

Annexe

Quatrième partie : Codage

Interpréteur du langage LIR

Nicolas CAMINADE, Sylvan COURTIOL,

Pierre DEBAS, Heïa DEXTER,

Lucàs VABRE

Sommaire :

[I. Paquetage interpreteurlir.donnees.litteraux 5](#_Toc72843376)

[1. Classe Litteral 5](#_Toc72843377)

[a. Litteral.java 5](#_Toc72843378)

[b. TestLitteral.java 6](#_Toc72843379)

[2. Classe Chaine 9](#_Toc72843380)

[a. Chaine.java 9](#_Toc72843381)

[b. TestChaine.java 10](#_Toc72843382)

[3. Classe Entier 13](#_Toc72843383)

[a. Entier.java 13](#_Toc72843384)

[b. TestEntier.java 15](#_Toc72843385)

[4. Classe Booleen 21](#_Toc72843386)

[a. Booleen.java 21](#_Toc72843387)

[b. TestBooleen.java 21](#_Toc72843388)

[II. Paquetage interpreteurlir.donnees 23](#_Toc72843389)

[1. Classe Identificateur 23](#_Toc72843390)

[a. Identificateur.java 23](#_Toc72843391)

[b. TestIdentificateur.java 24](#_Toc72843392)

[2. Classe IdentificateurChaine 27](#_Toc72843393)

[a. IdentificateurChaine.java 27](#_Toc72843394)

[b. TestIdentificateurChaine.java 27](#_Toc72843395)

[3. Classe IdentificateurEntier 30](#_Toc72843396)

[a. IdentificateurEntier.java 30](#_Toc72843397)

[b. TestIdentificateurEntier.java 31](#_Toc72843398)

[4. Classe Variable 33](#_Toc72843399)

[a. Variable.java 33](#_Toc72843400)

[b. TestVariable.java 34](#_Toc72843401)

[III. Paquetage interpreteurlir.expressions 38](#_Toc72843402)

[1. Classe Expression 38](#_Toc72843403)

[a. Expression.java 38](#_Toc72843404)

[b. TestExpression.java 41](#_Toc72843405)

[2. Classe ExpressionChaine 43](#_Toc72843406)

[a. ExpressionChaine.java 43](#_Toc72843407)

[b. TestExpressionChaine.java 45](#_Toc72843408)

[3. Classe ExpressionEntier 49](#_Toc72843409)

[a. ExpressionEntier.java 49](#_Toc72843410)

[b. TestExpressionEntier.java 51](#_Toc72843411)

[4. Classe ExpressionBooleenne 55](#_Toc72843412)

[a. ExpressionBooleenne.java 55](#_Toc72843413)

[b. TestExpressionBooleenne.java 60](#_Toc72843414)

[IV. Paquetage interpreteurlir 68](#_Toc72843415)

[1. Classe InterpreteurException 68](#_Toc72843416)

[a. InterpreteurException.java 68](#_Toc72843417)

[b. EssaiInterpreteurException.java 68](#_Toc72843418)

[2. Classe ExecutionException 69](#_Toc72843419)

[a. ExecutionException.java 69](#_Toc72843420)

[b. Tests 69](#_Toc72843421)

[3. Classe Contexte 70](#_Toc72843422)

[a. Contexte.java 70](#_Toc72843423)

[b. TestContexte.java 72](#_Toc72843424)

[4. Classe Analyseur 76](#_Toc72843425)

[a. Analyseur.java 76](#_Toc72843426)

[b. Tests 80](#_Toc72843427)

[5. ProgrammeDeTest.java 81](#_Toc72843428)

[V. Paquetage interpreteurlir.programmes 83](#_Toc72843429)

[1. Classe Etiquette 83](#_Toc72843430)

[a. Etiquette.java 83](#_Toc72843431)

[b. TestEtiquette.java 84](#_Toc72843432)

[2. Classe Programme 89](#_Toc72843433)

[a. Programme.java 89](#_Toc72843434)

[b. TestProgramme.java 93](#_Toc72843435)

[VI. Paquetage interpreteurlir.motscles 101](#_Toc72843436)

[1. Classe Commande 101](#_Toc72843437)

[a. Commande.java 101](#_Toc72843438)

[b. TestCommande.java 102](#_Toc72843439)

[2. Classe CommandeCharge 104](#_Toc72843440)

[a. CommandeCharge.java 104](#_Toc72843441)

[b. TestCommandeCharge.java 107](#_Toc72843442)

[3. Classe CommandeDebut 109](#_Toc72843443)

[a. CommandeDebut.java 109](#_Toc72843444)

[b. TestCommandeDebut.java 110](#_Toc72843445)

[4. Classe CommandeDefs 112](#_Toc72843446)

[a. CommandeDefs.java 112](#_Toc72843447)

[b. TestCommandeDefs.java 113](#_Toc72843448)

[5. Classe CommandeEfface 115](#_Toc72843449)

[a. CommandeEfface.java 115](#_Toc72843450)

[b. TestCommandeEfface.java 116](#_Toc72843451)

[6. Classe CommandeFin 118](#_Toc72843452)

[a. CommandeFin.java 118](#_Toc72843453)

[b. TestCommandeFin.java 118](#_Toc72843454)

[7. Classe CommandeLance 120](#_Toc72843455)

[a. CommandeLance.java 120](#_Toc72843456)

[b. TestCommandeLance.java 121](#_Toc72843457)

[8. Classe CommandeListe 123](#_Toc72843458)

[a. CommandeListe.java 123](#_Toc72843459)

[b. TestCommandeListe.java 124](#_Toc72843460)

[9. Classe CommandeSauve 128](#_Toc72843461)

[a. CommandeSauve.java 128](#_Toc72843462)

[b. TestCommandeSauve.java 129](#_Toc72843463)

[10. EssaiCommande.java 133](#_Toc72843464)

[VII. Paquetage interpreteurlir.motscles.instructions 135](#_Toc72843465)

[1. Classe Instruction 135](#_Toc72843466)

[a. Instruction.java 135](#_Toc72843467)

[b. TestInstruction.java 136](#_Toc72843468)

[2. Classe InstructionAffiche 137](#_Toc72843469)

[a. InstructionAffiche.java 137](#_Toc72843470)

[b. TestInstructionAffiche.java 138](#_Toc72843471)

[3. Classe InstructionEntre 141](#_Toc72843472)

[a. InstructionEntre.java 141](#_Toc72843473)

[b. TestInstructionEntre.java 142](#_Toc72843474)

[4. Classe InstructionProcedure 145](#_Toc72843475)

[a. InstructionProcedure.java 145](#_Toc72843476)

[b. TestInstructionProcedure.java 146](#_Toc72843477)

[5. Classe InstructionRetour 149](#_Toc72843478)

[a. InstructionRetour.java 149](#_Toc72843479)

[b. TestInstructionRetour.java 150](#_Toc72843480)

[6. Classe InstructionSi(Vaen) 153](#_Toc72843481)

[a. InstructionSi.java 153](#_Toc72843482)

[b. TestInstructionSi.java 154](#_Toc72843483)

[7. Classe InstructionStop 158](#_Toc72843484)

[a. InstructionStop.java 158](#_Toc72843485)

[b. TestInstructionStop.java 159](#_Toc72843486)

[8. Classe InstructionVaen 161](#_Toc72843487)

[a. InstructionVaen.java 161](#_Toc72843488)

[b. TestInstructionVaen.java 162](#_Toc72843489)

[9. Classe InstructionVar 165](#_Toc72843490)

[a. InstructionVar.java 165](#_Toc72843491)

[b. TestInstructionVar.java 166](#_Toc72843492)

# Paquetage interpreteurlir.donnees.litteraux

## Classe Litteral

### Litteral.java

/\*\*

\* Litteral.java 7 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.donnees.litteraux;

/\*\*

\* Valeur littérale utilisée dans une expression.

\* Chaque littérale est reconnue par sont type.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **abstract** **class** Litteral **implements** Comparable<Litteral> {

/\*\* valeur de ce littéral \*/

**protected** Object valeur;

/\*\*

\* Initialise ce littéral par défaut.

\*/

**protected** Litteral() {

**super**();

}

/\*\*

\* Initialise cette valeur avec un objet argument.

\* **@param** valeur valeur du littéral à construire

\*/

**public** Litteral(Object valeur) {

**super**();

**this**.valeur = valeur;

}

/\*\*

\* **@return** la valeur de valeur

\*/

**public** Object getValeur() {

**return** valeur;

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

**return** valeur.toString();

}

/\* non javadoc

\* @see java.lang.Comparable#compareTo(java.lang.Object)

\*/

@Override

**public** **abstract** **int** compareTo(Litteral autre);

}

### TestLitteral.java

// Tests faits avant le passage en abstract de la classe

/\*\*

\* TestLitteraux.java 7 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.donnees.litteraux.tests;

**import** interpreteurlir.donnees.litteraux.Litteral;

/\*\*

\* Test unitaires des constantes littérales de l'interpréteurlir

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestLitteraux {

/\*\* Jeux de littéraux pour test. \*/

**private** **static** **final** Litteral[] ***VALIDES*** = {

/\* Caractères \*/

**new** Litteral('a'),

**new** Litteral('!'),

**new** Litteral('\"'),

**new** Litteral('1'),

**new** Litteral('\t'),

/\* Chaînes \*/

**new** Litteral("ceci est une chaine"),

**new** Litteral("bonjour"),

**new** Litteral(" bonjour "),

**new** Litteral(""),

**new** Litteral(" "),

/\* Entier \*/

**new** Litteral(123),

**new** Litteral(-123),

**new** Litteral(0),

**new** Litteral(Integer.***MAX\_VALUE***),

/\* Double \*/

**new** Litteral(14.258),

**new** Litteral(-14.128),

**new** Litteral(0.0),

**new** Litteral(Double.***NaN***),

**new** Litteral(Double.***NEGATIVE\_INFINITY***),

**new** Litteral(Double.***MAX\_VALUE***),

**new** Litteral(Double.***MIN\_VALUE***),

**new** Litteral(Double.***MIN\_NORMAL***),

/\* Boolean \*/

**new** Litteral(**true**),

**new** Litteral(**false**),

**new** Litteral(3 >= 4),

**new** Litteral(**true**),

**new** Litteral(**true**),

**new** Litteral(**true**)

};

/\*\* test de getValeur \*/

**public** **static** **void** testGetValeur() {

**final** Object[] VALEURS\_ATTENDUES = {

'a', '!', '\"', '1', '\t' ,"ceci est une chaîne", "bonjour",

" bonjour ", "", " ", 123, -123, 0, 2147483647, 14.258, -14.128,

0.0, Double.***NaN***, Double.***NEGATIVE\_INFINITY***, Double.***MAX\_VALUE***,

Double.***MIN\_VALUE***, Double.***MIN\_NORMAL***, **true**, **false**, **false**, **true**, **true**,

**true**

};

System.***out***.println("test de getValeur\n");

**for** (**int** i = 0 ; i < ***VALIDES***.length ; i++) {

**try** {

**assert** (***VALIDES***[i].getValeur().equals(VALEURS\_ATTENDUES[i]));

} **catch** (AssertionError lancee) {

System.***err***.println("Echec du test a l'indice " + i);

}

}

System.***out***.println("==>test terminé\n");

}

/\*\* test de toString \*/

**public** **static** **void** testToString() {

**final** String[] STRING\_ATTENDUE = {

"a", "!", "\"", "1", "\t", "ceci est une chaîne", "bonjour",

" bonjour ", "", " ", "123", "-123", "0", "2147483647", "14.258",

"-14.128", "0.0", "NaN", "-Infinity", "1.7976931348623157E308",

"4.9E-324", "2.2250738585072014E-308", "true", "false", "false",

"true", "true", "true"

};

System.***out***.println("test de toString\n");

**for** (**int** i = 0 ; i < ***VALIDES***.length ; i++ ) {

**try** {

**assert** (***VALIDES***[i].toString().equals(STRING\_ATTENDUE[i]));

} **catch** (AssertionError lancee) {

System.***err***.println("Echec du test a l'indice " + i);

}

}

System.***out***.println("==>test terminé\n");

}

/\*\* test de compareTo \*/

**public** **static** **void** testCompareTo() {

**final** Litteral[] MEMES\_TYPES = {

**new** Litteral('a'),

**new** Litteral('!'),

**new** Litteral('\"'),

**new** Litteral('Z'),

**new** Litteral('s'),

**new** Litteral("bonjour"),

**new** Litteral("bonjour"),

**new** Litteral("arar"),

**new** Litteral("zarar za "),

**new** Litteral("CAFE\_BABE"),

**new** Litteral(123),

**new** Litteral(123),

**new** Litteral(0),

**new** Litteral(-123),

**new** Litteral(Double.***MAX\_VALUE***),

**new** Litteral(Double.***NaN***),

**new** Litteral(12.3),

**new** Litteral(Double.***NaN***),

**new** Litteral(45.7),

**new** Litteral(-12.6),

**new** Litteral(0.0),

**new** Litteral(Double.***MIN\_NORMAL***),

**new** Litteral(**false**),

**new** Litteral(**false**),

**new** Litteral(**true**),

**new** Litteral(**true**),

**new** Litteral(**true**),

**new** Litteral(**true**)

};

System.***out***.println("test de compareTo\nAvec des types identiques");

**for** (**int** i = 0 ; i < ***VALIDES***.length ; i++ ) {

**try** {

**assert** (***VALIDES***[i].compareTo(MEMES\_TYPES[i]) == 0);

} **catch** (AssertionError lancee) {

System.***err***.println("Echec du test a l'indice " + i);

}

}

System.***out***.println("Avec des types différents");

**for** (**int** i = 0 ; i < ***VALIDES***.length ; i++ ) {

**try** {

**assert** (***VALIDES***[i].compareTo(MEMES\_TYPES[MEMES\_TYPES.length

- (i + 1)]) != 0);

} **catch** (AssertionError lancee) {

System.***err***.println("Echec du test a l'indice " + i);

}

}

System.***out***.println("==>test terminé\n");

}

/\*\*

\* Lancement des test

\* **@param** args non utilisé

\*/

**public** **static** **void** main(String[] args) {

*testGetValeur*();

*testToString*();

*testCompareTo*();

}

}

## Classe Chaine

### Chaine.java

/\*\*

\* Chaine.java 7 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.donnees.litteraux;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Constante littérale de type chaîne de caractères.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** Chaine **extends** Litteral {

/\*\* Longueur maximale d'une chaîne \*/

**public** **static** **final** **int** ***LG\_MAX\_CHAINE*** = 70;

/\*\* Erreur chaîne trop longue \*/

**private** **static** **final** String ***ERREUR\_LG\_MAX*** =

"longueur maximale d'une chaîne dépassée";

/\*\* Erreur constante littéral chaîne invalide \*/

**private** **static** **final** String ***ERREUR\_INVALIDE*** =

" n'est pas une constante de type chaîne";

/\*\*

\* initialise cette chaîne avec une valeur par défaut.

\*/

**public** Chaine() {

valeur = "";

}

/\*\*

\* Initialise une chaîne avec la séquence

\* de caractères passée en argument.

\* **@param** uneValeur valeur de la chaine à construire (entre guillemets)

\*/

**public** Chaine(String uneValeur) {

uneValeur = uneValeur.trim();

**if** (!*isChaine*(uneValeur)) {

**throw** **new** InterpreteurException(uneValeur + ***ERREUR\_INVALIDE***);

}

uneValeur = uneValeur.substring(1, uneValeur.length() - 1);

**if** (uneValeur.length() > ***LG\_MAX\_CHAINE***)

**throw** **new** InterpreteurException(***ERREUR\_LG\_MAX***);

valeur = uneValeur;

}

/\*\*

\* Prédicat de validité d'une constante littérale de type chaîne.

\* **@param** uneValeur valeur aTester

\* **@return** true si uneValeur est une chaîne sinon false

\*/

**public** **static** **boolean** isChaine(String uneValeur) {

uneValeur = uneValeur.trim();

**return** uneValeur.length() >= 2

&& uneValeur.startsWith("\"") && uneValeur.endsWith("\"");

}

/\*\*

\* Concatène deux chaînes ensemble. Opération non commutative:

\* a + b != b + a

\* **@param** a une Chaîne

\* **@param** b une autre Chaîne

\* **@return** une nouvelle Chaîne.

\*/

**public** **static** Chaine concatener(Chaine a, Chaine b) {

**return** **new** Chaine("\"" + a.valeur + b.valeur + "\"");

}

/\* non javadoc

\* @see Litteral#compareTo(Litteral)

\*/

@Override

**public** **int** compareTo(Litteral autre) {

**return** valeur.toString().compareTo(autre.valeur.toString());

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

**return** '\"' + valeur.toString() + '\"';

}

}

### TestChaine.java

/\*\*

\* TestChaine.java 8 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.donnees.litteraux.tests;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.litteraux.Chaine;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Tests unitaires de la classe Chaine

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestChaine {

/\*\* Test unitaire de {@link Chaine#Chaine(String)} \*/

**public** **static** **void** testChaine() {

**final** String[] VALIDE = {

"\"arztyehjklmpoijhghnbghjklmpoiuytrf" +

"ghjnklmpoiuytrezaqsdfghnjklmpjbfrtyu\"",

"\"\"",

"\"coucou \"",

"\"" + 42 + "\""

};

**final** String INVALIDE =

"arztyehjklmpoijhghnbghjklmpoiuytrf" +

"yeryghjnklmpoiuytrezaqsdfghnjklmpjbfrtyu";

System.***out***.println("\tExécution du test de Chaine(String)");

**for**(String aTester : VALIDE) {

**try** {

**new** Chaine(aTester);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

**try** {

**new** Chaine(INVALIDE);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

/\*\* Test unitaire de {@link Chaine#compareTo(Litteral)} \*/

**public** **static** **void** testCompareTo() {

**final** Chaine[][] EGALITES = {

{ **new** Chaine("\"coucou\""), **new** Chaine("\"coucou\"") },

{ **new** Chaine("\" \""), **new** Chaine("\" \"") },

{ **new** Chaine("\"\""), **new** Chaine() }

};

**final** Chaine[][] DIFFERENCES = {

{ **new** Chaine("\"coucou\""), **new** Chaine("\"camomille\"") },

{ **new** Chaine("\"tarentule\""), **new** Chaine("\"coucou\"") },

{ **new** Chaine("\"coucou\""), **new** Chaine("\" \"") },

{ **new** Chaine("\"coucou\""), **new** Chaine() },

{ **new** Chaine("\" \""), **new** Chaine() }

};

System.***out***.println("\tExécution du test de compareTo(Chaine)\n"

+ "\tAvec égalités");

**for** (Chaine[] couple : EGALITES) {

*assertEquivalence*(couple[0].compareTo(couple[1]), 0);

}

System.***out***.println("\tAvec des inégalités");

**for** (Chaine[] couple : DIFFERENCES) {

*assertTrue*(couple[0].compareTo(couple[1]) > 0);

}

}

/\*\* Test unitaire de {@link Chaine#toString()} \*/

**public** **static** **void** testToString() {

**final** Chaine[] A\_AFFICHER = {

**new** Chaine(),

**new** Chaine("\" \""),

**new** Chaine("\"coucou\""),

**new** Chaine("\" coucou \""),

**new** Chaine("\"coucou monsieur\"")

};

**final** String[] AFFICHAGE\_GUILLEMETS = {

"\"\"",

"\" \"",

"\"coucou\"",

"\" coucou \"",

"\"coucou monsieur\""

};

System.***out***.println("\tExécution du test de toString");

**for** (**int** i = 0 ; i < A\_AFFICHER.length ; i++) {

*assertTrue*(AFFICHAGE\_GUILLEMETS[i]

.equals(A\_AFFICHER[i].toString()));

}

}

/\*\*

\* Tests unitaires de concaténer

\*/

**public** **static** **void** testConcatener() {

**final** Chaine[] ATTENDU = {

**new** Chaine(),

**new** Chaine("\"Bonjour le monde ! \""),

**new** Chaine("\" \""),

**new** Chaine("\"3,1415\""),

};

**final** Chaine[][] A\_CONCATENER = {

{ **new** Chaine(), **new** Chaine("\"\"") },

{ **new** Chaine("\"Bonjour \""), **new** Chaine("\"le monde ! \"") },

{ **new** Chaine("\"\""), **new** Chaine("\" \"") },

{ **new** Chaine("\"3,\""), **new** Chaine("\"1415\"") },

};

System.***out***.println("\tExécution du test de concaténer");

**for** (**int** numTest = 0 ; numTest < ATTENDU.length ; numTest++ ) {

*assertTrue*(ATTENDU[numTest].compareTo(

Chaine.*concatener*(A\_CONCATENER[numTest][0],

A\_CONCATENER[numTest][1])) == 0);

}

}

}

## Classe Entier

### Entier.java

/\*\*

\* Entier.java 8 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.donnees.litteraux;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Constante littérale de type entier

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** Entier **extends** Litteral {

/\*\*

\* Initialisation de cet entier avec une valeur passée en argument

\* **@param** unEntier valeur de l'entier à construire

\*/

**public** Entier(**int** unEntier) {

**super**.valeur = unEntier;

}

/\*\*

\* Initialisation de cet entier avec une valeur passée en argument

\* **@param** uneValeur chaîne contenant l'entier à construire

\*/

**public** Entier(String uneValeur) {

**if** (!*isEntier*(uneValeur)) {

**throw** **new** InterpreteurException(uneValeur

+ " n'est pas un nombre entier");

}

valeur = Integer.*valueOf*(uneValeur);

}

/\*\*

\* Prédicat de validité d'une chaîne en tant que nombre entier signé

\* **@param** uneValeur la chaîne à tester

\* **@return** true si uneValeur est un entier sinon false

\*/

**public** **static** **boolean** isEntier(String uneValeur) {

**if** (uneValeur == **null**) {

**return** **false**;

}

**try** {

Integer.*valueOf*(uneValeur.trim());

} **catch** (NumberFormatException lancee) {

**return** **false**;

}

**return** **true**;

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

**return** valeur.toString();

}

@Override

**public** Integer getValeur() {

**return** (Integer) **super**.valeur;

}

/\*\*

\* Compare cet entier à un autre entier

\* **@param** autre entier avec lequel this est comparé

\* **@return** une {@code valeur < 0} lorsque {@code autre > cet entier}

\* une {@code valeur > 0} lorsque {@code autre < cet entier}

\* une {@code valeur = 0} lorsque autre et cet entier sont égaux

\*/

**public** **int** compareTo(Entier autre) {

**return** ((Integer) valeur).compareTo(autre.getValeur());

}

/\*\*

\* Somme de deux entiers

\* **@param** premier Entier

\* **@param** second Entier

\* **@return** la somme des deux entiers

\*/

**public** **static** Entier somme(Entier premier, Entier second) {

**return** **new** Entier((**int**) premier.getValeur() + (**int**) second.getValeur());

}

/\*\*

\* Soustraction de deux entiers

\* **@param** premier Entier

\* **@param** second Entier

\* **@return** le résultat de premier auquel on soustrait second

\*/

**public** **static** Entier soustrait(Entier premier, Entier second) {

**return** **new** Entier((**int**) premier.getValeur() - (**int**) second.getValeur());

}

/\*\*

\* Multiplication de deux entiers

\* **@param** premier Entier

\* **@param** second Entier

\* **@return** le résultat de premier multiplié par second

\*/

**public** **static** Entier multiplie(Entier premier, Entier second) {

**return** **new** Entier((**int**) premier.getValeur() \* (**int**) second.getValeur());

}

/\*\*

\* Division de deux entiers

\* **@param** premier Entier

\* **@param** second Entier

\* **@return** le quotient de la division de premier par second

\* **@throws** ExecutionException lorsque le diviseur est nul

\*/

**public** **static** Entier quotient(Entier premier, Entier second) {

**if** (second.compareTo(**new** Entier (0)) == 0) {

**throw** **new** ExecutionException("division par 0");

}

**return** **new** Entier((**int**) premier.getValeur() / (**int**) second.getValeur());

}

/\*\*

\* Division de deux entiers

\* **@param** premier Entier

\* **@param** second Entier

\* **@return** le reste de la division de premier par second

\* **@throws** ExecutionException lorsque le diviseur est nul

\*/

**public** **static** Entier reste(Entier premier, Entier second) {

**if** (second.compareTo(**new** Entier (0)) == 0) {

**throw** **new** ExecutionException("division par 0");

}

**return** **new** Entier((**int**) premier.getValeur() % (**int**) second.getValeur());

}

/\* non javadoc

\* @see Litteral#compareTo(Litteral)

\*/

@Override

**public** **int** compareTo(Litteral autre) {

**return** ((Integer)valeur).compareTo((Integer)autre.valeur);

}

}

### TestEntier.java

/\*\*

\* TestEntier.java 13 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.donnees.litteraux.tests;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.litteraux.Entier;

**import** **static** interpreteurlir.donnees.litteraux.Entier.\*;

**import** **static** info1.outils.glg.Assertions.\*;

**import** **static** java.lang.Integer.***MIN\_VALUE***;

**import** **static** java.lang.Integer.***MAX\_VALUE***;

/\*\*

\* Tests des méthode de la classe Entier

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestEntier {

/\*\* Jeu d'entiers correctement instanciés à partir d'un entier \*/

**private** **final** Entier[] ENTIERS\_INT = {

**new** Entier(***MIN\_VALUE***),

**new** Entier(***MAX\_VALUE***),

**new** Entier(1),

**new** Entier(-4587),

**new** Entier(-569),

**new** Entier(-3),

**new** Entier(0),

**new** Entier(2),

**new** Entier(78),

**new** Entier(781),

**new** Entier(179892),

};

/\*\* Jeu d'integers correspondants \*/

**private** **static** **final** **int**[] ***INT\_VALIDES*** = {

***MIN\_VALUE***,

***MAX\_VALUE***,

1,

-4587,

-569,

-3,

0,

2,

78,

781,

179892,

};

/\*\*

\* Test unitaire du constructeur Entier(String)

\*/

**public** **static** **void** testEntierString() {

**final** String[] INVALIDES = {

**null**,

"",

" ",

"\t",

"\n",

"a",

"michel",

"Janis Joplin",

"(93)",

" 78.3",

"2147483648756",

"2147483648",

"+2147483648",

"-2147483649",

"-",

"+",

};

System.***out***.println("\tExécution du test de Entier(String)");

**for** (**int** i = 0; i < INVALIDES.length; i++) {

**try** {

**new** Entier(INVALIDES[i]);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

}

/\*\*

\* Test unitaire de la méthode toString()

\*/

**public** **void** testToString() {

System.***out***.println("\tExécution du test de Entier(String)");

**for** (**int** i = 0; i < ***INT\_VALIDES***.length; i ++) {

*assertTrue*(ENTIERS\_INT[i].toString()

.compareTo(Integer.*toString*(***INT\_VALIDES***[i])) == 0);

}

}

/\*\*

\* Test unitaire de la méthode compareTo()

\*/

**public** **void** testCompareTo() {

**final** Entier REF\_MIN = **new** Entier(***MIN\_VALUE***);

**final** Entier REF\_MAX = **new** Entier(***MAX\_VALUE***);

System.***out***.println("\tExécution du test de compareTo()");

**for** (**int** i = 2; i < ENTIERS\_INT.length; i++) {

*assertTrue*(REF\_MIN.compareTo(ENTIERS\_INT[i]) < 0);

*assertTrue*(REF\_MAX.compareTo(ENTIERS\_INT[i]) > 0);

}

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

*assertTrue*(ENTIERS\_INT[i].compareTo(**new** Entier(***INT\_VALIDES***[i])) == 0);

}

}

/\*\*

\* Test unitaire de la méthode getValeur()

\*/

@SuppressWarnings("boxing")

**public** **void** testGetValeur() {

**final** Integer[] ATTENDUS = {

***MIN\_VALUE***,

***MAX\_VALUE***,

1,

-4587,

-569,

-3,

0,

2,

78,

781,

179892,

};

System.***out***.println("\tExécution du test de getValeur()");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

*assertTrue*(ENTIERS\_INT[i].getValeur().compareTo(ATTENDUS[i]) == 0);

}

}

/\*\*

\* Test unitaire de la méthode somme(Entier, Entier)

\*/

**public** **void** testSomme() {

**final** Entier[] ATTENDUS = {

**new** Entier(***MIN\_VALUE*** + ***MIN\_VALUE***),

**new** Entier(***MAX\_VALUE*** + ***MAX\_VALUE***),

**new** Entier(1 + 1),

**new** Entier(-9174),

**new** Entier(-1138),

**new** Entier(-6),

**new** Entier(0),

**new** Entier(4),

**new** Entier(156),

**new** Entier(1562),

**new** Entier(359784),

};

System.***out***.println("\tExécution du test de somme(Entier, Entier)");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

*assertTrue*(*somme*(ENTIERS\_INT[i], ENTIERS\_INT[i])

.compareTo(ATTENDUS[i]) == 0);

}

}

/\*\*

\* Test unitaire de la méthode soustrait()

\*/

**public** **void** testSoustrait() {

Entier zero = **new** Entier(0);

System.***out***.println("\tExécution du test de soustrait(Entier, Entier)");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

*assertTrue*(*soustrait*(ENTIERS\_INT[i], ENTIERS\_INT[i])

.compareTo(zero) == 0);

}

}

/\*\*

\* Test unitaire de la méthode multiplie()

\*/

**public** **void** testMultiplie() {

**final** Entier[] ATTENDUS = {

**new** Entier(***MIN\_VALUE*** \* ***MIN\_VALUE***),

**new** Entier(***MAX\_VALUE*** \* ***MAX\_VALUE***),

**new** Entier(1 \* 1),

**new** Entier(-4587 \* (-4587)),

**new** Entier(-569 \* (-569)),

**new** Entier(-3 \* (-3)),

**new** Entier(0 \* 0),

**new** Entier(2 \* 2),

**new** Entier(78 \* 78),

**new** Entier(781 \* 781),

**new** Entier(179892 \* 179892),

};

System.***out***.println("\tExécution du test de multiplie(Entier, Entier)");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

*assertTrue*(*multiplie*(ENTIERS\_INT[i], ENTIERS\_INT[i])

.compareTo(ATTENDUS[i]) == 0);

}

}

/\*\*

\* Test unitaire de la méthode quotient()

\*/

**public** **void** testQuotient() {

**final** Entier DIVISEUR = **new** Entier(2);

**final** Entier[] ATTENDUS = {

**new** Entier(-1073741824),

**new** Entier(1073741823),

**new** Entier(0),

**new** Entier(-2293),

**new** Entier(-284),

**new** Entier(-1),

**new** Entier(0),

**new** Entier(1),

**new** Entier(39),

**new** Entier(390),

**new** Entier(89946)

};

System.***out***.println("\tExécution du test de quotient(Entier, Entier)");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

*assertTrue*(*quotient*(ENTIERS\_INT[i], DIVISEUR)

.compareTo(ATTENDUS[i]) == 0);

}

}

/\*\*

\* Test unitaire de la méthode quotient()

\*/

**public** **void** testQuotientParZero() {

**final** Entier DIVISEUR = **new** Entier(0);

System.***out***.println("\tExécution du test de "

+ "quotient(Entier, Entier) par 0");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

**try** {

*quotient*(ENTIERS\_INT[i], DIVISEUR);

*echec*();

} **catch** (ExecutionException lancee) {

// Test OK

}

}

}

/\*\*

\* Test unitaire de la méthode quotient()

\*/

**public** **void** testReste() {

**final** Entier DIVISEUR = **new** Entier(2);

**final** Entier[] ATTENDUS = {

**new** Entier(0),

**new** Entier(1),

**new** Entier(1),

**new** Entier(-1),

**new** Entier(-1),

**new** Entier(-1),

**new** Entier(0),

**new** Entier(0),

**new** Entier(0),

**new** Entier(1),

**new** Entier(0)

};

System.***out***.println("\tExécution du test de reste(Entier, Entier)");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

*assertTrue*(*reste*(ENTIERS\_INT[i], DIVISEUR)

.compareTo(ATTENDUS[i]) == 0);

}

}

/\*\*

\* Test unitaire de la méthode quotient()

\*/

**public** **void** testResteParZero() {

**final** Entier DIVISEUR = **new** Entier(0);

System.***out***.println("\tExécution du test de "

+ "reste(Entier, Entier) par 0");

**for** (**int** i = 0; i < ENTIERS\_INT.length; i++) {

**try** {

*reste*(ENTIERS\_INT[i], DIVISEUR);

*echec*();

} **catch** (ExecutionException lancee) {

// Test OK

}

}

}

}

## Classe Booleen

### Booleen.java

/\*\*

\* Booleen.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.donnees.litteraux;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Constante littérale de type booléen

