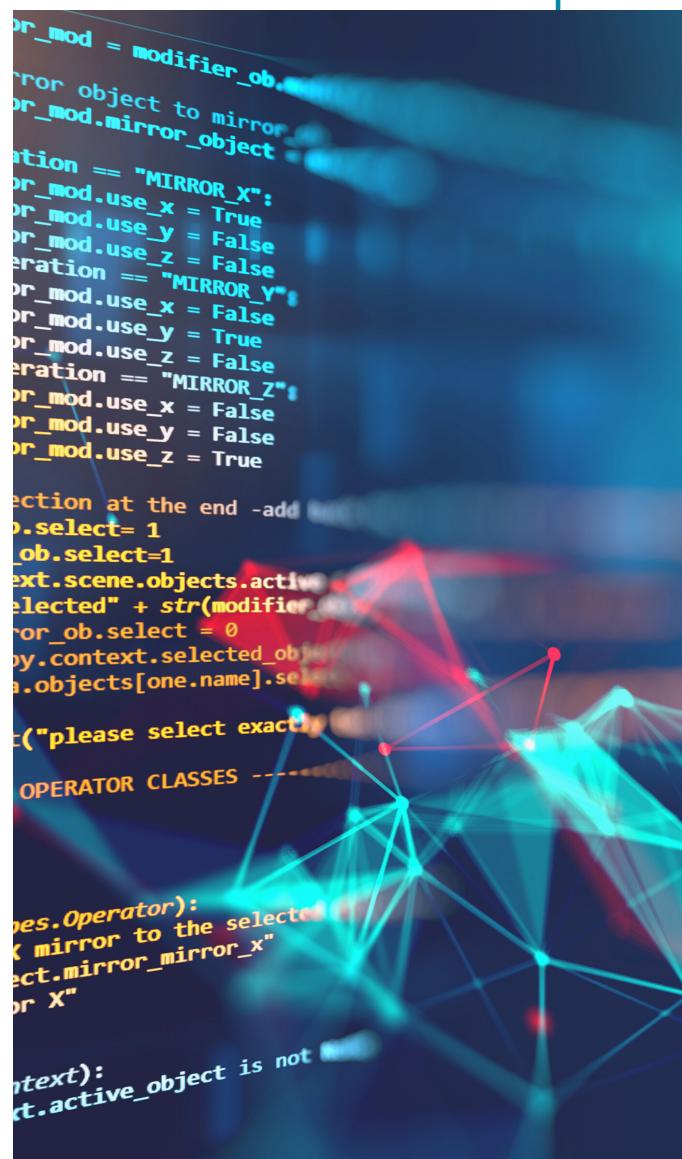


MARS 2020

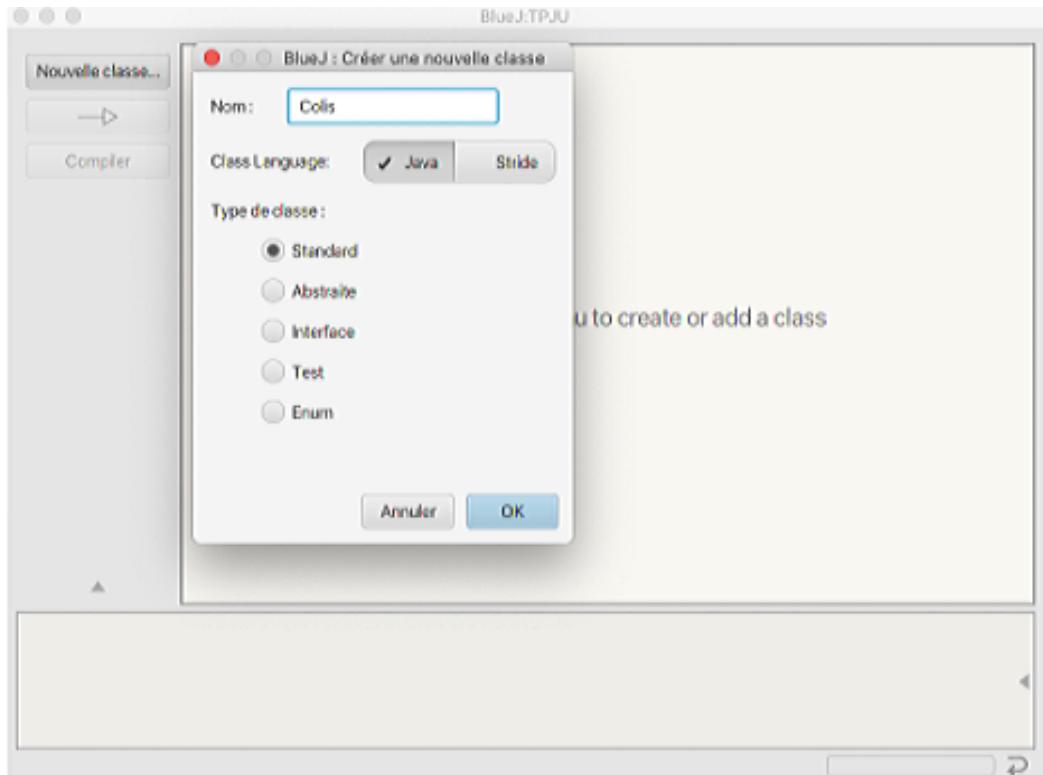
TUTORIAL CONCEPT ORIENTE OBJET

Julienne ISHA &
Marie VOUILLOT



Nous allons suivre un tutoriel pour comprendre ensemble la programmation orientée objets.

Le Colis et le Bon de livraison seront nos deux exemples pour une meilleure compréhension.



On commence par créer notre classe fétiche qui "fabriquera" des colis

```
public class Colis
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre

    private double poids;
    private String adresse;
    private String destinataire;

    /**
     * Constructeur d'objets de classe Colis
     */
    public Colis(String adresse, String destinataire)
    {
        // initialisation des variables
        this.adresse=adresse;
        poids=0.1;
        this.destinataire=destinataire;

        /**
         * @un getter pour le poids
         */
        public double getPoids()
        {
            return poids;
        }

        /**
         * @un getter pour l'adresse
         */
        public String getAdresse()
        {
            return adresse;
        }

        /**
         * @un getter pour le destinataire
         */
        public String getDestinataire()
        {
            return destinataire;
        }

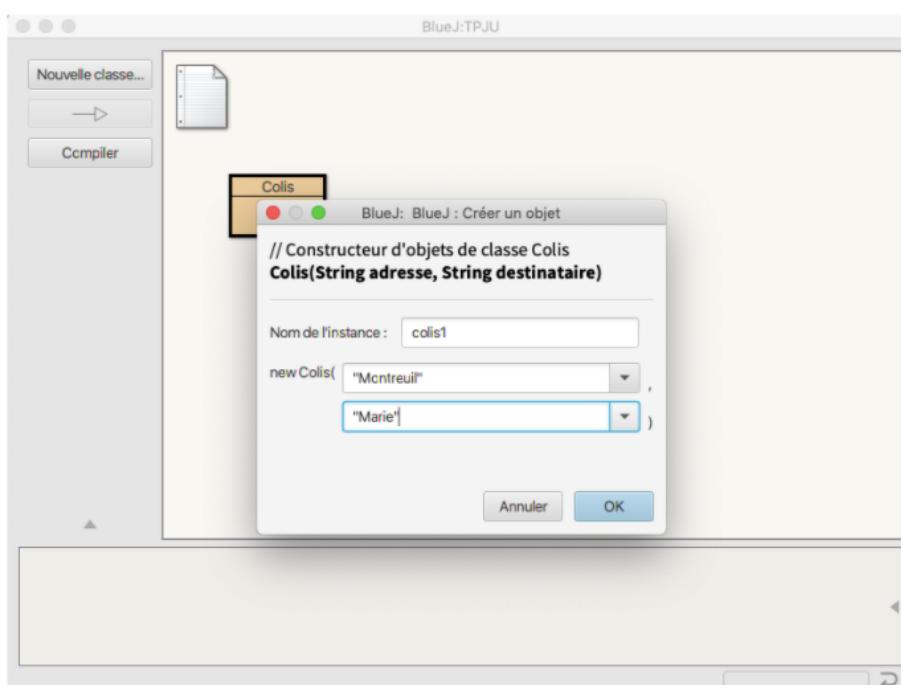
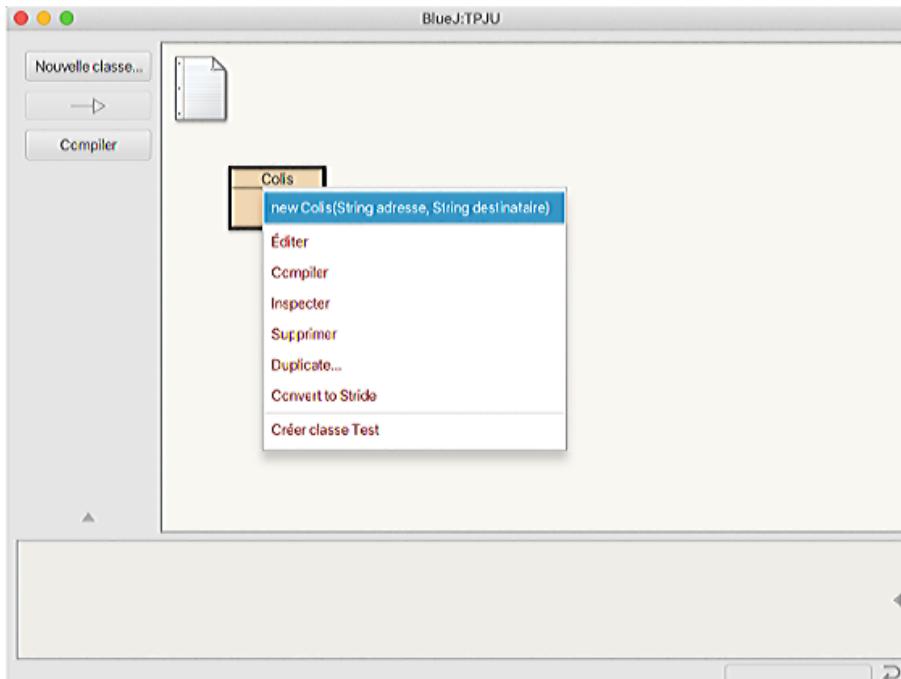
        /**
         * Méthode pour ajouter de la charge dans le colis
         */
        public double chargementColis( double charge)
        {
            poids+=charge;
            return poids;
        }
    }
}
```

