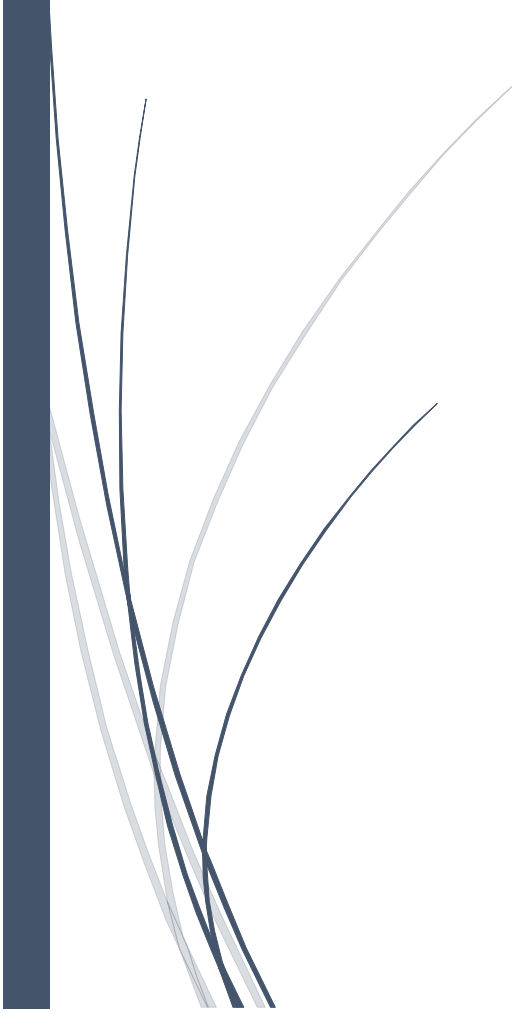
28/04/2020

SuperHero

La création de classe fait par les grands



Simona ARIZANOVA - Mickael PEEREN

Table des matières

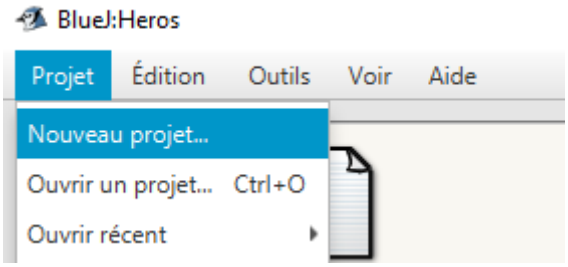
Bluej et Tests

Bienvenue à vous, super-héros ! Je suis Java-man, et je vais vous apprendre le Java et les concepts objets avec l'aide d'une application appelée BlueJ.

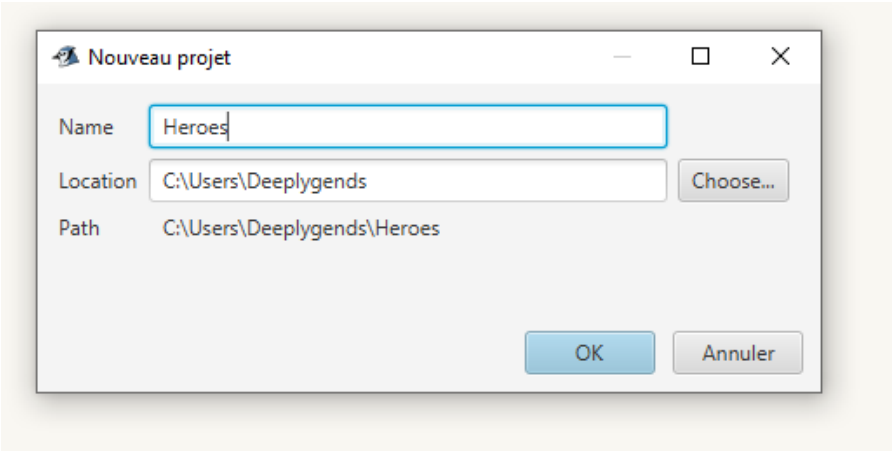
Vous allez m'aider à créer un super héro afin de combattre les méchants.