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** Booleen **extends** Litteral {

/\*\*

\* Initialise ce booléen

\* **@param** unBooleen valeur du booléen à construire

\* **@throws** InterpreteurException si la condition n'est pas un

\*

\*/

**public** Booleen(Boolean unBooleen) {

**super**();

valeur = unBooleen;

}

/\* non javadoc

\* @see interpreteurlir.donnees.litteraux.Litteral#getValeur()

\*/

@Override

**public** Boolean getValeur() {

**return** (Boolean) **super**.valeur;

}

/\* non javadoc

\* @deprecated

\* @see Litteral#compareTo(Litteral)

\*/

@Override

**public** **int** compareTo(Litteral autre) {

**return** (Boolean)**this**.valeur == (Boolean)autre.valeur ? 0 : 1; // égalité

}

}

### TestBooleen.java

/\*\*

\* TestBooleen.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.donnees.litteraux.tests;

**import** interpreteurlir.donnees.litteraux.Booleen;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Tests unitaires des méthodes de la classe Booleen

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestBooleen {

**private** **static** **final** Booleen[] ***FIXTURE*** = {

**new** Booleen(**false**),

**new** Booleen(**true**),

**new** Booleen(1<2),

**new** Booleen(1>=2),

**new** Booleen("bob".equals("bob")),

**new** Booleen(Character.*isDigit*('a')),

**new** Booleen(!Double.*isNaN*(1/0.0)),

};

/\*\*

\* Tests unitaire de {@link Booleen#getValeur()}

\*/

**public** **void** testGetValeur() {

**final** Boolean[] ATTENDUS = {

**false**,

**true**,

**true**,

**false**,

**true**,

**false**,

**true**

};

System.***out***.println("\tExécution du test de getValeur");

**for** (**int** i = 0 ; i < ATTENDUS.length ; i++) {

*assertTrue*(ATTENDUS[i].compareTo(***FIXTURE***[i].getValeur())

== 0);

}

}

}

# Paquetage interpreteurlir.donnees

## Classe Identificateur

### Identificateur.java

/\*

\* Identificateur.java , 8 mai 2021

\* IUT Rodez 2020-2021, info1

\* pas de copyright, aucun droits

\*/

**package** interpreteurlir.donnees;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Identificateur d'une variable.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **abstract** **class** Identificateur **implements** Comparable<Identificateur> {

/\*\* Longueur maximale d'un identificateur (ne prend pas en compte le $) \*/

**public** **static** **final** **int** ***LONGUEUR\_MAX*** = 25;

/\*\* Nom de cet identificateur \*/

**private** String nom;

/\*\*

\* Instantiation de l'identificateur

\* **@param** identificateur nom de l'identificateur

\*/

**public** Identificateur(String identificateur) {

**super**();

identificateur = identificateur.trim();

**if**(!*isIdentificateur*(identificateur)) {

**throw** **new** InterpreteurException(identificateur

+ " produit un résultat inattendu");

}

nom = identificateur;

}

/\*\*

\* Prédicat qui vérifie si une chaîne correspond à un identificateur

\* <ul>

\* <li>Longueur comprise entre 1 et 24 caractères</li>

\* <li>N'est pas une chaîne vide</li>

\* <li>N'est pas null</li>

\* </ul>

\* **@param** aTester chaîne à tester

\* **@return** true si le prédicat est vérifié

\* false sinon

\*/

**public** **static** **boolean** isIdentificateur(String aTester) {

**return** aTester != **null**

&& aTester.length() > 0

&& !*removeDollar*(aTester).isBlank()

&& *removeDollar*(aTester).length() <= ***LONGUEUR\_MAX***

&& *isAlphanumerique*(*removeDollar*(aTester));

}

/\*\*

\* Supprime le caractère $ si il est présent en premier dans la chaîne

\* **@param** chaine a modifier

\* **@return** la chaîne modifiée

\*/

**public** **static** String removeDollar(String chaine) {

**if**(chaine.charAt(0) == '$') **return** chaine.substring(1);

**return** chaine;

}

/\*\*

\* Prédicat testant si une chaîne est composée de chiffre ou de lettres

\* **@param** aTester chaîne a tester

\* **@return** true si le prédicat est vérifié

\* false sinon

\*/

**public** **static** **boolean** isAlphanumerique(String aTester) {

**int** index;

**for** (index = 0 ;

index < aTester.length()

&& Character.*isLetterOrDigit*(aTester.charAt(index)) ;

index++)

; /\* empty body \*/

**return** index >= aTester.length();

}

/\*\* **@return** la valeur de nom \*/

**public** String getNom() {

**return** nom;

}

/\* non javadoc - @see java.lang.Object#toString() \*/

@Override

**public** String toString() {

**return** nom;

}

/\* non javadoc - @see java.lang.String#Comparable() \*/

@Override

**public** **int** compareTo(Identificateur aComparer) {

**return** nom.compareTo(aComparer.nom);

}

}

### TestIdentificateur.java

// Tests faits avant le passage en abstract de la classe

/\*\*

\* TestIdentificateur.java 8 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.donnees.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.Identificateur;

/\*\*

\* Test de la classe donnees.Identificateur

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestIdentificateur {

/\*\* Jeu d'identificateurs correctement instanciés \*/

**public** **static** **final** Identificateur[] ***FIXTURE*** = {

**new** Identificateur("b"),

**new** Identificateur("A"),

**new** Identificateur("zalpha"),

**new** Identificateur("Alpha"),

**new** Identificateur("Alpha5"),

**new** Identificateur("jeSuisUnTresLongIdentifi"),

**new** Identificateur("$b"),

**new** Identificateur("z"),

**new** Identificateur("$zalpha"),

**new** Identificateur("$Alpha"),

**new** Identificateur("$Alpha5"),

**new** Identificateur("$jeSuisUnTresLongIdentifi")

};

/\*\*

\* Test de Identificateur(String identificateur)

\*/

**public** **static** **void** testIdentificateurString() {

**final** String[] INVALIDE = {

**null**,

"",

// Fait au maximum 25 caractères

"$jeSuisUnTresLongIdentificateur", // 30 char

"$jeSuisUnTresLongIdentifica",

// Espaces

"id 3a",

"$id 3a",

" ",

"$ ",

// caractères d'échapements

"\t",

"\n",

"$\t",

"$\n",

// , cas particulier

"$"

};

**for**(**int** noJeu = 0 ; noJeu < INVALIDE.length ; noJeu++) {

**try** {

**new** Identificateur(INVALIDE[noJeu]);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

}

/\*\*

\* Test de compareTo(Identificateur aComparer)

\*/

**public** **static** **void** testCompareTo() {

**final** Identificateur REF\_MIN = **new** Identificateur(

"$AAAAAAAAAAAAAAAAAAAAAAAA");

**final** Identificateur REF\_MAX = **new** Identificateur(

"zzzzzzzzzzzzzzzzzzzzzzzz");

**for**(**int** noJeu = 0; noJeu < ***FIXTURE***.length; noJeu++) {

*assertTrue*(***FIXTURE***[noJeu].compareTo(REF\_MIN) >= 0);

*assertTrue*(***FIXTURE***[noJeu].compareTo(REF\_MAX) <= 0);

*assertTrue*(***FIXTURE***[noJeu].compareTo(***FIXTURE***[noJeu]) == 0);

}

}

/\*\*

\* Tests unitaires de toString

\*/

**public** **static** **void** testToString() {

**final** String[] CHAINES\_VALIDES = {

"b",

"A",

"zalpha",

"Alpha",

"Alpha5",

"jeSuisUnTresLongIdentifi",

"$b",

"z",

"$zalpha",

"$Alpha",

"$Alpha5",

"$jeSuisUnTresLongIdentifi"

};

**for** (**int** noJeu = 0 ; noJeu < CHAINES\_VALIDES.length ; noJeu++) {

*assertEquivalence*(CHAINES\_VALIDES[noJeu],

***FIXTURE***[noJeu].toString());

}

}

## Classe IdentificateurChaine

### IdentificateurChaine.java

/\*

\* IdentificateurChaine.java, 08/05/2021

\* IUT Rodez 2020-2021, info1

\* pas de copyright, aucun droits

\*/

**package** interpreteurlir.donnees;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Identificateur de chaîne

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** IdentificateurChaine **extends** Identificateur {

/\*\*

\* Instantiation d'identificateur de chaîne

\* **@param** identificateur a instancier

\* **@throws** InterpreteurException si l'identificateur est invalide

\*/

**public** IdentificateurChaine(String identificateur) {

**super**(identificateur);

identificateur = identificateur.trim();

**if**(!*isIdentificateurChaine*(identificateur)) {

**throw** **new** InterpreteurException(identificateur

+ " n'est pas un identificateur"

+ " de chaine");

}

}

/\*\*

\* Prédicat attestant la validité de l'identificateur

\* **@param** identificateur à tester

\* **@return** true si l'identificateur est bien un identificateur d'entier

\* false sinon

\*/

**public** **static** **boolean** isIdentificateurChaine(String identificateur) {

**return** identificateur.length() >= 2

&& identificateur.charAt(0) == '$'

&& Character.*isLetter*(identificateur.charAt(1));

}

}

### TestIdentificateurChaine.java

/\*

\* TestIdentificateurChaine.java, 08/05/2021

\* IUT Rodez 2020-2021, info1

\* pas de copyright, aucun droits

\*/

**package** interpreteurlir.donnees.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurChaine;

/\*\*

\* Tests unitaires de la classe donnees.IdentificateurEntier

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestIdentificateurChaine {

/\*\* Jeu d'identificateurs de chaîne correctement instanciés \*/

**private** **static** IdentificateurChaine[] *FIXTURE* = {

**new** IdentificateurChaine("$a"),

**new** IdentificateurChaine("$A"),

**new** IdentificateurChaine("$alpha"),

**new** IdentificateurChaine("$Alpha"),

**new** IdentificateurChaine("$Alpha5"),

**new** IdentificateurChaine("$jeSuisUnTresLongIdentifi")

};

/\*\*

\* Tests unitaires du constructeur

\* IdentificateurEntier(String identificateur)

\*/

**public** **static** **void** testIdentificateurChaineString() {

**final** String[] INVALIDE = {

"",

// Commence par une lettre

"9alpha",

" 5alpha",

// Fait au maximum 25 caractères

"$jeSuisUnTresLongIdentificateur", // 30 char

"$jeSuisUnTresLongIdentifica",

// Espaces

"id 3a",

"$id 3a",

" ",

"$ ",

// caractères d'échapements

"\t",

"\n",

"$\t",

"$\n",

// , cas particulier

"$",

"$1"

};

System.***out***.println("\tExécution du test de "

+ "IdentificateurEntier(String identificateur)");

**for**(**int** noJeu = 0; noJeu < INVALIDE.length ; noJeu++) {

**try** {

**new** IdentificateurChaine(INVALIDE[noJeu]);

*echec*();

} **catch** (InterpreteurException lancee) {

// test OK

}

}

}

/\*\*

\* Tests unitaires de getNom()

\*/

**public** **static** **void** testGetNom() {

**final** String[] NOM\_VALIDES = {

"$a",

"$A",

"$alpha",

"$Alpha",

"$Alpha5",

"$jeSuisUnTresLongIdentifi"

};

System.***out***.println("\tExécution du test de getNom()");

**for** (**int** noJeu = 0 ; noJeu < NOM\_VALIDES.length ; noJeu++) {

*assertEquivalence*(NOM\_VALIDES[noJeu], *FIXTURE*[noJeu].getNom());

}

}

}

## Classe IdentificateurEntier

### IdentificateurEntier.java

/\*

\* IdentificateurEntier.java , 08/05/2021

\* IUT Rodez 2020-2021, info1

\* pas de copyright, aucun droits

\*/

**package** interpreteurlir.donnees;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Identificateur d'entier utilisé pour instancier des variables du même type.

\* Un identificateur entier commence par une lettre suivie d'au plus 24

\* caractère alphanumériques.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** IdentificateurEntier **extends** Identificateur {

/\*\*

\* Instantiation de cet identificateur d'entier avec le nom spécifié

\* en argument. Lève une exception si l'identificateur n'est pas

\* valide.

\* **@param** identificateur à instancier

\* **@throws** InterpreteurException si l'identificateur est invalide

\*/

**public** IdentificateurEntier(String identificateur) {

**super**(identificateur);

**if**(!*isIdentificateurEntier*(identificateur)) {

**throw** **new** InterpreteurException(identificateur

+ " n'est pas un identificateur"

+ " d'entier");

}

}

/\*\*

\* Prédicat attestant la validité de l'identificateur

\*

\* Un identificateur d'entier est valide si

\* - Il contient au maximum 24 caractères

\* - Commence obligatoirement par une lettre (majuscule ou minuscule)

\* - suivie uniquement de lettres (majuscule ou minuscule) ou de chiffres

\*

\* **@param** aTester à tester

\* **@return** true si l'identificateur est bien un identificateur d'entier

\* false sinon

\*/

**public** **static** **boolean** isIdentificateurEntier(String aTester) {

**return** aTester.length() <= 25 && Character.*isLetter*(aTester.charAt(0))

&& *isAlphanumerique*(aTester.substring(1));

}

}

### TestIdentificateurEntier.java

/\*

\* TestIdentificateurEntier.java , 08/05/2021

\* IUT Rodez 2020-2021, info1

\* pas de copyright, aucun droits

\*/

**package** interpreteurlir.donnees.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurEntier;

/\*\*

\* Tests unitaires de la classe donnees.IdentificateurEntier

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestIdentificateurEntier {

/\*\* Jeu d'identificateurs d'entier correctement instanciés \*/

**private** **static** IdentificateurEntier[] *FIXTURE* = {

**new** IdentificateurEntier("a"),

**new** IdentificateurEntier("A"),

**new** IdentificateurEntier("alpha"),

**new** IdentificateurEntier("Alpha"),

**new** IdentificateurEntier("Alpha5"),

**new** IdentificateurEntier("jeSuisUnTresLongIdentific")

};

/\*\*

\* Tests unitaires du constructeur IdentificateurEntier(String identificateur)

\*/

**public** **static** **void** testIdentificateurEntierString() {

**final** String[] INVALIDE = {

// Ne commence pas par une lettre

"9alpha",

" 5alpha",

"$beta",

// Fait plus de 25 caractères

"jeSuisUnTresLongIdentificateur", // 30 char

"jeSuisUnTresLongIdentifica",

// Espaces, caractères d'échapements, cas particulier

"id 3a",

"",

" ",

"\t",

"\n",

**null**

};

**for**(**int** noJeu = 0; noJeu < INVALIDE.length ; noJeu++) {

**try** {

**new** IdentificateurEntier(INVALIDE[noJeu]);

*echec*();

} **catch** (InterpreteurException lancee) {

// test OK

}

}

}

/\*\*

\* Tests unitaires de getNom()

\*/

**public** **static** **void** testGetNom() {

**final** String[] NOM\_VALIDES = {

"a",

"A",

"alpha",

"Alpha",

"Alpha5",

"jeSuisUnTresLongIdentific"

};

**for** (**int** noJeu = 0 ; noJeu < NOM\_VALIDES.length ; noJeu++) {

*assertEquivalence*(NOM\_VALIDES[noJeu], *FIXTURE*[noJeu].getNom());

}

}

}

## Classe Variable

### Variable.java

/\*\*

\* Variable.java 8 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.donnees;

**import** interpreteurlir.donnees.litteraux.Chaine;

**import** interpreteurlir.donnees.litteraux.Entier;

**import** interpreteurlir.donnees.litteraux.Litteral;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Associe un littéral à un identificateur

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*

\*/

**public** **class** Variable **extends** Object **implements** Comparable<Variable> {

/\*\* Identificateur de cette variable \*/

**private** Identificateur identificateur;

/\*\* Valeur de cette variable \*/

**private** Litteral valeur;

/\*\*

\* Initialise une variable

\* **@param** identificateur associé à cette variable

\* **@param** valeur associé à cette variable

\* **@throws** InterpreteurException en cas d'incompatibilité entre

\* l'identificateur et le type de la valeur à lui associer

\*/

**public** Variable(Identificateur identificateur, Litteral valeur) {

**if** (!*isVariable*(identificateur, valeur)) {

**throw** **new** InterpreteurException("Identificateur '"

+ identificateur.toString()

+ "' et type de "

+ valeur.toString()

+ "incompatible.");

}

**this**.identificateur = identificateur;

**this**.valeur = valeur;

}

**private** **static** **boolean** isVariable(Identificateur id, Litteral valeur) {

**return** id **instanceof** IdentificateurChaine && valeur **instanceof** Chaine

|| id **instanceof** IdentificateurEntier && valeur **instanceof** Entier;

}

/\*\*

\* **@return** la valeur de cette variable

\*/

**public** Litteral getValeur() {

**return** valeur;

}

/\*\*

\* Modifie la valeur de cette variable

\* **@param** nouvelleValeur valeur à affecter à cette variable

\*/

**public** **void** setValeur(Litteral nouvelleValeur) {

**if** (*isVariable*(identificateur, nouvelleValeur)) {

**this**.valeur = nouvelleValeur;

}

}

/\*\*

\* **@return** l'identificateur de cette variable

\*/

**public** Identificateur getIdentificateur() {

**return** identificateur;

}

/\* non javadoc @see java.lang.Object#toString() \*/

@Override

**public** String toString() {

**return** identificateur.toString() + " = " + valeur.toString();

}

/\*

\* non javadoc

\* @see java.lang.Comparable#compareTo(java.lang.Object)

\*/

**public** **int** compareTo(Variable aComparer) {

**return** identificateur.compareTo(aComparer.identificateur);

}

}

### TestVariable.java

/\*\*

\* TestVariable.java 8 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.donnees.tests;

**import** interpreteurlir.donnees.Variable;

**import** interpreteurlir.donnees.IdentificateurChaine;

**import** interpreteurlir.donnees.IdentificateurEntier;

**import** interpreteurlir.donnees.litteraux.\*;

**import** interpreteurlir.InterpreteurException;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Tests unitaires de la classe Variable

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucàs Vabre

\*/

**public** **class** TestVariable {

/\*\* Jeu d'identificateurs de chaîne valides \*/

**private** **static** **final** IdentificateurChaine[] ***ID\_CHAINE*** = {

**new** IdentificateurChaine("$a"),

**new** IdentificateurChaine("$B"),

**new** IdentificateurChaine("$alpha"),

**new** IdentificateurChaine("$Alpha"),

**new** IdentificateurChaine("$Alpha5"),

**new** IdentificateurChaine("$jeSuisUnTresLongIdentifi"),

**new** IdentificateurChaine("$R2D2"),

**new** IdentificateurChaine("$MichelSardou"),

**new** IdentificateurChaine("$PhilippePoutou2022")

};

/\*\* Jeu d'identificateurs d'entier valides \*/

**private** **static** **final** IdentificateurEntier[] ***ID\_ENTIER*** = {

**new** IdentificateurEntier("a"),

**new** IdentificateurEntier("A"),

**new** IdentificateurEntier("alpha"),

**new** IdentificateurEntier("Alpha"),

**new** IdentificateurEntier("Alpha5"),

**new** IdentificateurEntier("jeSuisUnTresLongIdentifi"),

**new** IdentificateurEntier("R2D2"),

**new** IdentificateurEntier("MichelSardou"),

**new** IdentificateurEntier("PhilippePoutou2022")

};

/\*\* Jeu de chaînes valides \*/

**private** **static** **final** Chaine[] ***VALEURS\_CHAINE*** = {

**new** Chaine(),

**new** Chaine("\"arztyehjklmpoijhghnbghjklmpoiuytrf"

+ "ghjnklmpoiuytrezaqsdfghnjklmpjbfrtyu\""),

**new** Chaine("\"\""),

**new** Chaine("\"coucou \""),

**new** Chaine("\"" + Integer.*toString*(42) + "\""),

**new** Chaine("\"Bidon\""),

**new** Chaine("\"toto\""),

**new** Chaine("\"tata\t\""),

**new** Chaine("\"titi\n\"")

};

/\*\* Jeu de variables chaîne valides\*/

**private** **static** Variable[] *fixtureChaine* = **new** Variable[***ID\_CHAINE***.length];

**private** **static** **void** fixtureReload() {

**for** (**int** i = 0; i < ***ID\_CHAINE***.length; i++) {

*fixtureChaine*[i] = **new** Variable(***ID\_CHAINE***[i], ***VALEURS\_CHAINE***[i]);

}

}

/\*\*

\* Test unitaire du constructeur Variable(Identificateur, Littéral)

\*/

**public** **static** **void** testVariableIdentificateurChaineLitteral() {

System.***out***.println("\tExécution du test de "

+ "Variable(Identificateur, Littéral)");

**for** (**int** noJeu = 0; noJeu < ***VALEURS\_CHAINE***.length; noJeu++) {

**try** {

**new** Variable(***ID\_ENTIER***[noJeu], ***VALEURS\_CHAINE***[noJeu]);

*echec*();

} **catch** (InterpreteurException lancee) {

// test OK

}

}

}

/\*\*

\* Test unitaire de getIdentificateur() d'une variable chaîne

\*/

**public** **static** **void** testGetIdentificateurChaine() {

*fixtureReload*();

System.***out***.println("\tExécution du test de getIdentificateur()");

**for** (**int** i = 0; i < ***VALEURS\_CHAINE***.length; i++ ) {

*assertTrue*(***ID\_CHAINE***[i].compareTo(*fixtureChaine*[i]

.getIdentificateur()) == 0);

}

}

/\*\*

\* Test unitaire de getValeur() d'une variable chaîne

\*/

**public** **static** **void** testGetValeurChaine() {

*fixtureReload*();

System.***out***.println("\tExécution du test de getValeur()");

**for** (**int** i = 0; i < ***VALEURS\_CHAINE***.length; i++ ) {

*assertTrue*(***VALEURS\_CHAINE***[i]

.compareTo(*fixtureChaine*[i].getValeur()) == 0);

}

}

/\*\*

\* Test unitaire de setValeur() d'une chaîne

\*/

**public** **static** **void** testSetValeurChaine() {

*fixtureReload*();

**final** Chaine[] NOUVELLE\_CHAINE = {

**new** Chaine("\"titi\""),

**new** Chaine("\"Mathématiques\""),

**new** Chaine("\"!?9563Message\""),

**new** Chaine("\"test TESTS\""),

**new** Chaine("\"-5 + 962\"")

};

System.***out***.println("\tExécution du test de setValeur()");

**for** (**int** i = 0; i < NOUVELLE\_CHAINE.length; i++) {

*fixtureChaine*[i].setValeur(NOUVELLE\_CHAINE[i]);

*assertTrue*(NOUVELLE\_CHAINE[i]

.compareTo(*fixtureChaine*[i].getValeur()) == 0);

}

}

/\*\*

\* Test unitaire de toString()

\*/

**public** **static** **void** testToString() {

*fixtureReload*();

**final** String[] ATTENDUS = {

"$a = \"\"",

"$B = \"arztyehjklmpoijhghnbghjklmpoiuytrf"

+ "ghjnklmpoiuytrezaqsdfghnjklmpjbfrtyu\"",

"$alpha = \"\"",

"$Alpha = \"coucou \"",

"$Alpha5 = \"42\"",

"$jeSuisUnTresLongIdentifi = \"Bidon\"",

"$R2D2 = \"toto\"",

"$MichelSardou = \"tata\t\"",

"$PhilippePoutou2022 = \"titi\n\""

};

System.***out***.println("\tExécution du test de toString()");

**for** (**int** noJeu = 0; noJeu < *fixtureChaine*.length; noJeu++ ) {

*assertEquivalence*(*fixtureChaine*[noJeu].toString(),

ATTENDUS[noJeu]);

}

}

/\*\*

\* Test unitaire de compareTo()

\*/

**public** **static** **void** testCompareTo() {

*fixtureReload*();

**final** Variable REF\_MIN

= **new** Variable(**new** IdentificateurChaine("$A"),

**new** Chaine("\"Min\""));

**final** Variable REF\_MAX

= **new** Variable(**new** IdentificateurChaine("$z"),

**new** Chaine("\"Max\""));

System.***out***.println("\tExécution du test de compareTo");

**for**(**int** noJeu = 0; noJeu < *fixtureChaine*.length; noJeu++) {

*assertTrue*(*fixtureChaine*[noJeu].compareTo(REF\_MIN) > 0);

*assertTrue*(*fixtureChaine*[noJeu].compareTo(REF\_MAX) < 0);

*assertTrue*(*fixtureChaine*[noJeu].compareTo(*fixtureChaine*[noJeu]) == 0);

}

}

}

# Paquetage interpreteurlir.expressions

## Classe Expression

### Expression.java

/\*\*

\* Expression.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.Identificateur;

**import** interpreteurlir.donnees.litteraux.Litteral;

/\*\*

\* Une expression contient tous les liens et données nécessaires à son calcul.

\* Une expression peut être calculée pour obtenir une valeur.

\* Elle peut affecter une valeur à une variable dans le contexte.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **abstract** **class** Expression {

/\*\* Index de l'operande gauche \*/

**protected** **static** **final** **int** ***INDEX\_OPERANDE\_G*** = 0;

/\*\* Index de l'operande droite \*/

**protected** **static** **final** **int** ***INDEX\_OPERANDE\_D*** = 1;

/\*\* Index de de l'identificateur pour l'affectation \*/

**protected** **static** **final** **int** ***INDEX\_AFFECTATION*** = 2;

/\*\* Index du premier symbole de l'opérateur \*/

**protected** **static** **final** **int** ***INDEX\_OPERATEUR\_G*** = 0;

/\*\* Index du second symbole de l'opérateur \*/

**protected** **static** **final** **int** ***INDEX\_OPERATEUR\_D*** = 1;

/\*\* Contexte global pour accéder aux données. \*/

**protected** **static** Contexte *contexteGlobal*;

/\*\* Identificateurs opérandes de cette l'expression \*/

**protected** Identificateur[] identificateursOperandes;

/\*\* Littéraux opérandes de cette expression \*/

**protected** Litteral[] litterauxOperandes;

/\*\*

\* Opérateur de cette expression potentiellement

\* composé de plusieurs symboles

\*/

**protected** **char**[] operateur;

/\*\*

\* Initialise une expression par défaut avec les liens nécessaires à

\* son calcul.

\*/

**protected** Expression() {

**super**();

**final** **int** NB\_IDENTIFICATEUR = 3;

**final** **int** NB\_LITTERAL = 2;

operateur = **new** **char**[2];

operateur[***INDEX\_OPERATEUR\_G***] = '\u0000';

operateur[***INDEX\_OPERATEUR\_D***] = '\u0000';

identificateursOperandes = **new** Identificateur[NB\_IDENTIFICATEUR];

litterauxOperandes = **new** Litteral[NB\_LITTERAL];

}

/\*\*

\* Calculer la valeur de cette expression à ce moment précis.

\* Peut accéder au contexte.

\* **@return** un Litteral de valeur du résultat de l'expression

\* **@throws** RuntimeException si le contexte n'est pas référencer

\* dans la classe Expression

\*/

**public** Litteral calculer() {

**if** (*contexteGlobal* == **null**) {

**throw** **new** RuntimeException("Le contexte doit être référencé "

+ "dans la classe Expression");

}

**return** **null**;

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

StringBuilder resultat = **new** StringBuilder("");

Identificateur affect = identificateursOperandes[***INDEX\_AFFECTATION***];

resultat.append(affect == **null** ? "" : (affect.toString() + " = "));

**if** (litterauxOperandes[***INDEX\_OPERANDE\_G***] != **null**) {

resultat.append(litterauxOperandes[***INDEX\_OPERANDE\_G***]);

} **else** {

resultat.append(identificateursOperandes[***INDEX\_OPERANDE\_G***]);

}

resultat.append(" ");

**if** (operateur[***INDEX\_OPERATEUR\_G***] != '\u0000') {

resultat.append(operateur[***INDEX\_OPERATEUR\_G***]);

}

**if** (operateur[***INDEX\_OPERATEUR\_D***] != '\u0000') {

resultat.append(operateur[***INDEX\_OPERATEUR\_D***]);

}

resultat.append(" ");

**if** (litterauxOperandes[***INDEX\_OPERANDE\_D***] != **null**) {

resultat.append(litterauxOperandes[***INDEX\_OPERANDE\_D***]);

} **else** **if** (identificateursOperandes[***INDEX\_OPERANDE\_D***] != **null**) {

resultat.append(identificateursOperandes[***INDEX\_OPERANDE\_D***]);

}

**return** resultat.toString().trim();

}

/\*\*

\* Référence le contexte pour accéder aux variables lors du calcul.

\* Le référencement vaut pour toutes les expressions

\* et est possible une unique fois.

\* **@param** aReferencer référence du contexte global

\* **@return** <ul><li>true si le contexte a pu être référencé</li>

\* <li>true si aReferencer == contexte déjà référencer</li>

\* <li>false si aReferencer est null</li>

\* <li>false si un contexte est déjà référencer</li>

\* </ul>

\*/

**public** **static** **boolean** referencerContexte(Contexte aReferencer) {

**if** (aReferencer != **null**

&& (*contexteGlobal* == **null** || aReferencer == *contexteGlobal*)) {

*contexteGlobal* = aReferencer;

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Détermine et créé une expression du bon type selon texteExpression.

\* Les types possibles sont ExpressionChaine ou ExpressionEntier.

\* **@param** texteExpression texte suivant la syntaxe d'une expression

\* **@return** l'expression du bon type correspondant à texteExpression

\* **@throws** InterpreteurException si texteExpression n'est pas valide

\* ou amène à une incohérence de type

\*/

**public** **static** Expression determinerTypeExpression(String texteExpression) {

String aTraiter = texteExpression.trim();

**if** (aTraiter.startsWith("$") || aTraiter.startsWith("\"")) {

**return** **new** ExpressionChaine(aTraiter);

} **else** {

**return** **new** ExpressionEntier(aTraiter);

}

}

/\*\*

\* Détermine l'index du caractère en dehors des constantes littérales

\* **@param** aTraiter chaîne à traiter

\* **@param** caractere opérateur à chercher hors guillemet

\* **@return** index dans à traiter du plus sinon -1 si aucun plus

\*/

**public** **static** **int** detecterCaractere(String aTraiter, **char** caractere) {

**char**[] aTester = aTraiter.toCharArray();

**int** indexPlus;

**int** nbGuillemet = 0;

**for** (indexPlus = 0 ;

indexPlus < aTester.length

&& (aTester[indexPlus] != caractere || nbGuillemet % 2 != 0) ;

indexPlus++) {

**if** (aTester[indexPlus] == '"') {

nbGuillemet++;

}

}

**return** indexPlus >= aTester.length ? -1 : indexPlus;

}

}

### TestExpression.java

/\*\*

\* TestExpression.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.expressions.ExpressionChaine;

**import** interpreteurlir.expressions.ExpressionEntier;

**import** interpreteurlir.Contexte;

/\*\*

\* Tests unitaires de {@link Expression}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestExpression {

/\*\*

\* Tests unitaires de {@link Expression#referencerContexte(Contexte)}

\*/

**public** **void** testReferencerContexte() {

Contexte reference = **new** Contexte();

Contexte[] contextes = {

**null**, reference, reference, **new** Contexte()

};

**boolean**[] resultatAttendu = { **false**, **true**, **true**, **false** };

System.***out***.println("\tExécution du test de "

+ "Expression#referencerContexte(Contexte)");

**for** (**int** numTest = 0 ; numTest < contextes.length ; numTest++) {

*assertTrue*( Expression.*referencerContexte*(contextes[numTest])

== resultatAttendu[numTest]);

}

}

/\*\*

\* Tests unitaires de {@link Expression#determinerTypeExpression(String)}

\*/

**public** **void** testDeterminerTypeExpression() {

**final** String[] TEXTE\_EXPRESSION = {

/\* Expression de type chaine \*/

"$chaine = \"texte\"",

"$chaine=\"tata\"",

" $tata \t ",

" \"une chaine de texte\"",

"$chaine= \"toto\"+\"titi\"",

" $chaine= $toto +\"titi\"",

"$chaine= \"toto\"+ $titi",

"$chaine=$toto +$titi",

" \"toto\"+\"titi\"",

"\t$toto +\"titi\"",

"\"toto\"+ $titi",

"$toto + $titi",

/\* Expression de type Entier \*/

"78",

" entier",

"78 %89",

" entier- nombre",

"\t entier/78",

"entier = 78 ",

" nombre= nombre + 78",

" entier =78 \*2"

};

**final** **int** INDEX\_DEBUT\_ENTIER = 12;

System.***out***.println("\tExécution du test de "

+ "Expression#determinerTypeExpression(String)");

**for** (**int** numTest = 0; numTest < TEXTE\_EXPRESSION.length ; numTest++) {

**if** (numTest < INDEX\_DEBUT\_ENTIER) {

*assertTrue*(Expression.*determinerTypeExpression*(

TEXTE\_EXPRESSION[numTest]) **instanceof** ExpressionChaine);

} **else** {

*assertTrue*(Expression.*determinerTypeExpression*(

TEXTE\_EXPRESSION[numTest]) **instanceof** ExpressionEntier);

}

}

}

}

## Classe ExpressionChaine

### ExpressionChaine.java

/\*\*

\* ExpressionChaine.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.Identificateur;

**import** interpreteurlir.donnees.IdentificateurChaine;

**import** interpreteurlir.donnees.litteraux.Chaine;

/\*\*

\* Expression de type Chaine qui peut être calculer.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** ExpressionChaine **extends** Expression {

/\*\* Opérateur possible pour ce type d'expression \*/

**private** **static** **final** **char** ***OPERATEUR*** = '+';

/\*\*

\* Initialise une expression de type Chaine

\* avec les liens nécessaires à son calcul.

