**Les tests unitaires**

**Avec JUnit**

# But

Mettre en pratique de la création de tests unitaires en Java au moyen de la bibliothèque JUnit.

Mettre en pratique le développement dirigé par les tests (TDD).

# Travail à affectuer

## La calculatrice

Vous devez implémenter une classe Calculatrice qui n’implémente pour l’instant qu’une méthode : addition().

Par contre, cette méthode prend en paramètre une liste d’entiers et doit gérer toutes les erreurs possibles, y compris vérifier que l’addition de tous les entiers de la liste ne dépasse pas la capacité d’un entier.



Implémentez cette classe selon les règles du développement dirigé par les tests :

* Implémenter une classe Calculatrice avec toutes ses méthodes publiques vides.
* Implémenter une suite de tests.
* Implémenter les méthodes de Calculatrice afin que la suite de tests de déroule avec succès.

**Calculatrice.java**

package calculatrice;

import java.util.List;

/\*\*

 \* Classe qui représente une calculatrice simplifiée.

 \* @author conujer

 \*/

public class Calculatrice {

 /\*\*

 \* Additionne tous les entiers de la liste donnée et retourne le résultat.

 \* @param liste Liste d'entiers.

 \* @return la somme des entiers de la liste.

 \*/

 public static int addition(List<Integer> liste) {

 if (liste == null)

 return 0;

 int somme = 0;

 for (int opérande : liste) {

 somme = Math.addExact(somme, opérande); // java 8, lance une exception ArithmeticException en cas d'overflow ou underflow

 }

 return somme;

 }

}

**CalculatriceTest.java**

/\*

 \* To change this license header, choose License Headers in Project Properties.

 \* To change this template file, choose Tools | Templates

 \* and open the template in the editor.

 \*/

package calculatrice;

import java.util.ArrayList;

import java.util.List;

import org.junit.Test;

import static org.junit.Assert.\*;

/\*\*

 \*

 \* @author conujer

 \*/

public class CalculatriceTest {

 public CalculatriceTest() {

 }

 @Test

 public void testAdditionListeVide() {

 List<Integer> liste = new ArrayList<>();

 assertEquals(0, Calculatrice.addition(liste));

 }

 @Test

 public void testAdditionListeNull() {

 assertEquals(0, Calculatrice.addition(null));

 }

 @Test

 public void testAdditionUnSeulElement() {

 List<Integer> liste = new ArrayList<>();

 liste.add(10);

 assertEquals(10, Calculatrice.addition(liste));

 }

 @Test

 public void testAdditionPlusieursElements() {

 List<Integer> liste = new ArrayList<>();

 liste.add(10);

 liste.add(20);

 liste.add(30);

 assertEquals(60, Calculatrice.addition(liste));

 }

 @Test

 public void testAdditionPlusieursElementsSigneDifférent() {

 List<Integer> liste = new ArrayList<>();

 liste.add(10);

 liste.add(20);

 liste.add(-15);

 assertEquals(15, Calculatrice.addition(liste));

 }

 @Test

 public void testAdditionOverflowIntPositif() {

 List<Integer> liste = new ArrayList<>();

 liste.add(Integer.MAX\_VALUE);

 liste.add(1);

 try {

 Calculatrice.addition(liste);

 fail("Exception attendue");

 } catch (ArithmeticException ex) {

 assertEquals("integer overflow", ex.getMessage());

 }

 }

 @Test

 public void testAdditionOverflowIntNegatif() {

 List<Integer> liste = new ArrayList<>();

 liste.add(Integer.MIN\_VALUE);

 liste.add(-1);

 try {

 Calculatrice.addition(liste);

 fail("Exception attendue");

 } catch (ArithmeticException ex) {

 assertEquals("integer overflow", ex.getMessage());

 }

 }

}

## La racine carrée

Friand de challenges, vous avez décidé d’implémenter votre propre classe mathématique, que vous allez appeler (par exemple) ZygoMath.

Vous vous êtes donné comme défi de n’utiliser au sein de cette classe que les opérations mathématiques élémentaires : addition, soustraction, multiplication, division et modulo.

À priori, cette classe ne contiendra que des méthodes statiques. Lorsqu’elle devra manipuler des nombres décimaux, elle se limitera aux nombres de type double.