1. Créer une classe SuperHero

Créer un nouveau projet dans **BlueJ** via **Projet** > **Nouveau Projet …**

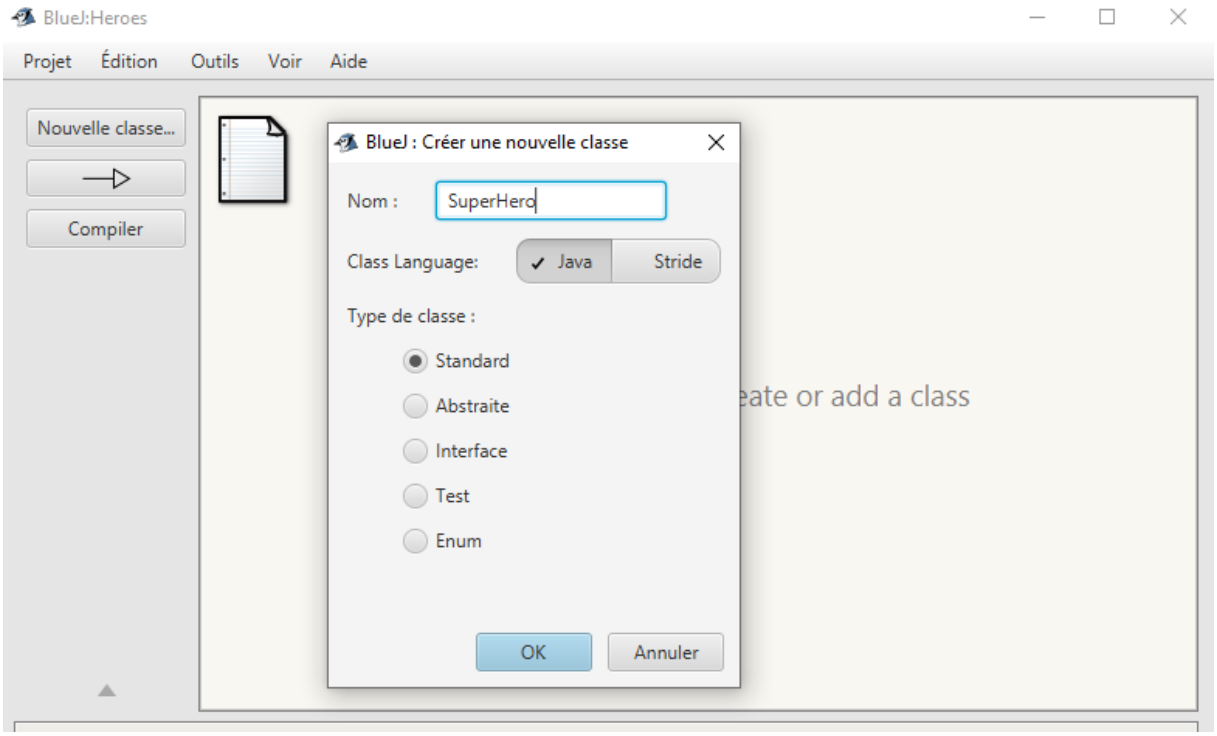


Entre le nom de projet que vous voulez (Avengers, Heroes, Rick et Morty, etc …)

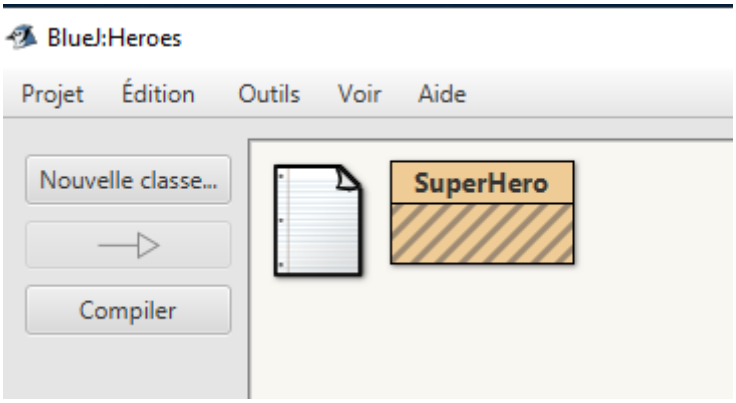


Ensuite, afin de créer notre équipe (constituée de super héros, je vous le rappelle)

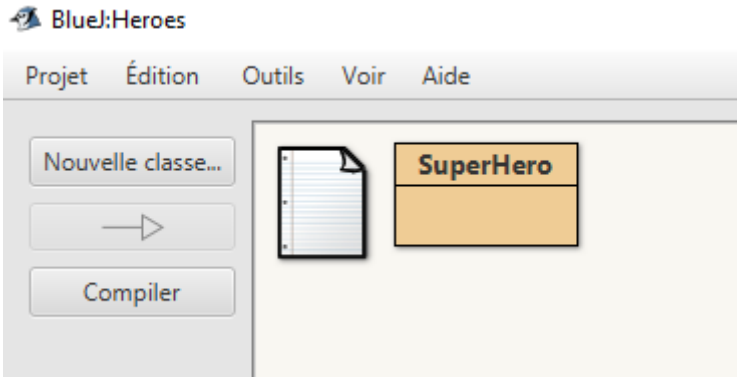
, nous devons créer la classe « *SuperHero* ».