\* **@param** texteExpression texte suivant la syntaxe d'une expression

\* **@throws** InterpreteurException si texteExpression n'est pas valide

\* ou amène à une incohérence de type

\*/

**public** ExpressionChaine(String texteExpression) {

**super**();

**final** String MESSAGE\_ERREUR = "une expression ne peut être vide";

String gauche;

String droite;

String aTraiter = texteExpression;

/\* chaîne null ou blanche \*/

**if** (aTraiter == **null** || aTraiter.isBlank()) {

**throw** **new** InterpreteurException(MESSAGE\_ERREUR);

}

aTraiter = aTraiter.trim();

/\* Traitement de la possible affectation \*/

**int** indexEgal = aTraiter.startsWith("$") ? aTraiter.indexOf('=') : -1;

**if** (indexEgal > -1) {

identificateursOperandes[***INDEX\_AFFECTATION***] =

**new** IdentificateurChaine(aTraiter.substring(0, indexEgal));

aTraiter = aTraiter.substring(indexEgal + 1);

}

/\* Traitement du nombre d'opérande \*/

**int** indexPlus = *detecterCaractere*(aTraiter, '+');

gauche = aTraiter;

**if** (indexPlus > -1) {

gauche = aTraiter.substring(0, indexPlus);

droite = aTraiter.substring(indexPlus + 1, aTraiter.length());

operateur[***INDEX\_OPERANDE\_G***] = ***OPERATEUR***;

initialiserOperande(droite, ***INDEX\_OPERANDE\_D***);

}

initialiserOperande(gauche, ***INDEX\_OPERANDE\_G***);

}

/\*\*

\* Initialise l'opérande à sa place dans l'expression.

\* **@param** operande représentation texte de l'opérande

\* **@param** index index de l'operande parmi INDEX\_OPERANDE\_G

\* et INDEX\_OPERANDE\_D

\* **@throws** IllegalArgumentException si index non valide

\*/

**private** **void** initialiserOperande(String operande, **int** index) {

**if** (***INDEX\_OPERANDE\_G*** != index && ***INDEX\_OPERANDE\_D*** != index) {

**throw** **new** IllegalArgumentException("index invalide");

}

**if** (Chaine.*isChaine*(operande)) {

litterauxOperandes[index] = **new** Chaine(operande);

} **else** {

identificateursOperandes[index] =

**new** IdentificateurChaine(operande);

}

}

/\* non javadoc

\* @see interpreteurlir.expressions.Expression#calculer()

\*/

@Override

**public** Chaine calculer() {

Chaine valeur;

**super**.calculer(); // exception levée si pas de contexte

/\* Détermine opérandeGauche \*/

Identificateur idGauche =

identificateursOperandes[***INDEX\_OPERANDE\_G***];

Chaine operandeG = (Chaine)(idGauche == **null**

? litterauxOperandes[***INDEX\_OPERANDE\_G***]

: *contexteGlobal*.lireValeurVariable(idGauche));

/\* Détermine possible operandeDroite \*/

Identificateur idDroite =

identificateursOperandes[***INDEX\_OPERANDE\_D***];

Chaine operandeD = **null**;

**if** (idDroite != **null** || litterauxOperandes[***INDEX\_OPERANDE\_D***] != **null**) {

operandeD = (Chaine)(idDroite == **null**

? litterauxOperandes[***INDEX\_OPERANDE\_D***]

: *contexteGlobal*.lireValeurVariable(idDroite));

}

/\* Calcul de la valeur \*/

valeur = operandeD == **null** ? operandeG

: Chaine.*concatener*(operandeG, operandeD);

/\* Affectation si nécessaire \*/

**if** (identificateursOperandes[***INDEX\_AFFECTATION***] != **null**) {

*contexteGlobal*.ajouterVariable(

identificateursOperandes[***INDEX\_AFFECTATION***], valeur);

}

**return** valeur;

}

}

### TestExpressionChaine.java

/\*\*

\* TestExpressionChaine.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurChaine;

**import** interpreteurlir.donnees.litteraux.Chaine;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.expressions.ExpressionChaine;

/\*\*

\* Tests unitaires de {@link ExpressionChaine}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestExpressionChaine {

/\*\* Jeu de tests d'expression chaîne valides\*/

**private** ExpressionChaine[] fixture = {

**new** ExpressionChaine("$chaine = \"texte\""),

**new** ExpressionChaine("$chaine=\"tata\""),

**new** ExpressionChaine(" $tata \t "),

**new** ExpressionChaine("\"une chaine de texte\""),

**new** ExpressionChaine("$chaine= \"toto\"+\"titi\""),

**new** ExpressionChaine("$chaine= $toto +\"titi\""),

**new** ExpressionChaine("$chaine= \"toto\"+ $titi"),

**new** ExpressionChaine("$chaine=$toto +$titi"),

**new** ExpressionChaine(" \"toto\"+\"titi\""),

**new** ExpressionChaine("$toto +\"titi\""),

**new** ExpressionChaine("\"toto\"+ $titi"),

**new** ExpressionChaine("$toto + $titi"),

**new** ExpressionChaine("\"ab=bc\""),

**new** ExpressionChaine("$chaine = \"ab+cd\"+$toto")

};

/\*\*

\* Tests unitaires de {@link ExpressionChaine#ExpressionChaine(String)}

\*/

**public** **void** testExpressionChaineString() {

**final** String[] INVALIDES = {

**null**,

"",

"3,1415", "3.1415", "1.7976931348623157E308",

"45", "-89",

"tata + $toto",

"\"chaine\" = $vie",

"$chaine / \"tata\"",

"£", "$"

};

**final** String[] VALIDES = {

"$chaine = \"texte\"",

"$chaine=\"tata\"",

" $tata \t ",

"\"une chaine de texte\"",

"$chaine= \"toto\"+\"titi\"",

"$chaine= $toto +\"titi\"",

"$chaine= \"toto\"+ $titi",

"$chaine=$toto +$titi",

" \"toto\"+\"titi\"",

"$toto +\"titi\"",

"\"toto\"+ $titi",

"$toto + $titi",

"\"ab=bc\"",

"$chaine = \"ab+cd\"+$toto"

};

System.***out***.println("\tExécution du test de "

+ "ExpressionChaine#ExpressionChaine(String)");

**for** (String texteArgs : INVALIDES) {

**try** {

**new** ExpressionChaine(texteArgs);

*echec*();

} **catch** (InterpreteurException lancee) {

// empty

}

}

**for** (String texteArgs : VALIDES) {

**try** {

**new** ExpressionChaine(texteArgs);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

}

/\*\*

\* Tests unitaires de {@link ExpressionChaine#calculer()}

\*/

**public** **void** testCalculer() {

**final** Chaine[] RESULTAT\_ATTENDU = {

**new** Chaine("\"texte\""),

**new** Chaine("\"tata\""),

**new** Chaine("\"\""),

**new** Chaine("\"une chaine de texte\""),

**new** Chaine("\"tototiti\""),

**new** Chaine("\"valTototiti\""),

**new** Chaine("\"toto\""),

**new** Chaine("\"valToto\""),

**new** Chaine("\"tototiti\""),

**new** Chaine("\"valTototiti\""),

**new** Chaine("\"toto\""),

**new** Chaine("\"valToto\""),

**new** Chaine("\"ab=bc\""),

**new** Chaine("\"ab+cdvalToto\"")

};

System.***out***.println("\tExécution du test de "

+ "ExpressionChaine#calculer()");

/\* Exception levée si contexte non référencé \*/

**try** {

fixture[0].calculer();

*echec*();

} **catch** (RuntimeException e) {

// vide

}

/\* Création contexte (avec $toto = "valToto") et référencement \*/

Contexte contexteGlobal = **new** Contexte();

contexteGlobal.ajouterVariable(**new** IdentificateurChaine("$toto"),

**new** Chaine("\"valToto\""));

Expression.*referencerContexte*(contexteGlobal);

System.***out***.print("\tContexte initial : \n" + contexteGlobal);

**for** (**int** numTest = 0; numTest < RESULTAT\_ATTENDU.length ; numTest++) {

System.***out***.println("\nCalcul de : " + fixture[numTest]);

*assertEquivalence*(fixture[numTest].calculer()

.compareTo(RESULTAT\_ATTENDU[numTest]), 0);

System.***out***.println("\tContexte : \n" + contexteGlobal);

}

}

/\*\*

\* Tests unitaires de {@link ExpressionChaine#toString()}

\*/

**public** **void** testToString() {

**final** String[] chaineAttendue = {

"$chaine = \"texte\"",

"$chaine = \"tata\"",

"$tata",

"\"une chaine de texte\"",

"$chaine = \"toto\" + \"titi\"",

"$chaine = $toto + \"titi\"",

"$chaine = \"toto\" + $titi",

"$chaine = $toto + $titi",

"\"toto\" + \"titi\"",

"$toto + \"titi\"",

"\"toto\" + $titi",

"$toto + $titi",

"\"ab=bc\"",

"$chaine = \"ab+cd\" + $toto",

};

System.***out***.println("\tExécution du test de "

+ "ExpressionChaine#toString()");

**for** (**int** numTest = 0 ; numTest < chaineAttendue.length ; numTest++) {

*assertEquivalence*(chaineAttendue[numTest],

fixture[numTest].toString());

}

}

}

## Classe ExpressionEntier

### ExpressionEntier.java

/\*\*

\* ExpressionEntier.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.Identificateur;

**import** interpreteurlir.donnees.IdentificateurEntier;

**import** interpreteurlir.donnees.litteraux.Entier;

/\*\*

\* Expression de type Entier qui peut être calculer.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** ExpressionEntier **extends** Expression {

/\*\* Liste des opérateurs possibles sur les entiers \*/

**private** **static** **final** **char**[] ***OPERATEURS*** = {'+', '-', '\*', '/', '%'};

/\*\* message d'erreur de chaîne null ou vide \*/

**private** **static** **final** String ***ERREUR\_VIDE*** =

"une expression ne peut être vide";

/\*\* Erreur opérande attendue \*/

**private** **static** **final** String ***OPERANDE\_D\_MANQUANT*** =

" attend un opérande droit";

/\*\*

\* Initialise une expression de type Entier avec les liens nécessaires à son

\* calcule.

\* **@param** texteExpression texte suivant la syntaxe d'une expression

\* **@throws** InterpreteurException si texteExpression n'est pas valide

\* ou amène à une incohérence de type

\*/

**public** ExpressionEntier(String texteExpression) {

**super**();

String gauche;

String droite;

String aTraiter;

**if** (texteExpression == **null** || texteExpression.isBlank()) {

**throw** **new** InterpreteurException(***ERREUR\_VIDE***);

}

aTraiter = texteExpression.trim();

/\* Traitement d'une éventuelle affectation \*/

**int** indexEgal = aTraiter.indexOf('=');

**if** (indexEgal > 0) {

identificateursOperandes[***INDEX\_AFFECTATION***] =

**new** IdentificateurEntier(aTraiter.substring(0, indexEgal)

.trim());

aTraiter = aTraiter.substring(indexEgal + 1).trim();

}

/\* Repérage de l'opérateur et de l'opérande droite s'ils existent \*/

**int** indexOperateur = *detecterOperateur*(aTraiter);

gauche = aTraiter.trim();

**if** (indexOperateur > 0) {

operateur[***INDEX\_OPERATEUR\_G***] = aTraiter.charAt(indexOperateur);

gauche = aTraiter.substring(0, indexOperateur).trim();

**if** (aTraiter.length() - 1 <= indexOperateur) {

**throw** **new** ExecutionException(aTraiter + ***OPERANDE\_D\_MANQUANT***);

}

droite = aTraiter.substring(indexOperateur + 1).trim();

initialiserOperande(droite, ***INDEX\_OPERANDE\_D***);

}

initialiserOperande(gauche, ***INDEX\_OPERANDE\_G***);

}

/\*\*

\* Détecte la présence d'un opérateur dans cette expression et renvoie

\* sa position

\* **@param** expression dont on cherche à connaître la position de l'opérande

\* **@return** position sous forme d'entier, -1 si pas d'opérateur

\*/

**private** **static** **int** detecterOperateur(String expression) {

**for** (**int** i = 1 ; i < expression.length() ; i++) {

**for** (**char** operateur : ***OPERATEURS***) {

**if** (operateur == expression.charAt(i)) {

**return** i;

}

}

}

**return** -1;

}

/\*\*

\* Initialise l'opérande à sa place dans l'expression.

\* **@param** operande opérande à initialiser

\* **@param** index de l'opérande

\*/

**private** **void** initialiserOperande(String operande, **int** index) {

**if** (***INDEX\_OPERANDE\_G*** != index && ***INDEX\_OPERANDE\_D*** != index) {

**throw** **new** IllegalArgumentException("index invalide");

}

**if** (Entier.*isEntier*(operande)) {

litterauxOperandes[index] = **new** Entier(operande);

} **else** {

identificateursOperandes[index] =

**new** IdentificateurEntier(operande);

}

}

/\* non javadoc

\* @see interpreteurlir.expressions.Expression#calculer()

\*/

@Override

**public** Entier calculer() {

Entier valeur;

**super**.calculer();

/\* Détermine opérandeGauche \*/

Identificateur idGauche =

identificateursOperandes[***INDEX\_OPERANDE\_G***];

Entier operandeG = (Entier)(idGauche == **null**

? litterauxOperandes[***INDEX\_OPERANDE\_G***]

: *contexteGlobal*.lireValeurVariable(idGauche));

/\* Détermine possible operandeDroite \*/

Identificateur idDroite =

identificateursOperandes[***INDEX\_OPERANDE\_D***];

Entier operandeD = **null**;

**if** (idDroite != **null** || litterauxOperandes[***INDEX\_OPERANDE\_D***] != **null**) {

operandeD = (Entier)(idDroite == **null**

? litterauxOperandes[***INDEX\_OPERANDE\_D***]

: *contexteGlobal*.lireValeurVariable(idDroite));

}

/\* Calcul de la valeur \*/

valeur = operandeD == **null**

? operandeG

: **switch** (operateur[***INDEX\_OPERATEUR\_G***]) {

**case** '+' -> Entier.*somme*(operandeG, operandeD);

**case** '-' -> Entier.*soustrait*(operandeG, operandeD);

**case** '\*' -> Entier.*multiplie*(operandeG, operandeD);

**case** '/' -> Entier.*quotient*(operandeG, operandeD);

**case** '%' -> Entier.*reste*(operandeG, operandeD);

**default** -> operandeG;

};

/\* Affectation si nécessaire \*/

**if** (identificateursOperandes[***INDEX\_AFFECTATION***] != **null**) {

*contexteGlobal*.ajouterVariable(

identificateursOperandes[***INDEX\_AFFECTATION***], valeur);

}

**return** valeur;

}

}

### TestExpressionEntier.java

/\*\*

\* TestExpressionEntier.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurEntier;

**import** interpreteurlir.donnees.litteraux.Entier;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.expressions.ExpressionEntier;

/\*\*

\* Tests unitaires de {@link ExpressionEntier}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestExpressionEntier {

/\* jeu de test d'expressions entières valides \*/

**private** **static** **final** ExpressionEntier[] ***FIXTURE*** = {

**new** ExpressionEntier("entier = 2 + 3"),

**new** ExpressionEntier("entier=2\*3"),

**new** ExpressionEntier("bob= marcel-2"),

**new** ExpressionEntier("45 +14"),

**new** ExpressionEntier("45 \* -2"),

**new** ExpressionEntier("affectation = 64"),

**new** ExpressionEntier("affectation= marcel"),

**new** ExpressionEntier("entier = j34n + pi3rr3"),

**new** ExpressionEntier(" entier = j34n"),

**new** ExpressionEntier(" 42"),

**new** ExpressionEntier("rep0ns3= 42"),

**new** ExpressionEntier("division = 12/0"),

**new** ExpressionEntier("modulo = 12%0")

};

/\*\*

\* Tests unitaires de {@link ExpressionEntier#ExpressionEntier(String)}

\*/

**public** **static** **void** testExpressionEntierString() {

**final** String[] INVALIDES = {

/\* identificateurs non valides \*/

"$bob =2",

"j@ck= 2+3",

"@75S= #michel",

"unidentificateurbeaucouptroplong = 0",

"truc.length = 9000",

/\* types non compatibles \*/

"resultat = \"50\"",

"resultat = 30.2",

"resultat = 10 / 2.0",

/\* Nombre incorrect d'opérandes \*/

"resultat = 10 \* 5 + 3",

"famille = marcel + jean + albert",

"divisionRatee = 5 /",

"ratee=\*7",

};

System.***out***.println("\tExécution du test de "

+ "ExpressionEntier#ExpressionEntier(String)");

**for** (String invalide : INVALIDES) {

**try** {

**new** ExpressionEntier(invalide);

*echec*();

} **catch**(InterpreteurException | ExecutionException lancee) {

// Empty body

}

}

}

/\*\*

\* Tests unitaires de {@link ExpressionEntier#calculer()}

\*/

**public** **static** **void** testCalculer() {

**final** Entier[] RESULTATS\_ATTENDUS = {

**new** Entier(5),

**new** Entier(6),

**new** Entier(-2),

**new** Entier(59),

**new** Entier(-90),

**new** Entier(64),

**new** Entier(0),

**new** Entier(3),

**new** Entier(1),

**new** Entier(42),

**new** Entier(42),

**new** Entier(0), // Bouchon

**new** Entier(0) // Bouchon

};

System.***out***.println("\tExécution du test de "

+ "ExpressionEntier#calculer()");

/\* Exception levée si contexte non référencé \*/

**try** {

***FIXTURE***[0].calculer();

*echec*();

} **catch** (RuntimeException e) {

// vide

}

/\*

\* Création contexte (avec marcel = 0 j34n = 1 et pi3rr3 = 2) et

\* référencement

\*/

Contexte contexteGlobal = **new** Contexte();

contexteGlobal.ajouterVariable(**new** IdentificateurEntier("marcel"),

**new** Entier(0));

contexteGlobal.ajouterVariable(**new** IdentificateurEntier("j34n"),

**new** Entier(1));

contexteGlobal.ajouterVariable(**new** IdentificateurEntier("pi3rr3"),

**new** Entier(2));

Expression.*referencerContexte*(contexteGlobal);

System.***out***.print("\tContexte initial : \n" + contexteGlobal);

**for** (**int** i = 0 ; i < ***FIXTURE***.length ; i++) {

**try** {

System.***out***.println("\nCalcul de : " + ***FIXTURE***[i]);

*assertTrue*(***FIXTURE***[i].calculer()

.compareTo(RESULTATS\_ATTENDUS[i]) == 0);

System.***out***.println("\tContexte : \n" + contexteGlobal);

} **catch** (ExecutionException divzero) {

System.***out***.println("Attention Division par 0");

}

}

}

/\*\*

\* test de toString()

\*/

**public** **static** **void** testToString() {

**final** String[] ATTENDUES = {

"entier = 2 + 3",

"entier = 2 \* 3",

"bob = marcel - 2",

"45 + 14",

"45 \* -2",

"affectation = 64",

"affectation = marcel",

"entier = j34n + pi3rr3",

"entier = j34n",

"42",

"rep0ns3 = 42",

"division = 12 / 0",

"modulo = 12 % 0"

};

System.***out***.println("\tExécution du test de "

+ "ExpressionEntier#toString()");

**for** (**int** i = 0 ; i < ***FIXTURE***.length ; i++) {

*assertTrue*(***FIXTURE***[i].toString().equals(ATTENDUES[i]));

}

}

}

## Classe ExpressionBooleenne

### ExpressionBooleenne.java

/\*\*

\* ExpressionBooleenne.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions;

**import** **static** interpreteurlir.donnees.IdentificateurChaine

.*isIdentificateurChaine*;

**import** **static** interpreteurlir.donnees.IdentificateurEntier

.*isIdentificateurEntier*;

**import** **static** interpreteurlir.donnees.litteraux.Entier.*isEntier*;

**import** **static** interpreteurlir.donnees.litteraux.Chaine.*isChaine*;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurChaine;

**import** interpreteurlir.donnees.IdentificateurEntier;

**import** interpreteurlir.donnees.litteraux.\*;

/\*\*

\* Expression de type Booleen qui peut être évaluée.

\* <p>

\* Syntaxe d'une expression logique : opérande1 oprel opérande2

\* <p>

\* Les expressions logiques concerneront donc toujours deux opérandes

\* séparés par un opérateur relationnel (notation infixe). Un opérande

\* est soit une constante, soit un identificateur.

\* <p>

\* L'’opérateur relationnel oprel est un symbole parmi : {@code = <> < <= > >=}

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** ExpressionBooleenne **extends** Expression {

/\*\* Liste des opérateurs relationnels utilisés \*/

**private** **static** **char**[] *OPERATEURS* = { '<', '>', '=' };

**private** **static** **final** String ***ERREUR\_ARGUMENT*** = "une expression ne peut être "

+ "vide";

**private** **static** **final** String ***ERREUR\_SYNTAXE*** =

"usage <opérande1> <oprel> <opérande2> \n"

+ "avec oprel comme opérateur relationnel "

+ "un des symboles suivants : <> < <= > >="

+ "et comme opérandes des constantes, "

+ "ou alors des identificateurs";

**private** **static** **final** String ***ERREUR\_TYPE*** = "opérande invalide "

+ "ou type incompatible";

**private** **static** **final** String ***ERREUR\_OPERATEUR*** = "opérateur inconnu";

/\*\*

\* Initialise une expression de type Booleen avec les liens

\* nécessaires à son calcul.

\*

\* **@param** texteExpression l'expression booléenne lue sous forme

\* de chaîne

\* **@throws** InterpreteurException si l'expression est vide ou les

\* types des opérandes sont incompatibles

\* ou si l'opérande droit est manquant

\*/

**public** ExpressionBooleenne(String texteExpression) {

**super**();

String gauche;

String droite;

String aTraiter;

**if** (texteExpression == **null** || texteExpression.isBlank()) {

**throw** **new** InterpreteurException(***ERREUR\_ARGUMENT***);

}

aTraiter = texteExpression.trim();

**int**[] indexOperateurs = affecterOperateur(aTraiter,

*detecterOperateurs*(aTraiter));

**if** (indexOperateurs[***INDEX\_OPERATEUR\_D***] > aTraiter.length() - 2

|| indexOperateurs[***INDEX\_OPERATEUR\_G***] <= 0) {

**throw** **new** InterpreteurException(aTraiter + ***ERREUR\_SYNTAXE***);

}

gauche = aTraiter.substring(0, indexOperateurs[***INDEX\_OPERATEUR\_G***])

.trim();

droite = aTraiter.substring(indexOperateurs[***INDEX\_OPERATEUR\_D***] + 1)

.trim();

**if** (!*isMemeType*(gauche, droite)) {

**throw** **new** InterpreteurException(***ERREUR\_TYPE***);

}

initialiserOperande(gauche, ***INDEX\_OPERANDE\_G***);

initialiserOperande(droite, ***INDEX\_OPERANDE\_D***);

}

/\*\*

\* Affecte les symboles de l'opérateur à partir des index de indexOperateur

\* correspondant à des caractères dans aTraiter.

\* **@param** aTraiter chaîne contenant les opérateurs

\* **@param** indexOperateurs index des symboles de l'opérateur.

\* **@return** tableau d'index avec l'index de début de l'opérateur en indice 0

\* et l'index de la fin de l'opérateur en indice 1.

\* Les index peuvent être égaux.

\* **@throws** InterpreteurException si les symboles de l'opérateur ne

\* se suivent pas dans aTraiter

\*

\*/

**private** **int**[] affecterOperateur(String aTraiter, **int**[] indexOperateurs) {

operateur[***INDEX\_OPERATEUR\_G***] = indexOperateurs[***INDEX\_OPERATEUR\_G***] <= 0

? '\u0000'

: aTraiter.charAt(indexOperateurs[***INDEX\_OPERATEUR\_G***]);

operateur[***INDEX\_OPERATEUR\_D***] = indexOperateurs[***INDEX\_OPERATEUR\_D***] <= 0

? '\u0000'

: aTraiter.charAt(indexOperateurs[***INDEX\_OPERATEUR\_D***]);

**if** (indexOperateurs[***INDEX\_OPERATEUR\_G***] <= 0) {

indexOperateurs[***INDEX\_OPERATEUR\_G***] =

indexOperateurs[***INDEX\_OPERATEUR\_D***];

} **else** **if** (indexOperateurs[***INDEX\_OPERATEUR\_D***] <= 0) {

indexOperateurs[***INDEX\_OPERATEUR\_D***] =

indexOperateurs[***INDEX\_OPERATEUR\_G***];

}

**if** (indexOperateurs[***INDEX\_OPERATEUR\_D***]

- indexOperateurs[***INDEX\_OPERATEUR\_G***] > 1

|| !isOperateurValide()) {

**throw** **new** InterpreteurException(***ERREUR\_OPERATEUR***);

}

**return** indexOperateurs;

}

/\*\*

\* Prédicat de validité de concordance des symboles

\* de l'opérateur à faire un opérateur valide

\* **@return** true si opérateur formé par les symboles est valide, false sinon

\*/

**private** **boolean** isOperateurValide() {

**final** String[] OPERATEUR\_VALIDE = {

"<", ">", "<=", ">=", "=", "<>"

};

String aTester = "";

**if** (operateur[***INDEX\_OPERATEUR\_G***] != '\u0000') {

aTester = aTester + operateur[***INDEX\_OPERATEUR\_G***];

}

**if** (operateur[***INDEX\_OPERATEUR\_D***] != '\u0000') {

aTester = aTester + operateur[***INDEX\_OPERATEUR\_D***];

}

**int** index;

**for** (index = 0 ;

index < OPERATEUR\_VALIDE.length

&& !OPERATEUR\_VALIDE[index].equals(aTester);

index++)

; /\* empty body \*/

**return** index < OPERATEUR\_VALIDE.length;

}

/\*\*

\* Prédicat de validité de compatibilité

\* entre les opérandes gauche et droite

\* **@param** gauche opérande gauche

\* **@param** droit opérande droit

\* **@return** true si deux opérandes sont de même type

\* sinon false

\*/

**private** **static** **boolean** isMemeType(String gauche, String droit) {

**return** ( (*isIdentificateurEntier*(gauche) || *isEntier*(gauche))

&& (*isIdentificateurEntier*(droit) || *isEntier*(droit)))

||

( (*isIdentificateurChaine*(gauche) || *isChaine*(gauche)))

&& (*isIdentificateurChaine*(droit) || *isChaine*(droit));

}

/\*\*

\* Détecte les opérateurs d'une expression logique

\* **@param** aTraiter chaine dont on cherche les opérateurs

\* **@return** les indexes de début et de fin du premier et unique

\* opérateur trouvé

\* **@throws** InterpreteurException s'il l'opérateur est invalide

\* ou inexistant

\*/

**private** **static** **int**[] detecterOperateurs(String aTraiter) {

**int**[] index = **new** **int**[2];

**char** charCourant;

index[***INDEX\_OPERATEUR\_G***] = -1;

index[***INDEX\_OPERATEUR\_D***] = -1;

**int** nbGuillemet = 0;

**for** (**int** i = 0 ; i < aTraiter.length() - 1 ; i++) {

charCourant = aTraiter.charAt(i);

**if** (charCourant == '"') {

nbGuillemet++;

}

**if** (index[***INDEX\_OPERATEUR\_G***] < 0 && (nbGuillemet & 1) == 0

&& ( charCourant == *OPERATEURS*[0]

|| charCourant == *OPERATEURS*[1])) {

index[***INDEX\_OPERATEUR\_G***] = i;

} **else** **if** ((nbGuillemet & 1) == 0

&& (charCourant == *OPERATEURS*[1]

|| charCourant == *OPERATEURS*[2])) {

index[***INDEX\_OPERATEUR\_D***] = i;

}

}

**if** (index[***INDEX\_OPERATEUR\_G***] == index[***INDEX\_OPERATEUR\_D***]) {

**throw** **new** InterpreteurException(***ERREUR\_OPERATEUR***);

}

**return** index;

}

/\*\*

\* Initialise l'opérande à sa place dans l'Expression.

\* **@param** operande

\* **@param** index

\* **@throws** IllegalArgumentException si index invalide

\*/

**private** **void** initialiserOperande(String operande, **int** index) {

**if** (***INDEX\_OPERANDE\_G*** != index && ***INDEX\_OPERANDE\_D*** != index) {

**throw** **new** IllegalArgumentException("index invalide");

}

**if** (*isIdentificateurEntier*(operande)) {

identificateursOperandes[index] =

**new** IdentificateurEntier(operande);

} **else** **if** (*isIdentificateurChaine*(operande)) {

identificateursOperandes[index] =

**new** IdentificateurChaine(operande);

} **else** **if** (*isChaine*(operande)) {

litterauxOperandes[index] = **new** Chaine(operande);

} **else** {

litterauxOperandes[index] = **new** Entier(operande);

}

}

/\* non javadoc

\* @see interpreteurlir.expressions.Expression#calculer()

\*/

@Override

**public** Booleen calculer() {

Litteral gauche = litterauxOperandes[***INDEX\_OPERANDE\_G***] != **null**

? litterauxOperandes[***INDEX\_OPERANDE\_G***]

: *contexteGlobal*.lireValeurVariable(

identificateursOperandes[***INDEX\_OPERANDE\_G***]);

Litteral droite = litterauxOperandes[***INDEX\_OPERANDE\_D***] != **null**

? litterauxOperandes[***INDEX\_OPERANDE\_D***]

: *contexteGlobal*.lireValeurVariable(

identificateursOperandes[***INDEX\_OPERANDE\_D***]);

**return** **new** Booleen(calculAvecOperateur(gauche, droite));

}

/\*\*

\* Calcule la valeur de l'expression selon l'opérateur

\* à partir de l'opérande gauche et droite

\* Les opérande doivent être du même type.