Pour commencer, vous avez décidé de développer une méthode sqrt() qui calcule la racine carrée d’un nombre *x*.

Voici comment procéder :

Soient les deux suites :

$$A\_{n+1}=\frac{A\_{n}+G\_{n}}{2}$$

et

$$G\_{n+1}=\frac{2A\_{n}G\_{n}}{A\_{n}+G\_{n}}$$

avec *A0 = x* et *G0 = 1*.

La limite, si *n* tend vers l’infini, de ces deux suites, est $\sqrt{n}$.

Cette suite doit être calculée jusqu’à obtenir une précision relative $epsilon= 10^{-6}$.

Le critère d’arrêt est donc : $epsilon> \left|\frac{A\_{n}-G\_{n}}{G\_{n}}\right|$

Rappel : le symbole mathématique |n| désigne la valeur absolue du nombre n.

**Attention** :

Si le nombre n vaut +∞, la méthode doit retourner +∞.

Si le nombre n est négatif ou n’est pas un nombre (NaN, not a number), la méthode doit retourner NaN.

Pour implémenter cet algorithme, vous aurez donc besoin au préalable de créer une méthode abs(), qui retourne la valeur absolue d’un nombre.

Implémentez cette classe selon les règles du développement dirigé par les tests :

* Implémenter une classe ZygoMath avec toutes ses méthodes publiques statiques vides.
* Implémenter une suite de tests.
* Implémenter les méthodes de ZygoMath afin que la suite de tests de déroule avec succès.

À noter qu’avec la méthode assertEquals() de *JUnit* pour comparer deux nombres décimaux, il faut fournir en troisième argument le delta, c’est-à-dire la tolérance acceptée pour considérer deux nombres décimaux identiques.

Par exemple, un delta de 0 implique que les deux nombres à comparer doivent être parfaitement égaux. Par contre, un delta de 0.0001 implique que les deux nombres à comparer sont considérés égaux si les trois chiffres après la virgule sont identiques et si le quatrième chiffre ne diffère pas de plus d’une unité.

**JMath.java**

package custommath;

/\*\*

 \* Classe mathématique personnelle.

 \* @author conujer

 \*/

public class JMath {

 public static double EPSILON = 1.0E-6;

 /\*\*

 \* Calcule la valeur absolue d'un nombre.

 \* @param nbre Nombre dont il faut déterminer la valeur absolue.

 \* @return la valeur absolue du nombre donné en paramètre.

 \*/

 public static double abs(double nbre) {

 return nbre < 0 ? -nbre : nbre;

 }

 /\*\*

 \* Calcule la racine carrée d'un nombre.

 \* @param nbre Nombre dont il faut calculer la racine carrée.

 \* @return la racine carrée du nombre.

 \*/

 public static double sqrt(double nbre) {

 if (nbre < 0.0 || Double.isNaN(nbre))

 return Double.NaN;

 final double E = 1.0E-6;

 double a = nbre;

 double g = 1;

 do {

 double nextA = (a + g) / 2.0;

 g = (2.0 \* a \* g) / (a + g);

 a = nextA;

 } while (EPSILON <= abs((a-g)/g));

 return a;

 }

}

**JMathTest.java**

package custommath;

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

/\*\*

 \*

 \* @author conujer

 \*/

public class JMathTest {

 public JMathTest() {

 }

 /\*\*

 \* Test of abs method, of class JMath.

 \*/

 @Test

 public void testAbsPositif() {

 assertEquals(3.14, JMath.abs(3.14), 0.0);

 }

 @Test

 public void testAbsNegatif() {

 assertEquals(3.14, JMath.abs(-3.14), 0.0);

 }

 @Test

 public void testAbsZéro() {

 assertEquals(0, JMath.abs(0), 0.0);

 }

 @Test

 public void testAbsNegZéro() {

 assertEquals(0, JMath.abs(-0), 0.0);

 }

 @Test

 public void testAbsInfiniPositif() {

 assertEquals(Double.POSITIVE\_INFINITY, JMath.abs(Double.POSITIVE\_INFINITY), 0.0);

 }

 @Test

 public void testAbsInfiniNegatif() {

 assertEquals(Double.POSITIVE\_INFINITY, JMath.abs(Double.NEGATIVE\_INFINITY), 0.0);

 }

 @Test

 public void testAbsNan() {

 assertTrue(Double.isNaN(JMath.abs(Double.NaN)));

 }

 /\*\*

 \* Test of sqrt method, of class JMath.