Félicitations, jeune padawan, la classe SuperHero vient d'être créée mais …… elle est toute barrée !



Pas de panique, Compile-man est ici pour vous aider. Cliquez sur **Compiler** et ... tadam !



Et si nous allons voir ce qu'il y a dans notre classe :

```java
/**
 * Décrivez votre classe SuperHero ici.
 *
 * @author (votre nom)
 * @version (un numéro de version ou une date)
 */
public class SuperHero
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private int x;

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

    /**
     * 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 int sampleMethod(int y)
    {
        // Insérez votre code ici
        return x + y;
    }
}
```

Cela ne représente pas vraiment un super-héros, apportons quelques modifications. Un Super-héros possède une identité et une certaine force. Ajoutons un champ « *identity* » de type string (chaine de caractère) et un champ « *strenght* » de type int (entier). Ces deux champs seront privés : on ne veut pas que les méchants puissent changer la force de notre héros et la mettre à 0.

```java
public class SuperHero
{
    // variables d'instance - remplacez l'exemple qui suit par le vôtre
    private String identity = "Spiderman";
    private int strength =  35;
```

Ensuite, nous allons ajouter les « *getter* » (méthodes d'accès pour récupérer une donnée) et les « *setter* » (pour initialiser une donnée privée) de nos deux champs.

```
public SuperHero()
{
    // initialisation des variables d'instance
}

public String getIdentity()
{
    return this.identity;
}

public void setIdentity(String identity)
{
    this.identity = identity;
}

public int getStrength()
{
    return this.strength;
}

public void setStrength(int strength)
{
    this.strength = strength;
}
```

Cela commence à ressembler à un super-héros. Mais pour rester un super-héros, il faut s'entrainer dur. Ajouter une méthode « *workOut* » qui prend en paramètre un entier (la force gagnée pendant l'entrainement) et qui va nous renvoyer la force totale du super-héros après son entrainement.

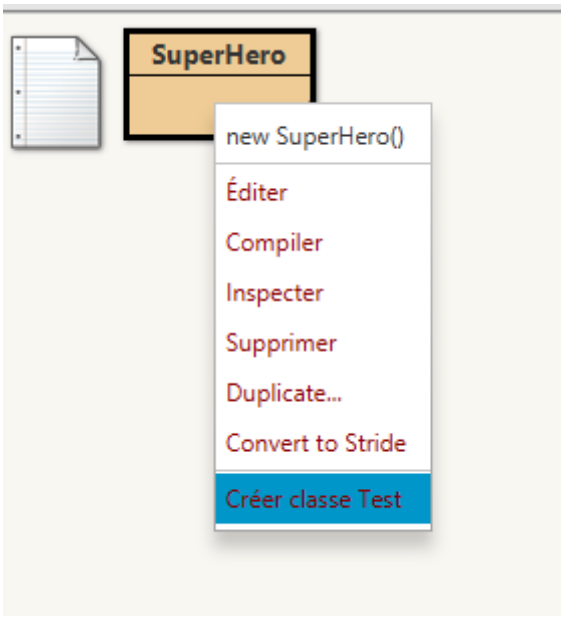
```
public int workOut(int addStrength)
{
    return this.strength + addStrength;
}
```

Très bien, notre super-héros est fort et entrainé. Mais on devrait vérifier s'il s'est bien entrainé non ? Même les super-héros trichent parfois …

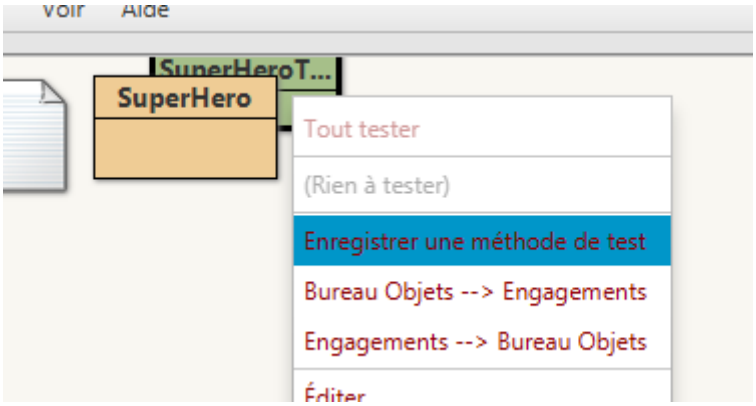
2. Tester les méthodes d'une classe

Créons une classe de test pour tester notre méthode workOut :