\* **@param** gauche opérande gauche

\* **@param** droite opérande droite

\* **@return** true si expression true sinon false

\*/

**private** **boolean** calculAvecOperateur(Litteral gauche, Litteral droite) {

**boolean** resultat = **false**;

**if** (operateur[***INDEX\_OPERATEUR\_G***] == *OPERATEURS*[0]) {

resultat = gauche.compareTo(droite) < 0;

} **else** **if** (operateur[***INDEX\_OPERATEUR\_G***] == *OPERATEURS*[1]) {

resultat = gauche.compareTo(droite) > 0;

}

**if** (operateur[***INDEX\_OPERATEUR\_D***] == *OPERATEURS*[1]) {

resultat = resultat || gauche.compareTo(droite) > 0;

} **else** **if** (operateur[***INDEX\_OPERATEUR\_D***] == *OPERATEURS*[2]) {

resultat = resultat || gauche.compareTo(droite) == 0;

}

**return** resultat;

}

}

### TestExpressionBooleenne.java

/\*\*

\* TestExpressionBooleenne.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.expressions.tests;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurChaine;

**import** interpreteurlir.donnees.IdentificateurEntier;

**import** interpreteurlir.donnees.litteraux.Chaine;

**import** interpreteurlir.donnees.litteraux.Entier;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.expressions.ExpressionBooleenne;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Tests unitaires des méthodes de la classe ExpressionBooleenne

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestExpressionBooleenne {

**private** **final** ExpressionBooleenne[] FIXTURE\_LITTERALE = {

/\* Expression logique sur des Entiers AVEC ESPACES \*/

**new** ExpressionBooleenne("1 = 1"), // true

**new** ExpressionBooleenne("1 = 2"), // false

**new** ExpressionBooleenne("1 < 2"),

**new** ExpressionBooleenne("1 < 1"),

**new** ExpressionBooleenne("1 <> 2"),

**new** ExpressionBooleenne("1 <> 1"),

**new** ExpressionBooleenne("1 <= 1"),

**new** ExpressionBooleenne("1 <= 5"),

**new** ExpressionBooleenne("1 > -3"),

**new** ExpressionBooleenne("1 > 56"),

**new** ExpressionBooleenne("1 >= 1"),

**new** ExpressionBooleenne("1 >= 45"),

/\* Expression logique sur des Entiers SANS ESPACES \*/

**new** ExpressionBooleenne("1=1"),

**new** ExpressionBooleenne("1=2"),

**new** ExpressionBooleenne("1<2"),

**new** ExpressionBooleenne("1<1"),

**new** ExpressionBooleenne("1<>2"),

**new** ExpressionBooleenne("1<>1"),

**new** ExpressionBooleenne("1<=1"),

**new** ExpressionBooleenne("1<=5"),

**new** ExpressionBooleenne("1>-3"),

**new** ExpressionBooleenne("1>56"),

**new** ExpressionBooleenne("1>=1"),

**new** ExpressionBooleenne("1>=45"),

/\* Expression logique sur des Entiers MOITIE ESPACES \*/

**new** ExpressionBooleenne(" 1=1"),

**new** ExpressionBooleenne("1=2 "),

**new** ExpressionBooleenne("1 <2"),

**new** ExpressionBooleenne("1< 1"),

**new** ExpressionBooleenne("1 <>2"),

**new** ExpressionBooleenne("1<> 1"),

**new** ExpressionBooleenne(" 1<=1 "),

**new** ExpressionBooleenne("1 <= 5"),

**new** ExpressionBooleenne(" 1 > -3 "),

**new** ExpressionBooleenne("1>56\t"),

**new** ExpressionBooleenne(" 1 >= 1 "),

**new** ExpressionBooleenne(" 1 >= 45 "),

/\* Expression logique sur des Chaines AVEC ESPACES \*/

**new** ExpressionBooleenne("\"TATA\" = \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" = \"TITI\""),

**new** ExpressionBooleenne("\"TATA\" < \"TITI\""),

**new** ExpressionBooleenne("\"TOTO\" < \"TITI\""),

**new** ExpressionBooleenne("\"TOTO\" <> \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" <> \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" <= \"TATA\""),

**new** ExpressionBooleenne("\"TITI\" <= \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" > \"FOO BAR\""),

**new** ExpressionBooleenne("\"FOO BAR\" > \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" >= \"TATA\""),

**new** ExpressionBooleenne("\"FOO BAR\" >= \"TATA\""),

/\* Expression logique sur des Chaines SANS ESPACES \*/

**new** ExpressionBooleenne("\"TATA\"=\"TATA\""),

**new** ExpressionBooleenne("\"TATA\"=\"TITI\""),

**new** ExpressionBooleenne("\"TATA\"<\"TITI\""),

**new** ExpressionBooleenne("\"TOTO\"<\"TITI\""),

**new** ExpressionBooleenne("\"TOTO\"<>\"TATA\""),

**new** ExpressionBooleenne("\"TATA\"<>\"TATA\""),

**new** ExpressionBooleenne("\"TATA\"<=\"TATA\""),

**new** ExpressionBooleenne("\"TITI\"<=\"TATA\""),

**new** ExpressionBooleenne("\"TATA\">\"FOO BAR\""),

**new** ExpressionBooleenne("\"FOO BAR\">\"TATA\""),

**new** ExpressionBooleenne("\"TATA\">=\"TATA\""),

**new** ExpressionBooleenne("\"FOO BAR\">=\"TATA\""),

/\* Expression logique sur des Chaines MOITIE ESPACES \*/

**new** ExpressionBooleenne(" \"TATA\" = \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" = \"TITI\""),

**new** ExpressionBooleenne("\"TATA\" < \"TITI\""),

**new** ExpressionBooleenne("\"TOTO\" < \"TITI\" "),

**new** ExpressionBooleenne(" \"TOTO\"<> \"TATA\" "),

**new** ExpressionBooleenne("\"TATA\" <> \"TATA\" "),

**new** ExpressionBooleenne(" \"TATA\" <=\"TATA\""),

**new** ExpressionBooleenne("\"TITI\" <= \"TATA\""),

**new** ExpressionBooleenne(" \"TATA\" > \"FOO BAR\""),

**new** ExpressionBooleenne(" \"FOO BAR \" > \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" >= \"TATA\""),

**new** ExpressionBooleenne(" \"FOO BAR\" >= \"TATA\""),

/\* Expression logique sur des Chaines AVEC OPERATEURS \*/

**new** ExpressionBooleenne("\"FOO BAR\"<>\"TATA=TOTO\""),

**new** ExpressionBooleenne("\"FOO BAR=FLEMME\">\"TOTO\""),

**new** ExpressionBooleenne("\"FOO BAR > FLEMME\"=\"TOTO\""),

**new** ExpressionBooleenne("\"FOO BAR<>FLEMME\">\"TOTO\""),

};

**private** **final** ExpressionBooleenne[] FIXTURE\_ID = {

/\* Expression logique sur des IdEntier et Entiers \*/

**new** ExpressionBooleenne("marcel <= 10"), // true

**new** ExpressionBooleenne("marcel > j34n"), // false

**new** ExpressionBooleenne("2 = pi3rr3"),

**new** ExpressionBooleenne("j34n = pi3rr3"),

/\* Expression logique sur des IdChaine et Chaines \*/

**new** ExpressionBooleenne("$sanchis < $barrios"),

**new** ExpressionBooleenne("$servieres > \"Windows\""),

**new** ExpressionBooleenne("$barrios <> $servieres"),

**new** ExpressionBooleenne("\"coucou\" = $barrios"),

};

/\*\*

\* Tests unitaire de {@link ExpressionBooleenne#ExpressionBooleenne(String)}

\*/

**public** **void** testExpressionBooleenne() {

**final** String[] INVALIDES = {

/\* Pas d'opérateur \*/

"",

"2 5",

"\"John Doe\"",

"\"Foo bar\" $serpillere",

"entier -20",

/\* Opérateurs invalides \*/

"-89 + 67",

"-8979 % 7",

"35 \* 12",

"89 / 12",

"65 - 74",

"\"Foo bar\" + $serpillere",

"ab >> cd",

/\* Expressions logiques avec opérateurs invalides \*/

"78 < = 45",

"entier > = 56",

"\"Foo bar\" < > $serpillere",

"$coucou >< $dollarchaine",

"78 => 45",

"32 =< 61",

"32 == 61",

/\* Plus de 2 opérandes et 1 opérateur \*/

"65 <> 45 = 45",

"entier > 85 && 45 = 12",

"entier <= 85 || 45 <> 12",

/\* Caractères entre les opérandes et l'opérateur \*/

"\"Foo bar\" . > $serpillere",

"\"Foo bar\" < \_ $serpillere",

"\"Foo bar\" + $balai > serpillere",

/\* opérande manquant \*/

">= entier",

"entier =",

"<>",

/\* Incompatibilité entre types d'opérandes \*/

"\"Foo bar\" > serpillere",

"serpillere <> \"Foo bar\"",

"15 > $coucou",

"\"coucou\" <> 45",

"$coucou = entier"

};

System.***out***.println("\tExécution du test de ExpressionBooleenne()");

**for** (String aTester : INVALIDES) {

**try** {

**new** ExpressionBooleenne(aTester);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

**try** {

/\* Expression logique sur des Entiers AVEC ESPACES \*/

**new** ExpressionBooleenne("1 = 1"); // true

**new** ExpressionBooleenne("1 = 2"); // false

**new** ExpressionBooleenne("1 < 2");

**new** ExpressionBooleenne("1 < 1");

**new** ExpressionBooleenne("1 <> 2");

**new** ExpressionBooleenne("1 <> 1");

**new** ExpressionBooleenne("1 <= 1");

**new** ExpressionBooleenne("1 <= 5");

**new** ExpressionBooleenne("1 > -3");

**new** ExpressionBooleenne("1 > 56");

**new** ExpressionBooleenne("1 >= 1");

**new** ExpressionBooleenne("1 >= 45");

/\* Expression logique sur des Entiers SANS ESPACES \*/

**new** ExpressionBooleenne("1=1");

**new** ExpressionBooleenne("1=2");

**new** ExpressionBooleenne("1<2");

**new** ExpressionBooleenne("1<1");

**new** ExpressionBooleenne("1<>2");

**new** ExpressionBooleenne("1<>1");

**new** ExpressionBooleenne("1<=1");

**new** ExpressionBooleenne("1<=5");

**new** ExpressionBooleenne("1>-3");

**new** ExpressionBooleenne("1>56");

**new** ExpressionBooleenne("1>=1");

**new** ExpressionBooleenne("1>=45");

/\* Expression logique sur des Entiers MOITIE ESPACES \*/

**new** ExpressionBooleenne(" 1=1");

**new** ExpressionBooleenne("1=2 ");

**new** ExpressionBooleenne("1 <2");

**new** ExpressionBooleenne("1< 1");

**new** ExpressionBooleenne("1 <>2");

**new** ExpressionBooleenne("1<> 1");

**new** ExpressionBooleenne(" 1<=1 ");

**new** ExpressionBooleenne("1 <= 5");

**new** ExpressionBooleenne(" 1 > -3 ");

**new** ExpressionBooleenne("1>56\t");

**new** ExpressionBooleenne(" 1 >= 1 ");

**new** ExpressionBooleenne(" 1 >= 45 ");

/\* Expression logique sur des Chaines AVEC ESPACES \*/

**new** ExpressionBooleenne("\"TATA\" = \"TATA\"");

**new** ExpressionBooleenne("\"TATA\" = \"TITI\"");

**new** ExpressionBooleenne("\"TATA\" < \"TITI\"");

**new** ExpressionBooleenne("\"TOTO\" < \"TITI\"");

**new** ExpressionBooleenne("\"TOTO\" <> \"TATA\"");

**new** ExpressionBooleenne("\"TATA\" <> \"TATA\"");

**new** ExpressionBooleenne("\"TATA\"<=\"TATA\"");

**new** ExpressionBooleenne("\"TITI\" <= \"TATA\"");

**new** ExpressionBooleenne("\"TATA\" > \"FOO BAR\"");

**new** ExpressionBooleenne("\"FOO BAR\" > \"TATA\"");

**new** ExpressionBooleenne("\"TATA\" >= \"TATA\"");

**new** ExpressionBooleenne("\"FOO BAR\" >= \"TATA\"");

/\* Expression logique sur des Chaines SANS ESPACES \*/

**new** ExpressionBooleenne("\"TATA\"=\"TATA\"");

**new** ExpressionBooleenne("\"TATA\"=\"TITI\"");

**new** ExpressionBooleenne("\"TATA\"<\"TITI\"");

**new** ExpressionBooleenne("\"TOTO\"<\"TITI\"");

**new** ExpressionBooleenne("\"TOTO\"<>\"TATA\"");

**new** ExpressionBooleenne("\"TATA\"<>\"TATA\"");

**new** ExpressionBooleenne("\"TATA\"<=\"TATA\"");

**new** ExpressionBooleenne("\"TITI\"<=\"TATA\"");

**new** ExpressionBooleenne("\"TATA\">\"FOO BAR\"");

**new** ExpressionBooleenne("\"FOO BAR\">\"TATA\"");

**new** ExpressionBooleenne("\"TATA\">=\"TATA\"");

**new** ExpressionBooleenne("\"FOO BAR\">=\"TATA\"");

/\* Expression logique sur des Chaines MOITIE ESPACES \*/

**new** ExpressionBooleenne(" \"TATA\" = \"TATA\"");

**new** ExpressionBooleenne("\"TATA\" = \"TITI\"");

**new** ExpressionBooleenne("\"TATA\" < \"TITI\"");

**new** ExpressionBooleenne("\"TOTO\" < \"TITI\" ");

**new** ExpressionBooleenne(" \"TOTO\"<> \"TATA\" ");

**new** ExpressionBooleenne("\"TATA\" <> \"TATA\" ");

**new** ExpressionBooleenne(" \"TATA\" <=\"TATA\"");

**new** ExpressionBooleenne("\"TITI\" <= \"TATA\"");

**new** ExpressionBooleenne(" \"TATA\" > \"FOO BAR\"");

**new** ExpressionBooleenne(" \"FOO BAR \" > \"TATA\"");

**new** ExpressionBooleenne("\"TATA\" >= \"TATA\"");

**new** ExpressionBooleenne(" \"FOO BAR\" >= \"TATA\"");

/\* Expression logique sur des Chaines AVEC OPERATEURS \*/

**new** ExpressionBooleenne("\"FOO BAR\"<>\"TATA=TOTO\"");

**new** ExpressionBooleenne("\"FOO BAR=FLEMME\">\"TOTO\"");

**new** ExpressionBooleenne("\"FOO BAR > FLEMME\"=\"TOTO\"");

**new** ExpressionBooleenne("\"FOO BAR<>FLEMME\">\"TOTO\"");

/\* Expression logique sur des IdEntier et Entiers \*/

**new** ExpressionBooleenne("marcel <= 10"); // true

**new** ExpressionBooleenne("marcel > j34n"); // false

**new** ExpressionBooleenne("2 = pi3rr3");

**new** ExpressionBooleenne("j34n = pi3rr3");

/\* Expression logique sur des IdChaine et Chaines \*/

**new** ExpressionBooleenne("$sanchis < $barrios");

**new** ExpressionBooleenne("$servieres > \"Windows\"");

**new** ExpressionBooleenne("$barrios <> $servieres");

**new** ExpressionBooleenne("\"coucou\" = $barrios");

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Tests unitaire de {@link ExpressionBooleenne#calculer()}

\*/

**public** **void** testCalculer() {

**final** Boolean[] VALEUR\_ATTENDU\_ID = {

**true**, **false**, **true**, **false**, **true**, **false**, **true**, **false**

};

Contexte contexteGlobal = **new** Contexte();

contexteGlobal.ajouterVariable(**new** IdentificateurEntier("marcel"),

**new** Entier(0));

contexteGlobal.ajouterVariable(**new** IdentificateurEntier("j34n"),

**new** Entier(1));

contexteGlobal.ajouterVariable(**new** IdentificateurEntier("pi3rr3"),

**new** Entier(2));

contexteGlobal.ajouterVariable(**new** IdentificateurChaine("$sanchis"),

**new** Chaine("\"coucou\""));

contexteGlobal.ajouterVariable(**new** IdentificateurChaine("$barrios"),

**new** Chaine("\"java\""));

contexteGlobal.ajouterVariable(**new** IdentificateurChaine("$servieres"),

**new** Chaine("\"WinDesign\""));

Expression.*referencerContexte*(contexteGlobal);

System.***out***.println("\tExécution du test de Calculer()");

**for** (**int** numTest = 0 ; numTest < VALEUR\_ATTENDU\_ID.length ; numTest++) {

*assertEquivalence*(VALEUR\_ATTENDU\_ID[numTest],

FIXTURE\_ID[numTest].calculer().getValeur());

}

**final** ExpressionBooleenne[] A\_TESTER = {

**new** ExpressionBooleenne("1=1"),

**new** ExpressionBooleenne("1=2"),

**new** ExpressionBooleenne("1<2"),

**new** ExpressionBooleenne("1<1"),

**new** ExpressionBooleenne("1<>2"),

**new** ExpressionBooleenne("1<>1"),

**new** ExpressionBooleenne("1<=1"),

**new** ExpressionBooleenne("1<=5"),

**new** ExpressionBooleenne("1>-3"),

**new** ExpressionBooleenne("1>56"),

**new** ExpressionBooleenne("1>=1"),

**new** ExpressionBooleenne("1>=45"),

**new** ExpressionBooleenne("\"TATA\" = \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" = \"TITI\""),

**new** ExpressionBooleenne("\"TATA\" < \"TITI\""),

**new** ExpressionBooleenne("\"TOTO\" < \"TITI\""),

**new** ExpressionBooleenne("\"TOTO\" <> \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" <> \"TATA\""),

**new** ExpressionBooleenne("\"TATA\"<=\"TATA\""),

**new** ExpressionBooleenne("\"TITI\" <= \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" > \"FOO BAR\""),

**new** ExpressionBooleenne("\"FOO BAR\" > \"TATA\""),

**new** ExpressionBooleenne("\"TATA\" >= \"TATA\""),

**new** ExpressionBooleenne("\"FOO BAR\" >= \"TATA\""),

};

**final** Boolean[] VALEUR\_ATTENDU\_L = {

**true**, **false**, **true**, **false**, **true**, **false**,

**true**, **true**, **true**, **false**, **true**, **false**,

**true**, **false**, **true**, **false**, **true**, **false**,

**true**, **false**, **true**, **false**, **true**, **false**

};

**for** (**int** numTest = 0 ; numTest < A\_TESTER.length ; numTest++) {

*assertEquivalence*(VALEUR\_ATTENDU\_L[numTest],

A\_TESTER[numTest].calculer().getValeur());

}

}

/\*\*

\* Tests unitaire de {@link ExpressionBooleenne#toString()}

\*/

**public** **void** testToString() {

System.***out***.println("\tExécution du test de toString()");

**final** String[] ATTENDU\_L = {

"1 = 1",

"1 = 2",

"1 < 2",

"1 < 1",

"1 <> 2",

"1 <> 1",

"1 <= 1",

"1 <= 5",

"1 > -3",

"1 > 56",

"1 >= 1",

"1 >= 45",

"1 = 1",

"1 = 2",

"1 < 2",

"1 < 1",

"1 <> 2",

"1 <> 1",

"1 <= 1",

"1 <= 5",

"1 > -3",

"1 > 56",

"1 >= 1",

"1 >= 45",

"1 = 1",

"1 = 2",

"1 < 2",

"1 < 1",

"1 <> 2",

"1 <> 1",

"1 <= 1",

"1 <= 5",

"1 > -3",

"1 > 56",

"1 >= 1",

"1 >= 45",

"\"TATA\" = \"TATA\"",

"\"TATA\" = \"TITI\"",

"\"TATA\" < \"TITI\"",

"\"TOTO\" < \"TITI\"",

"\"TOTO\" <> \"TATA\"",

"\"TATA\" <> \"TATA\"",

"\"TATA\" <= \"TATA\"",

"\"TITI\" <= \"TATA\"",

"\"TATA\" > \"FOO BAR\"",

"\"FOO BAR\" > \"TATA\"",

"\"TATA\" >= \"TATA\"",

"\"FOO BAR\" >= \"TATA\"",

"\"TATA\" = \"TATA\"",

"\"TATA\" = \"TITI\"",

"\"TATA\" < \"TITI\"",

"\"TOTO\" < \"TITI\"",

"\"TOTO\" <> \"TATA\"",

"\"TATA\" <> \"TATA\"",

"\"TATA\" <= \"TATA\"",

"\"TITI\" <= \"TATA\"",

"\"TATA\" > \"FOO BAR\"",

"\"FOO BAR\" > \"TATA\"",

"\"TATA\" >= \"TATA\"",

"\"FOO BAR\" >= \"TATA\"",

"\"TATA\" = \"TATA\"",

"\"TATA\" = \"TITI\"",

"\"TATA\" < \"TITI\"",

"\"TOTO\" < \"TITI\"",

"\"TOTO\" <> \"TATA\"",

"\"TATA\" <> \"TATA\"",

"\"TATA\" <= \"TATA\"",

"\"TITI\" <= \"TATA\"",

"\"TATA\" > \"FOO BAR\"",

"\"FOO BAR \" > \"TATA\"",

"\"TATA\" >= \"TATA\"",

"\"FOO BAR\" >= \"TATA\"",

"\"FOO BAR\" <> \"TATA=TOTO\"",

"\"FOO BAR=FLEMME\" > \"TOTO\"",

"\"FOO BAR > FLEMME\" = \"TOTO\"",

"\"FOO BAR<>FLEMME\" > \"TOTO\"",

};

**for** (**int** numTest = 0 ; numTest < ATTENDU\_L.length ; numTest++) {

*assertEquivalence*(ATTENDU\_L[numTest],

FIXTURE\_LITTERALE[numTest].toString());

}

**final** String[] ATTENDU\_I = {

"marcel <= 10",

"marcel > j34n",

"2 = pi3rr3",

"j34n = pi3rr3",

"$sanchis < $barrios",

"$servieres > \"Windows\"",

"$barrios <> $servieres",

"\"coucou\" = $barrios",

};

**for** (**int** numTest = 0 ; numTest < ATTENDU\_I.length ; numTest++) {

*assertEquivalence*(ATTENDU\_I[numTest],

FIXTURE\_ID[numTest].toString());

}

}

}

# Paquetage interpreteurlir

## Classe InterpreteurException

### InterpreteurException.java

/\*\*

\* InterpreteurException.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir;

/\*\*

\* Exception levée lors d'une erreur dans l'interpreteur LIR.

\* (Erreur de syntaxe, erreur de types)

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

@SuppressWarnings("serial")

**public** **class** InterpreteurException **extends** RuntimeException {

/\*\*

\* Une exception de syntaxe expliquée par un message

\* **@param** message explication succincte de cette exception

\*/

**public** InterpreteurException(String message) {

**super**(message);

}

}

### EssaiInterpreteurException.java

/\*\*

\* EssaiInterpreteurException.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.tests;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Essai des {@link InterpreteurException}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** EssaiInterpreteurException {

/\*\*

\* Lancement des essais.

\* **@param** args non utilisé

\*/

**public** **static** **void** main(String[] args) {

String[] messages = {

**null**,

"",

"la commande fin n'accepte pas d'arguments"

};

**for** (String msg : messages) {

System.***out***.print("Message de l'exception : ");

**try** {

**throw** **new** InterpreteurException(msg);

} **catch** (InterpreteurException lancee) {

System.***out***.println(lancee.getMessage());

}

}

}

}

## Classe ExecutionException

### ExecutionException.java

/\*\*

\* ExecutionException.java 12 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir;

/\*\*

\* Exception levée lors d'une erreur dans l'exécution d'un programme

\* dans l'interpréteurLIR.

\* (Ex: division par 0)

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

@SuppressWarnings("serial")

**public** **class** ExecutionException **extends** RuntimeException {

/\*\*

\* Initialise cette exception avec un message.

\* **@param** message explication succincte de cette exception

\*/

**public** ExecutionException(String message) {

**super**(message);

}

}

### Tests

Les tests reviendraient à tester RuntimeException car aucune fonctionnalité n’est ajoutée. De plus InterpreteurException qui est très similaire a été testée.

## Classe Contexte

### Contexte.java

/\*\*

\* Contexte.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir;

**import** java.util.ArrayList;

**import** interpreteurlir.donnees.\*;

**import** interpreteurlir.donnees.litteraux.\*;

/\*\*

\* Le contexte contiens l'ensemble des variables définies au cours d'un session

\* d'interpréteur LIR.

\* Il est le seul moyen d'accéder et modifier ces variables.

\* Il sert donc d'interface entre le programme et les variables.

\* L'accès à une variable se fait à partir d'un Identificateur.

\* Les variables sont listée par ordre alphabétique des identificateurs.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** Contexte {

/\*\* Valeur par défaut pour une Chaine \*/

**public** **static** **final** Chaine ***CHAINE\_DEFAUT*** = **new** Chaine("\"\"");

/\*\* Valeur par défaut pour un Entier \*/

**public** **static** **final** Entier ***ENTIER\_DEFAUT*** = **new** Entier(0);

/\*\* Listes des variables définies dans ce contexte \*/

**private** ArrayList<Variable> variables;

/\*\*

\* Initialise un contexte vide (aucune variable de définie)

\*/

**public** Contexte() {

**super**();

variables = **new** ArrayList<Variable>();

}

/\*\*

\* Affecte valeur à la variable dans le contexte

\* ayant pour identificateur id.

\* Si cette variable n'existe pas alors elle est créée

\* et référence dans le contexte.

\* **@param** id identificateur typé de la Variable

\* **@param** valeur valeur typée à affecter à la variable

\* **@throws** InterpreteurException si Variable lance l'exception.

\* {@link Variable#Variable(Identificateur, Litteral)}

\*/

**public** **void** ajouterVariable(Identificateur id, Litteral valeur) {

**int** indexVar = indexVariable(id);

/\* Ajout de variable absente à la fin \*/

**if** (variables.size() == indexVar) {

variables.add(indexVar, **new** Variable(id, valeur));

**return**;

}

// else

Variable varAIndex = variables.get(indexVar);

/\* Variable déjà présente \*/

**if** (varAIndex.getIdentificateur().compareTo(id) == 0) {

varAIndex.setValeur(valeur);

} **else** {

/\* Variable absente : ajout à son index trié \*/

variables.add(indexVar, **new** Variable(id, valeur));

}

}

/\*\*

\* Lecture de la valeur d'une variable du contexte.

\* **@param** id identificateur de la variable dont on veut la valeur

\* **@return** Si la variable est définie alors sa valeur est retournée.

\* Sinon la valeur par défaut (CHAINE\_DEFAUT ou ENTIER\_DEFAUT)

\* selon type de id est retournée.

\*/

**public** Litteral lireValeurVariable(Identificateur id) {

Litteral valeur;

**int** indexVar = indexVariable(id);

**if** (variables.size() > indexVar

&& variables.get(indexVar).getIdentificateur().compareTo(id)

== 0) {

/\* La variables est présente \*/

valeur = variables.get(indexVar).getValeur();

} **else** **if** (id **instanceof** IdentificateurChaine) {

valeur = ***CHAINE\_DEFAUT***;

} **else** {

valeur = ***ENTIER\_DEFAUT***;

}

**return** valeur;

}

/\*\*

\* Recherche de l'index théorique ou réel de la variable ayant id comme

\* identificateur.

\* **@param** id identificateur dont on cherche l'index de la variable

\* **@return** index de la Variable ayant id comme identificateur.

\* Si la variable n'existe pas alors c'est l'index de la place

\* théorique de variable pour qu'elle soit triée qui est renvoyé.

\*/

**private** **int** indexVariable(Identificateur id) {

**int** index;

**int** taille = variables.size();

**for** (index = 0; index < taille; index++) {

**if** (variables.get(index).getIdentificateur().compareTo(id) >= 0) {

**return** index;

}

}

**return** index;

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

**final** String MSG\_VIDE = "aucune variable n'est définie\n";

**int** taille = variables.size();

**if** (taille == 0) {

**return** MSG\_VIDE;

}

// else

StringBuilder resultat = **new** StringBuilder("");

**for** (**int** index = 0 ; index < taille ; index++) {

resultat.append(variables.get(index) + "\n");

}

**return** resultat.toString();

}

/\*\*

\* Remise à zéro du contexte.

\* Efface toutes les variables mémorisée par le contexte.

\*/

**public** **void** raz() {

variables.clear();

}

}

### TestContexte.java

/\*\*

\* TestContexte.java 8 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.donnees.\*;

**import** interpreteurlir.donnees.litteraux.\*;

/\*\*

\* Tests unitaires de {@link Contexte}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestContexte {

/\*\* Jeux de tests de Contexte \*/

**private** Contexte[] fixture = {

**new** Contexte(), **new** Contexte(), **new** Contexte(),

};

/\*\*

\* Tests unitaires de {@link Contexte#Contexte()}

\*/

**public** **void** testContexte() {

System.***out***.println("\tExécution du test de Contexte#Contexte()");

**try** {

**new** Contexte();

} **catch** (Exception e) {

*echec*();

}

}

/\*\*

\* Tests unitaires de

\* {@link Contexte#ajouterVariable(Identificateur, Litteral)}

\*/

**public** **void** testAjouterVariable() {

Identificateur[] id = {

**new** IdentificateurChaine("$chaine"), // ajout dans liste vide

**new** IdentificateurEntier("entier"), // ajout fin

**new** IdentificateurChaine("$zoro"), // ajout milieu

// modif variable présente

**new** IdentificateurChaine("$chaine"), // ajout dans liste vide

**new** IdentificateurEntier("entier"), // ajout fin

**new** IdentificateurChaine("$zoro"), // ajout milieu

**new** IdentificateurChaine("$abcd"), // ajout debut

};

Litteral[] valeur = {

**new** Chaine("\"blabla\""),

**new** Entier(25),

**new** Chaine("\"Zoro le héro\""),

**new** Chaine("\"viveLa Vie\""),

**new** Entier(-1),

**new** Chaine("\" ah ah ! \""),

**new** Chaine("\"lol\""),

};

System.***out***.println("\tExécution du test de "

+ "Contexte#ajouterVariable(Identificateur, Litteral)");

**for** (**int** numAjout = 0 ; numAjout < id.length ; numAjout++) {

fixture[0].ajouterVariable(id[numAjout], valeur[numAjout]);

**if** (numAjout == 2) {

System.***out***.println(fixture[0].toString());

}

}

System.***out***.println(fixture[0].toString());

}

/\*\*

\* Tests unitaires de {@link Contexte#toString()}

\*/

**public** **void** testToString() {

String[] chaineAttendue = {

"aucune variable n'est définie\n",

"aucune variable n'est définie\n",

"aucune variable n'est définie\n",

};

System.***out***.println("\tExécution du test de Contexte#toString()");

**for** (**int** numTest = 0 ; numTest < chaineAttendue.length ; numTest++) {

*assertEquivalence*(fixture[numTest].toString(),

chaineAttendue[numTest]);

}

}

/\*\*

\* Tests unitaires de {@link Contexte#raz()}

\*/

**public** **void** testRaz() {

String toStringVide = "aucune variable n'est définie\n";

// fixture 0 est vide

// fixture 1 a 3 éléments à vider

fixture[1].ajouterVariable(**new** IdentificateurChaine("$chaine"),

**new** Chaine("\"blabla\""));

fixture[1].ajouterVariable(**new** IdentificateurEntier("entier"),

**new** Entier(25));

fixture[1].ajouterVariable(**new** IdentificateurChaine("$zoro"),

**new** Chaine("\"Zoro le héro\""));

// fixture 2 a 1 éléments unique

fixture[1].ajouterVariable(**new** IdentificateurChaine("$zer"),

**new** Chaine("\"blvzgr\""));

System.***out***.println("\tExécution du test de Contexte#raz()");

**for** (Contexte aTester : fixture) {

aTester.raz();

// toString doit être celui d'un contexte vide

*assertEquivalence*(toStringVide, aTester.toString());

}

}

/\*\*

\* Tests unitaire de {@link Contexte#lireValeurVariable(Identificateur)}

\*/

**public** **void** testLireValeurVariable() {

System.***out***.println("\tExécution du test de "

+ "Contexte#lireValeurVariable(Identificateur)");

// lire valeur défaut contexte vid

*assertEquivalence*(fixture[0].lireValeurVariable(

**new** IdentificateurChaine("$chaine")).getValeur(), "");

*assertEquivalence*(fixture[0].lireValeurVariable(

**new** IdentificateurEntier("entier")).getValeur(),

Integer.*valueOf*(0));

// lire valeur par défaut dans contexte non vide

fixture[1].ajouterVariable(**new** IdentificateurChaine("$zoro"),

**new** Chaine("\"Zoro le héro\""));

*assertEquivalence*(fixture[1].lireValeurVariable(

**new** IdentificateurChaine("$chaine")).getValeur(), "");

*assertEquivalence*(fixture[1].lireValeurVariable(

**new** IdentificateurEntier("entier")).getValeur(), 0);

// lire valeur qui existent déjà

fixture[1].ajouterVariable(**new** IdentificateurChaine("$chaine"),

**new** Chaine("\"blabla\""));

fixture[1].ajouterVariable(**new** IdentificateurEntier("entier"),

**new** Entier(25));

System.***out***.println(fixture[1].lireValeurVariable(

**new** IdentificateurChaine("$zoro")).getValeur());

*assertEquivalence*(fixture[1].lireValeurVariable(

**new** IdentificateurChaine("$chaine")).getValeur(), "blabla");

*assertEquivalence*(fixture[1].lireValeurVariable(

**new** IdentificateurEntier("entier")).getValeur(), 25);

*assertEquivalence*(fixture[1].lireValeurVariable(

**new** IdentificateurChaine("$zoro")).getValeur(),

"Zoro le héro");

}

}

## Classe Analyseur

### Analyseur.java

/\*\*

\* Analyseur.java 9 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir;

**import** java.lang.reflect.InvocationTargetException;

**import** java.util.Scanner;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.instructions.Instruction;

**import** interpreteurlir.programmes.\*;

/\*\*

\* Analyseur de l'entrée standard du programme interpréteur LIR.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** Analyseur {

/\*\* Symbole d'invite de commande \*/

**public** **static** **final** String ***INVITE*** = "? ";

/\*\* Message feedback ok \*/

**public** **static** **final** String ***OK\_FEEDBACK*** = "ok";

/\*\* Message feedback not ok \*/

**public** **static** **final** String ***NOK\_FEEDBACK*** = "nok : ";

/\*\* analyseur de l'entrée standard \*/

**private** Scanner entree;

/\*\* contexte de cet analyseur \*/

**private** Contexte contexteGlobal;

/\*\* programme de cet analyseur \*/

**private** Programme programme;

/\*\*

\* Initialise un analyseur ayant son propre contexte.