 \*/

 @Test(timeout=1000)

 public void testSqrtPositifDecimal() {

 assertEquals(1.772004514666935, JMath.sqrt(3.14), JMath.EPSILON);

 }

 @Test(timeout=1000)

 public void testSqrtPositifEntier() {

 assertEquals(5.0, JMath.sqrt(25), JMath.EPSILON);

 }

 @Test(timeout=1000)

 public void testSqrtZero() {

 assertEquals(0, JMath.sqrt(0), JMath.EPSILON);

 }

 @Test(timeout=1000)

 public void testSqrtUn() {

 assertEquals(1, JMath.sqrt(1), JMath.EPSILON);

 }

 @Test(timeout=1000)

 public void testSqrtNegtif() {

 assertTrue(Double.isNaN(JMath.sqrt(-4)));

 }

 @Test(timeout=1000)

 public void testSqrtNan() {

 assertTrue(Double.isNaN(JMath.sqrt(Double.NaN)));

}

 @Test(timeout=1000)

 public void testSqrtInfiniPositif() {

 assertEquals(Double.POSITIVE\_INFINITY, JMath.sqrt(Double.POSITIVE\_INFINITY), JMath.EPSILON);

 }

 @Test(timeout=1000)

 public void testSqrtInfiniNegatif() {

 assertTrue(Double.isNaN(JMath.sqrt(Double.NEGATIVE\_INFINITY)));

 }

}

## La pile

### Description

Une pile (stack en anglais) est une structure de donnée permettant de stocker des informations sur le principe du dernier arrivé, premier sorti.

Le fonctionnement d’une pile est souvent comparé à une pile d’assiettes : on empile les assiettes les unes sur les autres. La dernière assiette mise sur la pile sera la première à être sortie de la pile.

Un développeur avait commencé le développement d’une implémentation d’une pile, selon le diagramme de classes suivant :



Il avait rapidement programmé une ébauche, qu’il n’avait jamais vraiment terminée, ni correctement testée.

De plus, il avait été convenu que dans un premier temps, il n’y aurait pas de gestion d’exceptions. Si l’utilisateur tentait d’accéder à un élément inexistant, une valeur null est simplement retournée.

Voici le code qu’il avait écrit :

package stack;

/\*\*

 \* Gère une pile (type LIFO) générique.

 \* @author Némard Jean

 \*/

public class Stack<T> {

 private class Node<T> {

 public T value;

 public Node bottomNode;

 public Node(T value) {

 this.value = value;

 }

 }

 // Mémorise le sommet de la pile

 private Node top = null;

 /\*\*

 \* Ajoute l'élément donné sur le sommet de la pile.

 \* @param value Elément à ajouter sur la pile.

 \*/

 public void push(T value) {

 Node newNode = new Node(value);

 top.bottomNode = newNode;

 top = newNode;

 }

 /\*\*

 \* Retire l'élément au sommet de la pile et le retourne.

 \* @return l'élément au sommet de la pile.

 \*/

 public T pop() {

 top = top.bottomNode;

 return top.value;

 }

 /\*\*

 \* Atteint l'élément au sommet de la pile et l'y laisse.

 \* @return l'élément au sommet de la pile.

 \*/

 public T peak() {

 return top.value;

 }

 /\*\*

 \* @return un booléen qui indique si la pile est vide ou non.

 \*/

 public boolean isEmpty() {

 return top == null;

 }

}

### Suite de tests

Vous êtes chargé dans un premier temps d’écrire des tests unitaires pour valider le fonctionnement de cette pile.

Pour cela, il faut commencer par remplir le tableau ci-contre, puis d’implémenter ces tests avec *JUnit*.