Les colis auront chacun un poids, une adresse de livraison ainsi qu'un destinataire.

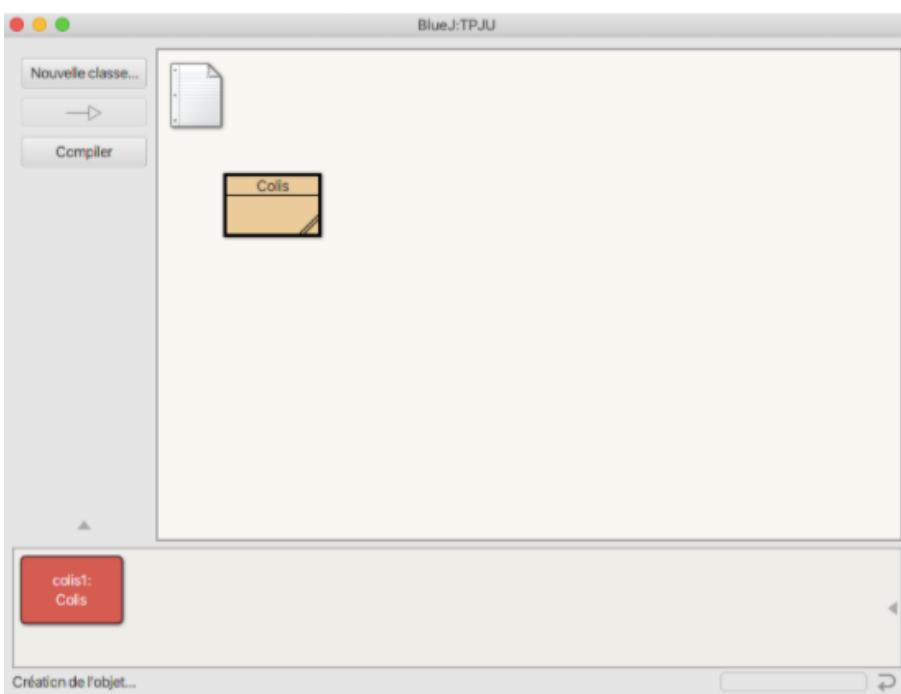
Le constructeur permet de "fabriquer" les colis. Ici on voit que l'adresse et le destinataire seront saisis par l'utilisateur mais le poids sera forcément de 0,100g lors de la création.

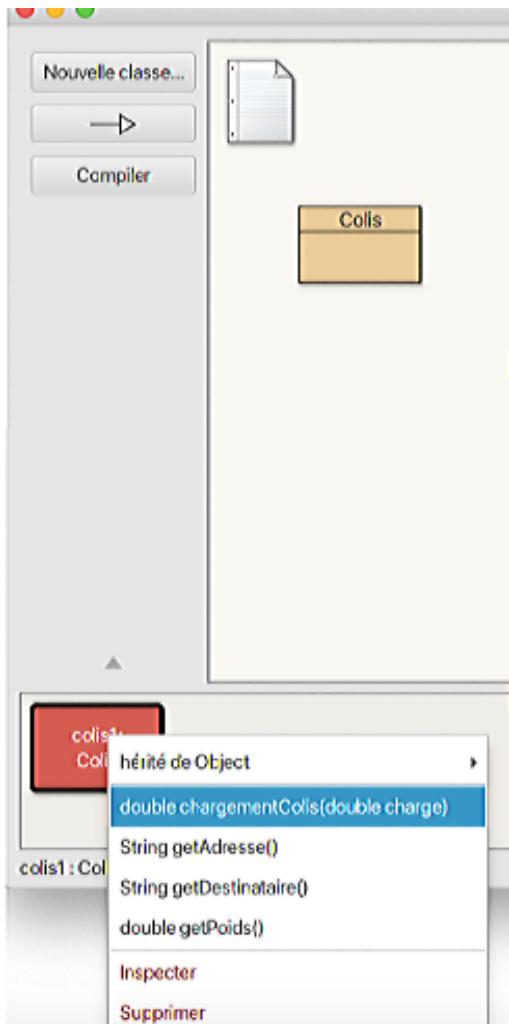
il y a également des méthodes pour récupérer les valeurs des variables.

Enfin on a une méthode chargementColis, qui prend en attribut une charge et qui l'ajoute au poids du colis. Cette méthode représente l'action de remplir un colis.

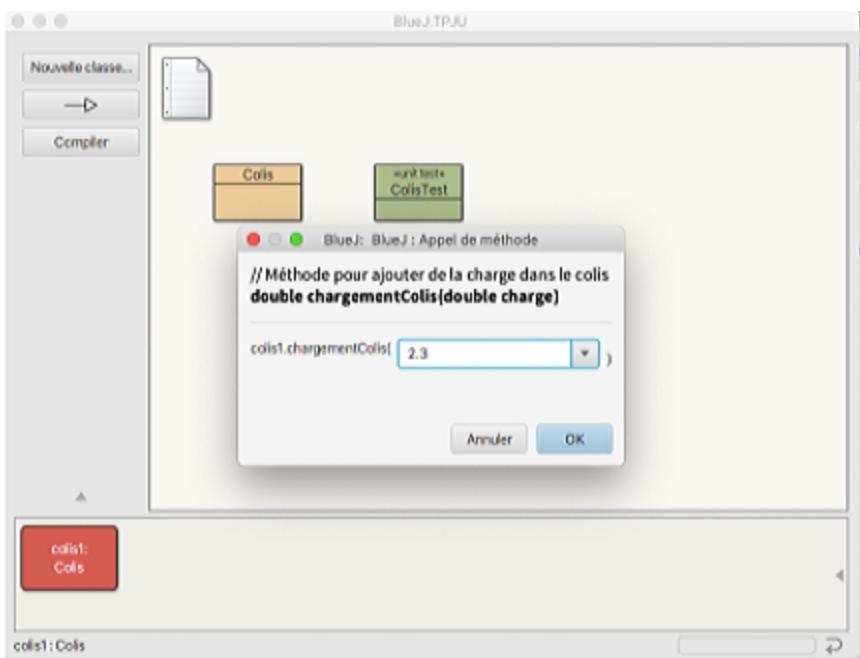


ici, on crée une instance de notre classe Colis.