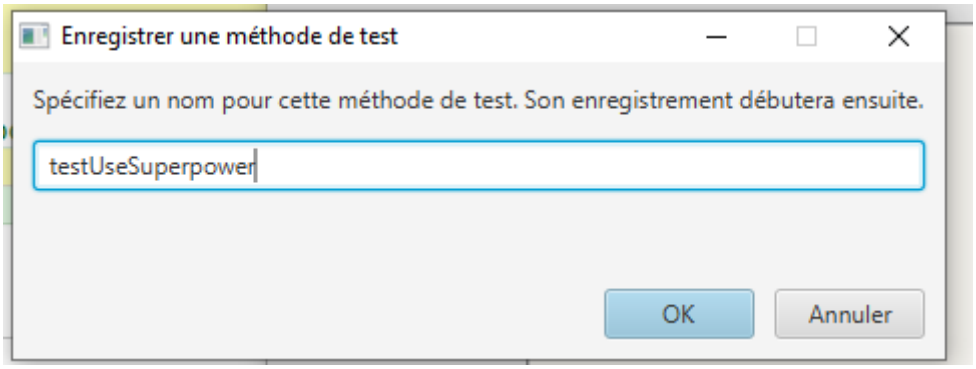
Clic droit sur la classe SuperHero > **Créer classe Test**



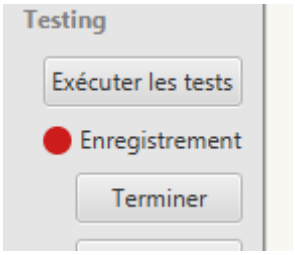
La classe de test apparait en vert, encore une fois : **Clic droit** dessus > **Enregistrer une méthode de test.**



Il faut faire attention à bien nommer notre méthode de test afin de bien savoir ce qu'elle teste, sinon on va oublier après.

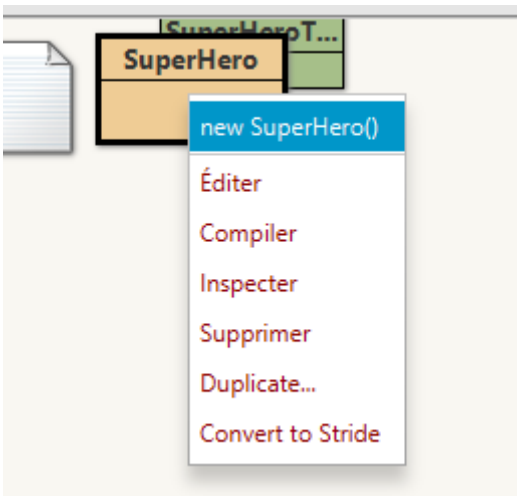


Tiens, tiens. L'enregistrement est lancé, à partir de maintenant tout ce que tu feras sera enregistré (Big Brother est là !).

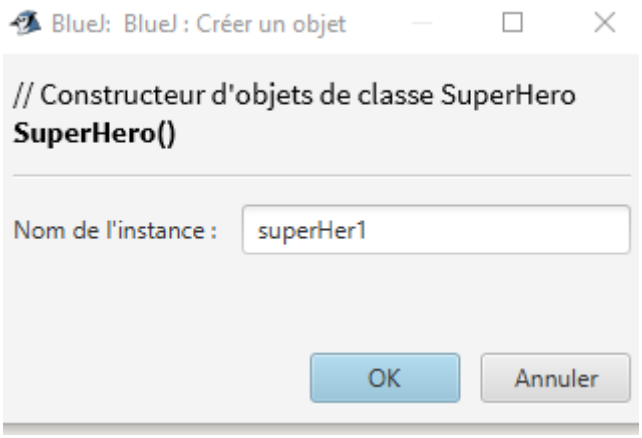


Pour tester l'entrainement d'un super-héros il faut créer une instance de « *SuperHero* » :