Lorsque tous les tests sont implémentés, corriger les éventuelles erreurs de codage de la classe Stack.

|  |  |
| --- | --- |
| **Test à réaliser** | **Résultat attendu si le test fonctionne** |
| Création d’une pile | La pile est vide. |
| Ajout d’un élément sur une pile vide | L’élément se trouve au sommet de la pile, la pile n’est pas vide. |
| Ajout d’un élément sur une pile déjà remplie | L’élément se trouve au sommet de la pile, la pile n’est pas vide. |
| Ajout d’un élément sur une pile à laquelle un élément vient d’être enlevé | L’élément se trouve au sommet de la pile, la pile n’est pas vide. |
| Ajout d’un élément sur une pile qui est devenue vide suite à un dépilement. | L’élément se trouve au sommet de la pile, la pile n’est pas vide. |
| Dépilement d’un élément sur une pile vide | Un élément null est retourné. La pile est vide. |
| Dépilement du seul élément de la pile | L’élément est correctement retourné, la pile est vide. |
| Dépilement d’un élément d’une pile remplie | L’élément est correctement retourné, la pile n’est pas vide. |
| Dépilement de tous les éléments d’une pile remplie | Les éléments sont correctement retournés, la pile est vide. |
| Atteint un élément d’une pile vide | Un élément null est retourné. La pile est vide. |
| Atteint le seul élément d’une pile | L’élément est correctement retourné, la pile n’est pas vide. L’élément au sommet de la pile est inchangé. |
| Atteint l’élément au sommet d’une pile remplie | L’élément est correctement retourné, la pile n’est pas vide. L’élément au sommet de la pile est inchangé. |
| Atteint l’élément au sommet d’une pile auquel ona retiré un élément | L’élément au sommet de la pile est correctement retourné. |
| Vérifie qu’une pile vide est bien vide | La pile est vide. |
| Vérifie qu’une pile pas vide n’est pas vide | La pile n’est pas vide. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Stack.java**

package stack;

/\*\*

 \* Gère une pile (type LIFO) générique.

 \* @author conujer

 \*/

public class Stack<T> {

 private class Node {

 public T value;

 public Node bottomNode;

 public Node(T value) {

 this.value = value;

 }

 }

 // Mémorise le sommet de la pile

 private Node top = null;

 /\*\*

 \* Ajoute l'élément donné sur le sommet de la pile.

 \* @param value Elément à ajouter sur la pile.

 \*/

 public void push(T value) {

 Node newNode = new Node(value);

 newNode.bottomNode = top;

 top = newNode;

 }

 /\*\*

 \* Retire l'élément au sommet de la pile et le retourne.

 \* @return l'élément au sommet de la pile.

 \*/

 public T pop() {

 if (top == null)

 return null;

 T val = top.value;

 top = top.bottomNode;

 return val;

 }

 /\*\*

 \* Récupère l'élément au sommet de la pile et l'y laisse.

 \* @return l'élément au sommet de la pile.

 \*/

 public T peak() {

 if (top == null)

 return null;

 return top.value;

 }

 /\*\*

 \* @return un booléen qui indique si la pile est vide ou non.

 \*/

 public boolean isEmpty() {

 return top == null;

 }

}

**StackTest.java**

package stack;

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

/\*\*

 \*

 \* @author conujer

 \*/

public class StackTest {

 public StackTest() {

 }

 /\*\*

 \* Test la création d'un stack

 \*/

 @Test

 public void testCréerStack() {

 Stack<Integer> myStack = new Stack<>();

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of push method, of class Stack.

 \*/

 @Test

 public void testPushSurStackVide() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertEquals(10, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of push method, of class Stack.

 \*/

 @Test

 public void testPushSurStackRempli() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 assertEquals(20, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of push method, of class Stack.

 \*/

 @Test

 public void testPushAprèsPop() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 myStack.pop();

 myStack.push(30);

 assertEquals(30, myStack.pop().intValue());

 assertEquals(10, myStack.pop().intValue());

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopSurStackVide() {

 Stack<Integer> myStack = new Stack<>();

 assertTrue(myStack.isEmpty());

 assertNull(myStack.pop());

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopSurStackUnSeulElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertEquals(10, myStack.pop().intValue());

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPushSurStackVideSuiteAPop() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.pop();

 myStack.push(20);

 assertEquals(20, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopSurStackPlusieursElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 assertEquals(20, myStack.pop().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopAllSurStackPlusieursElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 assertEquals(20, myStack.pop().intValue());

 assertEquals(10, myStack.pop().intValue());

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackVide() {

 Stack<Integer> myStack = new Stack<>();

 assertNull(myStack.peak());

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackUnElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertEquals(10, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackPlusieursElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 myStack.push(30);

 assertEquals(30, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackAprèsPop() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 myStack.pop();

 assertEquals(10, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of isEmpty method, of class Stack.