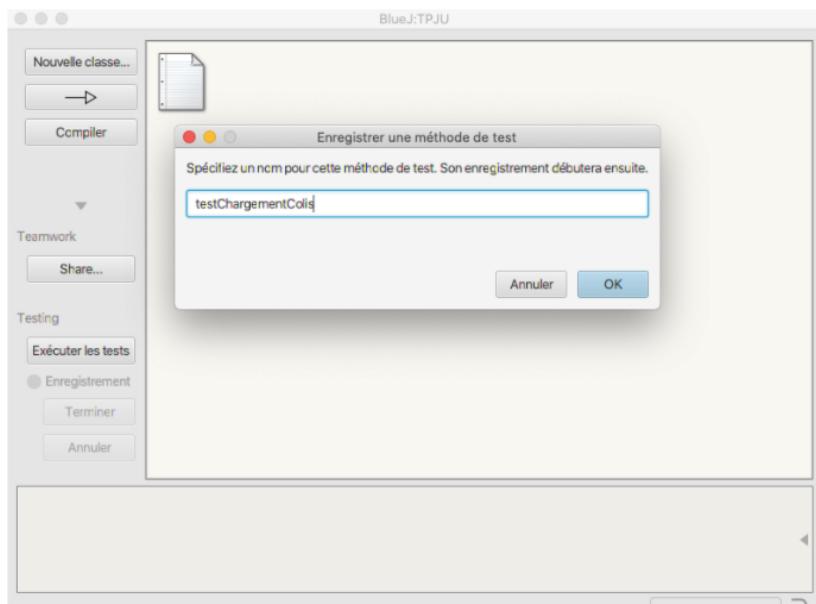
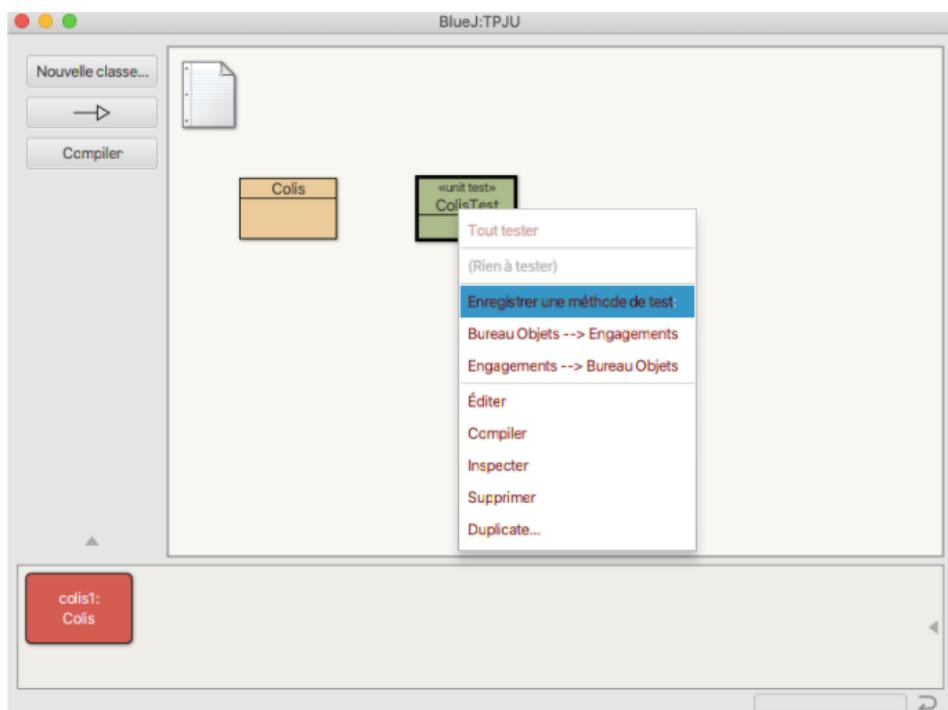
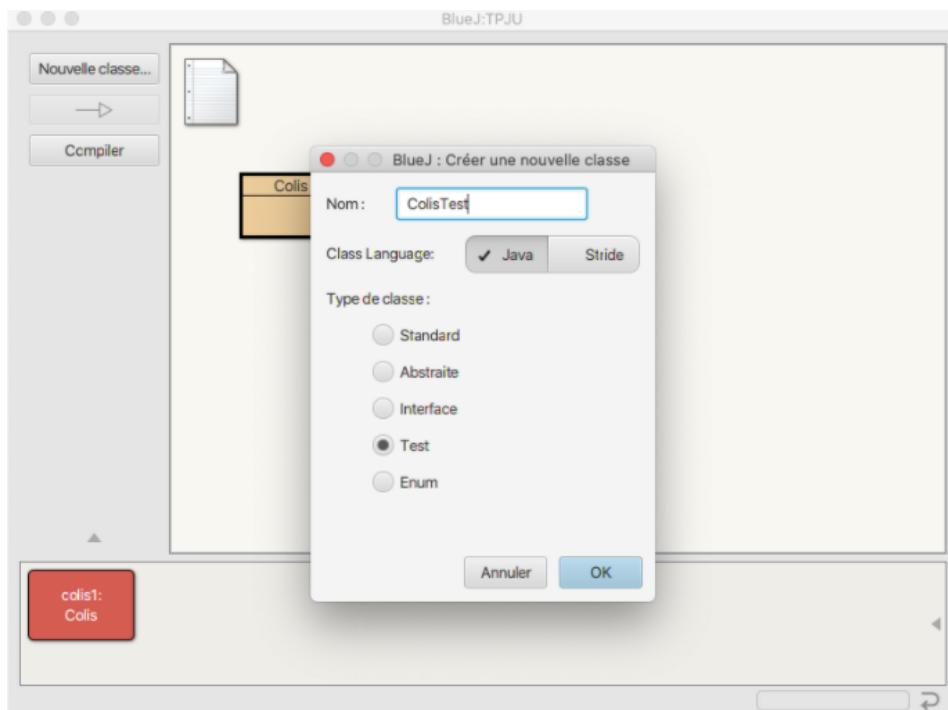




On ajoute de la charge dans le colis avec la méthode chargementColis()

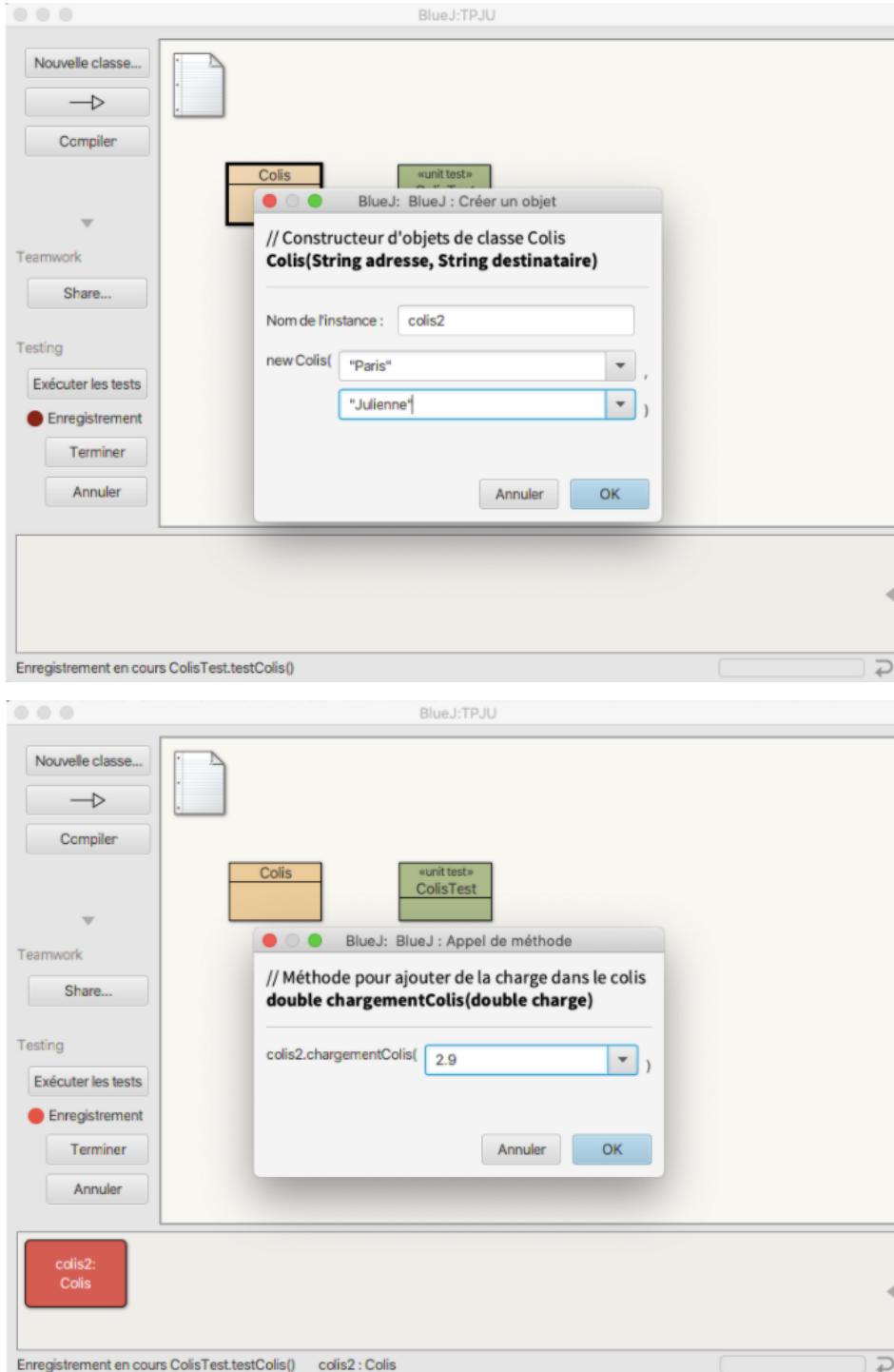


Ici par exemple on ajoute 2,3 kg dans notre colis.