\*/

**public** Analyseur() {

**super**();

entree = **new** Scanner(System.***in***);

contexteGlobal = **new** Contexte();

Expression.*referencerContexte*(contexteGlobal);

programme = **new** Programme();

Commande.*referencerProgramme*(programme);

}

/\*\*

\* Lance la boucle qui demande puis exécute des commandes ou instructions

\* saisies par l'utilisateur.

\*/

**public** **void** mainLoop() {

String ligneSaisie;

String[] decoupage;

String motCle;

String arguments;

String texteEtiquette;

**for** (;;) {

texteEtiquette = **null**;

System.***out***.print(***INVITE***);

ligneSaisie = entree.nextLine().trim();

/\* Instruction avec étiquette \*/

**if** (!ligneSaisie.isBlank()

&& Character.*isDigit*(ligneSaisie.charAt(0))) {

decoupage = ligneSaisie.split(" ", 2);

texteEtiquette = decoupage.length >= 1 ? decoupage[0] : "";

ligneSaisie = decoupage.length >= 2 ? decoupage[1] : "";

}

decoupage = ligneSaisie.split(" ", 2);

motCle = decoupage.length >= 1 ? decoupage[0] : "";

arguments = decoupage.length >= 2 ? decoupage[1] : "";

**if** (texteEtiquette == **null**) {

executerCommande(motCle, arguments.trim());

} **else** {

editerProgramme(texteEtiquette, motCle, arguments.trim());

}

}

}

/\*\*

\* Ajoute une ligne de code (étiquette associée à une instruction)

\* au programme chargé.

\* **@param** texteEtiquette représentation texte de l'étiquette

\* **@param** motCle mot clé de l'instruction

\* **@param** arguments reste de la ligne saisie après le mot clé

\*/

**private** **void** editerProgramme(String texteEtiquette, String motCle,

String arguments) {

Class<?> aAjouter;

**try** {

aAjouter = *rechercheInstruction*(motCle);

Class<?> classeArg = String.**class**;

Class<?> classeContexte = Contexte.**class**;

Instruction inst = (Instruction)aAjouter

.getConstructor(classeArg, classeContexte)

.newInstance(arguments, contexteGlobal);

Etiquette etiquette = **new** Etiquette(texteEtiquette);

programme.ajouterLigne(etiquette, inst);

*feedback*(**false**);

} **catch** ( InvocationTargetException | IllegalAccessException

| InstantiationException | NoSuchMethodException

| InterpreteurException | ExecutionException lancee) {

System.***err***.println(***NOK\_FEEDBACK***

+ (lancee.getMessage() != **null**

? lancee.getMessage()

: lancee.getCause().getMessage()));

}

}

/\*\*

\* Recherche la commande et exécute cette commande si présente.

\* Affiche un feedback si la commande ne s'en occupe pas ou erreur.

\* **@param** motCle chaîne contenant le mot clé de la commande/instruction

\* **@param** arguments reste de la ligne saisie après le mot clé

\*/

**private** **void** executerCommande(String motCle, String arguments) {

Class<?> aExecuter;

/\* recherche de la class \*/

**try** {

aExecuter = *rechercheCommande*(motCle);

Class<?> classeArg = String.**class**;

Class<?> classeContexte = Contexte.**class**;

Commande cmd = (Commande)aExecuter

.getConstructor(classeArg, classeContexte)

.newInstance(arguments, contexteGlobal);

*feedback*(cmd.executer());

**if** (motCle.equals("lance")

|| (motCle.equals("affiche") && !arguments.isBlank())) {

System.***out***.println();

}

} **catch** ( InvocationTargetException | IllegalAccessException

| InstantiationException | NoSuchMethodException

| InterpreteurException | ExecutionException lancee) {

System.***err***.println(***NOK\_FEEDBACK***

+ (lancee.getMessage() != **null**

? lancee.getMessage()

: lancee.getCause().getMessage()));

}

}

/\*\*

\* Affiche feedback de bon déroulement

\* d'une commande si celle ci n'affiche rien.

\* **@param** afficheRien true si feedback déjà fait par la commande sinon false

\*/

**private** **static** **void** feedback(**boolean** afficheRien) {

**if** (!afficheRien) {

System.***out***.println(***OK\_FEEDBACK***);

}

}

/\*\*

\* Recherche la commande ou instruction correspondant au mot clé.

\* <ul><li>Les commandes doivent être

\* dans le package interpreteurlir.motscles</li>

\* <li>Les instructions doivent être

\* dans le package interpreteurlir.motscles.instructions</li>

\* <ul>

\* La classe correspondant doit avoir un nom qui se finit avec le mot clé

\* (première lettre en majuscule)

\* **@param** motCle mot clé de la commande/ instruction

\* **@return** Classe de cette commande.

\* **@throws** InterpreteurException si motCle est vide, null ou non reconnue

\*/

**private** **static** Class<?> rechercheCommande(String motCle) {

**final** String ERREUR\_VIDE = "ligne vide";

**final** String ERREUR\_INCONNU = "mot clé inconnu";

**final** String CLASS\_PATH\_CMD = "interpreteurlir.motscles.Commande";

**final** String CLASS\_PATH\_INST =

"interpreteurlir.motscles.instructions.Instruction";

**if** (motCle == **null** || motCle.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_VIDE);

} **else** **if** (!motCle.equals(motCle.toLowerCase())) {

**throw** **new** InterpreteurException(ERREUR\_INCONNU);

}

motCle = (Character.*toUpperCase*(motCle.charAt(0)))

+ (motCle.length() > 1 ? motCle.substring(1) : "");

Class<?> aChercher;

**try** {

aChercher = Class.*forName*(CLASS\_PATH\_CMD + motCle);

} **catch**(ClassNotFoundException | NoClassDefFoundError nonCmd) {

**try** {

aChercher = Class.*forName*(CLASS\_PATH\_INST + motCle);

} **catch**(ClassNotFoundException nonInst) {

**throw** **new** InterpreteurException(ERREUR\_INCONNU);

}

}

**return** aChercher;

}

/\*\*

\* Recherche l'instruction correspondant au mot clé.

\* <ul><li>Les instructions doivent être

\* dans le package interpreteurlir.motscles.instructions</li>

\* </ul>

\* La classe correspondant doit avoir un nom qui se finit avec le mot clé

\* (première lettre en majuscule)

\* **@param** motCle mot clé de l'instruction

\* **@return** Classe de cette instruction.

\* **@throws** InterpreteurException si motCle est vide, null ou non reconnue

\*/

**public** **static** Class<?> rechercheInstruction(String motCle) {

**final** String ERREUR\_VIDE = "ligne vide";

**final** String ERREUR\_INCONNU = "mot clé inconnu";

**final** String CLASS\_PATH\_INST =

"interpreteurlir.motscles.instructions.Instruction";

**if** (motCle == **null** || motCle.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_VIDE);

} **else** **if** (!motCle.equals(motCle.toLowerCase())) {

**throw** **new** InterpreteurException(ERREUR\_INCONNU);

}

motCle = (Character.*toUpperCase*(motCle.charAt(0)))

+ (motCle.length() > 1 ? motCle.substring(1) : "");

Class<?> aChercher;

**try** {

aChercher = Class.*forName*(CLASS\_PATH\_INST + motCle);

} **catch**(ClassNotFoundException | NoClassDefFoundError nonCmd) {

**throw** **new** InterpreteurException(ERREUR\_INCONNU);

}

**return** aChercher;

}

/\*\*

\* Lancement de l'interpréteur LIR.

\* Un analyseur est créé.

\* **@param** args non utilisé

\*/

**public** **static** **void** main(String[] args) {

**final** String MESSAGE\_LANCEMENT =

"Interpréteur Langage IUT de Rodez, bienvenue !\n"

+ "Entrez vos commandes et instructions après l’invite "

+ ***INVITE*** + "\n";

System.***out***.println(MESSAGE\_LANCEMENT);

**new** Analyseur().mainLoop();

}

}

### Tests

Les tests d’Analyseur ont été faits en intégration.

## ProgrammeDeTest.java

// Cette classe a été utilisée pour faciliter les démonstrations au moment où les // commandes sauve et charge n’existaient pas encore.

/\*\*

\* ProgrammeDeTest.java 18 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.tests;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.motscles.instructions.\*;

**import** interpreteurlir.programmes.Etiquette;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Permet de générer un programme pour les tests contenant plusieurs lignes.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** ProgrammeDeTest {

/\*\*

\* Génère en ajoutant au programme des lignes

\* pour les tests

\* **@param** aGenerer programme auquel est ajouté les lignes

\* **@param** contexte contexte pour les instructions

\*/

**public** **static** **void** genererProgramme(Programme aGenerer, Contexte contexte) {

Object[][] lignes = {

{**new** Etiquette(10),

**new** InstructionAffiche("\"Bienvenue dans le programme\"",

contexte)},

{**new** Etiquette(20), **new** InstructionAffiche("", contexte)},

{**new** Etiquette(30),

**new** InstructionVar("instant = 2021", contexte)},

{**new** Etiquette(40), **new** InstructionProcedure("500", contexte)},

{**new** Etiquette(50), **new** InstructionVar("$message = \"Vous êtes \"

+ $prenom", contexte)},

{**new** Etiquette(60),

**new** InstructionVar("$message = $message+ \" \"", contexte)},

{**new** Etiquette(65),

**new** InstructionVar("$message = $message+ $nom", contexte)},

{**new** Etiquette(70), **new** InstructionAffiche("$message", contexte)},

{**new** Etiquette(80), **new** InstructionAffiche("", contexte)},

{**new** Etiquette(90), **new** InstructionAffiche("\"age : \"",

contexte)},

{**new** Etiquette(100), **new** InstructionAffiche("age", contexte)},

{**new** Etiquette(110), **new** InstructionAffiche("\" ans\"",

contexte)},

{**new** Etiquette(120), **new** InstructionVaen("130", contexte)},

{**new** Etiquette(124), **new** InstructionAffiche("", contexte)},

{**new** Etiquette(125),

**new** InstructionAffiche("\"erreur vaen si affiché\"", contexte)},

{**new** Etiquette(150), **new** InstructionAffiche("", contexte)},

{**new** Etiquette(200), **new** InstructionAffiche(

"\"Merci d'avoir utilisé ce programme !\"", contexte)},

{**new** Etiquette(400), **new** InstructionStop("", contexte)},

// procedure saisie

{**new** Etiquette(500),

**new** InstructionAffiche("\"Saisissez votre nom : \"", contexte)},

{**new** Etiquette(510), **new** InstructionEntre("$nom", contexte)},

{**new** Etiquette(520),

**new** InstructionAffiche("\"Saisissez votre prénom : \"",

contexte)},

{**new** Etiquette(530), **new** InstructionEntre("$prenom", contexte)},

{**new** Etiquette(540),

**new** InstructionAffiche("\"Saisissez votre année de naissance "

+ " (entier) : \"", contexte)},

{**new** Etiquette(550), **new** InstructionEntre("naissance", contexte)},

{**new** Etiquette(560), **new** InstructionProcedure("1000", contexte)},

{**new** Etiquette(570), **new** InstructionRetour("", contexte)},

// procedure calcul age

{**new** Etiquette(1000),

**new** InstructionVar("age = instant - naissance", contexte)},

{**new** Etiquette(1010), **new** InstructionRetour("", contexte)},

};

**for** (**int** index = 0 ; index < lignes.length ; index++) {

aGenerer.ajouterLigne((Etiquette)lignes[index][0],

(Instruction)lignes[index][1]);

}

}

}

# Paquetage interpreteurlir.programmes

## Classe Etiquette

### Etiquette.java

/\*\*

\* Etiquette.java 13 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.programmes;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Etiquette permettant de définir les ordres d'exécution

\* des instructions d'un programme.

\* Le compteur ordinal pointe vers une étiquette.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** Etiquette **implements** Comparable<Etiquette> {

/\*\* Valeur minimale d'une étiquette \*/

**public** **static** **final** **int** ***VALEUR\_ETIQUETTE\_MIN*** = 1;

/\*\* Valeur maximale d'une étiquette \*/

**public** **static** **final** **int** ***VALEUR\_ETIQUETTE\_MAX*** = 99999;

/\*\* Message d'erreur pour une étiquette invalide \*/

**private** **static** **final** String ***MSG\_INVALIDE*** =

"valeur invalide pour une étiquette";

/\*\* valeur de cette étiquette \*/

**private** **int** valeur;

/\*\*

\* Initialise une étiquette qui défini l'ordre d'exécution

\* d'une instruction à partir d'un entier

\* **@param** valeur valeur de l'étiquette

\* **@throws** InterpreteurException si valeur n'appartient pas

\* à [ VALEUR\_ETIQUETTE\_MIN, VALEUR\_ETIQUETTE\_MAX ]

\*/

**public** Etiquette(**int** valeur) {

**super**();

**if** (valeur < ***VALEUR\_ETIQUETTE\_MIN*** || valeur > ***VALEUR\_ETIQUETTE\_MAX***) {

**throw** **new** InterpreteurException(***MSG\_INVALIDE***);

}

**this**.valeur = valeur;

}

/\*\*

\* Initialise une étiquette qui défini l'ordre d'exécution

\* d'une instruction à partir d'un entier dans une chaîne de texte

\* **@param** valeur chaîne contenant la valeur de l'étiquette

\* **@throws** InterpreteurException si l'entier de valeur n'appartient pas

\* à [ VALEUR\_ETIQUETTE\_MIN, VALEUR\_ETIQUETTE\_MAX ]

\*/

**public** Etiquette(String valeur) {

**super**();

**int** valeurEntiere;

**try** {

valeurEntiere = Integer.*valueOf*(valeur.trim());

} **catch** (NumberFormatException | NullPointerException lancee) {

**throw** **new** InterpreteurException(***MSG\_INVALIDE***);

}

**if** (valeurEntiere < ***VALEUR\_ETIQUETTE\_MIN***

|| valeurEntiere > ***VALEUR\_ETIQUETTE\_MAX***) {

**throw** **new** InterpreteurException(***MSG\_INVALIDE***);

}

**this**.valeur = valeurEntiere;

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

**return** Integer.*toString*(valeur);

}

/\*\*

\* **@return** valeur de valeur

\*/

**public** **int** getValeur() {

**return** valeur;

}

/\* non javadoc

\* @see java.lang.Comparable#compareTo(java.lang.Object)

\*/

@Override

**public** **int** compareTo(Etiquette aComparer) {

**return** valeur - aComparer.valeur;

}

}

### TestEtiquette.java

/\*\*

\* TestEtiquette.java 13 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.programmes.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Etiquette;

/\*\*

\* Tests unitaires de {@link Etiquette}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestEtiquette {

/\*\* Jeu de données valides pour les tests \*/

**private** Etiquette[] fixture = {

**new** Etiquette(Etiquette.***VALEUR\_ETIQUETTE\_MIN***),

**new** Etiquette(10),

**new** Etiquette(15),

**new** Etiquette(8),

**new** Etiquette(18),

**new** Etiquette(1500),

**new** Etiquette(1501),

**new** Etiquette(Etiquette.***VALEUR\_ETIQUETTE\_MAX***),

**new** Etiquette("" +Etiquette.***VALEUR\_ETIQUETTE\_MIN***),

**new** Etiquette(" 10"),

**new** Etiquette("15 "),

**new** Etiquette("8"),

**new** Etiquette("18"),

**new** Etiquette("1500 "),

**new** Etiquette(" 1501 "),

**new** Etiquette("" + Etiquette.***VALEUR\_ETIQUETTE\_MAX***),

};

/\*\*

\* Tests unitaires de {@link Etiquette#Etiquette(int)}

\*/

**public** **void** testEtiquetteInt() {

System.***out***.println("\tExécution du test de Etiquette#Etiquette(int)");

**final** **int**[] INVALIDES = {

Integer.***MIN\_VALUE***, -1, 0, 100000, Integer.***MAX\_VALUE***

};

**for** (**int** valeur : INVALIDES) {

**try** {

**new** Etiquette(valeur);

*echec*();

} **catch** (InterpreteurException lancee) {

}

}

**try** {

**new** Etiquette(Etiquette.***VALEUR\_ETIQUETTE\_MIN***);

**new** Etiquette(10);

**new** Etiquette(15);

**new** Etiquette(8);

**new** Etiquette(18);

**new** Etiquette(1500);

**new** Etiquette(1501);

**new** Etiquette(Etiquette.***VALEUR\_ETIQUETTE\_MAX***);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link Etiquette#Etiquette(String)}

\*/

**public** **void** testEtiquetteString() {

System.***out***.println("\tExécution du test de "

+ "Etiquette#Etiquette(String)");

**final** String[] INVALIDES = {

**null**, "", "cinq",

"" + Integer.***MIN\_VALUE***, "-1", " 0",

"100000 ", "" + Integer.***MAX\_VALUE***

};

**for** (String valeur : INVALIDES) {

**try** {

**new** Etiquette(valeur);

*echec*();

} **catch** (InterpreteurException lancee) {

}

}

**try** {

**new** Etiquette("" +Etiquette.***VALEUR\_ETIQUETTE\_MIN***);

**new** Etiquette(" 10");

**new** Etiquette("15 ");

**new** Etiquette("8");

**new** Etiquette("18");

**new** Etiquette("1500 ");

**new** Etiquette(" 1501 ");

**new** Etiquette("" + Etiquette.***VALEUR\_ETIQUETTE\_MAX***);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link Etiquette#toString()}

\*/

**public** **void** testToString() {

System.***out***.println("\tExécution du test de Etiquette#toString()");

**final** String[] TEXTE\_ATTENDU = {

"1",

"10",

"15",

"8",

"18",

"1500",

"1501",

"99999",

"1",

"10",

"15",

"8",

"18",

"1500",

"1501",

"99999",

};

**for** (**int** numTest = 0 ; numTest < TEXTE\_ATTENDU.length ; numTest++) {

*assertEquivalence*(fixture[numTest].toString(),

TEXTE\_ATTENDU[numTest]);

}

}

/\*\*

\* Tests unitaires de {@link Etiquette#getValeur()}

\*/

**public** **void** testGetValeur() {

System.***out***.println("\tExécution du test de Etiquette#getValeur()");

**final** **int**[] VALEUR\_ATTENDUE = {

1,

10,

15,

8,

18,

1500,

1501,

99999,

1,

10,

15,

8,

18,

1500,

1501,

99999,

};

**for** (**int** numTest = 0 ; numTest < VALEUR\_ATTENDUE.length ; numTest++) {

*assertEquivalence*(fixture[numTest].getValeur(),

VALEUR\_ATTENDUE[numTest]);

}

}

/\*\*

\* Test unitaires de {@link Etiquette#compareTo(Etiquette)}

\*/

**public** **void** testCompareTo() {

**final** Etiquette[] CROISSANTS = {

**new** Etiquette(Etiquette.***VALEUR\_ETIQUETTE\_MIN***),

**new** Etiquette(8),

**new** Etiquette(10),

**new** Etiquette(15),

**new** Etiquette(18),

**new** Etiquette(1500),

**new** Etiquette(1501),

**new** Etiquette(Etiquette.***VALEUR\_ETIQUETTE\_MAX***),

};

System.***out***.println("\tExécution du test de "

+ "Etiquette#compareTo(Etiquette)");

/\*\* Test croissant \*/

**for** (**int** reference = 0 ; reference < CROISSANTS.length ; reference++) {

**for** (**int** numtest = reference + 1 ;

numtest < CROISSANTS.length ;

numtest++) {

*assertTrue*(CROISSANTS[reference].compareTo(

CROISSANTS[numtest]) < 0);

}

}

/\*\* Test décroissant \*/

**for** (**int** reference = CROISSANTS.length - 1 ;

reference > 0 ;

reference--) {

**for** (**int** numtest = reference - 1 ;

numtest >= 0 ;

numtest--) {

*assertTrue*(CROISSANTS[reference].compareTo(

CROISSANTS[numtest]) > 0);

}

}

Etiquette referenceEgalite = **new** Etiquette(666);

*assertTrue*(referenceEgalite.compareTo(referenceEgalite) == 0);

*assertTrue*(referenceEgalite.compareTo(**new** Etiquette("666")) == 0);

}

}

## Classe Programme

### Programme.java

/\*\*

\* Programme.java 14 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.programmes;

**import** java.util.EmptyStackException;

**import** java.util.Map;

**import** java.util.Stack;

**import** java.util.TreeMap;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.instructions.Instruction;

**import** **static** interpreteurlir.programmes.Etiquette.***VALEUR\_ETIQUETTE\_MAX***;

**import** **static** interpreteurlir.programmes.Etiquette.***VALEUR\_ETIQUETTE\_MIN***;

/\*\*

\* Enregistrement des lignes de code (instruction associée à une

\* étiquette) et gestion de l'exécution des lignes de code dans

\* l'ordre des étiquettes

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** Programme {

**private** **static** **final** String ***ERREUR\_INTERVALLE*** = "erreur dans l'intervalle "

+ "d'étiquettes";

/\*\* Pile LIFO pour la gestion des étiquettes \*/

**private** Stack<Etiquette> compteurOrdinnal;

/\*\* Détermine la poursuite d'exécution de ce programme \*/

**private** **boolean** enExecution;

**private** TreeMap<Etiquette, Instruction> lignesCode;

/\*\*

\* Initialisation de ce programme sans lignes de code

\*/

**public** Programme() {

**super**();

lignesCode = **new** TreeMap<Etiquette, Instruction>();

enExecution = **false**;

compteurOrdinnal = **new** Stack<Etiquette>();

}

/\*\*

\* Remise à zero de ce programme

\* <p>

\* Vide ce programme de toute lignes de code (instruction associée

\* à une étiquette)

\*/

**public** **void** raz() {

lignesCode.clear();

compteurOrdinnal.clear();

}

/\*\*

\* Ajoute une ligne de code (instruction associée à une étiquette)

\* à ce programme

\* <p>

\* Une ligne ajoutée à une même étiquette écrase le contenu associé

\* à cette dernière

\*

\* **@param** etiquette pour donner l'ordre d'exécution de ce programme

\* **@param** instruction associée une étiquette à insérer dans ce

\* programme

\* **@throws** NullPointerException si l'étiquette ou l'instruction

\* est nulle

\*/

**public** **void** ajouterLigne(Etiquette etiquette, Instruction instruction) {

**if** (etiquette == **null** || instruction == **null**) {

**throw** **new** NullPointerException();

}

lignesCode.put(etiquette, instruction);

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

Object[] tableauEtiquette = lignesCode.keySet().toArray();

Object[] tableauInstruction = lignesCode.values().toArray();

StringBuilder aAfficher = **new** StringBuilder("");

**for** (**int** i = 0; i < tableauEtiquette.length; i++) {

aAfficher.append(tableauEtiquette[i] + " "

+ tableauInstruction[i] + '\n');

}

**return** aAfficher.toString();

}

/\*\*

\* Liste des lignes de code comprise entre les étiquettes de

\* début et les étiquettes de fin

\* <p>

\* S'il n'y aucune ligne de code dans l'intervalle, alors la

\* chaîne renvoyée contient un message qui l'indique.

\*

\* **@param** debut étiquette à partir de laquelle le programme

\* est listé

\* **@param** fin dernière étiquette associée à son contenu à lister

\* **@return** la représentation texte des lignes de code comprise

\* entre les étiquettes de début et les étiquettes de fin

\* **@throws** InterpreteurException si fin est strictement inférieur

\* à debut

\*/

**public** String listeBornee(Etiquette debut, Etiquette fin) {

**if** (fin.compareTo(debut) < 0) {

**throw** **new** InterpreteurException(***ERREUR\_INTERVALLE***);

}

StringBuilder aAfficher = **new** StringBuilder("");

Etiquette cleCourante = debut;

Instruction instCourante = **null**;

Map.Entry<Etiquette, Instruction> entreeCourante;

**boolean** lignesRestantes;

**do** {

entreeCourante = lignesCode.ceilingEntry(cleCourante);

lignesRestantes = entreeCourante != **null**;

**if** (lignesRestantes) {

cleCourante = entreeCourante.getKey();

instCourante = entreeCourante.getValue();

lignesRestantes = cleCourante.compareTo(fin) <= 0;

}

**if** (lignesRestantes) {

aAfficher.append(cleCourante + " " + instCourante + '\n');

cleCourante = **new** Etiquette(cleCourante.getValeur() + 1);

}

} **while** (lignesRestantes);

**return** aAfficher.toString().equals("") ? "aucune ligne à afficher\n"

: aAfficher.toString();

}

/\*\*

\* Efface les lignes de code comprises entre les étiquettes debut

\* et fin

\*

\* **@param** debut étiquette à partir de laquelle le programme

\* est à effacer

\* **@param** fin dernière étiquette associée à son contenu à effacer

\* **@throws** InterpreteurException si fin est strictement inférieur

\* à debut

\*/

**public** **void** effacer(Etiquette debut, Etiquette fin) {

**if** (fin.compareTo(debut) < 0) {

**throw** **new** InterpreteurException(***ERREUR\_INTERVALLE***);

}

Etiquette cleCourante = debut;

**boolean** lignesRestantes;

**do** {

cleCourante = lignesCode.ceilingKey(cleCourante);

lignesRestantes = cleCourante != **null**

&& cleCourante.compareTo(fin) <= 0;

**if** (lignesRestantes) {

lignesCode.remove(cleCourante);

cleCourante = **new** Etiquette(cleCourante.getValeur() + 1);

}

} **while** (lignesRestantes);

}

/\*\*

\* Arrête l'exécution du programme

\*/

**public** **void** stop() {

enExecution = **false**;

}

/\*\*

\* Boucle d'exécution du programme

\*/

**private** **void** execution() {

Etiquette etiquetteCourante;

**while** (enExecution) {

etiquetteCourante = compteurOrdinnal.pop();

etiquetteCourante = lignesCode.ceilingKey(etiquetteCourante);

enExecution = etiquetteCourante != **null**

&& etiquetteCourante.getValeur() + 1 <= ***VALEUR\_ETIQUETTE\_MAX***;

**if** (enExecution) {

compteurOrdinnal.push(

**new** Etiquette(etiquetteCourante.getValeur() + 1));

lignesCode.get(etiquetteCourante).executer();

}

}

}

/\*\*

\* Lance l'exécution du programme à partir de l'étiquette

\* passée en argument

\*

\* **@param** etiquetteDepart étiquette à partir de laquelle

\* l'exécution du programme est lancée

\*/

**public** **void** lancer(Etiquette etiquetteDepart) {

compteurOrdinnal.clear();

compteurOrdinnal.push(etiquetteDepart);

enExecution = **true**;

execution();

}

/\*\*

\* Lance l'exécution du programme à partir de l'étiquette

\* la plus petite

\*/

**public** **void** lancer() {

lancer(**new** Etiquette(***VALEUR\_ETIQUETTE\_MIN***));

}

/\*\*

\* Change le compteur ordinal avec l'étiquette argument

\* **@param** destination étiquette où continuer l'exécution

\* **@throws** ExecutionException si aucune instruction dans le programme

\* n'a comme étiquette destination

\*/

**public** **void** vaen(Etiquette destination) {

**final** String ERR\_ETIQUETTE = "vaen impossible car l'étiquette "

+ destination

+ " ne correspond à aucune instruction";

**if** (!lignesCode.containsKey(destination)) {

**throw** **new** ExecutionException(ERR\_ETIQUETTE);

}

// else

**if** (!compteurOrdinnal.isEmpty()) {

compteurOrdinnal.pop();

}

compteurOrdinnal.push(destination);

enExecution = **true**;

execution();

}

/\*\*

\* Appel une procédure en empilant l'étiquette de départ dans

\* le compteurOrdinal

\* **@param** depart étiquette du début de la procédure

\*/

**public** **void** appelProcedure(Etiquette depart) {

compteurOrdinnal.push(depart);

}

/\*\*

\* Retour d'une procédure en dépilant l'étiquette de départ dans

\* le compteurOrdinal

\* **@throws** ExecutionException lorsque retourProcedure vide le

\* compteurOrdinnal

\*/

**public** **void** retourProcedure() {

**final** String ERREUR\_RETOUR = "retour nécessite un appel de "

+ "procédure au préalable";

**try** {

compteurOrdinnal.pop();

} **catch** (EmptyStackException lancee) {

// empty body

}

**if** (compteurOrdinnal.isEmpty()) {

**throw** **new** ExecutionException(ERREUR\_RETOUR);

}

}

}

### TestProgramme.java

/\*\*

\* TestProgramme.java 14 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.programmes.tests;

**import** interpreteurlir.programmes.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.instructions.\*;

**import** interpreteurlir.motscles.instructions.tests.TestInstructionStop;

**import** interpreteurlir.motscles.instructions.tests.TestInstructionVaen;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Tests unitaires de {@link Programme}

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestProgramme {

**private** Programme programmeTest = **new** Programme();

**private** Contexte contexteTest = **new** Contexte();

**private** **final** Etiquette[] JEU\_ETIQUETTES = {

**new** Etiquette(1),

**new** Etiquette(10),

**new** Etiquette(13),

**new** Etiquette(5),

**new** Etiquette(31),

**new** Etiquette(40),

**new** Etiquette(5),

**new** Etiquette(89)

};

**private** **final** Instruction[] JEU\_INSTRUCTIONS = {

**new** InstructionVar("$toto = \"toto\"", contexteTest),

**new** InstructionVar("tata = 0 + 0", contexteTest),

**new** InstructionVar("$titi = \"titi\"", contexteTest),

**new** InstructionEntre("agreu", contexteTest),

**new** InstructionEntre("tutu", contexteTest),

**new** InstructionVar("entier = 93", contexteTest),

**new** InstructionVar("$agreuagreu = \"agreu\"", contexteTest),

**new** InstructionVar("$youpi = \"youpi lapin\"", contexteTest)

};

**private** **static** **final** Etiquette[][] ***BORNES*** = {

{ **new** Etiquette(6), **new** Etiquette(6) },

{ **new** Etiquette(1), **new** Etiquette(90) },

{ **new** Etiquette(31), **new** Etiquette(39) },

{ **new** Etiquette(9), **new** Etiquette(41) }

};

**private** **static** **final** **int** ***DEBUT*** = 0;

**private** **static** **final** **int** ***FIN*** = 1;

**private** **void** ajoutLigne() {

**for** (**int** i = 0; i < JEU\_ETIQUETTES.length; i++) {

programmeTest.ajouterLigne(JEU\_ETIQUETTES[i], JEU\_INSTRUCTIONS[i]);

}

}

/\*\*

\* Test unitaire de {@link Programme#Programme()}

\*/

**public** **void** testProgramme() {

System.***out***.println("\tExécution du test de Programme() : ");

**try** {

**new** Programme();

} **catch** (Exception lancee) {

*echec*();

}

}

/\*\*

\* Test unitaire de {@link Programme#ajouterLigne(Etiquette, Instruction)}

\*/

**public** **void** testAjouterLigne() {

**final** Etiquette[] ETIQUETTES\_INVALIDES = {

**null**,

**new** Etiquette(1),

**null**,

};

**final** Instruction[] INSTRUCTIONS\_INVALIDES = {

**new** InstructionEntre("janis", contexteTest),

**null**,

**null**

};

System.***out***.println("\tExécution du test de ajouterLigne() : ");

**for** (**int** i = 0; i < ETIQUETTES\_INVALIDES.length; i++) {

**try** {

programmeTest.ajouterLigne(ETIQUETTES\_INVALIDES[i],

INSTRUCTIONS\_INVALIDES[i]);

*echec*();

} **catch** (NullPointerException lancee) {

// Test OK

}

}

**for** (**int** i = 0; i < JEU\_ETIQUETTES.length; i++) {

**try** {

programmeTest.ajouterLigne(JEU\_ETIQUETTES[i],

JEU\_INSTRUCTIONS[i]);

} **catch** (NullPointerException lancee) {

*echec*();

}

}

}

/\*\*

\* Test unitaire de {@link Programme#toString()}

\*/

**public** **void** testToString() {

**final** String TEXTE\_ATTENDU = "1 var $toto = \"toto\"\n"

+ "5 var $agreuagreu = \"agreu\"\n"