 \*/

 @Test

 public void testIsEmptyStackVide() {

 Stack<Integer> myStack = new Stack<>();

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of isEmpty method, of class Stack.

 \*/

 @Test

 public void testIsEmptyStackRempli() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertFalse(myStack.isEmpty());

 }

}

### Améliorer la pile

Si la classe Stack est complètement fonctionnelle et que les tests passent tous avec succès, il faut l’améliorer en implémentant les fonctionnalités suivantes :

* Tenir compte d’une taille maximum pour la pile. Par défaut, cette taille est de 127 éléments. Elle est spécifiée au moment de la construction de la pile et ne peut plus être changée.
* Ajouter une méthode count() permettant de connaître le nombre d’élément dans la pile.
* Mettre en place les exceptions implicites suivantes :
	1. Une exception lorsqu’on tente d’accéder à un élément sur une pile vide
	2. Une exception lorsqu’on dépasse la capacité de la pile

Ces exceptions seront des spécialisations d’une exception de base spécifique à la pile.

**Attention :** Avant d’implémenter ces améliorations, mettez à jour vos tests unitaires !

**Stack.java**

package stack;

/\*\*

 \* Gère une pile (type LIFO) générique.

 \* @author conujer

 \*/

public class Stack<T> {

 private class Node {

 public T value;

 public Node bottomNode;

 public Node(T value) {

 this.value = value;

 }

 }

 // Mémorise le sommet de la pile

 private Node top = null;

 private int count = 0;

 private int maxSize = 127;

 /\*\*

 \* Construit une pile de taille max par défaut.

 \*/

 public Stack() {

 }

 /\*\*

 \* Construit une pile de tailel max donnée.

 \* @param maxSize Taille maximale de la pile.

 \*/

 public Stack(int maxSize) {

 this.maxSize = maxSize;

 }

 /\*\*

 \* Ajoute l'élément donné sur le sommet de la pile.

 \* @param value Elément à ajouter sur la pile.

 \*/

 public void push(T value) {

 if (count == maxSize) {

 throw new StackOverflowException(maxSize);

 }

 Node newNode = new Node(value);

 newNode.bottomNode = top;

 top = newNode;

 count++;

 }

 /\*\*

 \* Retire l'élément au sommet de la pile et le retourne.

 \* @return l'élément au sommet de la pile.

 \*/

 public T pop() {

 if (top == null)

 throw new StackEmptyException();

 count--;

 T value = top.value;

 top = top.bottomNode;

 return value;

 }

 /\*\*

 \* Récupère l'élément au sommet de la pile et l'y laisse.

 \* @return l'élément au sommet de la pile.

 \*/

 public T peak() {

 if (top == null)

 throw new StackEmptyException();

 return top.value;

 }

 /\*\*

 \* @return un booléen qui indique si la pile est vide ou non.

 \*/

 public boolean isEmpty() {

 return top == null;

 }

 /\*\*

 \* @return le nombre d'éléments dans cette pile.

 \*/

 public int count() {

 return count;

 }

}

**StackException.java**

package stack;

/\*\*

 \* Exception concernant la pile.

 \* @author conujer

 \*/

public class StackException extends RuntimeException{

 public StackException() {

 }

 public StackException(String message) {

 super(message);

 }

}

**StackEmptyException.java**

package stack;

/\*\*

 \* Exception d'accès à un élément de la pile vide.

 \* @author conujer

 \*/

public class StackEmptyException extends StackException {

 public StackEmptyException() {

 super("Stack empty.");

 }

}

**StackOverflowException.java**

package stack;

/\*\*

 \* Exception de dépassement de la capacité de la pile.