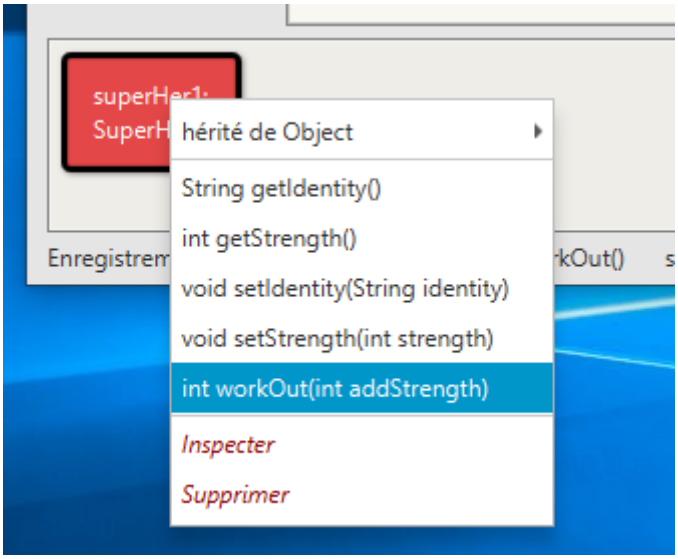
Clic droit sur la classe SuperHero > **new SuperHero()**



On nomme la variable comme on veut (ici superHer1). On peut aussi l'appeler Batman, Superman, Green Lantern si on veut être original.



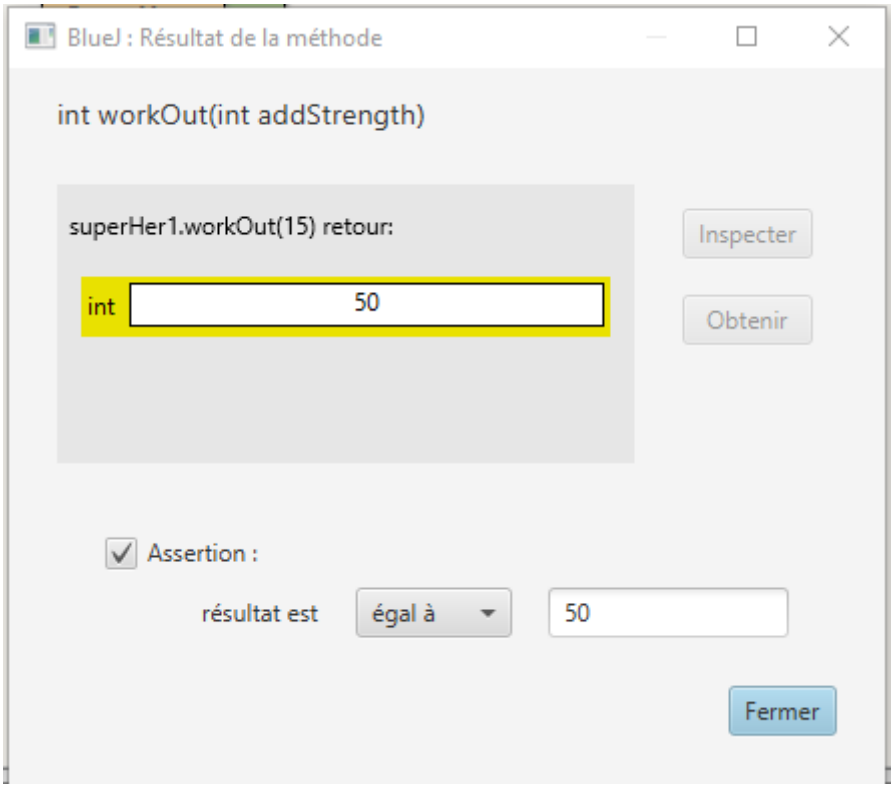
Et enfin, on lance la méthode « *workOut* »de superHer1, notre premier super-héros. Il va faire un peu de corde à sauter, de course à pied, de natation mais aussi d'autres entrainements dignes d'un vrai super-héros !



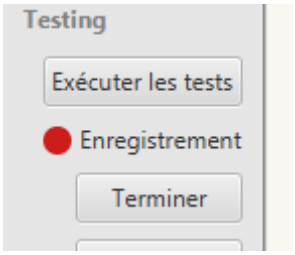
Après son entrainement, on imagine qu'il va gagner 15 de force :

De base, un nouveau super-héros a 35 de force, donc on devrait obtenir 50. On écrit 50 à coté de « **égal à ».**

