commande ainsi que pour l'affichage du message (en read only).
- Un **QTableWidget** pour l'affichage de la pile.
- Un **QCheckBox** pour afficher ou non le clavier.
- Un **QComboBox** pour dérouler un menu avec l'ensemble des atomes existant.

L'application doit aussi pouvoir émettre un son lorsqu'un nouveau message s'affiche. Pour celà, un attribut *QMediaPlayer* a été défini (avec accès à un fichier .mp3 placé dans les ressources).

La classe **MainWindow** hérite de *QObject*, on peut donc lui définir des slots. Ces slots seront appelés suite à l'émission de signaux (comme ceux vu dans les classes *Pile* et *Calculatrice*). Les connections entres slots et signaux sont définis dans le constructeur, voici une petite selection :

- *refresh()* : ce slot est appelé lors de l'émission d'un signal *modificationEtat*(), provenant de la classe *Pile*. Il permet de réinitialiser le message affiché dans le *QLineEdit* dédié, puis de réactualiser l'affichage de la *Pile* dans la *QTableWidget*.
- *playsound()* : appelé lorsqu'un son doit être joué, donc lorsqu'un *newMessage()* est émis depuis la pile.
- *addAtom(const QString&, const QString&)* : appelé lorsqu'un nouvel atome a été créé (émission d'un signal *newAtom(const QString&, const QString&)* provenant de la calculatrice). Celui ci permet de rajouter une ligne dans le menu déroulant avec le nouvel atome et sa valeur.
- *on_PushButtonX_Clicked()* : Lorsque l'utilisateur clique sur le PushButton X, un signal est émis et le slot correspondant est déclenché. Si c'est un chiffre, il s'écrit juste dans la ligne de commande, si

c'est un "+", on vérifie si l'on se trouve dans une expression ou pas (si oui on l'écrit juste). Si ce n'est pas le cas, on l'écrit dans la ligne de commande et on appelle la méthode *GetNextCommand()* pour évaluer la ligne.

EVOLUTIONS POSSIBLES DE L'ARCHITECTURE

La classe calculatrice, qui gère toutes les opérations de la notre calculateur, gère une pile de pointeurs vers litterales. Il est donc très facile de rajouter un nouveau type de litterale qui pourra être pris en charge par la pile sans modification : en effet, si ce nouveau type de littérale hérite de la classe littérale, il pourra être ajouté à la pile grâce au principe de substitution. C'est donc le premier point qui favorise grandement les évolutions de notre calculatrice : il est possible d'ajouter à l'infini des nouveaux types de littérales (il faudra cependant surcharger tous les opérateurs usuels dans cette nouvelle classe, car sinon elle sera abstraite).

Il sera également facile un nouveau type de littérale, car pour les afficher, nous utilisons la méthode *toString()*, (méthode virtuelle pure définie dans la classe littérale). Il suffira donc d'ajouter cette méthode à notre nouvelle classe la méthode, et il sera ensuite possible de l'afficher.

Nous pouvons également ajouter des nouveaux opérateurs : il faudra les surcharger dans les méthodes correspondantes, et les ajouter aux fonction d'identification (*estUnOperateurBinaire()*, *estUnOperateurUnaire()*, etc…) ou pourquoi pas ajouter un nouveau type d'opérateur si il n'est ni unaire, ni binaire.

Toutes ces évolutions ne changent rien à l'implémentation de la calculatrice, et il est donc relativement facile de faire des évolutions comme celles-ci.

La classe MainWindow, grâce à l'environnement QtDesigner est elle aussi très maléable. On peut facilement ajouter des Widgets et les personnaliser visuellement ainsi qu'attribuer des slots à leurs signaux.

C'est ce qui fait la force de Qt : on peut par exemple prendre le cas du son enclenché par un message d'erreur (slot *playsound()*): dans notre application, il était demandé à ce que cette fonctionnalité ne soit appelée seulement quand un nouveau message apparaît. Si l'on souhaite que du son s'enclenche pour d'autres fonctionnalités, on devra seulement envoyé le signal correspondant depuis n'importe quelle méthodes ou fonctions et le son s'enclenchera.