 \* @author conujer

 \*/

public class StackOverflowException extends StackException {

 public final int stackSize;

 public StackOverflowException(int stackSize) {

 super("Stack overflow. Stack size : " + stackSize);

 this.stackSize = stackSize;

 }

}

**StackTest.java**

package stack;

import org.junit.After;

import org.junit.AfterClass;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

/\*\*

 \*

 \* @author conujer

 \*/

public class StackTest {

 public StackTest() {

 }

 @BeforeClass

 public static void setUpClass() {

 }

 @AfterClass

 public static void tearDownClass() {

 }

 @Before

 public void setUp() {

 }

 @After

 public void tearDown() {

 }

 /\*\*

 \* Test la création d'un stack

 \*/

 @Test

 public void testCréerStack() {

 Stack<Integer> myStack = new Stack<>();

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of push method, of class Stack.

 \*/

 @Test

 public void testPushSurStackVide() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertEquals(10, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of push method, of class Stack.

 \*/

 @Test

 public void testPushSurStackRempli() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 assertEquals(20, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of push method, of class Stack.

 \*/

 @Test

 public void testPushAprèsPop() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 myStack.pop();

 myStack.push(30);

 assertEquals(30, myStack.pop().intValue());

 assertEquals(10, myStack.pop().intValue());

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopSurStackVide() {

 Stack<Integer> myStack = new Stack<>();

 assertTrue(myStack.isEmpty());

 try {

 assertNull(myStack.pop());

 fail("Exception attentue");

 } catch (StackEmptyException ex) {

 assertTrue(myStack.isEmpty());

 }

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopSurStackUnSeulElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertEquals(10, myStack.pop().intValue());

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPushSurStackVideSuiteAPop() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.pop();

 myStack.push(20);

 assertEquals(20, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopSurStackPlusieursElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 assertEquals(20, myStack.pop().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of pop method, of class Stack.

 \*/

 @Test

 public void testPopAllSurStackPlusieursElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 assertEquals(20, myStack.pop().intValue());

 assertEquals(10, myStack.pop().intValue());

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackVide() {

 Stack<Integer> myStack = new Stack<>();

 try {

 assertNull(myStack.peak());

 fail("Exception attendue");

 } catch (StackEmptyException ex) {

 assertTrue(myStack.isEmpty());

 }

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackUnElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertEquals(10, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackPlusieursElement() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 myStack.push(30);

 assertEquals(30, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of peak method, of class Stack.

 \*/

 @Test

 public void testPeakStackAprèsPop() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 myStack.push(20);

 myStack.pop();

 assertEquals(10, myStack.peak().intValue());

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of isEmpty method, of class Stack.

 \*/

 @Test

 public void testIsEmptyStackVide() {

 Stack<Integer> myStack = new Stack<>();

 assertTrue(myStack.isEmpty());

 }

 /\*\*

 \* Test of isEmpty method, of class Stack.

 \*/

 @Test

 public void testIsEmptyStackRempli() {

 Stack<Integer> myStack = new Stack<>();

 myStack.push(10);

 assertFalse(myStack.isEmpty());

 }

 /\*\*

 \* Test of count method, of class Stack.

 \*/

 @Test

 public void testCount() {

 Stack<Integer> myStack = new Stack<>();

 assertEquals(0, myStack.count());

 myStack.push(10);

 assertEquals(1, myStack.count());

 myStack.push(20);

 assertEquals(2, myStack.count());

 myStack.pop();

 assertEquals(1, myStack.count());

 myStack.peak();

 assertEquals(1, myStack.count());

 myStack.pop();

 assertEquals(0, myStack.count());

 }

 /\*\*

 \* Test of push method, of class Stack.

 \*/

 @Test

 public void testPushSurStackDeTaillePleine() {

 Stack<Integer> myStack = new Stack<>(5);

 myStack.push(10);

 myStack.push(20);

 myStack.push(30);

 myStack.push(40);

 myStack.push(50);

 try {

 myStack.push(60);

 fail("Exception attendue");

 } catch (StackOverflowException ex) {

 assertEquals(5, ex.stackSize);

 assertEquals(5, myStack.count());

 assertEquals(50, myStack.peak().intValue());

 }

 }

}