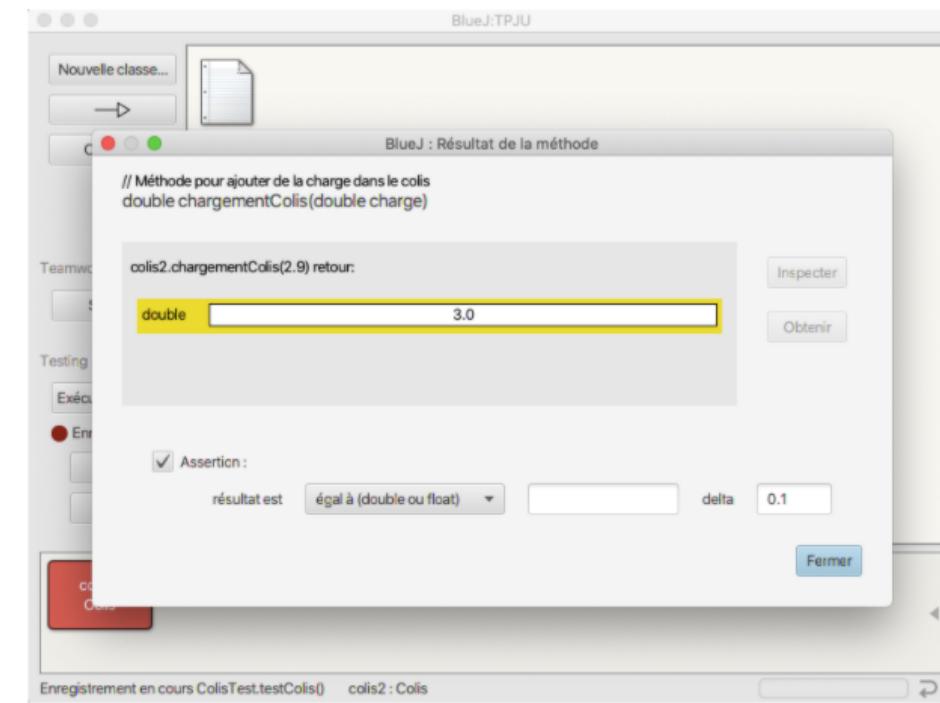


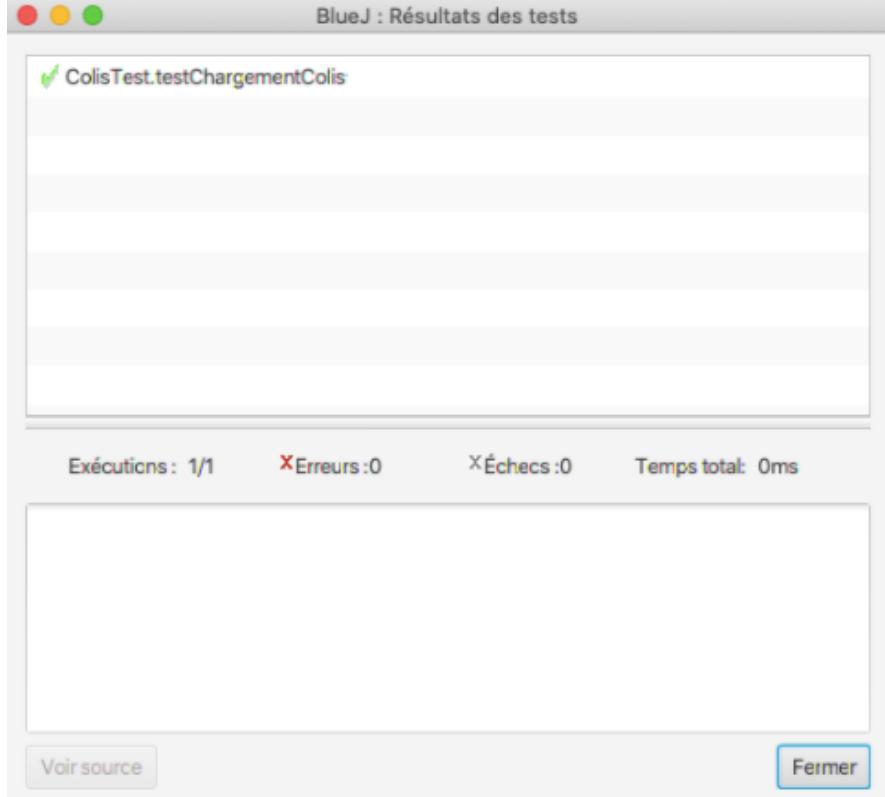
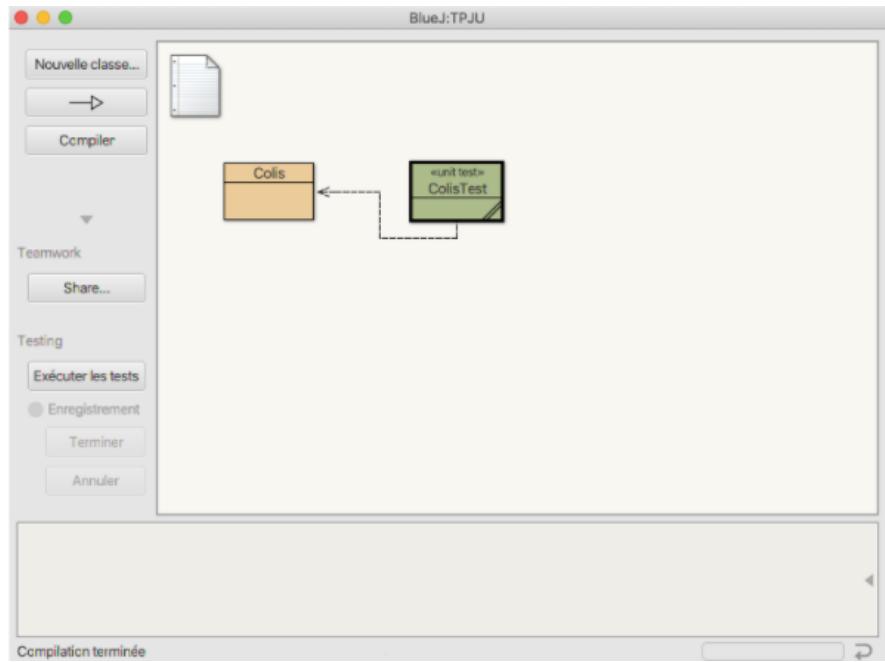
On crée une classe de test pour vérifier que notre méthode de chargement renvoie le résultat que l'on attend.

Pour cela on va refaire le processus de création d'un colis et de son chargement et le test sera enregistré tout le long.



Ici il faut entrer le poids que l'on est censé trouver suite à l'ajout d'une charge





Une fois l'enregistrement du test terminé, on peut exécuter le test
On peut voir que le test a fonctionné. La fonction ChargementColis fonctionne donc bien.

```

/*
public class bonLivraison
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private double prix;
    public Colis c;

    /**
     * Constructeur d'objets de classe bonLivraison
     */
    public bonLivraison()
    {
        // initialisation des variables d'instance
        prix = 0;
    }

    public void setColis(Colis c){
        this.c=c;
    }
    public Colis getColis() {
        return this.c;
    }

    /**
     * @un getter pour le prix
     */
    public double getPrix()
    {
        return prix;
    }

    /**
     * @un getter pour le poids
     */
    public double getPoids()
    {
        double poids=this.c.getPoids();
        return poids;
    }

    /**
     * Un exemple de méthode - remplacez ce commentaire par le vôtre
     *
     * @param y    le paramètre de la méthode
     * @return      la somme de x et de y
     */
    public double calculPrix()
    {
        prix=c.getPoids()*10.5;
        return prix;
    }
}

```

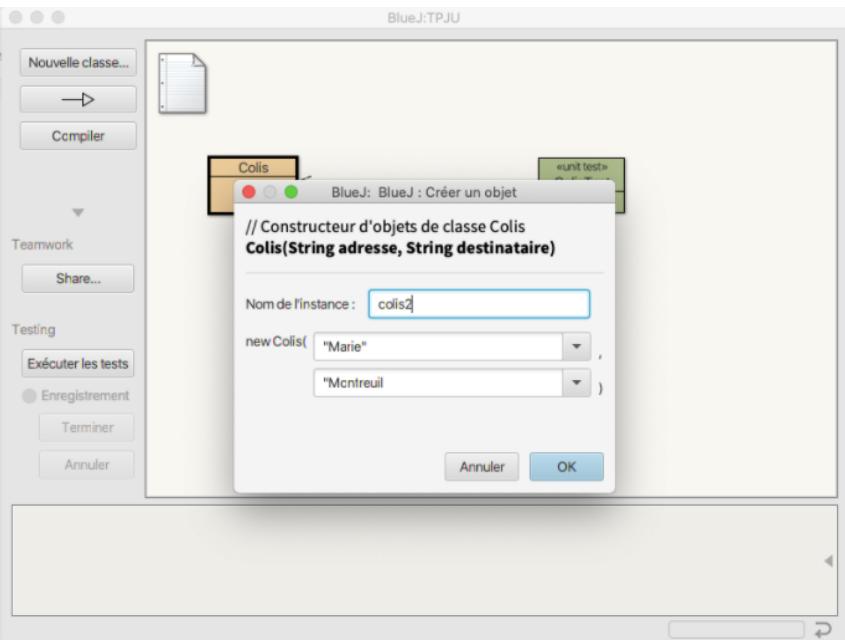
la classe compilée - pas d'erreur de syntaxe