UTILISATION DE LA CALCULATRICE

Premièrement les parties du sujet non-traitées (par manque de temps) sont :

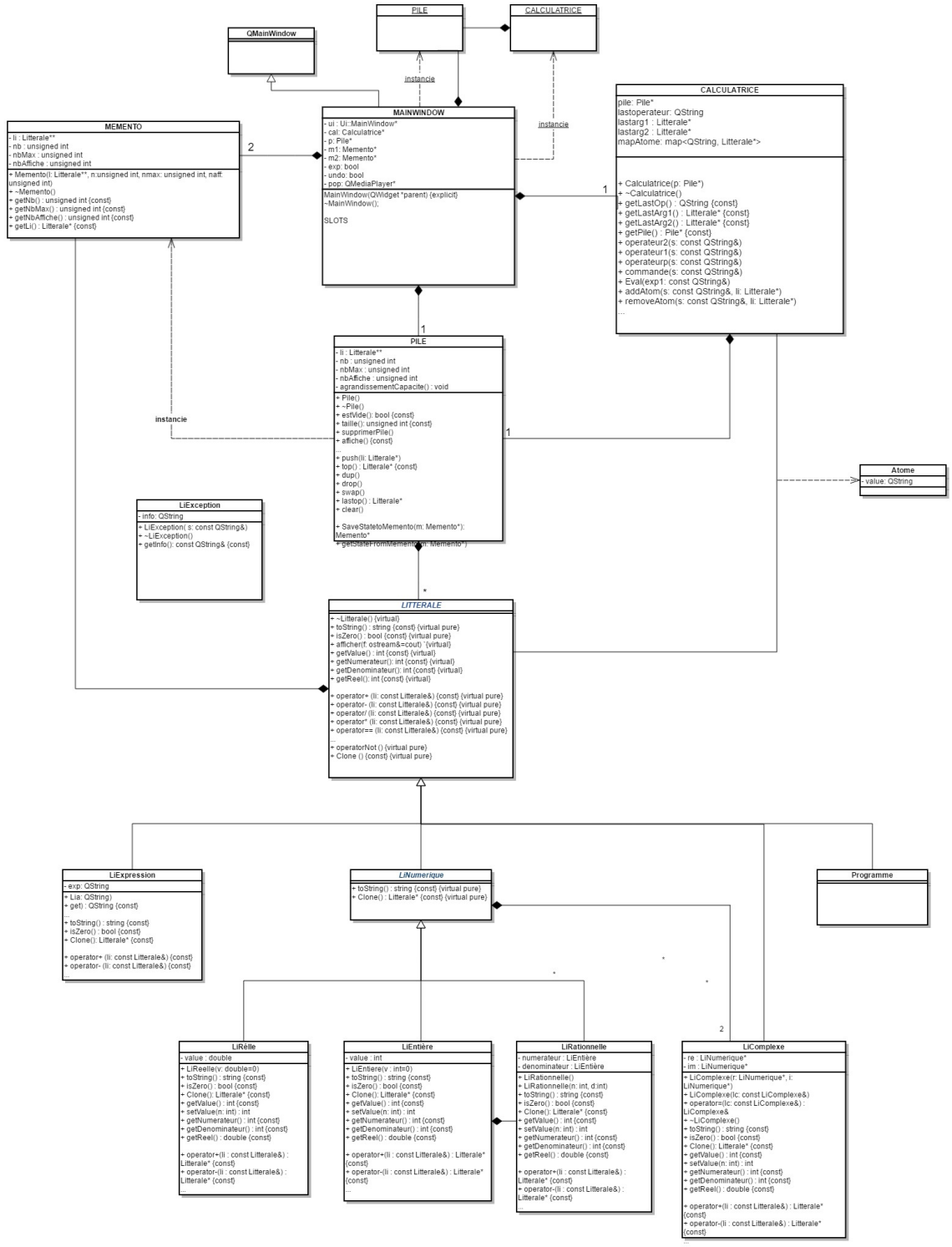
- Les littérales expression
- Les littérales atomes dans les expressions (pas reconnue)
- La sauvegarde du context
- Impossibilité de gérer les atomes depuis une nouvelle fenêtre d'édition

Les raccourcis claviers qu'il est possible d'utiliser sont :

- CTRL + Z : raccourcis de la commande UNDO (il est possible de revenir en arrière jusqu'à ce que l'état initial du calculateur soit rencontré)
- CTRL + Y : raccourcis de la commande REDO

Aucun exécutable n'a été fourni dans le rendu (non demandé). Pour lancer l'application, il faut ouvrir le fichier **UTComputerQt.pro** puis l'exécuter depuis l'environnement.