+ "10 var tata = 0 + 0\n"

+ "13 var $titi = \"titi\"\n"

+ "31 entre tutu\n"

+ "40 var entier = 93\n"

+ "89 var $youpi = \"youpi lapin\"\n";

ajoutLigne();

System.***out***.println("\tExécution du test de toString() : ");

*assertEquivalence*(TEXTE\_ATTENDU, programmeTest.toString());

}

/\*\*

\* Test unitaire de {@link Programme#raz()}

\*/

**public** **void** testRaz() {

System.***out***.println("\tExécution du test de raz() : ");

programmeTest.raz();

*assertEquivalence*(programmeTest.toString(), "");

ajoutLigne();

programmeTest.raz();

*assertEquivalence*(programmeTest.toString(), "");

}

/\*\*

\* Test unitaire de {@link Programme#listeBornee(Etiquette, Etiquette)}

\*/

**public** **void** testListeBornee() {

**final** String[] TEXTES\_ATTENDUS = {

"aucune ligne à afficher\n",

"1 var $toto = \"toto\"\n"

+ "5 var $agreuagreu = \"agreu\"\n"

+ "10 var tata = 0 + 0\n"

+ "13 var $titi = \"titi\"\n"

+ "31 entre tutu\n"

+ "40 var entier = 93\n"

+ "89 var $youpi = \"youpi lapin\"\n",

"31 entre tutu\n",

"10 var tata = 0 + 0\n"

+ "13 var $titi = \"titi\"\n"

+ "31 entre tutu\n"

+ "40 var entier = 93\n",

};

**final** Etiquette[][] BORNES\_INVALIDES = {

{ **new** Etiquette(8), **new** Etiquette(6) },

{ **new** Etiquette(10000), **new** Etiquette(90) }

};

ajoutLigne();

System.***out***.println("\tExécution du test de listeBornee() : ");

**for** (**int** i = 0; i < TEXTES\_ATTENDUS.length; i++) {

*assertEquivalence*(TEXTES\_ATTENDUS[i],

programmeTest.listeBornee(***BORNES***[i][***DEBUT***],

***BORNES***[i][***FIN***]));

}

**for** (**int** i = 0; i < BORNES\_INVALIDES.length; i++) {

**try** {

programmeTest.listeBornee(BORNES\_INVALIDES[i][***DEBUT***],

BORNES\_INVALIDES[i][***FIN***]);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

}

/\*\*

\* Test unitaire de {@link Programme#effacer(Etiquette, Etiquette)}

\*/

**public** **void** testEffacer() {

**final** String[] TEXTES\_ATTENDUS = {

"1 var $toto = \"toto\"\n"

+ "5 var $agreuagreu = \"agreu\"\n"

+ "10 var tata = 0 + 0\n"

+ "13 var $titi = \"titi\"\n"

+ "31 entre tutu\n"

+ "40 var entier = 93\n"

+ "89 var $youpi = \"youpi lapin\"\n",

"",

"1 var $toto = \"toto\"\n"

+ "5 var $agreuagreu = \"agreu\"\n"

+ "10 var tata = 0 + 0\n"

+ "13 var $titi = \"titi\"\n"

+ "40 var entier = 93\n"

+ "89 var $youpi = \"youpi lapin\"\n",

"1 var $toto = \"toto\"\n"

+ "5 var $agreuagreu = \"agreu\"\n"

+ "89 var $youpi = \"youpi lapin\"\n",

};

**final** Etiquette[][] BORNES\_INVALIDES = {

{ **new** Etiquette(8), **new** Etiquette(6) },

{ **new** Etiquette(10000), **new** Etiquette(90) }

};

System.***out***.println("\tExécution du test de effacer() : ");

**for** (**int** i = 0; i < ***BORNES***.length ; i++) {

ajoutLigne();

programmeTest.effacer(***BORNES***[i][***DEBUT***], ***BORNES***[i][***FIN***]);

*assertEquivalence*(programmeTest.toString(), TEXTES\_ATTENDUS[i]);

}

**for** (**int** i = 0; i < BORNES\_INVALIDES.length; i++) {

**try** {

programmeTest.effacer(BORNES\_INVALIDES[i][***DEBUT***],

BORNES\_INVALIDES[i][***FIN***]);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

}

/\*\*

\* Test unitaire de {@link Programme#stop()}

\* **@see** TestInstructionStop#testExecuter()

\*/

**public** **void** testStop() {

System.***out***.println("\tExécution du test de Programme#stop() "

+ ": voir TestInstructionStop#testExecuter()");

}

/\*\*

\* Test unitaire de {@link Programme#lancer(Etiquette)}

\*/

**public** **void** testLancerEtiquette() {

**final** Etiquette[] ETIQUETTES\_DEPART = {

**new** Etiquette(1),

**new** Etiquette(10),

**new** Etiquette(25),

**new** Etiquette(90)

};

Expression.*referencerContexte*(contexteTest);

ajoutLigne();

System.***out***.println("\tExécution du test de lancer(Etiquette) "

+ "TEST INTERACTIF : ");

**for** (**int** i = 0; i < ETIQUETTES\_DEPART.length; i++) {

System.***out***.println(programmeTest.listeBornee(ETIQUETTES\_DEPART[i],

**new** Etiquette(9999)));

contexteTest.raz();

programmeTest.lancer(ETIQUETTES\_DEPART[i]);

System.***out***.println(contexteTest.toString());

}

}

/\*\*

\* Test unitaire de {@link Programme#lancer()}

\*/

**public** **void** testLancer() {

Expression.*referencerContexte*(contexteTest);

contexteTest.raz();

ajoutLigne();

System.***out***.println("\tExécution du test de lancer() "

+ "TEST INTERACTIF : ");

System.***out***.println(programmeTest.toString());

programmeTest.lancer();

System.***out***.println(contexteTest.toString());

}

/\*\*

\* Test unitaire de {@link Programme#appelProcedure(Etiquette)}

\*/

**public** **void** testAppelProcedure() {

System.***out***.println("\tExécution du test de appelProcedure(Etiquette) "

+ ": ");

/\* Cas Valides \*/

**try** {

/\* Simulation du lancement du programme \*/

programmeTest.appelProcedure(**new** Etiquette(1));

/\* Lancement de 2 procédures \*/

programmeTest.appelProcedure(**new** Etiquette(100));

programmeTest.appelProcedure(**new** Etiquette(50));

} **catch** (InterpreteurException lancee) {

*echec*();

}

/\* Cas Invalides \*/

**try** {

/\* Simulation du lancement du programme \*/

programmeTest.appelProcedure(**new** Etiquette(1));

/\* Lancement de 2 procédures \*/

programmeTest.appelProcedure(**new** Etiquette(-30));

programmeTest.appelProcedure(**new** Etiquette(10000000));

*echec*();

} **catch** (InterpreteurException lancee) {

/\* Test OK \*/

}

}

/\*\*

\* Test unitaire de {@link Programme#retourProcedure()}

\*/

**public** **void** testRetourProcedure() {

System.***out***.println("\tExécution du test de retourProcedure() : ");

// Simulation du lancement du programme

programmeTest.appelProcedure(**new** Etiquette(1));

// Lancement de 2 procédures

programmeTest.appelProcedure(**new** Etiquette(100));

programmeTest.appelProcedure(**new** Etiquette(50));

**try** {

programmeTest.retourProcedure();

programmeTest.retourProcedure();

} **catch** (ExecutionException lancee) {

*echec*();

}

**try** {

programmeTest.retourProcedure();

*echec*();

} **catch** (ExecutionException lancee) {

// Test OK

}

}

/\*\*

\* Test unitaire de {@link Programme#vaen(Etiquette)}

\* **@see** TestInstructionVaen#testExecuter()

\*/

**public** **void** testVaen() {

System.***out***.println("\tExécution du test de vaen(Etiquette) "

+ ": voir TestInstructionVaen#testExecuter()");

}

}

# Paquetage interpreteurlir.motscles

## Classe Commande

### Commande.java

/\*\*

\* Commande.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Une commande (générale) n'a aucun comportement.

\* Voir les sous-classes pour les comportements.

\* Une commande contient tous les liens nécessaires à son exécution.

\* Une commande peut être exécutée.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **abstract** **class** Commande {

/\*\* référence du programme global \*/

**protected** **static** Programme *programmeGlobal*;

/\*\* référence du contexte possiblement manié directement par la commande \*/

**protected** Contexte contexte;

/\*\*

\* Initialise une commande avec les liens dont elle a besoin pour

\* s'exécuter à partir des arguments.

\* Cependant la commande ne s'exécute pas à la construction.

\* La commande a accès au contexte passé en argument.

\* **@param** arguments chaîne de texte représentant les arguments

\* **@param** contexte référence du contexte global

\* **@throws** InterpreteurException est propagée si Commande la reçoit

\* **@throws** NullPointerException si contexte ou arguments est null

\*/

**public** Commande(String arguments, Contexte contexte) {

**super**();

**if** (arguments == **null** || contexte == **null**) {

**throw** **new** NullPointerException();

}

// arguments non utilisés dans Commande générale

**this**.contexte = contexte;

}

/\*\*

\* Commande d'exécution de la commande.

\* **@return** true si la commande affiche un feedback directement sur la sortie

\* standard, sinon false

\*/

**public** **abstract** **boolean** executer();

/\*\*

\* Référence le programme pour accéder et modifier le programme chargé.

\* Le référencement vaut pour toutes les commandes/instructions

\* et est possible une unique fois.

\* **@param** aReferencer référence du programme global

\* **@return** <ul><li>true si le programme a pu être référencé</li>

\* <li>true si aReferencer == programme déjà référencer</li>

\* <li>false si aReferencer est null</li>

\* <li>false si un programme est déjà référencer</li>

\* </ul>

\*/

**public** **static** **boolean** referencerProgramme(Programme aReferencer) {

**if** (aReferencer != **null**

&& ( *programmeGlobal* == **null**

|| aReferencer == *programmeGlobal*)) {

*programmeGlobal* = aReferencer;

**return** **true**;

}

**return** **false**;

}

}

### TestCommande.java

// Tests faits avant le passage en abstract de la classe

/\*\*

\* TestCommande.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Tests unitaires de {@link interpreteurlir.motscles.Commande}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommande {

/\*\* Jeux d'essais de Commande valides pour les tests \*/

**private** Commande[] fixture = {

**new** Commande("", **new** Contexte()),

**new** Commande("coucou", **new** Contexte()),

**new** Commande("$chaine = \"toto\" + $tata", **new** Contexte())

};

/\*\*

\* Tests unitaires de {@link Commande#referencerProgramme(Programme)}

\*/

**public** **void** testReferencerProgramme() {

Programme reference = **new** Programme();

Programme[] programmes = {

**null**, reference, reference, **new** Programme()

};

**boolean**[] resultatAttendu = { **false**, **true**, **true**, **false** };

System.***out***.println("\tExécution du test de "

+ "Commande#referencerProgramme(Programme)");

**for** (**int** numTest = 0 ; numTest < programmes.length ; numTest++) {

*assertTrue*( Commande.*referencerProgramme*(programmes[numTest])

== resultatAttendu[numTest]);

}

}

/\*\*

\* Tests unitaires de {@link Commande#Commande(String, Contexte)}

\*/

**public** **void** testCommandeStringContexte() {

System.***out***.println(

"\tExécution du test de Commande#Commande(String, Contexte)");

/\* Tests Commande invalide \*/

String[] arguments = { **null**, **null**, "" };

Contexte[] contexte = { **null**, **new** Contexte(), **null**};

**for** (**int** numTest = 0 ; numTest < arguments.length ; numTest++) {

**try** {

**new** Commande(arguments[numTest], contexte[numTest]);

*echec*();

} **catch** (NullPointerException lancee) {

}

}

**try** {

**new** Commande("", **new** Contexte());

**new** Commande("coucou", **new** Contexte());

**new** Commande("$chaine = \"toto\" + $tata", **new** Contexte());

} **catch** (NullPointerException e) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link Commande#executer()}

\*/

**public** **void** testExecuter() {

System.***out***.println("\tExécution du test de Commande#executer()");

**for** (Commande aTester : fixture) {

*assertFalse*(aTester.executer());

}

}

}

## Classe CommandeCharge

### CommandeCharge.java

/\*\*

\* CommandeCharge.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.instructions.Instruction;

**import** interpreteurlir.programmes.Etiquette;

**import** interpreteurlir.Analyseur;

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.lang.reflect.InvocationTargetException;

/\*\*

\* Charge les lignes de programme dans le fichier texte indiqué en argument

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeCharge **extends** Commande{

/\*\* Chemin du fichier dans lequel sera le programme chargé \*/

**private** String cheminFichier;

/\*\*

\* Initialise la commande Charge a partir de ses argument et

\* de son contexte passé en paramètre.

\* **@param** arguments Chemin du fichier dans lequel sera le programme chargé

\* **@param** contexte Contexte du programme

\* **@throws** InterpreteurException Si l'argument est null, vide,

\* contient uniquement des espaces ou

\* ne se termine pas par ".lir".

\*/

**public** CommandeCharge(String arguments, Contexte contexte ) {

**super**(arguments, contexte);

**final** String extension = ".lir";

**if** (arguments == **null**

|| arguments.isEmpty()

|| arguments.isBlank()

|| !arguments.trim().endsWith(extension)) {

**throw** **new** InterpreteurException("\t" + arguments

+ " n'est pas un chemin valide");

}

**this**.cheminFichier = arguments.trim();

}

/\*\*

\* Charge le programme contenu dans le fichier de this

\* **@return** false car elle n'affiche aucun feedback directement

\* **@throws** InterpreteurException Si le fichier a charger n'as pas été trouvé

\*/

**public** **boolean** executer() {

*programmeGlobal*.raz();

/\* Chemin du fichier \*/

String nomFichier = **new** File(cheminFichier).getAbsolutePath();

/\* Fichier logique en entrée \*/

BufferedReader entree;

entree = **null**;

**try** {

entree = **new** BufferedReader(

**new** InputStreamReader(

**new** FileInputStream(nomFichier)));

analyserFichier(entree);

entree.close();

} **catch** (IOException e) {

**throw** **new** InterpreteurException(nomFichier + " est introuvable");

}

**return** **false**;

}

/\*\*

\* Analyse chaque ligne de l'entrée et les ajoute dans programme global

\* **@param** entree Tampon du fichier à lire

\* **@throws** IOException Problème de lecture du fichier

\*/

**private** **void** analyserFichier(BufferedReader entree) **throws** IOException {

/\* Index de la ligne découpée \*/

**final** **int** ETIQUETTE = 0;

**final** **int** MOT\_CLE = 1;

**final** **int** ARGUMENT = 2;

String ligneLue;

**int** numLigne = 0;

**do** {

ligneLue = entree.readLine();

**if** (ligneLue != **null** && !ligneLue.isBlank()) {

numLigne++;

String[] decoupage = *splitter*(ligneLue);

Class<?> aAjouter;

**try** {

aAjouter = Analyseur

.*rechercheInstruction*(decoupage[MOT\_CLE]);

Class<?> classeArg = String.**class**;

Class<?> classeContexte = Contexte.**class**;

Instruction inst = (Instruction)aAjouter

.getConstructor(classeArg, classeContexte)

.newInstance(decoupage[ARGUMENT], contexte);

Etiquette etiquette = **new** Etiquette(decoupage[ETIQUETTE]);

*programmeGlobal*.ajouterLigne(etiquette, inst);

} **catch** (InvocationTargetException| IllegalAccessException

|InstantiationException | NoSuchMethodException

|InterpreteurException | ExecutionException lancee) {

*programmeGlobal*.raz();

**throw** **new** InterpreteurException(ligneLue + " => ligne "

+ numLigne);

}

}

} **while** (ligneLue != **null**);

}

/\*\*

\* Sépare la ligne en etiquette/mot clé/argument

\* **@param** ligneLue

\* **@return** Tableau comportant en :

\* <ul><li>indice 1 : l'étiquette</li>

\* <li>indice 2 : mot clé</li>

\* <li>indice 3 : argument</li></ul>

\* **@throws** InterpreteurException Si la ligne ne contient pas les 2 éléments:

\* <ul><li>Etiquette</li>

\* <li>Mot clé</li></ul>

\*/

**private** **static** String[] splitter(String ligneLue) {

/\* Sépare l'étiquette, la commande et l'argument \*/

String[] ligne = ligneLue.split(" ", 3);

**if** (ligne.length < 2) {

*programmeGlobal*.raz();

**throw** **new** InterpreteurException(ligneLue + " n'est pas "

+ "une ligne valide");

}

String[] decoupage = **new** String[3];

/\* Ajouter la ligne dans le contexte \*/

decoupage[0] = ligne[0];

decoupage[1] = ligne[1];

decoupage[2] = ligne.length >= 3 ? ligne[2] : "";

**return** decoupage;

}

}

### TestCommandeCharge.java

/\*\*

\* TestCommandeCharge.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.CommandeCharge;

**import** interpreteurlir.motscles.CommandeListe;

**import** interpreteurlir.programmes.Programme;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Tests unitaires de {@link interpreteurlir.motscles.CommandeCharge}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeCharge {

/\*\* Contexte pour tests \*/

**private** **final** **static** Contexte ***CONTEXTE\_TESTS*** = **new** Contexte();

/\*\* Programme global pour tests \*/

**private** **static** Programme *progGlobal* = **new** Programme();

/\*\* jeu de test valide \*/

**public** **static** **final** CommandeCharge[] ***FIXTURE*** = {

**new** CommandeCharge("F:\\Programmation\\WorkspaceInterpreteurLIR"

+ "\\outilTest\\dossierFichier\\"

+ "lefichier1.lir", ***CONTEXTE\_TESTS***),

**new** CommandeCharge("dossierFichier\\lefichier2.lir",

***CONTEXTE\_TESTS***),

**new** CommandeCharge("dossierFichier\\lefichier3.lir",

***CONTEXTE\_TESTS***),

**new** CommandeCharge("dossierFichier\\test\\lefichier4.lir",

***CONTEXTE\_TESTS***),

**new** CommandeCharge("dossierFichier\\test\\test2\\lefichier5.lir",

***CONTEXTE\_TESTS***),

**new** CommandeCharge(" dossierFichier\\lefichier6.lir ",

***CONTEXTE\_TESTS***),

**new** CommandeCharge("dossierFichier\\test\\test2\\..\\lefichier7.lir",

***CONTEXTE\_TESTS***)

};

/\*\*

\* Tests unitaires de

\* {@link CommandeCharge#CommandeCharge(String, Contexte)}

\*/

**public** **static** **void** testCommandeCharge() {

**final** String[] INVALIDE = {

**null**,

" ",

"",

"lefichier",

"dossier\\ lefichier",

"dossier \\lefichier",

};

**for** (**int** i = 0; i < INVALIDE.length ; i++) {

**try** {

**new** CommandeCharge(INVALIDE[i], ***CONTEXTE\_TESTS***);

*echec*();

} **catch** (InterpreteurException | NullPointerException lancee) {

// Test OK

}

}

}

/\*\*

\* Tests unitaires de

\* {@link CommandeCharge#executer()}

\*/

**public** **static** **void** testExecuter() {

Commande.*referencerProgramme*(*progGlobal*);

**final** CommandeCharge[] ERREUR = {

**new** CommandeCharge("dossierFichier\\erreur1.lir",

***CONTEXTE\_TESTS***),

**new** CommandeCharge("dossierFichier\\erreur2.lir",

***CONTEXTE\_TESTS***)

};

**final** **int** NB\_TESTS = ***FIXTURE***.length;

System.***out***.println("\nTest valides de CommandeCharge#executer():");

**for** (**int** i = 0; i < NB\_TESTS ; i++) {

System.***out***.println("Test " + (i+1) + '\\' + NB\_TESTS + ":");

***FIXTURE***[i].executer();

**new** CommandeListe("", ***CONTEXTE\_TESTS***).executer();

}

System.***out***.println("\nTest invalides de CommandeCharge#executer():");

**for**(**int** i = 0; i < ERREUR.length ; i++) {

**try** {

ERREUR[i].executer();

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

**new** CommandeListe("", ***CONTEXTE\_TESTS***).executer();

}

}

}

}

## Classe CommandeDebut

### CommandeDebut.java

/\*\*

\* CommandeDebut.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* La commande debut n'a aucun arguments.

\* Lors de son exécution :

\* <ul><li>tous les identificateurs du contexte sont supprimés</li>

\* <li>toutes les lignes de programmes mémorisée sont effacées</li>

\* </ul>

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeDebut **extends** Commande {

/\*\*

\* Initialise une commande debut qui est sans arguments

\* et qui a besoin du contexte.

\* **@param** arguments arguments de debut soit chaîne blanche ou vide

\* **@param** contexte référence du contexte global

\* **@throws** InterpreteurException si arguments n'est pas une chaîne blanche

\* **@throws** NullPointerException si contexte ou arguments est null

\*/

**public** CommandeDebut(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERREUR\_ARGUMENTS = "la commande debut n'a pas d'arguments";

**if** (!arguments.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_ARGUMENTS);

}

}

/\*\*

\* Commande d'exécution de la commande.

\* Efface le contexte.

\* **@return** false car aucun feedback afficher directement

\*/

@Override

**public** **boolean** executer() {

contexte.raz();

*programmeGlobal*.raz();

**return** **false**;

}

}

### TestCommandeDebut.java

/\*\*

\* TestCommandeDebut.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.CommandeDebut;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Tests unitaires de {@link interpreteurlir.motscles.CommandeDebut}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeDebut {

/\*\* Jeux d'essais de CommandeDebut valides pour les tests \*/

**private** CommandeDebut[] fixture = {

**new** CommandeDebut("", **new** Contexte()),

**new** CommandeDebut(" ", **new** Contexte()),

**new** CommandeDebut("\t", **new** Contexte()),

};

/\*\*

\* Tests unitaires de {@link CommandeDebut#CommandeDebut(String, Contexte)}

\*/

**public** **void** testCommandeDebutStringContexte() {

System.***out***.println("\tExécution du test de CommandeDebut"

+ "#CommandeDebut(String, Contexte)");

/\* Tests Commande invalide \*/

String[] arguments = { "$chaine", " a ", "fin" };

Contexte contexte = **new** Contexte();

**for** (**int** numTest = 0 ; numTest < arguments.length ; numTest++) {

**try** {

**new** CommandeDebut(arguments[numTest], contexte);

*echec*();

} **catch** (InterpreteurException lancee) {

}

}

**try** {

**new** CommandeDebut("", **new** Contexte());

**new** CommandeDebut(" ", **new** Contexte());

**new** CommandeDebut("\t", **new** Contexte());

} **catch** (InterpreteurException e) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link CommandeDebut#executer()}

\*/

**public** **void** testExecuter() {

Commande.*referencerProgramme*(**new** Programme());

System.***out***.println("\tExécution du test de CommandeDebut#executer()");

**for** (CommandeDebut cmd : fixture) {

*assertFalse*(cmd.executer());

}

}

}

## Classe CommandeDefs

### CommandeDefs.java

/\*\*

\* CommandeDefs.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* La commande defs n'a aucun argument.

\* Lors de son exécution, elle affiche le contenu du contexte

\* (liste des identificateurs avec leurs valeurs)

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeDefs **extends** Commande {

/\*\*

\* Initialise une commande defs qui est sans arguments et qui a

\* besoin du contexte

\* **@param** arguments arguments de defs soit chaîne blanche ou vide

\* **@param** contexte référence du contexte global

\* **@throws** InterpreteurException si arguments n'est pas une chaîne blanche

\* **@throws** NullPointerException si contexte ou arguments est null

\*/

**public** CommandeDefs(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERREUR\_ARGUMENTS = "la commande defs n'a pas d'arguments";

**if** (!arguments.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_ARGUMENTS);

}

}

/\*\*

\* Commande d'exécution de la commande.

\* Affiche le contexte (liste des identificateurs avec leurs valeurs).

\* **@return** true car l'affichage est un feedback directe de la commande

\*/

@Override

**public** **boolean** execut

er() {

System.***out***.print(contexte.toString());

**return** **true**;

}

}

### TestCommandeDefs.java

/\*\*

\* TestCommandeDefs.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.CommandeDefs;

/\*\*

\* Tests unitaires de {@link interpreteurlir.motscles.CommandeDefs}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeDefs {

/\*\* Jeux d'essais de CommandeDefs valides pour les tests \*/

**private** CommandeDefs[] fixture = {

**new** CommandeDefs("", **new** Contexte()),

**new** CommandeDefs(" ", **new** Contexte()),

**new** CommandeDefs("\t", **new** Contexte()),

};

/\*\*

\* Tests unitaires de {@link CommandeDefs#CommandeDefs(String, Contexte)}

\*/

**public** **void** testCommandeDefsStringContexte() {

System.***out***.println("\tExécution du test de CommandeDefs"

+ "#CommandeDefs(String, Contexte)");

/\* Tests Commande invalide \*/

String[] arguments = { "$chaine", " a ", "fin" };

Contexte contexte = **new** Contexte();

**for** (**int** numTest = 0 ; numTest < arguments.length ; numTest++) {

**try** {

**new** CommandeDefs(arguments[numTest], contexte);

*echec*();

} **catch** (InterpreteurException lancee) {

}

}

**try** {

**new** CommandeDefs("", **new** Contexte());

**new** CommandeDefs(" ", **new** Contexte());

**new** CommandeDefs("\t", **new** Contexte());

} **catch** (InterpreteurException e) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link CommandeDefs#executer()}

\*/

**public** **void** testExecuter() {

System.***out***.println("\tExécution du test de CommandeDefs#executer()");

**for** (CommandeDefs cmd : fixture) {

System.***out***.println("Affichage du contexte :");

*assertTrue*(cmd.executer());

}

}

}

## Classe CommandeEfface

### CommandeEfface.java

/\*\*

\* CommandeEfface.java 16 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Etiquette;

/\*\*

\* Instruction permettant d'effacer une, plusieurs, ou l'intégralité des lignes

\* de code d'un programme écrit dans l'interpréteur LIR.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeEfface **extends** Commande {

/\*\* Erreur nombre incorrect d'arguments \*/

**private** **static** **final** String ***ERREUR\_NB\_ARGS*** =

"usage efface <étiquette\_début>:<étiquette\_fin>";

/\*\* Plage de suppression des lignes de code \*/

**private** Etiquette[] plageSuppression;

/\*\*

\* Initialise cette InstructionEfface à partir des arguments et du

\* contexte passés en paramètres. Modifie le programme global référencé

\* par l'analyseur.

\* **@param** arguments lignes à effacer (tout le programme si vide)

\* **@param** contexte référence du contexte global

\*/

**public** CommandeEfface(String arguments, Contexte contexte) {

**super**(arguments, contexte);

plageSuppression = *analyserArguments*(arguments);

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#executer()

\*/

@Override

**public** **boolean** executer() {

*programmeGlobal*.effacer(plageSuppression[0], plageSuppression[1]);

**return** **false**;

}

/\*\*

\* Découpe la chaîne argument en une plage de deux étiquettes

\* **@param** aDecouper chaîne à analyser

\* **@return** la plage de lignes de code à supprimer sous forme de tableau

\* d'étiquettes.

\*/

**private** **static** Etiquette[] analyserArguments(String aDecouper) {

String[] valeurs = aDecouper.split(":");

**if** (valeurs.length != 2)

**throw** **new** InterpreteurException(***ERREUR\_NB\_ARGS***);

Etiquette[] aRenvoyer = {

**new** Etiquette(valeurs[0]),

**new** Etiquette(valeurs[1])

};

**return** aRenvoyer;

}

}

### TestCommandeEfface.java

/\*\*

\* TestCommandeEfface.java 16 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.CommandeEfface;

**import** interpreteurlir.motscles.instructions.InstructionAffiche;

**import** interpreteurlir.programmes.Etiquette;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Tests unitaires de la commande d'effacement de lignes de codes pour

\* l'interpréteur LIR.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeEfface {

/\*\* Contexte pour tests \*/

**public** **static** **final** Contexte ***CONTEXTE\_TESTS*** = **new** Contexte();

/\*\* Programme global pour tests \*/

**public** **static** **final** Programme ***PGM\_TESTS*** = **new** Programme();

/\*\* Jeu de test valide \*/

**public** **static** **final** CommandeEfface[] ***FIXTURE*** = {

**new** CommandeEfface("1:99999", ***CONTEXTE\_TESTS***),

**new** CommandeEfface(" 1 : 99999 ", ***CONTEXTE\_TESTS***),

**new** CommandeEfface("99999 :1 ", ***CONTEXTE\_TESTS***),

**new** CommandeEfface("1 : 1", ***CONTEXTE\_TESTS***),

**new** CommandeEfface(" 250: 150 ", ***CONTEXTE\_TESTS***)

};

/\*\* Test du constructeur \*/

**public** **static** **void** testCommandeEfface() {

**final** String[] INVALIDES = {

"",

"23 : ",

" : 15",

"coucou",

"50 : coucou",

"12.4: 24",

"-14 : 90",

"\'a\' : 99"

};

System.***out***.println("\tExécution du test de CommandeEfface"

+ "(String, Contexte)");

**for** (String aTester : INVALIDES) {

**try** {

**new** CommandeEfface(aTester, ***CONTEXTE\_TESTS***);

*echec*();

} **catch** (InterpreteurException e) {

// test OK

}

}

}

/\*\* Test de executer() \*/

**public** **static** **void** testExecuter() {

System.***out***.println("\tExécution du test d'executer()\nTest visuel :");

Commande.*referencerProgramme*(***PGM\_TESTS***);

***PGM\_TESTS***.ajouterLigne(**new** Etiquette(10),

**new** InstructionAffiche("Bonjour", ***CONTEXTE\_TESTS***));

***PGM\_TESTS***.ajouterLigne(**new** Etiquette(20),

**new** InstructionAffiche("Comment", ***CONTEXTE\_TESTS***));

***PGM\_TESTS***.ajouterLigne(**new** Etiquette(30),

**new** InstructionAffiche("Allez", ***CONTEXTE\_TESTS***));

***PGM\_TESTS***.ajouterLigne(**new** Etiquette(40),

**new** InstructionAffiche("Vous", ***CONTEXTE\_TESTS***));

***PGM\_TESTS***.ajouterLigne(**new** Etiquette(50),

**new** InstructionAffiche("foobar", ***CONTEXTE\_TESTS***));

System.***out***.println(***PGM\_TESTS***);

CommandeEfface effacement = **new** CommandeEfface("20:30", ***CONTEXTE\_TESTS***);

effacement.executer();

System.***out***.println(***PGM\_TESTS***);

}

}

## Classe CommandeFin

### CommandeFin.java

/\*\*

\* CommandeFin.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* La commande fin n'a aucun argument.

\* Lors de son exécution, elle permet de quitter l'interpreteur en affichant un

\* message d'aurevoir.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeFin **extends** Commande {

/\*\*

\* Initialise une commande fin qui est sans arguments.

\* **@param** arguments arguments de fin soit chaîne blanche ou vide

\* **@param** contexte référence du contexte global

\* **@throws** InterpreteurException si arguments n'est pas une chaîne blanche

\* **@throws** NullPointerException si contexte ou arguments est null

\*/

**public** CommandeFin(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERREUR\_ARGUMENTS = "la commande fin n'a pas d'arguments";

**if** (!arguments.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_ARGUMENTS);

}

}

/\*\*

\* Commande d'exécution de la commande.

\* Quitte l'interpreteur en affichant un message d'aurevoir.

\* **@return** true si la commande affiche un feedback directement sur la sortie

\* standard, sinon false

\*/

@Override

**public** **boolean** executer() {

**final** String MESSAGE\_AUREVOIR = "Au revoir, à bientôt !";

System.***out***.println(MESSAGE\_AUREVOIR);

System.*exit*(0);

**return** **true**;

}

}

### TestCommandeFin.java

/\*\*

\* TestCommandeFin.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.CommandeFin;

/\*\*

\* Tests unitaires de {@link interpreteurlir.motscles.CommandeFin}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeFin {

/\*\*

\* Tests unitaires de {@link CommandeFin#CommandeFin(String, Contexte)}

\*/

**public** **void** testCommandeFinStringContexte() {

System.***out***.println("\tExécution du test de CommandeFin"

+ "#CommandeFin(String, Contexte)");

/\* Tests Commande invalide \*/

String[] arguments = { "$chaine", " a ", "fin" };

Contexte contexte = **new** Contexte();

**for** (**int** numTest = 0 ; numTest < arguments.length ; numTest++) {

**try** {

**new** CommandeFin(arguments[numTest], contexte);

*echec*();

} **catch** (InterpreteurException lancee) {

}

}

**try** {

**new** CommandeFin("", **new** Contexte());

**new** CommandeFin(" ", **new** Contexte());

**new** CommandeFin("\t", **new** Contexte());

} **catch** (InterpreteurException e) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link CommandeFin#executer()}

\*/

**public** **void** testExecuter() {

System.***out***.println("\tExécution du test de CommandeFin#executer()");

System.***out***.println("\tLe programme doit s'éteindre en affichant un "

+ "message d'aurevoir :");

System.***out***.println("Test exécuter désactiver");

//fixture[0].executer();

}

}

## Classe CommandeLance

### CommandeLance.java

/\*\*

\* CommandeLance.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Etiquette;

/\*\*

\* Démarre l'exécution d'un programme à partir de son plus petit numéro

\* d'étiquette.