enregistré

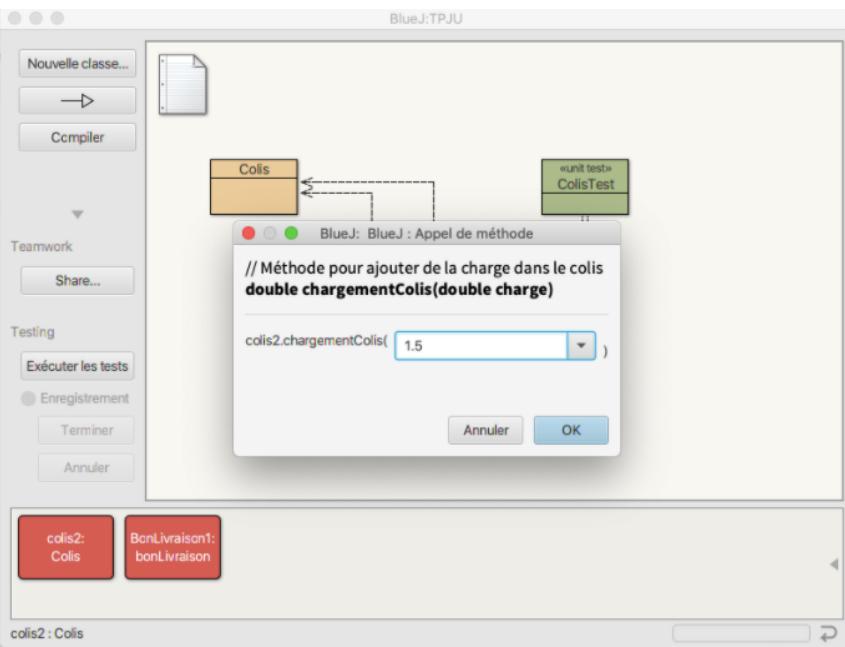
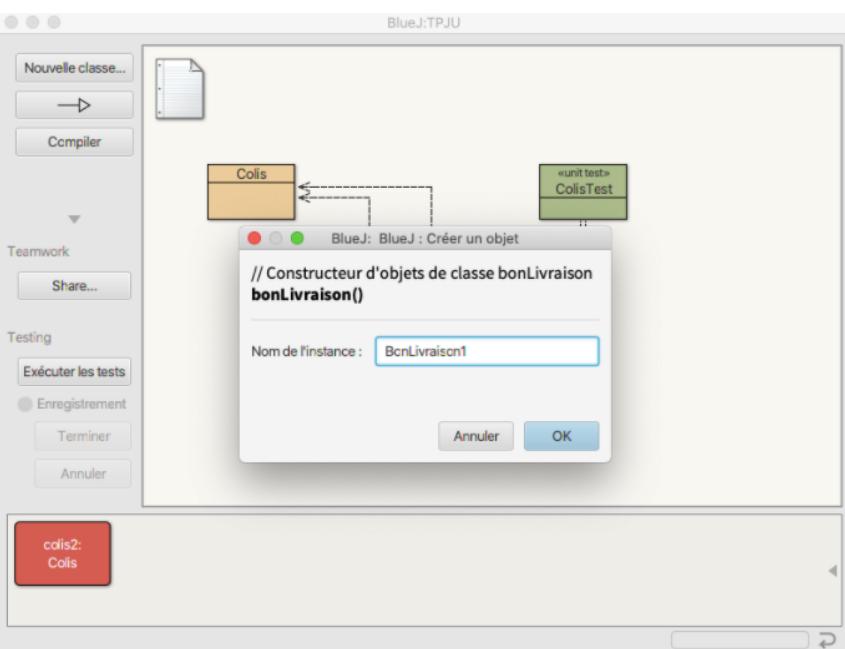
On crée maintenant une classe "bonLivraison". Sur le bon de livraison on aura le prix de la livraison, et il sera "lié" à un seul colis.

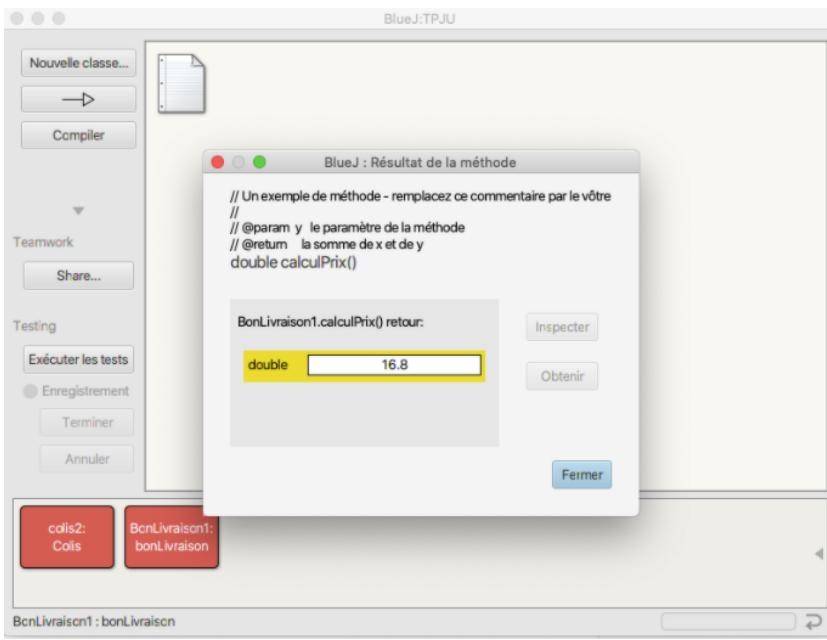
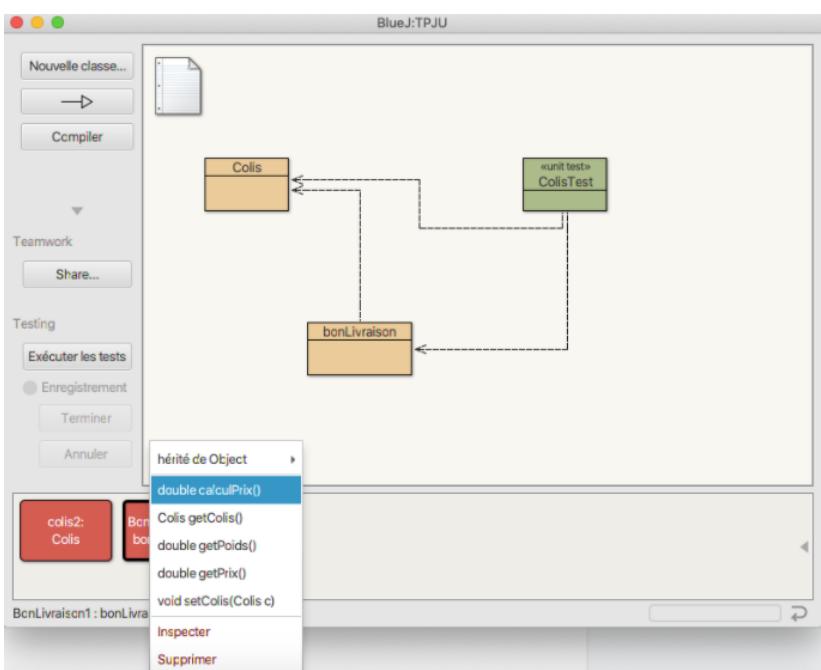
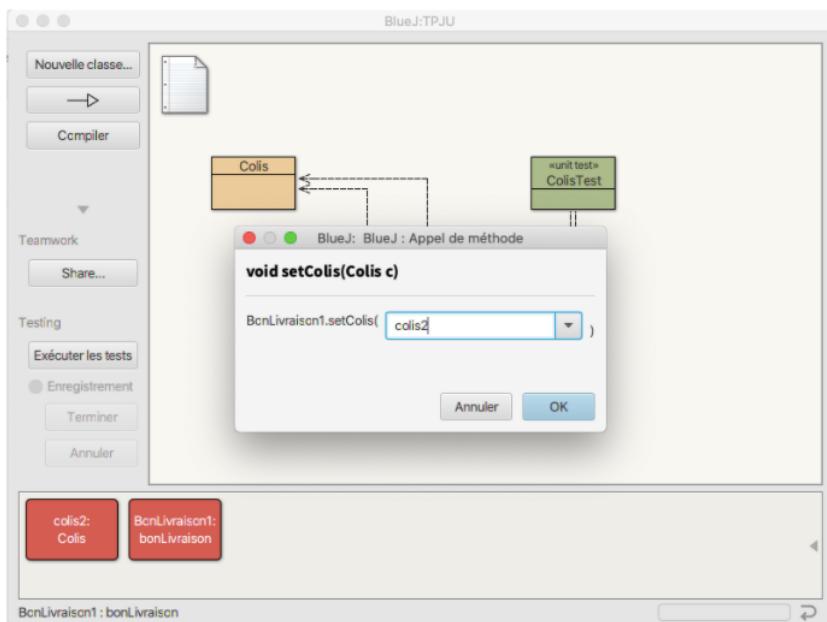
La méthode setColis permet d'attribuer un colis à un bon de livraison et la méthode getPoids() permettra de récupérer son poids par exemple.

Enfin la méthode calculPrix() permettra de calculer le prix de la livraison en fonction du poids du colis. Ici, 1kg coûtera 10,50€.