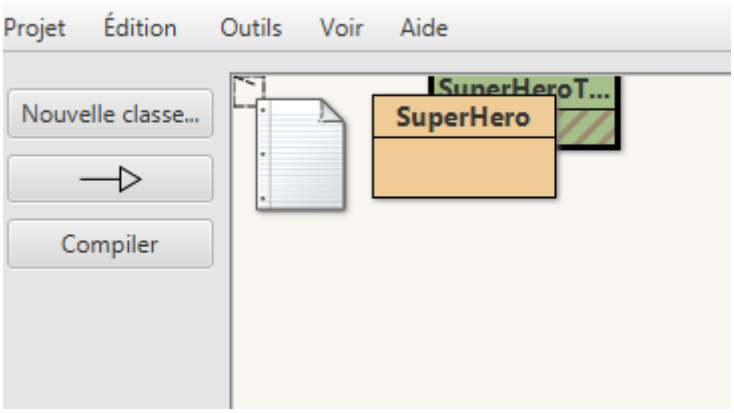
Encadré en jaune est le résultat de notre opération (ouf c'est le bon d'ailleurs)



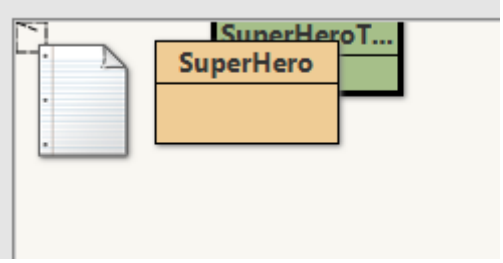
Notre test est terminé donc ……. On clique sur **Terminer**



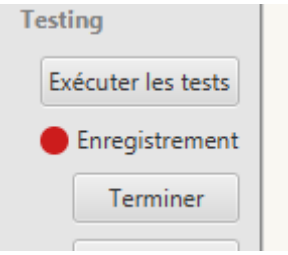
On n'oublie pas de compiler, sinon Compile-man ne va pas être content.



Ce qui nous enlève les rayures de notre joli carré vert.



Et on clique sur **Exécuter les tests :**



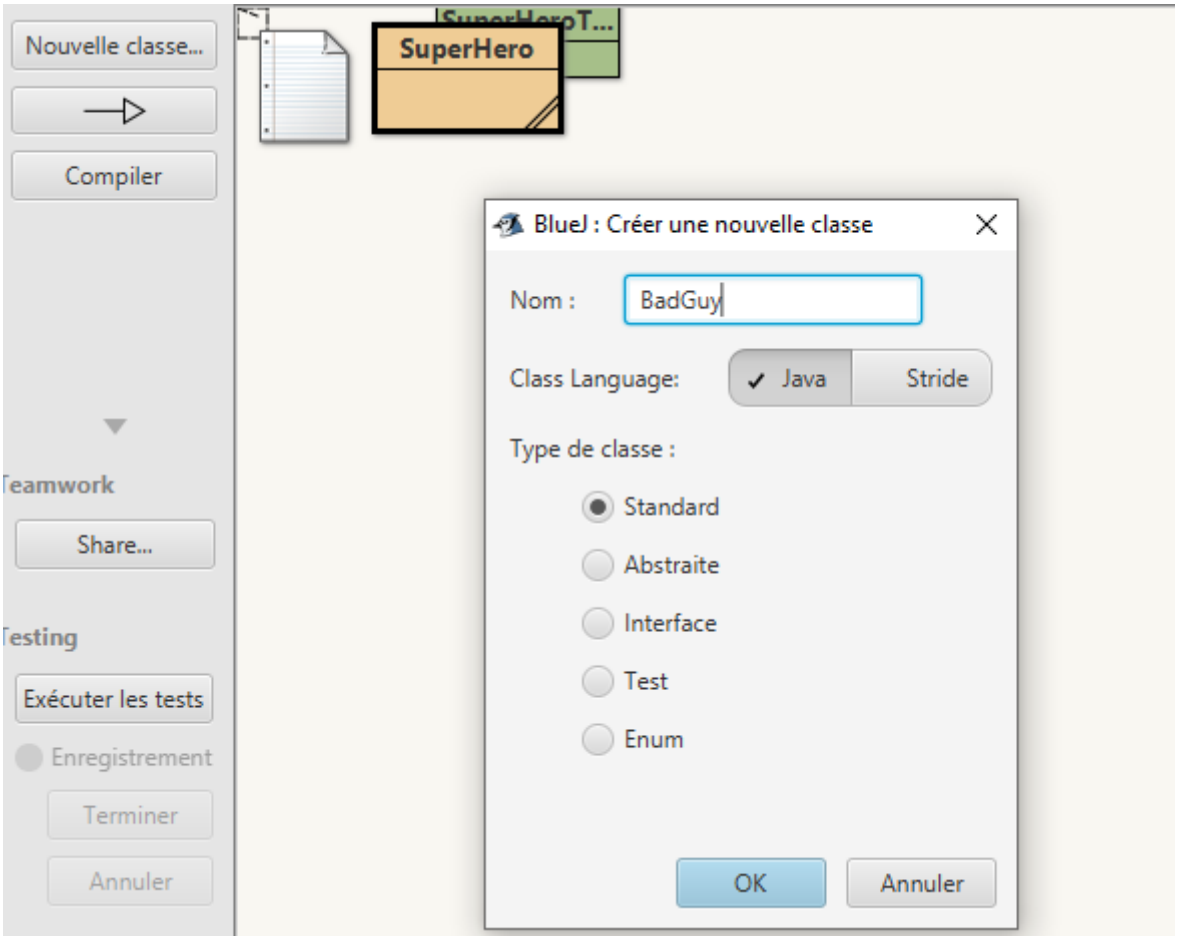
Et on obtient le résultat de notre test :