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeLance **extends** Commande {

**private** Etiquette debutLancer;

/\*\*

\* Initialise la commande lance avec ses arguments et le contexte

\* **@param** arguments vide ou étiquette de lancement

\* **@param** contexte référence du contexte global

\* **@throws** InterpreteurException en cas d'erreur de syntaxe à

\* la création d'une étiquette

\*/

**public** CommandeLance(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**if** (arguments.isBlank()) {

debutLancer = **null**;

} **else** {

debutLancer = **new** Etiquette(arguments);

}

}

/\*\*

\* Exécution de la commande :

\* <ul><li>Lance le programme à partir de l'étiquette la plus

\* petite s'il n'y a pas d'argument</li>

\* <li>Lance le programme à partir de l'étiquette passée

\* en paramètre</li>

\* </ul>

\* **@return** true car un feedback est affiché

\* **@throws** RuntimeException si un programme n'est pas référencé

\* dans la classe {@link Commande}

\*/

**public** **boolean** executer() {

**final** String ERREUR = "erreur exécution";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException(ERREUR);

}

**if** (debutLancer == **null**) {

*programmeGlobal*.lancer();

} **else** {

*programmeGlobal*.lancer(debutLancer);

}

**return** **true**;

}

}

### TestCommandeLance.java

/\*\*

\* TestCommandeLance.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.tests;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.CommandeLance;

**import** **static** info1.outils.glg.Assertions.\*;

**import** info1.outils.glg.TestException;

/\*\*

\* Tests unitaires de la classe CommandeLance

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeLance {

**private** Contexte contexteTest = **new** Contexte();

**private** **final** CommandeLance[] FIXTURE = {

**new** CommandeLance("", contexteTest),

**new** CommandeLance("10", contexteTest),

**new** CommandeLance("9", contexteTest),

**new** CommandeLance("20", contexteTest),

**new** CommandeLance("70", contexteTest),

**new** CommandeLance("40", contexteTest),

};

**private** **final** String[] ARGS\_VALIDES = {

"",

"10",

"9",

"20",

"70",

"40"

};

/\*\*

\* Test unitaire de

\* {@link CommandeLance#CommandeLance(String, interpreteurlir.Contexte)}

\*/

**public** **void** testCommandeLance() {

**final** String[] ARGS\_INVALIDES = {

"greuuuuuu",

" motus 5800",

"100000",

"-4",

"$$$$£££"

};

Expression.*referencerContexte*(contexteTest);

System.***out***.println("\tExécution du test de "

+ "CommandeLance#CommandeLance(String, Contexte)");

**for** (**int** i = 0; i < ARGS\_INVALIDES.length; i++) {

**try** {

**new** CommandeLance(ARGS\_INVALIDES[i], contexteTest);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

**for** (**int** i = 0 ; i < ARGS\_VALIDES.length ; i++) {

**try** {

contexteTest.raz();

**new** CommandeLance(ARGS\_VALIDES[i], contexteTest);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

}

/\*\*

\* Test unitaire de {@link CommandeLance#executer()}

\*/

**public** **void** testExecuter() {

Expression.*referencerContexte*(contexteTest);

System.***out***.println("\tExécution du test de CommandeLance#executer()");

**for** (**int** i = 0 ; i < FIXTURE.length ; i++) {

**try** {

FIXTURE[i].executer();

*echec*();

} **catch** (RuntimeException lancee) {

**if** (lancee **instanceof** TestException) {

*echec*();

}

}

}

// Tests valides faits en intégration

}

}

## Classe CommandeListe

### CommandeListe.java

/\*\*

\* CommandeListe.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Etiquette;

**import** **static** interpreteurlir.programmes.Etiquette.\*;

/\*\*

\* Commande liste affiche les lignes de codes du programme,

\* soit dans leur intégralité, soit dans un intervalle donné

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeListe **extends** Commande {

**private** Etiquette debut;

**private** Etiquette fin;

/\*\*

\* Initialise la commande liste avec ses arguments et le contexte

\*

\* **@param** arguments arguments vide ou contenant les étiquettes à afficher

\* **@param** contexte référence du contexte global

\* **@throws** InterpreteurException en cas d'erreur de syntaxe lors

\* de l'instanciation des étiquettes

\*/

**public** CommandeListe(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** **int** ARGS\_DEBUT = 0;

**final** **int** ARGS\_FIN = 1;

**final** String ERREUR\_INTERVALLE = "usage liste <étiquette\_début>:"

+ "<étiquette\_fin> avec "

+ "<étiquette\_début> <= "

+ "<étiquette\_fin> ";

**if** (arguments.isBlank()) {

debut = **new** Etiquette(***VALEUR\_ETIQUETTE\_MIN***);

fin = **new** Etiquette(***VALEUR\_ETIQUETTE\_MAX***);

} **else** {

String[] decoupage = arguments.split(":");

**if** (decoupage.length < 2) {

**throw** **new** InterpreteurException(ERREUR\_INTERVALLE);

}

debut = **new** Etiquette(decoupage[ARGS\_DEBUT]);

fin = **new** Etiquette(decoupage[ARGS\_FIN]);

**if** (debut.compareTo(fin) > 0) {

**throw** **new** InterpreteurException(ERREUR\_INTERVALLE);

}

}

}

/\*\*

\* Exécution de la commande :

\* <ul><li>Affiche les lignes de code du programme entre l'étiquette

\* de début et celle de fin passées en argument</li>

\* <li>Affiche l'intégralité des lignes de code du programme</li>

\* </ul>

\* **@return** true car un feedback est affiché

\* **@throws** RuntimeException si un programme n'est pas référencé

\* dans la classe {@link Commande}

\*/

**public** **boolean** executer() {

**final** String ERREUR = "erreur exécution";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException(ERREUR);

}

**if** (debut != **null** || fin != **null**) {

System.***out***.print(*programmeGlobal*.listeBornee(debut, fin));

}

**return** **true**;

}

}

### TestCommandeListe.java

/\*\*

\* TestCommandeListe.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.tests;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.CommandeListe;

**import** interpreteurlir.motscles.instructions.Instruction;

**import** interpreteurlir.motscles.instructions.InstructionVar;

**import** interpreteurlir.programmes.Etiquette;

**import** interpreteurlir.programmes.Programme;

**import** **static** info1.outils.glg.Assertions.\*;

**import** info1.outils.glg.TestException;

/\*\*

\* Tests unitaires de la classe Commande liste

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeListe {

**private** Programme programmeTest = **new** Programme();

**private** Contexte contexteTest = **new** Contexte();

**private** **final** CommandeListe[] FIXTURE = {

**new** CommandeListe("1:89", contexteTest),

**new** CommandeListe("13:30", contexteTest),

**new** CommandeListe("17:54", contexteTest),

**new** CommandeListe("40:108", contexteTest),

};

**private** **final** String[] ARGS\_VALIDES = {

"1:90",

"5:45",

"40:56"

};

**private** **final** Etiquette[] JEU\_ETIQUETTES = {

**new** Etiquette(1),

**new** Etiquette(10),

**new** Etiquette(13),

**new** Etiquette(25),

**new** Etiquette(31),

**new** Etiquette(40),

**new** Etiquette(78),

**new** Etiquette(89)

};

**private** **final** Instruction[] JEU\_INSTRUCTIONS = {

**new** InstructionVar("$res = \"1 \"", contexteTest),

**new** InstructionVar("$res = $res + \"10 \"", contexteTest),

**new** InstructionVar("$res = $res + \"13 \"", contexteTest),

**new** InstructionVar("$res = $res + \"25 \"", contexteTest),

**new** InstructionVar("$res = $res + \"31 \"", contexteTest),

**new** InstructionVar("$res = $res + \"40 \"", contexteTest),

**new** InstructionVar("$res = $res + \"78 \"", contexteTest),

**new** InstructionVar("$res = $res + \"89 \"", contexteTest)

};

**private** **void** ecrireProgrammeTest() {

**for** (**int** i = 0 ; i < JEU\_ETIQUETTES.length ; i++) {

programmeTest.ajouterLigne(JEU\_ETIQUETTES[i],

JEU\_INSTRUCTIONS[i]);

}

}

/\*\*

\* Test unitaire de

\* {@link CommandeListe#CommandeListe(String, interpreteurlir.Contexte)}

\*/

**public** **void** testCommandeListe() {

**final** String[] ARGS\_INVALIDES = {

"agreu",

"0:0",

"-4:9",

"45:-8",

"78:12",

"1:",

":4",

"1:100000"

};

System.***out***.println("\tExécution du test de "

+ "CommandeListe#CommandeListe(String, Contexte)");

**for** (**int** i = 0; i < ARGS\_INVALIDES.length; i++) {

**try** {

**new** CommandeListe(ARGS\_INVALIDES[i], contexteTest);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

**for** (**int** i = 0 ; i < ARGS\_VALIDES.length ; i++) {

**try** {

**new** CommandeListe(ARGS\_VALIDES[i], contexteTest);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

}

/\*\*

\* Test unitaire de {@link CommandeListe#executer()}

\*/

**public** **void** testExecuter() {

**for** (**int** i = 0 ; i < FIXTURE.length ; i++) {

**try** {

FIXTURE[i].executer();

*echec*();

} **catch** (RuntimeException lancee) {

**if** (lancee **instanceof** TestException) {

*echec*();

}

// Test OK

}

}

ecrireProgrammeTest();

Commande.*referencerProgramme*(programmeTest);

Expression.*referencerContexte*(contexteTest);

System.***out***.println("\tExécution du test de "

+ "CommandeListe#executer()");

**for** (**int** i = 0 ; i < FIXTURE.length ; i++) {

**try** {

FIXTURE[i].executer();

} **catch** (RuntimeException lancee) {

*echec*();

}

}

}

}

## Classe CommandeSauve

### CommandeSauve.java

/\*\*

\* CommandeSauve.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles;

**import** java.io.PrintStream;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Commande sauve prenant en argument le chemin du fichier

\* (avec une extension .lir) dans lequel est enregistré le programme chargé.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** CommandeSauve **extends** Commande {

/\*\* chemin du fichier d'extension .lir pour la sauvegarde \*/

**private** String cheminFichier;

/\*\*

\* Initialise une commande sauve avec un chemin de fichier .lir passé en

\* argument.

\* **@param** arguments chemin du fichier dans lequel

\* la sauvegarde sera effectuée

\* **@param** contexte référence du contexte global

\* **@throws** InterpreteurException si le chemin du fichier n'a pas

\* l'extension .lir

\* ou si arguments est chaîne blanche

\*/

**public** CommandeSauve(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String EXTENSION = ".lir";

**final** String USAGE = "usage sauve <chemin\_et\_nom\_du\_fichier>.lir";

arguments = arguments.trim();

**if** (arguments.isBlank() || !arguments.endsWith(EXTENSION)) {

**throw** **new** InterpreteurException(USAGE);

}

// else

cheminFichier = arguments;

}

/\*\*

\* Commande d'exécution de la commande.

\* Sauvegarde le programme référencé dans la classe Commande

\* dans le fichier de cette CommandeSauve.

\* **@return** false car aucun feedback afficher directement

\* **@throws** ExecutionException si l'enregistrement est impossible

\* **@throws** RuntimeException si aucun programme référencé dans la classe

\* Commande avec la méthode

\* {@link Commande#referencerProgramme(Programme)}

\*/

@Override

**public** **boolean** executer() {

**final** String MSG\_ERREUR = "impossible de sauvegarder le programme "

+ "dans le fichier (le chemin est peut-être invalide)";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException("Programme non référencé dans Commande");

}

PrintStream aEcrire = **null**;

**try** {

aEcrire = **new** PrintStream(cheminFichier);

aEcrire.print(*programmeGlobal*.toString());

aEcrire.close();

} **catch** (Exception lancee) {

**throw** **new** ExecutionException(MSG\_ERREUR);

}

**return** **false**;

}

}

### TestCommandeSauve.java

/\*\*

\* TestCommandeSauve.java 21 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.CommandeSauve;

**import** interpreteurlir.programmes.Programme;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.tests.ProgrammeDeTest;

**import** java.io.BufferedReader;

**import** java.io.FileInputStream;

**import** java.io.InputStreamReader;

/\*\*

\* Tests unitaires de {@link CommandeSauve}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestCommandeSauve {

/\*\* contexte pour les tests \*/

**private** Contexte contexte = **new** Contexte();

/\*\* Programme pour les tests \*/

**private** Programme progGlobal = **new** Programme();

/\*\* Jeu de donnée de commandeSauve valides pour les tests \*/

**private** CommandeSauve[] fixture = {

/\* chemin valide \*/

**new** CommandeSauve("monProgramme.lir", contexte),

**new** CommandeSauve("programmationLIR\\monProgramme.lir", contexte),

**new** CommandeSauve("D:\\testInterpreteurLIR\\test1.lir", contexte),

**new** CommandeSauve(" D:\\testInterpreteurLIR\\test2.lir\t", contexte),

/\* chemin invalide à l'exécution\*/

**new** CommandeSauve("\\\\monProgramme.lir", contexte),

**new** CommandeSauve("monPro//??!gr<>amme.lir", contexte),

/\* chemin inexistant \*/

**new** CommandeSauve("D:\\testInterpreteurLIR\\dossierNonCree\\test1.lir",

contexte),

/\* lecteur inexistant \*/

**new** CommandeSauve("X:\\testInterpreteurLIR\\test1.lir", contexte),

};

/\*\*

\* Tests unitaires de {@link CommandeSauve#CommandeSauve(String, Contexte)}

\*/

**public** **void** testCommandeSauveStringContexte() {

**final** String[] ARGS\_INVALIDES = {

"",

" \t ",

"D:\\utilisateurs\\defaut\\bureau\\",

"D:\\utilisateurs\\defaut\\bureau\\monProgramme.txt",

"D:\\utilisateurs\\defaut\\bureau\\monProgramme",

"nouveau dossier\\monProgramme.java",

"nouveau dossier\\monProgramme",

"monProgramme.class",

"monProgramme"

};

System.***out***.println("\tExécution du test de "

+ "CommandeSauve#CommandeSauve(String, Contexte)");

**for** (String aTester : ARGS\_INVALIDES) {

**try** {

**new** CommandeSauve(aTester, contexte);

*echec*();

} **catch** (InterpreteurException e) {

// test ok

}

}

**try** {

/\* chemin valide \*/

**new** CommandeSauve("monProgramme.lir", contexte);

**new** CommandeSauve("programmationLIR\\monProgramme.lir", contexte);

**new** CommandeSauve("D:\\testInterpreteurLIR\\test1.lir", contexte);

**new** CommandeSauve(" D:\\testInterpreteurLIR\\test2.lir\t",

contexte);

/\* chemin invalide à l'exécution\*/

**new** CommandeSauve("\\\\monProgramme.lir", contexte);

**new** CommandeSauve("monPro//??!gr<>amme.lir", contexte);

/\* chemin inexistant \*/

**new** CommandeSauve("D:\\testInterpreteurLIR\\dossierNonCree\\"

+ "test1.lir", contexte);

/\* lecteur inexistant \*/

**new** CommandeSauve("X:\\testInterpreteurLIR\\test1.lir", contexte);

} **catch** (InterpreteurException e) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link CommandeSauve#executer()}

\*/

**public** **void** testExecuter() {

**final** **int** INDEX\_INVALIDES = 4;

Commande.*referencerProgramme*(progGlobal);

System.***out***.println("\tExécution du test de CommandeSauve#executer()");

/\* Tests des chemins invalides \*/

**for** (**int** index = INDEX\_INVALIDES ; index < fixture.length ; index++) {

**try** {

fixture[index].executer();

*echec*();

} **catch** (ExecutionException lancee) {

// test OK

}

}

**try** {

*assertFalse*(fixture[0].executer());

*assertEquivalence*(progGlobal.toString(),

*lireFichier*("monProgramme.lir"));

*assertFalse*(fixture[2].executer());

*assertEquivalence*(progGlobal.toString(),

*lireFichier*("D:\\testInterpreteurLIR\\test1.lir"));

ProgrammeDeTest.*genererProgramme*(progGlobal, contexte);

*assertFalse*(fixture[1].executer());

*assertEquivalence*(progGlobal.toString(),

*lireFichier*("programmationLIR\\monProgramme.lir"));

*assertFalse*(fixture[3].executer());

*assertEquivalence*(progGlobal.toString(),

*lireFichier*("D:\\testInterpreteurLIR\\test2.lir"));

} **catch** (ExecutionException lancee) {

*echec*();

}

}

/\*\*

\* Lit un fichier et retourne le contenu entier du fichier

\* **@param** cheminFichier chemin du fichier à lire

\* **@return** contenu du fichier

\*/

**private** **static** String lireFichier(String cheminFichier) {

BufferedReader aTester;

StringBuilder contenu = **new** StringBuilder("");

aTester = **null**;

**try** {

aTester = **new** BufferedReader(

**new** InputStreamReader(

**new** FileInputStream(cheminFichier)));

String ligneLue;

**do** {

ligneLue = aTester.readLine();

**if** (ligneLue != **null**) {

contenu.append(ligneLue).append("\n");

}

} **while** (ligneLue != **null**);

aTester.close();

} **catch** (Exception e) {

*echec*();

}

**return** contenu.toString();

}

}

## EssaiCommande.java

// Classe utilisée pour des tests « d’intégrations » avant la mise en place de

// l’Analyseur

/\*\*

\* EssaiCommande.java 7 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.tests;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.\*;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Essais des commandes (création + éxécution)

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** EssaiCommande {

/\*\*

\* Essais de commandes avec arguments invalides puis valides

\* **@param** args non utilisé

\*/

**public** **static** **void** main(String[] args) {

Contexte contexte = **new** Contexte();

Commande.*referencerProgramme*(**new** Programme());

/\* Erreur dans commande \*/

System.***out***.println("? debut args");

**try** {

**new** CommandeDebut("args", contexte);

} **catch** (InterpreteurException lancee) {

*feedback*(lancee);

}

System.***out***.println("? defs args");

**try** {

**new** CommandeDefs("args", contexte);

} **catch** (InterpreteurException lancee) {

*feedback*(lancee);

}

System.***out***.println("? fin args");

**try** {

**new** CommandeFin("args", contexte);

} **catch** (InterpreteurException lancee) {

*feedback*(lancee);

}

/\* Commande valide et exécution \*/

System.***out***.println("? debut");

*feedback*(**new** CommandeDebut("", contexte).executer());

System.***out***.println("? defs");

*feedback*(**new** CommandeDefs("", contexte).executer());

System.***out***.println("? liste");

*feedback*(**new** CommandeListe("", contexte).executer());

System.***out***.println("? fin");

*feedback*(**new** CommandeFin("", contexte).executer());

System.***err***.println("Erreur, la commande fin n'a pas quitter");

}

**private** **static** **void** feedback(**boolean** nonBesoinFeedback) {

**if** (!nonBesoinFeedback) {

System.***out***.println("ok");

}

}

**private** **static** **void** feedback(InterpreteurException lancee) {

System.***out***.println("nok : " + lancee.getMessage());

}

}

# Paquetage interpreteurlir.motscles.instructions

## Classe Instruction

### Instruction.java

/\*\*

\* Instruction.java 9 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.Commande;

/\*\*

\* Instruction du langage LIR. Chaque instruction se caractérise par une

\* expression et un contexte d'exécution

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **abstract** **class** Instruction **extends** Commande {

/\*\* Expression qui sera exécutée par la commande \*/

**protected** Expression aExecuter;

/\*\*

\* Initialise une instruction à partir du contexte d'éxécution et de

\* l'expression à exécuter

\* **@param** arguments expression qui sera exécutée

\* **@param** contexte global de l'application

\*/

**public** Instruction(String arguments, Contexte contexte) {

**super**(arguments,contexte);

// arguments non utilisés pour Instruction générale

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.Commande#executer()

\*/

@Override

**public** **abstract** **boolean** executer();

/\*

\* Non - javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** **abstract** String toString();

}

### TestInstruction.java

// Tests faits avant le passage en abstract de la classe

/\*\*

\* TestInstruction.java 9 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.instructions.Instruction;

/\*\*

\* Tests unitaires des instructions

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestInstruction {

/\*\*

\* Test unitaire de {@link Instruction#Instruction(String, Contexte)}

\*/

**public** **static** **void** testInstruction() {

System.***out***.println("\tExécution du test de Instruction()");

**try** {

**new** Instruction("Bonjour", **new** Contexte());

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Test unitaire de {@link Instruction#toString()}

\*/

**public** **static** **void** testToString() {

System.***out***.println("\tExécution du test de toString()");

Instruction aTester = **new** Instruction("Bonjour", **new** Contexte());

*assertEquivalence*(aTester.toString(), "Instruction null");

}

}

## Classe InstructionAffiche

### InstructionAffiche.java

/\*\*

\* InstructionAffiche.java 13 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

/\*\*

\* Affiche sur la sortie standard une expression passée en argument. Cette

\* expression ne doit pas contenir d'affectation. Si aucune expression n'est

\* passée en argument, effectue un retour à la ligne.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** InstructionAffiche **extends** Instruction {

/\*\* Erreur d'affectation illégale \*/

**private** **static** **final** String ***AFFECTATION\_ILLEGALE*** =

"affectation impossible avec la commande affiche";

/\*\*

\* Initialise cette InstructionAffiche à partir de son contexte global

\* d'exécution et de l'expression passée en argument. Lève une exception

\* si cette expression contient une affectation.

\* **@param** arguments contenant l'expression dont le résultat doit être

\* affiché.

\* **@param** contexte global de la session d'interpreteurlir

\* **@throws** InterpreteurException si la chaîne arguments contient un signe

\* égal en dehors d'un littéral de chaîne de caractères.

\*/

**public** InstructionAffiche(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**if** (Expression.*detecterCaractere*(arguments, '=') >= 0) {

**throw** **new** InterpreteurException(***AFFECTATION\_ILLEGALE***);

}

aExecuter = arguments.isBlank()

? **null**

: Expression.*determinerTypeExpression*(arguments.trim());

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#executer()

\*/

@Override

**public** **boolean** executer() {

**if** (aExecuter == **null**) {

System.***out***.println();

} **else** {

System.***out***.print(aExecuter.calculer().getValeur());

}

**return** **true**;

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#toString()

\*/

@Override

**public** String toString() {

**return** "affiche" + (aExecuter == **null** ? ""

: " " + aExecuter.toString());

}

}

### TestInstructionAffiche.java

/\*\*

\* TestInstructionAffiche.java 13 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.instructions.InstructionAffiche;

/\*\*

\* Tests unitaires de l'instruction affiche, avec et sans arguments.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestInstructionAffiche {

/\*\* Contexte d'execution pour jeux de tests \*/

**private** **static** **final** Contexte ***CONTEXTE\_GBL*** = **new** Contexte();

/\*\* Jeu données valides pour test de InstructionAffiche \*/

**private** **static** **final** InstructionAffiche[] ***FIXTURE*** = {

**new** InstructionAffiche("", ***CONTEXTE\_GBL***),

**new** InstructionAffiche(" ", ***CONTEXTE\_GBL***),

**new** InstructionAffiche("\"Hello World !!!\"", ***CONTEXTE\_GBL***),

**new** InstructionAffiche("3 + 3", ***CONTEXTE\_GBL***),

**new** InstructionAffiche("marcel", ***CONTEXTE\_GBL***),

**new** InstructionAffiche("marcel + -3", ***CONTEXTE\_GBL***),

**new** InstructionAffiche("$fraysse", ***CONTEXTE\_GBL***),

**new** InstructionAffiche("$sanchis + \"coucou\"", ***CONTEXTE\_GBL***),

**new** InstructionAffiche("\"300000000000000000 ça passe\"", ***CONTEXTE\_GBL***)

};

/\*\*

\* Tests unitaires de

\* {@link InstructionAffiche#InstructionAffiche(String, Contexte)}

\*/

**public** **static** **void** testInstructionAffiche() {

**final** String[] INVALIDES = {

"a = b + c",

"3 +",

"une chaine de plus de soixtante quinze caractères ne devrait pas"

+ "pouvoir s'afficher parce qu'elle est trop longue !",

"30000000000000000000000",

"12aveyron",

"$aveyron + 12"

};

Expression.*referencerContexte*(***CONTEXTE\_GBL***);

System.***out***.println("\tExécution du test de InstructionAffiche(String"

+ ", Contexte)");

**for** (String argInvalide : INVALIDES) {

**try** {

**new** InstructionAffiche(argInvalide, ***CONTEXTE\_GBL***);

*echec*();

} **catch** (InterpreteurException | ExecutionException lancee) {

// Empty body

}

}

}

/\*\*

\* Tests unitaires de {@link InstructionAffiche#executer()}

\*/

**public** **static** **void** testExecuter() {

System.***out***.println("\tExécution du test de executer()\nTEST VISUEL SUR "

+ "CONSOLE :");

Expression.*referencerContexte*(***CONTEXTE\_GBL***);

**for** (InstructionAffiche aLancer : ***FIXTURE***) {

System.***out***.println("\n\ttest visuel suivant : ");

aLancer.executer();

}

System.***out***.println();

}

/\*\*

\* Tests unitaires de {@link InstructionAffiche#toString()}

\*/

**public** **static** **void** testToString() {

**final** String[] ATTENDUS = {

"affiche",

"affiche",

"affiche \"Hello World !!!\"",

"affiche 3 + 3",

"affiche marcel",

"affiche marcel + -3",

"affiche $fraysse",

"affiche $sanchis + \"coucou\"",

"affiche \"300000000000000000 ça passe\""

};

System.***out***.println("\tExécution du test de toString()");

**for** (**int** i = 0 ; i < ***FIXTURE***.length ; i++) {

*assertTrue*(***FIXTURE***[i].toString().compareTo(ATTENDUS[i]) == 0);

}

}

}

## Classe InstructionEntre

### InstructionEntre.java

/\*\*

\* InstructionEntre.java 13 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.\*;

**import** interpreteurlir.donnees.litteraux.Chaine;

**import** interpreteurlir.donnees.litteraux.Entier;

**import** java.util.Scanner;

/\*\*

\* Instruction qui attend que l'utilisateur

\* entre une valeur sur l'entrée standard du type de l'identificateur argument

\* cette valeur sera affectée dans une variable ayant cet

\* identificateur dans le contexte.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** InstructionEntre **extends** Instruction {

/\*\*

\* Identificateur à affecter à partir de la valeur

\* saisie à l'execution de cette instruction

\*/

**private** Identificateur id;

/\*\*

\* Initialise une instruction entre avec un

\* identificateur chaine ou entier en argument

\* **@param** arguments représentation texte de l'indentificateur

\* **@param** contexte contexte pour l'enregistrement de la valeur

\* saisie à l'execution

\* **@throws** InterpreteurException si argument n'est pas

\* un identificateur valide

\* **@throws** NullPointerException si contexte ou argument est null

\*/

**public** InstructionEntre(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERREUR\_ARG = "usage entre <identificateur>";

**if** (arguments.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_ARG);

}

**if** (arguments.indexOf("$") >= 0) {

id = **new** IdentificateurChaine(arguments.trim());

} **else** {

id = **new** IdentificateurEntier(arguments.trim());

}

}

/\*\*

\* Execution de l'instruction :

\* L'utilisateur saisi une valeur sur l'entrée standard qui sera affectée

\* à une variable dans le contexte ayant comme identificateur celui

\* de l'instruction si le type est compatible

\* **@return** false car aucun feedback affiché directement

\* **@throws** ExecutionException si la valeur saisie n'est pas compatible

\* avec le type de l'identificateur de l'instruction

\*/

**public** **boolean** executer() {

**final** String MESSAGE\_ERREUR\_TYPE = "type saisi "

+ "et type demandé incompatibles";

@SuppressWarnings("resource") // ne pas fermer sinon crash

Scanner entree = **new** Scanner(System.***in***);

String valeurSaisie = entree.nextLine();

**try** {

**if** (id **instanceof** IdentificateurEntier) {

contexte.ajouterVariable(id, **new** Entier(valeurSaisie.trim()));

} **else** {

contexte.ajouterVariable(id, **new** Chaine("\""

+ valeurSaisie + "\""));

}

} **catch** (InterpreteurException lancee) {

**throw** **new** ExecutionException(MESSAGE\_ERREUR\_TYPE);

}

**return** **false**;

}

/\* non javadoc

\* @see java.lang.Object#toString()

\*/

@Override

**public** String toString() {

**return** "entre " + id;

}

}

### TestInstructionEntre.java

/\*\*

\* TestInstructionEntre.java 13 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** interpreteurlir.motscles.instructions.InstructionEntre;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Test unitaire de {@link InstructionEntre}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestInstructionEntre {

/\*\*

\* Contexte pour les tests

\*/

**private** **final** Contexte CONTEXTE\_GLB = **new** Contexte();

/\*\*

\* Jeux de données de instructionEntre valides

\*/

**private** InstructionEntre[] fixture = {

**new** InstructionEntre("$chaine ", CONTEXTE\_GLB),

**new** InstructionEntre(" $toto", CONTEXTE\_GLB),

**new** InstructionEntre("\t entier ", CONTEXTE\_GLB),

**new** InstructionEntre("resultat", CONTEXTE\_GLB),

};

/\*\*

\* Test unitaire de

\* {@link InstructionEntre#InstructionEntre(String, Contexte)}

\*/

**public** **void** testInstructionEntreStringContexte() {

System.***out***.println("\tExécution du test de "

+ "InstructionEntre#InstructionEntre(String, Contexte)");

**final** Contexte CONTEXTE = **new** Contexte();

**final** String[] ARGS\_INVALIDES = {

"",

"$hhjdkeliyehozrbnjkm236khl749k",

"$chaine = $toto + \"\"",

"entier/2",

"45"

};

**for** (String arg : ARGS\_INVALIDES) {

**try** {

**new** InstructionEntre(arg, CONTEXTE);

*echec*();

} **catch** (InterpreteurException lancee) {

}

}

**try** {

**new** InstructionEntre("$chaine ", CONTEXTE);

**new** InstructionEntre(" $toto", CONTEXTE);

**new** InstructionEntre("\t entier ", CONTEXTE);

**new** InstructionEntre("resultat", CONTEXTE);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Test unitaire de {@link InstructionEntre#toString()}

\*/

**public** **void** testToString() {

**final** String[] TEXTE\_ATTENDU = {

"entre $chaine", "entre $toto", "entre entier", "entre resultat"

};

System.***out***.println("\tExécution du test de "

+ "InstructionEntre#toString()");

**for** (**int** numTest = 0 ; numTest < TEXTE\_ATTENDU.length ; numTest++) {

*assertEquivalence*(TEXTE\_ATTENDU[numTest],

fixture[numTest].toString());

}

}

/\*\*

\* Test unitaire de {@link InstructionEntre#executer()}

\*/

**public** **void** testExecuter() {

System.***out***.println("Execution du test de InstructionEntre#executer()");

**for** (InstructionEntre entre : fixture) {

System.***out***.println("? " + entre);

**try** {

*assertFalse*(entre.executer());

System.***out***.println("ok");

} **catch** (ExecutionException lancee) {

System.***err***.println("nok : " + lancee.getMessage());

}

}

System.***out***.println("Contexte : \n" + CONTEXTE\_GLB);

}

}

## Classe InstructionProcedure

### InstructionProcedure.java

/\*\*

\* InstructionProcedure.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Etiquette;

/\*\*

\* Instruction qui transfère l'exécution au numéro d'étiquette spécifié

\* et reprendra en séquence lorsque la procédure sera terminée

\* (instruction retour)

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*

\*/

**public** **class** InstructionProcedure **extends** Instruction {

/\*\* Etiquette désignant le début de la procedure à exécuter \*/

**private** Etiquette debutProcedure;

/\*\*

\* Initialise une procédure avec une étiquette en argument.