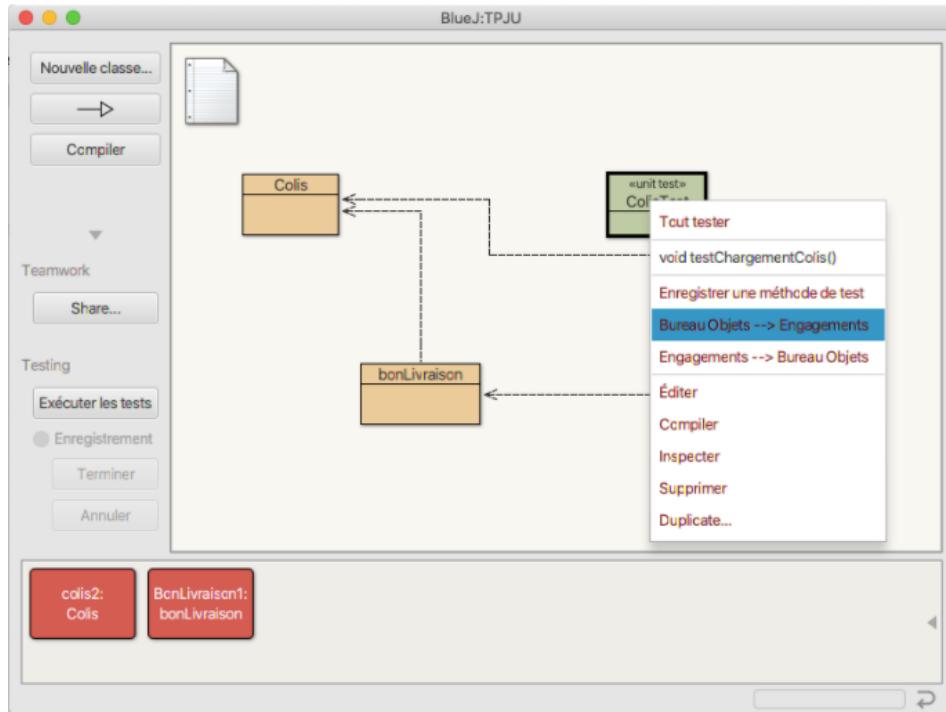


Ici, on va créer un colis, le remplir, puis créer le bon de livraison correspondant (avec le prix de livraison calculé grâce à la méthode `calculPrix`).





**Par exemple ici le
prix sera de 16,80€
pour un colis de
1,6kg.**



On ajoute ces objets à la fixture de la classe de test (dans la méthode `setUp()` de la classe `testColis`).

Pourquoi faire ça ?

Lorsqu'on lancera un test, ces objets seront automatiquement créés afin que le test puisse se dérouler.

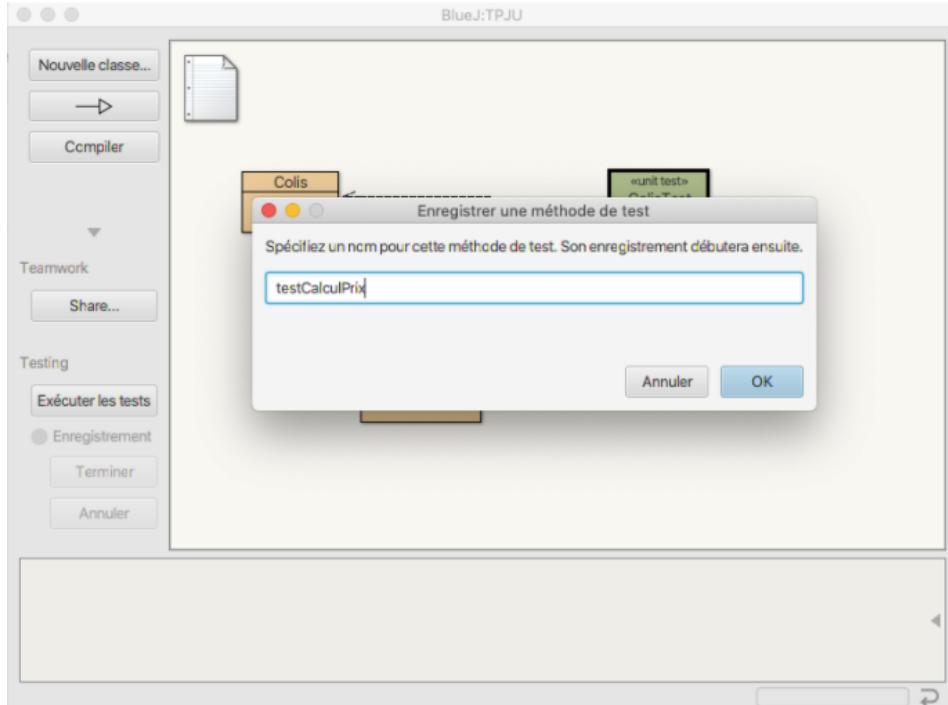
The screenshot shows the BlueJ IDE with the title 'ColisTest - TPJU'. The code editor contains the following Java code:

```

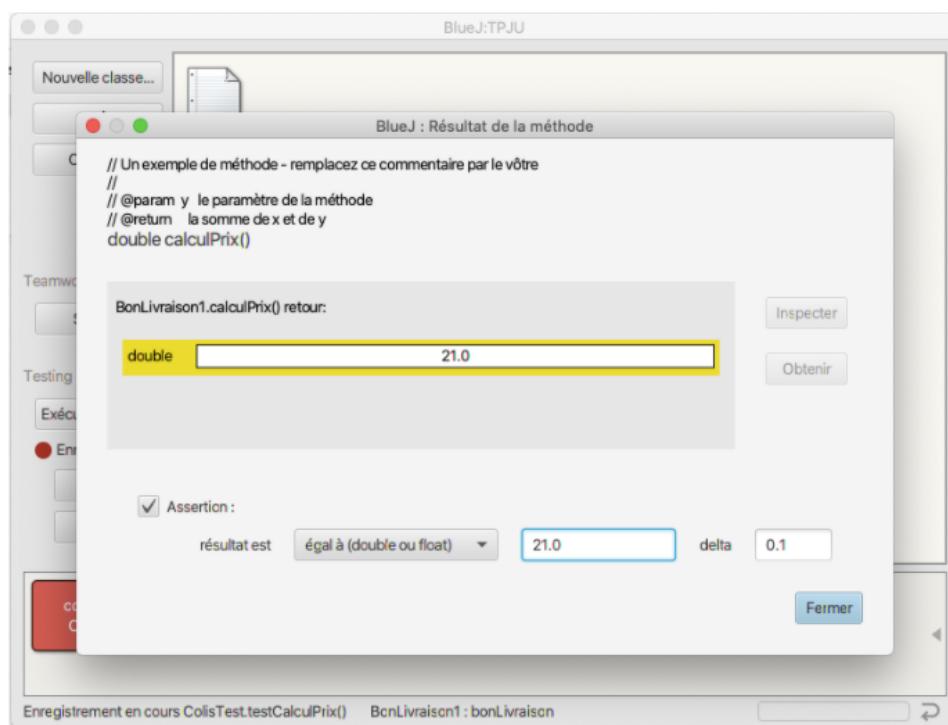
/*
 * Constructeur de la classe-test ColisTest
 */
public ColisTest()
{
}

/**
 * Met en place les engagements.
 *
 * Méthode appelée avant chaque appel de méthode de test.
 */
@Before
public void setUp() // throws java.lang.Exception
{
    colis2 = new Colis("Marie", "Montreuil");
    BonLivraison1 = new bonLivraison();
    colis2.chargementColis(1.5);
    BonLivraison1.setColis(colis2);
    BonLivraison1.getPoids();
    BonLivraison1.getPrix();
    BonLivraison1.calculPrix();
}

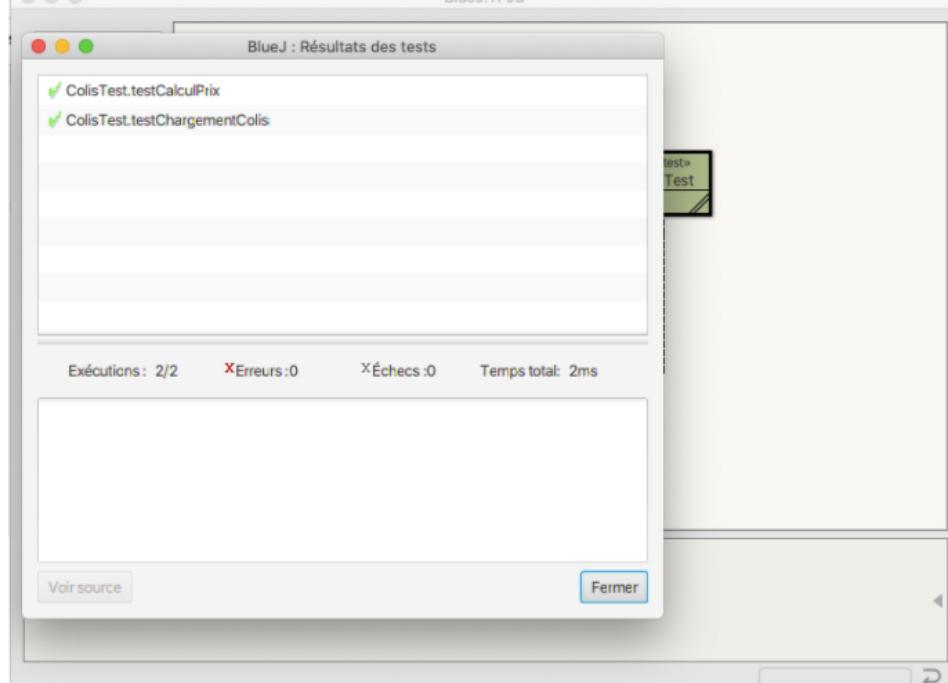

```



On finit en ajoutant un test qui vérifie que le calcul du prix de livraison s'effectue bien.