Yyyeeaaaaahhhh !! Première victoire de notre super-héros. C'est en grande partie grâce à toi !

Partie 2 – IntelliJ et JUnit

3. Liens entre deux classes

Mais passons aux choses sérieuses : un grand méchant arrive pour mettre des bâtons dans les roues de notre super-héros (Mouhahahaa). Créons une classe BadGuy.



Le méchant, il n'est jamais très malin (donc on ne va pas lui mettre trop de champs). En général, c'est une brute donc il n'aura qu'un champ « *strenght* » (avec son accesseur getStrength)

```java
/**
 * Décrivez votre classe BadGuy ici.
 *
 * @author (votre nom)
 * @version (un numéro de version ou une date)
 */
public class BadGuy
{
    // variables d'instance - remplacez l'exemple qui suit
    private int strength = 10;

    /**
     * Constructeur d'objets de classe BadGuy
     */
    public BadGuy()
    {
        // initialisation des variables d'instance
    }

    public int getStrength()
    {
        // Insérez votre code ici
        return this.strength;
    }
}
```

La ville n'est plus sûre depuis que notre méchant est arrivé. Notre super-héros va devoir lui apprendre les bonnes manières. Dans notre classe « *SuperHero* », nous allons ajouter un champ « *BadGuy* ». En effet, il vient sans vergogne dans la ville de notre héros.

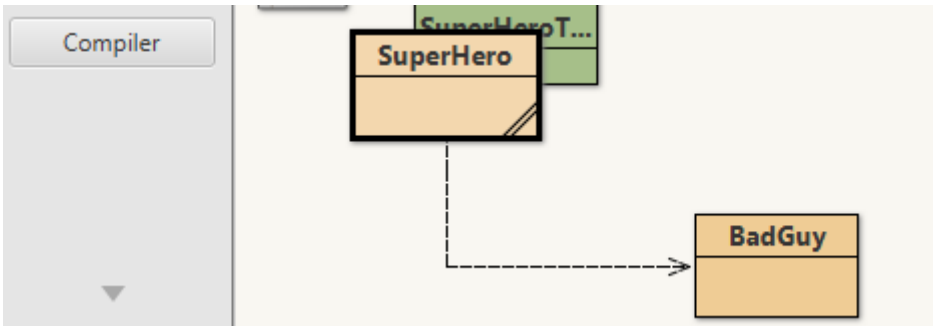
```java
public class SuperHero
{
    // variables d'instance - remplacez l'exemple
    private String identity = "Spiderman";
    private int strength =  35;

    private BadGuy badGuy = new BadGuy();
```

Nous venons d'associer un méchant à notre héros. Maintenant, place à la bagarre. Notre super-héros va essayer de se débarrasser de notre super-méchant. Nous allons créer une méthode « *fightBadGuy* » dans notre classe « *SuperHero* » pour qu'il combatte son méchant.