\* **@param** arguments Représentation texte d'une étiquette

\* **@param** contexte Contexte de la session de l'interpreteur LIR

\* **@throws** InterpreteurException Si un arguments ne corresponds

\* pas a une étiquette valide

\* **@throws** NullPointerException Si contexte ou argument est null

\*/

**public** InstructionProcedure(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERREUR\_ARG = "usage procedure <étiquette>";

**if**(arguments.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_ARG);

}

debutProcedure = **new** Etiquette(arguments);

}

/\* non javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#toString()

\*/

@Override

**public** String toString() {

**return** "procedure " + debutProcedure;

}

/\*\*

\* Execution de l'instruction :

\* Appel d'une procedure située à l'étiquette de l'instruction.

\* L'appel s'empile sur le contexte appellant pour ce qui est du

\* compteur ordinal.

\* **@return** false car aucun feedback affiché directement

\* **@throws** RuntimeException si un programme n'est pas référencé en membre

\* de classe de Commande.

\*/

**public** **boolean** executer() {

**final** String ERREUR\_REFERENCEMENT = "Le programme doit être référencé "

+ "dans la classe commande";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException(ERREUR\_REFERENCEMENT);

}

*programmeGlobal*.appelProcedure(debutProcedure);

**return** **false**;

}

}

### TestInstructionProcedure.java

/\*\*

\* TestInstructionProcedure.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.instructions.InstructionProcedure;

**import** interpreteurlir.motscles.instructions.InstructionVar;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurEntier;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.programmes.\*;

**import** **static** info1.outils.glg.Assertions.\*;

**import** info1.outils.glg.TestException;

/\*\*

\* Tests unitaires de {@link InstructionProcedure}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*

\*/

**public** **class** TestInstructionProcedure {

/\*\* Contexte global pour les tests \*/

**private** **final** Contexte CONTEXTE = **new** Contexte();

/\*\* Jeu de donnée d'InstructionProcedure valides \*/

**private** **final** InstructionProcedure[] FIXTURE = {

**new** InstructionProcedure(" 1 ", CONTEXTE),

**new** InstructionProcedure(" 10", CONTEXTE),

**new** InstructionProcedure("5 ", CONTEXTE),

**new** InstructionProcedure("1549", CONTEXTE),

**new** InstructionProcedure("99999", CONTEXTE)

};

/\*\* Programme utilisé dans les tests \*/

**private** **final** Programme PROG\_REFERENCE = **new** Programme();

/\*\*

\* Tests unitaires de

\* {@link InstructionProcedure#InstructionProcedure(String, Contexte)}

\*/

**public** **void** testInstructionProcedureStringContexte() {

System.***out***.println("\tExecution du test de "

+ "InstructionProcedure"

+ "#InstructionProcedure(String, Contexte)");

**final** String[] ARGS\_INVALIDES = {

/\* Sans arguments \*/

"",

"\t",

" ",

"\n",

/\* Arguments invalides \*/

"LETTRE",

"6messages",

"-5",

"100000"

};

**for** (**int** i = 0 ; i < ARGS\_INVALIDES.length ; i++) {

**try** {

**new** InstructionProcedure(ARGS\_INVALIDES[i], CONTEXTE);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

**try** {

**new** InstructionProcedure(" 1 ", CONTEXTE);

**new** InstructionProcedure(" 10", CONTEXTE);

**new** InstructionProcedure("5 ", CONTEXTE);

**new** InstructionProcedure("1549", CONTEXTE);

**new** InstructionProcedure("99999", CONTEXTE);

} **catch** (InterpreteurException e) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link InstructionProcedure#toString()}

\*/

**public** **void** testToString() {

System.***out***.println("\tExecution du test de "

+ "InstructionProcedure#toString()");

**final** String[] FORMAT\_ATTENDU = {

"procedure 1",

"procedure 10",

"procedure 5",

"procedure 1549",

"procedure 99999",

};

**for** (**int** i = 0 ; i < FORMAT\_ATTENDU.length ; i++) {

*assertEquivalence*(FORMAT\_ATTENDU[i], FIXTURE[i].toString());

}

}

/\*\*

\* Tests unitaires de {@link InstructionProcedure#executer()}

\*/

**public** **void** testExecuter() {

System.***out***.println("\tExecution du test de "

+ "InstructionProcedure#executer()");

**for**(InstructionProcedure instruction : FIXTURE) {

**try** {

instruction.executer();

*echec*();

} **catch** (RuntimeException e) {

**if** (e **instanceof** TestException) {

*echec*();

}

// Test OK

}

}

Commande.*referencerProgramme*(PROG\_REFERENCE);

Expression.*referencerContexte*(CONTEXTE);

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(3),

**new** InstructionVar("test=5", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(4), FIXTURE[1]);

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(5),

**new** InstructionVar("test=-1", CONTEXTE));

PROG\_REFERENCE.lancer();

*assertEquivalence*(CONTEXTE.lireValeurVariable(

**new** IdentificateurEntier("test")).getValeur(), 5);

}

}

## Classe InstructionRetour

### InstructionRetour.java

/\*\*

\* InstructionRetour.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Instruction qui transfère l'exécution au numéro d'étiquette

\* appelant (continue en séquence après l'instruction procedure qui à généré

\* l'appel).

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*

\*/

**public** **class** InstructionRetour **extends** Instruction {

/\*\*

\* Initialise une procedure qui est sans argument

\* **@param** arguments Argument de retour soit une chaine blanche ou vide

\* **@param** contexte Contexte de la session de l'interpreteur LIR

\* **@throws** InterpreteurException Si arguments n'est pas une chaîne blanche

\* ou vide.

\* **@throws** NullPointerException Si contexte ou argument est null

\*/

**public** InstructionRetour(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERREUR\_ARG = "l'instruction retour n'a pas d'arguments";

**if** (!arguments.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_ARG);

}

}

/\* non javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#toString()

\*/

@Override

**public** String toString() {

**return** "retour";

}

/\*\*

\* Execution de l'instruction :

\* Retour d'une procédure en séquence après l'instruction procédure

\* appelante.

\* **@return** false car aucun feedback affiché directement

\* **@throws** RuntimeException si un programme n'est pas référencé en membre

\* de classe de Commande.

\* **@throws** ExecutionException Lorsque retour est exécuté alors qu'aucune

\* Instruction procedure n'a été exécutée avant.

\*/

**public** **boolean** executer() {

**final** String ERREUR\_REFERENCEMENT = "Le programme doit être référencé "

+ "dans la classe commande";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException(ERREUR\_REFERENCEMENT);

}

*programmeGlobal*.retourProcedure();

**return** **false**;

}

}

### TestInstructionRetour.java

/\*\*

\* TestInstructionRetour.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.instructions.InstructionProcedure;

**import** interpreteurlir.motscles.instructions.InstructionRetour;

**import** interpreteurlir.motscles.instructions.InstructionVar;

**import** interpreteurlir.programmes.Etiquette;

**import** interpreteurlir.programmes.Programme;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.IdentificateurEntier;

**import** interpreteurlir.expressions.Expression;

**import** **static** info1.outils.glg.Assertions.\*;

**import** info1.outils.glg.TestException;

/\*\*

\* Tests unitaires de {@link InstructionRetour}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*

\*/

**public** **class** TestInstructionRetour {

/\*\* Contexte global pour les tests \*/

**private** **final** Contexte CONTEXTE = **new** Contexte();

/\*\* Programme utilisé dans les tests \*/

**private** **final** Programme PROG\_REFERENCE = **new** Programme();

/\*\* Jeu de donnée d'InstructionRetour valides \*/

**private** **final** InstructionRetour[] FIXTURE = {

**new** InstructionRetour("", CONTEXTE),

**new** InstructionRetour(" ", CONTEXTE),

**new** InstructionRetour("\t", CONTEXTE),

**new** InstructionRetour("\t ", CONTEXTE)

};

/\*\*

\* Tests unitaires de

\* {@link InstructionRetour#InstructionRetour(String, Contexte)}

\*/

**public** **void** testInstructionRetourStringContexte() {

System.***out***.println("\t Exécution du test de "

+ "InstructionRetour#InstructionRetour(String, Contexte)");

**final** String[] ARGS\_INVALIDES = {

" a ",

"bonjour bonsoir",

"513",

"@!?/",

"^p65Na@"

};

**for** (**int** i = 0 ; i < ARGS\_INVALIDES.length ; i++) {

**try** {

**new** InstructionRetour(ARGS\_INVALIDES[i], CONTEXTE);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

**try** {

**new** InstructionRetour("", CONTEXTE);

**new** InstructionRetour(" ", CONTEXTE);

**new** InstructionRetour("\t", CONTEXTE);

**new** InstructionRetour("\t ", CONTEXTE);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link InstructionRetour#toString()}

\*/

**public** **void** testToString() {

System.***out***.println("\tExecution du test de "

+ "InstructionRetour#toString()");

**for** (**int** i = 0 ; i < FIXTURE.length ; i++) {

*assertEquivalence*("retour", FIXTURE[i].toString());

}

}

/\*\*

\* Tests unitaires de {@link InstructionRetour#executer()}

\*/

**public** **void** testExecuter() {

System.***out***.println("\tExecution du test de "

+ "InstructionRetour#executer()");

**for**(InstructionRetour instruction : FIXTURE) {

**try** {

instruction.executer();

*echec*();

} **catch** (RuntimeException e) {

**if** (e **instanceof** TestException) {

*echec*();

}

// Test OK

}

}

Commande.*referencerProgramme*(PROG\_REFERENCE);

Expression.*referencerContexte*(CONTEXTE);

/\* Test retour procedure invalide \*/

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(1), FIXTURE[0]);

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(4),

**new** InstructionProcedure("1", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(10), FIXTURE[1]);

**try** {

PROG\_REFERENCE.lancer(**new** Etiquette(2));

*echec*();

} **catch** (ExecutionException lancee) {

// Test OK

}

PROG\_REFERENCE.raz();

/\* Tests retour procedure valide \*/

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(1),

**new** InstructionVar("test=5", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(2), FIXTURE[0]);

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(3),

**new** InstructionVar("test=-1", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(4),

**new** InstructionProcedure("1", CONTEXTE));

PROG\_REFERENCE.lancer(**new** Etiquette(3));

*assertEquivalence*(CONTEXTE.lireValeurVariable(

**new** IdentificateurEntier("test")).getValeur(), 5);

}

}

## Classe InstructionSi(Vaen)

### InstructionSi.java

/\*\*

\* InstructionSi.java 22 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.ExecutionException;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.ExpressionBooleenne;

**import** interpreteurlir.programmes.Etiquette;

/\*\*

\* Instruction de saut conditionnel.

\* La syntaxe est "si expression\_booléenne vaen etiquette".

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** InstructionSi **extends** Instruction {

/\*\* expression booléenne de this qui est la condition du saut \*/

**private** ExpressionBooleenne condition;

/\*\* etiquette pour le saut conditionnel \*/

**private** Etiquette saut;

/\*\*

\* Initialise une instruction de saut conditionnel à partir des arguments

\* **@param** arguments chaîne argument de la forme

\* "si expression\_booléenne vaen etiquette"

\* **@param** contexte contexte global

\* **@throws** InterpreteurException si syntaxe invalide (si ... vaen ...)

\* ou si expression\_booléenne invalide

\* ou si etiquette invalide

\*/

**public** InstructionSi(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERR\_SYNTAXE = "usage si <expression\_booléenne>"

+ " vaen <étiquette>";

arguments = arguments.trim();

**if** (arguments.isBlank()) {

**throw** **new** InterpreteurException(ERR\_SYNTAXE);

}

**int** indexVaen = arguments.lastIndexOf("vaen");

**if** (indexVaen < 1) {

**throw** **new** InterpreteurException(ERR\_SYNTAXE);

}

String expression = arguments.substring(0, indexVaen);

String etiquette = arguments.substring(indexVaen + 4,

arguments.length());

condition = **new** ExpressionBooleenne(expression);

saut = **new** Etiquette(etiquette);

}

/\*\*

\* Execution de l'instruction :

\* Realise un saut a l'étiquette spécifiée

\* si l'expression booléenne est true.

\* **@return** false car aucun feedback affiché directement

\* **@throws** RuntimeException si un programme n'est pas référencé en membre

\* de classe de Commande.

\* **@throws** ExecutionException si l'étiquette n'existe pas dans le programme

\*/

@Override

**public** **boolean** executer() {

**final** String ERREUR\_REFERENCEMENT = "Le programme doit être référencé "

+ "dans la classe commande";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException(ERREUR\_REFERENCEMENT);

}

**if** (condition.calculer().getValeur()) {

*programmeGlobal*.vaen(saut);

}

**return** **false**;

}

/\* non javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#toString()

\*/

@Override

**public** String toString() {

**return** "si " + condition + " vaen " + saut;

}

}

### TestInstructionSi.java

/\*\*

\* TestInstructionSi.java 22 mai 2021

\* IUT Rodez info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.instructions.InstructionSi;

**import** interpreteurlir.motscles.instructions.InstructionVar;

**import** interpreteurlir.programmes.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.donnees.\*;

**import** interpreteurlir.donnees.litteraux.\*;

**import** interpreteurlir.expressions.Expression;

**import** **static** info1.outils.glg.Assertions.\*;

/\*\*

\* Tests unitaires de {@link InstructionSi}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestInstructionSi {

/\*\* contexte pour les tests \*/

**private** Contexte contexte = **new** Contexte();

/\*\* programme pour les tests \*/

**private** Programme prog = **new** Programme();

/\*\* Jeu de donnée d'instruction si vaen valides pour les tests\*/

**private** InstructionSi[] fixture = {

**new** InstructionSi("45 = 2 vaen 15", contexte),

**new** InstructionSi("age >= 130 vaen 1000", contexte),

**new** InstructionSi("$prenom <>\"défaut\" vaen 16", contexte),

**new** InstructionSi("resultat < 20 vaen 17", contexte),

**new** InstructionSi("resultat < moyenne vaen 18", contexte),

**new** InstructionSi("age > 20 vaen 19", contexte),

**new** InstructionSi("\"tata\" = \"tata\"vaen 20", contexte),

**new** InstructionSi("\"toto \" > $toto vaen1502", contexte),

**new** InstructionSi("-5 <= 0 vaen 21", contexte),

**new** InstructionSi("resultat < 20 vaen 22", contexte),

};

/\*\*

\* Tests unitaires de {@link InstructionSi#InstructionSi(String, Contexte)}

\*/

**public** **void** testInstructionSiStringContexte() {

**final** String[] ARGS\_INVALIDES = {

"",

" \t",

" entier < index",

"vaen 1050",

"age = 10 vaen",

" $prenom = \"défaut\" va 10",

"$prenom <> $nom goto 45",

"$prenom <> $nom vaen dix",

"$prenom != $nom vaen 45",

/\* erreur de type \*/

"$prenom <> 5 vaen 450",

"age > \"\" vaen 450",

"age >= $prenom vaen 450",

"\"dix\" = 10 vaen 4500",

};

System.***out***.println("\tExécution du test de "

+ "InstructionSi#InstructionSi(String, Contexte)");

**for** (String aTester : ARGS\_INVALIDES) {

**try** {

**new** InstructionSi(aTester, contexte);

*echec*();

} **catch** (InterpreteurException lancee) {

// testok

}

}

**try** {

**new** InstructionSi("45 = 2 vaen 15", contexte);

**new** InstructionSi("age >= 130 vaen 1000", contexte);

**new** InstructionSi("$prenom <>\"défaut\" vaen 15", contexte);

**new** InstructionSi("resultat < 20 vaen 15", contexte);

**new** InstructionSi("resultat < moyenne vaen 15", contexte);

**new** InstructionSi("age > 20 vaen 15", contexte);

**new** InstructionSi("\"tata\" = \"tata\"vaen 15", contexte);

**new** InstructionSi("\"toto \" > $toto vaen1502", contexte);

**new** InstructionSi("-5 <= 0 vaen 15", contexte);

**new** InstructionSi("resultat < 20 vaen 15", contexte);

**new** InstructionSi("$chaine <= \"vaen 15\" vaen 15", contexte);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link InstructionSi#toString()}

\*/

**public** **void** testToString() {

**final** String[] ATTENDU = {

"si 45 = 2 vaen 15",

"si age >= 130 vaen 1000",

"si $prenom <> \"défaut\" vaen 16",

"si resultat < 20 vaen 17",

"si resultat < moyenne vaen 18",

"si age > 20 vaen 19",

"si \"tata\" = \"tata\" vaen 20",

"si \"toto \" > $toto vaen 1502",

"si -5 <= 0 vaen 21",

"si resultat < 20 vaen 22",

};

System.***out***.println("\tExécution du test de InstructionSi#toString()");

**for** (**int** numTest = 0 ; numTest < ATTENDU.length ; numTest++) {

*assertEquivalence*(ATTENDU[numTest], fixture[numTest].toString());

}

}

/\*\*

\* Tests unitaires de {@link InstructionSi#executer()}

\*/

**public** **void** testExecuter() {

Commande.*referencerProgramme*(prog);

prog.ajouterLigne(**new** Etiquette(15),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(16),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(17),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(18),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(19),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(20),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(21),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(22),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(1000),

**new** InstructionVar("valeur = valeur -1", contexte));

prog.ajouterLigne(**new** Etiquette(1502),

**new** InstructionVar("valeur = valeur -1", contexte));

Expression.*referencerContexte*(contexte);

**final** **int**[] VALEUR\_ATTENDU = {

0, // pas de saut

0,

-9, // saut en 16

-8, // saut en 17

0,

-6, // saut en 19

-5, // saut en 20

-1, // saut en 1502

-4, // saut en 21

-3, // saut en 22

};

System.***out***.println("\tExécution du test de InstructionSi#executer()");

**for** (**int** numTest = 0 ; numTest < VALEUR\_ATTENDU.length ; numTest++) {

/\* initialisation du contexte \*/

contexte.raz();

contexte.ajouterVariable(**new** IdentificateurEntier("moyenne"),

**new** Entier("-2"));

contexte.ajouterVariable(**new** IdentificateurEntier("age"),

**new** Entier("99"));

contexte.ajouterVariable(**new** IdentificateurChaine("$toto"),

**new** Chaine("\"toto\""));

fixture[numTest].executer();

*assertEquivalence*(VALEUR\_ATTENDU[numTest],

((Integer)contexte.lireValeurVariable(

**new** IdentificateurEntier("valeur"))

.getValeur()).intValue());

}

}

}

## Classe InstructionStop

### InstructionStop.java

/\*\*

\* InstructionStop.java 16 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

/\*\*

\* Instruction stop servant à marquer la fin d'un programme de l'interpréteur

\* LIR. Aucune ligne de code portant une étiquette supérieure ne sera donc lue.

\* L'usage de cette instruction en ligne de commande directe n'a aucun effet.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** InstructionStop **extends** Instruction {

/\*\* Message d'erreur si instruction passée avec des arguments \*/

**private** **static** **final** String ***ERREUR\_ARGUMENTS*** =

"l'instruction stop n'a pas d'arguments";

/\*\*

\* Initialise cette instruction stop à partir des arguments, du contexte

\* et du programme passés en paramètres. Cette instruction ne modifie que

\* le programme. Si arguments n'est pas vide, une exception sera levée.

\* **@param** arguments doit être vide

\* **@param** contexte global de la session de l'interpréteur

\* **@throws** InterpreteurException si arguments non vide

\*/

**public** InstructionStop(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**if** (!arguments.isBlank())

**throw** **new** InterpreteurException(***ERREUR\_ARGUMENTS***);

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#executer()

\*/

@Override

**public** **boolean** executer() {

**final** String ERREUR\_REFERENCEMENT = "Le programme doit être référencé "

+ "dans la classe commande";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException(ERREUR\_REFERENCEMENT);

}

*programmeGlobal*.stop();

**return** **false**;

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#toString()

\*/

@Override

**public** String toString() {

**return** "stop";

}

}

### TestInstructionStop.java

/\*\*

\* TestInstructionStop.java 16 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.instructions.InstructionAffiche;

**import** interpreteurlir.motscles.instructions.InstructionStop;

**import** interpreteurlir.programmes.Etiquette;

**import** interpreteurlir.programmes.Programme;

/\*\*

\* Tests unitaires de l'instruction stop de l'interpréteur LIR.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestInstructionStop {

/\*\* Contexte d'exécution nécessaire à instanciation \*/

**private** **static** **final** Contexte ***CONTEXTE\_TESTS*** = **new** Contexte();

/\*\* Instruction stop valide \*/

**public** **static** **final** InstructionStop[] ***FIXTURE*** = {

**new** InstructionStop("", ***CONTEXTE\_TESTS***),

**new** InstructionStop("\t", ***CONTEXTE\_TESTS***),

**new** InstructionStop(" ", ***CONTEXTE\_TESTS***)

};

/\*\* Tests du constructeur \*/

**public** **static** **void** testInstructionStop() {

**final** String[] INVALIDES = {

"coucou",

" Bonjour",

**null**,

"entier = 2 + 3"

};

System.***out***.println("\tExécution du test de InstructionStop"

+ "(String, Contexte)");

**for** (String aTester : INVALIDES) {

**try** {

**new** InstructionStop(aTester, ***CONTEXTE\_TESTS***);

*echec*();

} **catch** (InterpreteurException | NullPointerException e) {

// Empty block

}

}

}

/\*\* Test de executer() \*/

**public** **static** **void** testExecuter() {

Programme pgmTest = **new** Programme();

System.***out***.println("\tExécution du test de executer()\nTest Visuels\n");

Commande.*referencerProgramme*(pgmTest);

Expression.*referencerContexte*(***CONTEXTE\_TESTS***);

pgmTest.ajouterLigne(**new** Etiquette(10),

**new** InstructionAffiche("\"Bonjour\"", ***CONTEXTE\_TESTS***));

pgmTest.ajouterLigne(**new** Etiquette(20),

**new** InstructionAffiche("\"Comment\"", ***CONTEXTE\_TESTS***));

pgmTest.ajouterLigne(**new** Etiquette(30),

**new** InstructionAffiche("\"Allez\"", ***CONTEXTE\_TESTS***));

pgmTest.ajouterLigne(**new** Etiquette(40),

**new** InstructionAffiche("\"Vous\"", ***CONTEXTE\_TESTS***));

pgmTest.ajouterLigne(**new** Etiquette(45),

**new** InstructionStop("", ***CONTEXTE\_TESTS***));

pgmTest.ajouterLigne(**new** Etiquette(50),

**new** InstructionAffiche("\"foobar\"", ***CONTEXTE\_TESTS***));

System.***out***.println(pgmTest);

System.***out***.println("lancement du programme : ne doit pas "

+ "afficher foobar");

pgmTest.lancer();

System.***out***.println();

}

/\*\* Tests de toString() \*/

**public** **static** **void** testToString() {

**final** String ATTENDUE = "stop";

System.***out***.println("\tExécution du test de toString()");

**for** (InstructionStop valide : ***FIXTURE***)

*assertTrue*(valide.toString().compareTo(ATTENDUE) == 0);

}

}

## Classe InstructionVaen

### InstructionVaen.java

/\*\*

\* InstructionVaen.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.programmes.Etiquette;

/\*\*

\* Instruction qui transfère l'execution au numéro étiquette spécifié.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*

\*/

**public** **class** InstructionVaen **extends** Instruction {

/\*\* Etiquette à laquelle le programme doit se rendre \*/

**private** Etiquette etiquette;

/\*\*

\* Initialise un saut de ligne avec une étiquette en argument.

\* **@param** arguments Etiquette à laquelle le programme doit se rendre

\* **@param** contexte Contexte de la session de l'interpreteur LIR

\*/

**public** InstructionVaen(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String ERREUR\_ARG = "usage vaen <étiquette>";

**if** (arguments.isBlank()) {

**throw** **new** InterpreteurException(ERREUR\_ARG);

}

**this**.etiquette = **new** Etiquette(arguments);

}

/\* non javadoc -

\* @see interpreteurlir.motscles.instructions.Instruction#toString()

\* \*/

@Override

**public** String toString() {

**return** "vaen " + etiquette;

}

/\*\*

\* Execution de l'instruction :

\* Réalise un saut à l'étiquette spécifiée.

\* **@return** false car aucun feedback affiché directement

\* **@throws** RuntimeException si un programme n'est pas référencé en membre

\* de classe de Commande.

\*/

**public** **boolean** executer() {

**final** String ERREUR = "Le programme doit être référencé "

+ "dans la classe commande";

**if** (*programmeGlobal* == **null**) {

**throw** **new** RuntimeException(ERREUR);

}

*programmeGlobal*.vaen(etiquette);

**return** **false**;

}

}

### TestInstructionVaen.java

/\*\*

\* TestInstructionVaen.java 15 mai 2021

\* IUT-Rodez info1 2020-2021, pas de droits, pas de copyrights

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** interpreteurlir.motscles.Commande;

**import** interpreteurlir.motscles.instructions.InstructionAffiche;

**import** interpreteurlir.motscles.instructions.InstructionVaen;

**import** interpreteurlir.programmes.Etiquette;

**import** interpreteurlir.programmes.Programme;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

**import** **static** info1.outils.glg.Assertions.\*;

**import** info1.outils.glg.TestException;

/\*\*

\* Tests unitaires de {@link InstructionVaen}

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heia Dexter

\* **@author** Lucas Vabre

\*

\*/

**public** **class** TestInstructionVaen {

/\*\* Contexte global pour les tests \*/

**private** **final** Contexte CONTEXTE = **new** Contexte();

/\*\* Programme utilisé dans les tests \*/

**private** **final** Programme PROG\_REFERENCE = **new** Programme();

/\*\* Jeu de donnée d'InstructionRetour valides \*/

**private** **final** InstructionVaen[] FIXTURE = {

**new** InstructionVaen("10", CONTEXTE),

**new** InstructionVaen("9", CONTEXTE),

**new** InstructionVaen("20", CONTEXTE),

**new** InstructionVaen("70", CONTEXTE),

**new** InstructionVaen("40", CONTEXTE),

};

/\*\*

\* Tests unitaires de

\* {@link InstructionVaen#InstructionVaen(String, Contexte)}

\*/

**public** **void** testInstructionVaenStringContexte() {

System.***out***.println("\tExecution du test de "

+ "InstructionVaen#InstructionVaen(String, Contexte)");

**final** String[] ARGS\_INVALIDES = {

"greuuuuuu",

" motus 5800",

"100000",

"-4",

"$$$$£££"

};

Expression.*referencerContexte*(CONTEXTE);

/\* Cas invalides \*/

**for** (**int** i = 0; i < ARGS\_INVALIDES.length; i++) {

**try** {

**new** InstructionVaen(ARGS\_INVALIDES[i], CONTEXTE);

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

/\* Cas Valides \*/

**try** {

**new** InstructionVaen("10", CONTEXTE);

**new** InstructionVaen("9", CONTEXTE);

**new** InstructionVaen("20", CONTEXTE);

**new** InstructionVaen("70", CONTEXTE);

**new** InstructionVaen("40", CONTEXTE);

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

/\*\*

\* Tests unitaires de {@link InstructionVaen#toString()}

\*/

**public** **void** testToString() {

System.***out***.println("\tExecution du test de "

+ "InstructionVaen#toString()");

**final** String[] FORMAT\_ATTENDU = {

"vaen 10",

"vaen 9",

"vaen 20",

"vaen 70",

"vaen 40"

};

**for** (**int** i = 0 ; i < FORMAT\_ATTENDU.length ; i++) {

*assertEquivalence*(FORMAT\_ATTENDU[i], FIXTURE[i].toString());

}

}

/\*\*

\* Tests unitaires de {@link InstructionVaen#executer()}

\*/

**public** **void** testExecuter() {

System.***out***.println("\tExecution du test de "

+ "InstructionVaen#executer()");

/\* Cas invalide : où le programme global est vide \*/

**for**(InstructionVaen instruction : FIXTURE) {

**try** {

instruction.executer();

*echec*();

} **catch** (RuntimeException e) {

**if** (e **instanceof** TestException) {

*echec*();

}

// Test OK

}

}

/\* Cas valide \*/

Commande.*referencerProgramme*(PROG\_REFERENCE);

Expression.*referencerContexte*(CONTEXTE);

System.***out***.println("Test visuel : Ne doit pas afficher "

+ "les étiquettes (25, 31, 40 )");

/\* 1,10,13 -> 78, 89 (saute 25, 31, 40) \*/

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(1),

**new** InstructionAffiche("\"1 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(10),

**new** InstructionAffiche("\"10 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(13),

**new** InstructionAffiche("\"13 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(25),

**new** InstructionAffiche("\"25 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(31),

**new** InstructionAffiche("\"31 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(40),

**new** InstructionAffiche("\"40 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(78),

**new** InstructionAffiche("\"78 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(89),

**new** InstructionAffiche("\"89 \"", CONTEXTE));

PROG\_REFERENCE.ajouterLigne(**new** Etiquette(14),

**new** InstructionVaen("78", CONTEXTE));

PROG\_REFERENCE.lancer();

System.***out***.println();

}

}

## Classe InstructionVar

### InstructionVar.java

/\*\*

\* InstructionVar.java 9 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.expressions.Expression;

/\*\*

\* Instruction de déclaration et d'affectation de variables. La syntaxe de

\* cette expression est de la forme var identificateur = expression. Si

\* expression non renseignée, l'interpréteur affichera un message d'erreur ;

\* l'instruction doit effectuer systématiquement une affectation.

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** InstructionVar **extends** Instruction {

/\*\*

\* Initialise une instruction var à partir de arguments.

\* Le contexte sera modifié à l'exécution dde l'instruction.

\* **@param** arguments expression à exécuter

\* **@param** contexte global de l'interpréteur

\*/

**public** InstructionVar(String arguments, Contexte contexte) {

**super**(arguments, contexte);

**final** String USAGE = "usage var <identificateur> = <expression>";

**if** (arguments == **null** || arguments.isBlank()

|| Expression.*detecterCaractere*(arguments, '=') <= 0)

**throw** **new** InterpreteurException(USAGE);

aExecuter = Expression.*determinerTypeExpression*(arguments.trim());

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#executer()

\*/

@Override

**public** **boolean** executer() {

aExecuter.calculer();

**return** **false**;

}

/\*

\* Non - javadoc

\* @see interpreteurlir.motscles.instructions.Instruction#toString()

\*/

@Override

**public** String toString() {

**return** "var " + aExecuter;

}

}

### TestInstructionVar.java

/\*\*

\* TestInstructionVar.java 9 mai 2021

\* IUT info1 2020-2021, pas de copyright, aucun droit

\*/

**package** interpreteurlir.motscles.instructions.tests;

**import** **static** info1.outils.glg.Assertions.\*;

**import** interpreteurlir.Contexte;

**import** interpreteurlir.InterpreteurException;

**import** interpreteurlir.motscles.instructions.InstructionVar;

/\*\*

\* Tests unitaires de la classe InstructionVar

\*

\* **@author** Nicolas Caminade

\* **@author** Sylvan Courtiol

\* **@author** Pierre Debas

\* **@author** Heïa Dexter

\* **@author** Lucas Vabre

\*/

**public** **class** TestInstructionVar {

/\*\* jeu de données pour tests \*/

**public** **static** **final** String[] ***VALIDES*** = {

"$toto = $tata", "entier=2+2", "$coucou = $toto + \"titi\"",

"anneeNaissance = 1898"

};

/\*\*

\* Test unitaire de {@link InstructionVar#InstructionVar(String, Contexte)}

\*/

**public** **static** **void** testInstructionVar() {

**final** String[] EXPRESSIONS\_INVALIDES = {

"bonjour", "", "$toto $tata",

};

System.***out***.println("\tExécution du test de InstructionVar(String, "

+ "Contexte)");

**for** (String aTester : EXPRESSIONS\_INVALIDES) {

**try** {

**new** InstructionVar(aTester, **new** Contexte());

*echec*();

} **catch** (InterpreteurException lancee) {

// Test OK

}

}

**for** (String aTester : ***VALIDES***) {

**try** {

**new** InstructionVar(aTester, **new** Contexte());

} **catch** (InterpreteurException lancee) {

*echec*();

}

}

}

/\*\*

\* Test unitaire de {@link InstructionVar#toString()}

\*/

**public** **static** **void** testToString() {

**final** String[] CHAINES\_ATTENDUES = {

"var $toto = $tata",

"var entier = 2 + 2",

"var $coucou = $toto + \"titi\"",

"var anneeNaissance = 1898"

};

System.***out***.println("\tExécution du tes de toString()");

**for** (**int** i = 0 ; i < ***VALIDES***.length ; i++) {

InstructionVar aTester = **new** InstructionVar(***VALIDES***[i],

**new** Contexte());

*assertTrue*(CHAINES\_ATTENDUES[i].equals(aTester.toString()));

}

}

}