Le prix pour un colis de 2kg doit être de 21€ (soit $2 \cdot 10.5$).



Une fois le test enregistré, on peut exécuter tous les tests. Ici tous les tests sont validés.

Passons à présent à un environnement de développement.
Vous verrez ainsi le code qui est derrière ce que nous avons pu faire
jusqu'à présent.

```
public class DeliveryNote
{
    //instance variables

    private double price;
    double totalWeight;
    protected List<Package> packageList = new ArrayList<~>();
}
```

```
protected DeliveryNote DeliveryNote;
//instance variables

private double weight;
private String address;
private String recipient;
```

Nous avons une classe DeliveryNote ainsi qu'une classe Package.

Nous pouvons voir ici que le colis peut être associé à un seul bon de livraison

Tandis qu'un bon de livraison peut contenir plusieurs colis. C'est pourquoi on a ici une "liste" de colis.

```

public double getTotalWeight()
{
    for (Package pack : packageList)
        totalWeight += pack.getWeight();

    return totalWeight;
}

/**
 * @return      the total price depending on the weight
 */
public double calculationPrice()
{
    price = totalWeight*10.5;
    return price;
}

```

Le refactoring c'est le fait par exemple de faire une classe "calculationPrice" qui utilise le le poids total retourné par la classe précédente alors qu'on aurait pu avoir une seule classe pour le calcul du poids total et le calcul du prix

```

/**
 * a setter for the address
 */
public void setAddress(String address) { this.address = address; }

```

Le "rename" est une méthode qui va permettre de modifier une valeur, par exemple ici l'adresse au cas ou l'utilisateur souhaite la modifier.

```

/**
 * a getter for the weight
 */
public double getWeight() { return weight; }

/**
 * a setter for the address
 */
public void setAddress(String address) { this.address = address; }

/**
 * a getter for the address
 */
public String getAddress() { return address; }

/**
 * a getter for the recipient
 */
public String getRecipient()
{
    return recipient;
}

/**
 *Method for adding charge to the package
 *
 */
public double loadingPackage(double charge)
{
    weight += charge;
    return weight;
}

```

Pour la classe package nous retrouvons les méthodes qui permettent de modifier l'adresse, de renvoyer l'adresse, de renvoyer le destinataire ainsi que de charger le colis (donc faire augmenter son poids).

Pour chacune de ses méthodes on va vérifier que ce qu'elles renvoient est correct.

```

class PackageTest extends TestCase {

    public void setAddress(){
        Package pack4=new Package( address: "Paris", recipient: "Marie");
        pack4.setAddress("Marseille");

        String expectedAddress = "Marseille";
        String obtainedAddress = pack4.getAddress();
        assertTrue(obtainedAddress.equals(expectedAddress));
    }

    public void getAddress() {
        Package pack1= new Package( address: "Paris", recipient: "Maris");

        String expectedAddress = "Paris";
        String returnedAddress = pack1.getAddress();
        assertTrue (returnedAddress.equals(expectedAddress));
    }

    void getRecipient() {
        Package pack2= new Package( address: "Paris", recipient: "Julia");

        String expectedRecipient = "Julia";
        String returnedRecipient = pack2.getRecipient();
        assertTrue (expectedRecipient.equals (returnedRecipient));
    }

    public void loadingPackage() {
        Package pack3= new Package( address: "Paris", recipient: "Julia");
        double expectedWeight =1;
        double returnedWeight = pack3.loadingPackage( charge: 0.9);
        assertTrue( condition: expectedWeight == returnedWeight);
    }
}

```

Les tests se feront dans une classe de test.

Le principe des tests est toujours le même, il faut enregistrer la valeur ou l'objet que l'on attend dans une variable puis la comparer avec ce que renvoie réellement la méthode.

Prenons par exemple le test de la méthode chargement du colis. On déclare un colis (qui pèse donc 0,1 kg vide)

On utilise la méthode loadingPackage qui ajoute 0,9 kg au colis, et on vérifie que le poids du colis une fois chargé est bien de 1kg
assertTrue permet de vérifier que la valeur attendue est bien égale à la valeur renvoyée

```
class DeliveryNoteTest extends TestCase {

    public void getPack() {
        DeliveryNote deliveryNote1 = new DeliveryNote();
        Package pack1 = new Package( address: "Paris", recipient: "marie");
        Package pack2 = new Package( address: "Lyon", recipient: "Julienne");

        deliveryNote1.setPack(pack1);
        deliveryNote1.setPack(pack2);

        List<Package> expectedPackageList = new ArrayList<>();
        expectedPackageList.add(pack1);
        expectedPackageList.add(pack2);

        List<Package> returnedPackageList = deliveryNote1.packageList;

        assertTrue( expectedPackageList.equals( returnedPackageList));
    }

    public void getPrice() {
        DeliveryNote deliveryNote1 = new DeliveryNote();

        double expectedPrice= 0;
        double returnedPrice = deliveryNote1.getPrice();
        assertTrue( condition: expectedPrice == returnedPrice);
    }

    void getTotalWeight() {
        DeliveryNote deliveryNote1 = new DeliveryNote();
        Package pack1 = new Package( address: "Paris", recipient: "marie", weight: 0.9);
        Package pack2 = new Package( address: "Lyon", recipient: "Julienne", weight: 0.2);

        double expectedTotalWeight = 1.1;
        double returnedTotalWeight = deliveryNote1.getTotalWeight();
        assertTrue( condition: expectedTotalWeight == returnedTotalWeight);
    }
}
```

```

public void calculationPrice() {
    DeliveryNote deliveryNote1 = new DeliveryNote();
    Package pack1 = new Package( address: "Paris", recipient: "marie", weight: 0.9);
    Package pack2 = new Package( address: "Lyon", recipient: "Julienne", weight: 0.2);

    deliveryNote1.packageList.add(pack1);
    deliveryNote1.packageList.add(pack2);

    double totalWeight = deliveryNote1.getTotalWeight();

    double expectedPrice = 11.55;
    double returnedPrice = deliveryNote1.calculationPrice();
    assertTrue( condition: expectedPrice == returnedPrice);
}
}

```

Il faut procéder de la même manière pour la classe des bons de livraison.

Prenons par exemple le test de la méthode qui calcule le prix affiché sur le bon de livraison.

Tout d'abord nous déclarons deux colis que nous ajoutons à notre bon de livraison.

Nous déclarons le prix que l'on attend dans une variable puis nous appelons la méthode du calcul du prix.

Nous comparons ensuite les deux pour vérifier qu'ils sont bien égaux.

```

zouhyr@Air-de-Zouhyr livraison % java -cp junit-4.13.jar:hamcrest-core-1.3.jar:. org.junit.runner.JUnitCore DeliveryNoteTest PackageTest
JUnit version 4.13
.E.E
Time: 0,009

OK (2 tests)

```

Il faut tout d'abord ajouter deux .jar à notre librairie IntelliJ (suivre cette vidéo <https://www.codejava.net/testing/how-to-compile-and-run-junit-tests-in-command-line>)

Une fois les tests JUnit lancés via ligne de commande on peut voir que nos deux tests sont passés.

Pourquoi la tartine tombe toujours du côté du beurre



D'après la loi de Murphy, "S'il existe au moins deux façons de faire quelque chose et qu'au moins l'une de ces façons peut entraîner une catastrophe, il se trouvera forcément quelqu'un quelque part pour emprunter cette voie."

Cette loi est à prendre en compte en développement. Effectivement il faut toujours s'attendre à ce que l'utilisateur fasse "ce qu'il ne faut pas". C'est pourquoi il faut être très vigilant et tester tout ce qui est codé dans les classes. Si le test n'est pas validé, il faudra alors faire en sorte de couvrir tous les cas possibles.



Nous allons maintenant utiliser Cucumber.

Cucumber est un outils de test open source qui prend en charge le développement dirigé par le comportement (BDD)

Le BDD consiste à écrire du code sous forme de scénario accessible à toute personne et non plus uniquement aux développeurs.

Les scénarios décrivent une exigence attendue par le client.

Cucumber lit les spécifications écrites en langage "naturel" et confirme que le logiciel est fidèle à la spécification et génère un rapport indiquant le succès ou l'échec pour chaque scénario.

Pour que Cucumber comprenne les scénarios, ils doivent suivre des règles de syntaxe, appelées Gherkin .

Gherkin est un ensemble de règles de grammaire qui rend le texte écrit par les "humains" structuré et qui permet à Cucumber de l'analyser et le comprendre.

- **Feature** : Nom de la fonctionnalité testée.
- **Description** (facultatif) : description de la fonctionnalité à tester.
- **Scénario** (Scenario outline): Quel est le scénario de test.
- **Étant donné** (Given) : Pré-requis avant que les étapes de test ne soient exécutées.
- **Quand** (When): Condition spécifique pour exécuter l'étape suivante.
- **Alors** (Then) : Que doit-il se passer si la condition mentionnée dans Quand est remplie?