UML


PILE
CALCULATRICE
QMainWindow
instancie
MAINWINDOW
- ui : Ui::MainWindow*
- cal: Calculatrice*
- p: Pile*
- m1: Memento*
- m2: Memento*
- exp: bool
- undo: bool
- pop: QMediaPlayer*
MainWindow(QWidget *parent) {explicit}
~MainWindow();
SLOTS
MEMENTO
- li : Litterale**
- nb : unsigned int
- nbMax : unsigned int
- nbAffiche : unsigned int
+ Memento(l: Litterale**, n:unsigned int, nmax: unsigned int, naff: unsigned int)
+ ~Memento()
+ getNb() : unsigned int {const}
+ getNbMax() : unsigned int {const}
+ getNbAffiche() : unsigned int {const}
+ getLi() : Litterale* {const}
2
1
CALCULATRICE
pile: Pile*
lastoperateur: QString
lastarg1 : Litterale*
lastarg2 : Litterale*
mapAtome: map<QString, Litterale*>
+ Calculatrice(p: Pile*)
+ ~Calculatrice()
+ getLastOp() : QString {const}
+ getLastArg1() : Litterale* {const}
+ getLastArg2() : Litterale* {const}
+ getPile() : Pile* {const}
+ operateur2(s: const QString&)
+ operateur1(s: const QString&)
+ operateurp(s: const QString&)
+ commande(s: const QString&)
+ Eval(exp1: const QString&)
+ addAtom(s: const QString&, li: Litterale*)
+ removeAtom(s: const QString&, li: Litterale*)
PILE
- li : Litterale**
- nb : unsigned int
- nbMax : unsigned int
- nbAffiche : unsigned int
- agrandissementCapacite() : void
+ Pile()
+ ~Pile()
+ estVide(): bool {const}
+ taille(): unsigned int {const}
+ supprimerPile()
+ affiche() {const}
+ push(li: Litterale*)
+ top() : Litterale* {const}
+ dup()
+ drop()
+ swap()
+ lastop() : Litterale*
+ clear()
+ SaveStatetoMemento(m: Memento*): Memento*
+ getStateFromMemento(m: Memento*)
instancie
1
1
Atome
- value: QString
LiException
- info: QString
+ LiException(s: const QString&)
+ ~LiException()
+ getInfo(): const QString& {const}
LITTERALE
+ ~Litterale() {virtual}
+ toString() : string {const} {virtual pure}
+ isZero() : bool {const} {virtual pure}
+ afficher(f: ostream&=cout) {virtual}
+ getValue() : int {const} {virtual}
+ getNumerateur() : int {const} {virtual}
+ getDenominateur(): int {const} {virtual}
+ getReel(): int {const} {virtual}
+ operator+ (li: const Litterale&) {const} {virtual pure}
+ operator- (li: const Litterale&) {const} {virtual pure}
+ operator/ (li: const Litterale&) {const} {virtual pure}
+ operator* (li: const Litterale&) {const} {virtual pure}
+ operator== (li: const Litterale&) {const} {virtual pure}
+ operatorNot () {virtual pure}
+ Clone () {const} {virtual pure}
*
LiExpression
- exp: QString
+ Lia: QString)
+ get) : QString {const}
+ toString() : string {const}
+ isZero() : bool {const}
+ Clone(): Litterale* {const}
+ operator+ (li: const Litterale&) {const}
+ operator- (li: const Litterale&) {const}
LiNumerique
+ toString() : string {const} {virtual pure}
+ Clone() : Litterale* {const} {virtual pure}
Programme
*
*
*
2
LiRéelle
- value : double
+ LiReelle(v: double=0)
+ toString() : string {const}
+ isZero() : bool {const}
+ Clone(): Litterale* {const}
+ getValue() : int {const}
+ setValue(n: int) : int
+ getNumerateur() : int {const}
+ getDenominateur() : int {const}
+ getReel() : double {const}
+ operator+(li : const Litterale&) : Litterale* {const}
+ operator-(li : const Litterale&) : Litterale* {const}
LiEntière
- value : int
+ LiEntiere(v : int=0)
+ toString() : string {const}
+ isZero() : bool {const}
+ Clone(): Litterale* {const}
+ getValue() : int {const}
+ setValue(n: int) : int
+ getNumerateur() : int {const}
+ getDenominateur() : int {const}
+ getReel() : double {const}
+ operator+(li : const Litterale&) : Litterale* {const}
+ operator-(li : const Litterale&) : Litterale* {const}
LiRationnelle
- numerateur : LiEntière
- denominateur : LiEntière
+ LiRationnelle()
+ LiRationnelle(n: int, d:int)
+ toString() : string {const}
+ isZero() : bool {const}
+ Clone(): Litterale* {const}
+ getValue() : int {const}
+ setValue(n: int) : int
+ getNumerateur() : int {const}
+ getDenominateur() : int {const}
+ getReel() : double {const}
+ operator+(li : const Litterale&) : Litterale* {const}
+ operator-(li : const Litterale&) : Litterale* {const}
LiComplexe
- re : LiNumerique*
- im : LiNumerique*
+ LiComplexe(r:LiNumerique*, i: LiNumerique*)
+ LiComplexe(lc: const LiComplexe&)
+ operator=(lc: const LiComplexe&) : LiComplexe&
+ ~LiComplexe()
+ toString() : string {const}
+ isZero() : bool {const}
+ Clone(): Litterale* {const}
+ getValue() : int {const}
+ setValue(n: int) : int
+ getNumerateur() : int {const}
+ getDenominateur() : int {const}
+ getReel() : double {const}
+ operator+(li : const Litterale&) : Litterale* {const}
+ operator-(li : const Litterale&) : Litterale* {const}