Le fichier de fonctionnalité ne suffit pas. Il faut également un fichier de définition intermédiaire afin que Cucumber sache quel morceau de code doit être exécuté pour un scénario (du fichier de fonctionnalité).

Le fichier de définition des étapes stocke le mappage entre chaque étape du scénario avec un code de fonction à exécuter.

Voyons ce que cela donne dans notre exemple de Colis

The screenshot shows a Java IDE interface with the following project structure:

- Project: DeliveryNoteTest
- resources
- test
 - java
 - livraison
 - DeliveryNoteTest.java
 - PackageTest.java
 - stepDefs
 - PoidsColisStepdefs.java
 - PrixColisStepdefs.java
 - RunCucumberTest.java
 - resources
 - DeliveryNoteTest.feature
 - PackageTest.feature

On the right, the `DeliveryNoteTest.feature` file is open, displaying a Cucumber feature and scenario outline:

```
Feature: US_001 Donner les prix des colis
  En tant que client
  Je veux être sûr de payer le prix correspondant au poids de mon colis
  Afin de m'assurer que le prix à payer est justifié

  Scenario Outline: calcul du prix du colis
    Given Un bon de livraison peut contenir plusieurs colis <packageList1>
    When le client pèse l'ensemble de colis de poids total <totalWeight1>
    Then le prix du colis <price1> sera automatiquement calculé

  Examples:
    | packageList1 | totalWeight1 | price1 |
    | colis1       | 0.1          | 1.05   |
    | colis2       | 2.0          | 21.0    |
```

Ici, on veut vérifier que le prix des colis est correctement calculé sur les bons de livraison. On décrit dans un langage naturel ce qu'on attend lors du calcul du prix de la livraison et on donne des exemples avec des valeurs.

Mais il faut également notre fichier de définition:

The screenshot shows a Java IDE interface with the `PrixColisStepdefs.java` file open:

```
package stepDefs;

import ...

public class PrixColisStepdefs {

    @Given("Un bon de livraison peut contenir plusieurs colis <packageList1>")
    public void poidsTotalColis() throws Exception {
        List<Package> packageList = new ArrayList<Package>();
        throw new PendingException();
    }

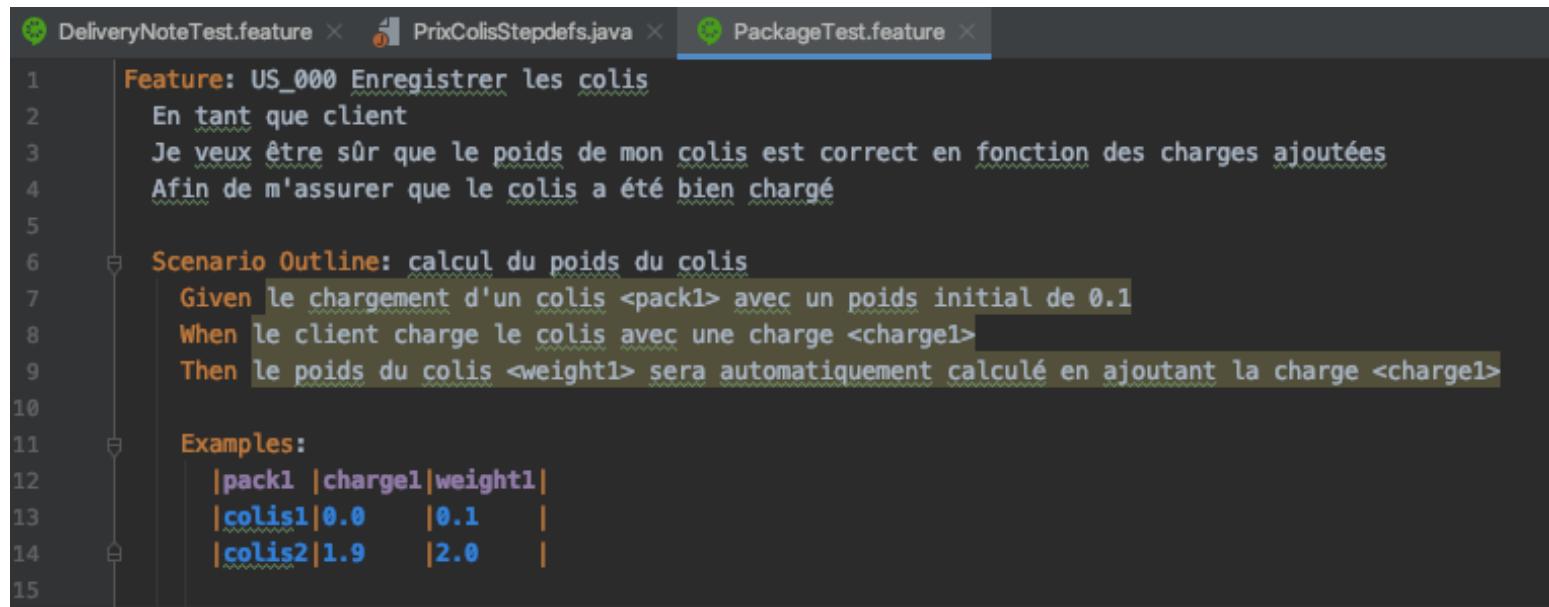
    @When("le client pèse l'ensemble de colis <packageList1> de poids total <totalWeight1> {double}")
    public void calculPoidsTotal(List<Package> packageList1, double totalWeight1) throws Exception {
        for (Package pack1 : packageList1)
            totalWeight1 += pack1.getWeight();
        throw new PendingException();
    }

    @Then("le prix du colis <price1> sera automatiquement calculé")
    public void calculPrix(double totalWeight1, double price1) throws Exception {
        price1 = totalWeight1*10.5;
        throw new PendingException();
    }
}
```

Dans le fichier de définition on "explique" à Cucumber où est ce qu'il doit aller chercher les validation. Par exemple lorsque dans le fichier de fonctionnalité on décrit qu'un bon de livraison peut contenir plusieurs colis, on indique dans notre fichier de définition que cette information est contenue dans la méthode `poidsTotalColis()` dans la création d'une liste de colis.

Lorsque dans le fichier de fonctionnalité on décrit que le client pèse l'ensemble de colis `<packageList1>` de poids total `<totalWeight1>` {double}, on indique dans le fichier de définition que cette information devra être vérifier dans la méthode `calculPoidsTotal` qui prend en paramètres l'ensemble de colis appelé `packageList1` et le poids total `TotalWeight1`.

On procède de la même façon pour faire les vérifications associées aux Colis :



```
DeliveryNoteTest.feature ✘ PrixColisStepdefs.java ✘ PackageTest.feature ✘
1 Feature: US_000 Enregistrer les colis
2   En tant que client
3     Je veux être sûr que le poids de mon colis est correct en fonction des charges ajoutées
4     Afin de m'assurer que le colis a été bien chargé
5
6 Scenario Outline: calcul du poids du colis
7   Given le chargement d'un colis <pack1> avec un poids initial de 0.1
8   When le client charge le colis avec une charge <charge1>
9   Then le poids du colis <weight1> sera automatiquement calculé en ajoutant la charge <charge1>
10
11 Examples:
12 | pack1 | charge1 | weight1 |
13 | colis1 | 0.0 | 0.1 |
14 | colis2 | 1.9 | 2.0 |
```

```
1 package stepDefs;
2
3 import ...
4
5 public class PoidsColisStepdefs {
6
7     @Given("le poids total du colis {Package} avec un poids initial weight1 de 0.1 {double}")
8     public void poidsColis(double weight1) throws Exception {
9         Package pack1 = new Package(weight1);
10        throw new PendingException();
11    }
12
13    @When("le client charge le colis")
14    public void chargeColis(double weight1) throws Exception {
15        Package pack1 = new Package(weight1);
16        pack1.getWeight();
17        throw new PendingException();
18    }
19
20    @Then("le poids du colis weight2 {double} sera automatiquement calculé")
21    public void calculPoids(double weight1, double charge1) throws Exception {
22        Package pack1 = new Package(weight1);
23        pack1.getWeight();
24        weight1 += charge1;
25        throw new PendingException();
26    }
27
28 }
29 }
```

Une fois nos fichier de fonctionnalités et de définitions écrits, On lance les tests et on vérifie qu'ils sont validés.

```
-----  
Running stepDefs.RunCucumberTest  
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0.226 sec <<< FAILURE! - in stepDefs.RunCucumberTest  
initializationError(stepDefs.RunCucumberTest)  Time elapsed: 0.006 sec  <<< ERROR!  
cucumber.runtime.CucumberException: No backends were found. Please make sure you have a backend module on your CLASSPATH.  
  
Results :  
  
Tests in error:  
  RunCucumberTest.initializationError » Cucumber No backends were found. Please ...  
  
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0  
  
[ERROR] There are test failures.  
  
Please refer to C:\Users\julie\IdeaProjects\Agility2\target\surefire-reports for the individual test results.  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time:  4.888 s  
[INFO] Finished at: 2020-05-03T23:14:19+02:00  
[INFO] -----
```

Ici on note qu'un des test n'est pas passé.