La méthode compare la force des deux opposants, et renvoie « *True* » si notre super-héros a une force supérieure à son méchant (et il lui botte allégrement les fesses). Sinon, on renvoie false.

```
/**
 * Return true if superhero beat the badGuy
 * else false
 */
public boolean fightBadGuy()
{
    return this.strength >= badGuy.getStrength();
}
```

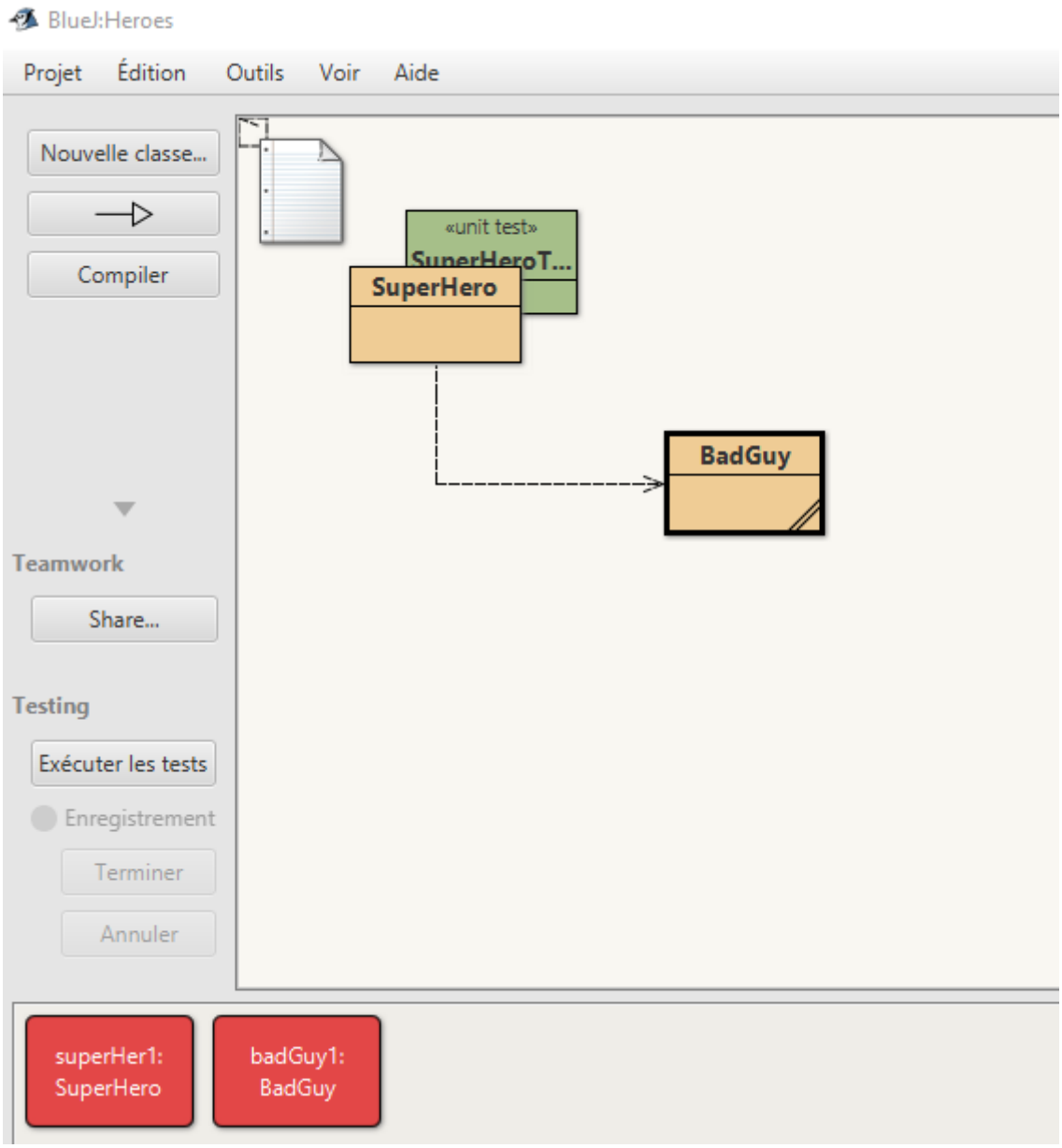
Compile-man vous demande de compiler. Exécution !



On avance bien. Essayons de sauvegarder une instance de « *Superhero* » avec une instance de « *BadGuy* » dans notre fichier de test.

Tout d'abord, sur la classe « *SuperHero* », **Clic Droit** sur la classe SuperHero > **new SuperHero()**

Ensuite, sur la classe BadGuy, **Clic Droit** sur la classe BadGuy > **new BadGuy()**



Comment faire pour dire à notre super-héros : « Tiens, ça c'est ton méchant, il est là pour t'embêter ? »

Nous sommes obligés de créer un « *setter* » dans notre classe « *SuperHero* » pour lui dire qui est son méchant. On supprime l'initialisation ci-dessous :

```
public class SuperHero
{
    // variables d'instance - remplacez l'exemple
    private String identity = "Spiderman";
    private int strength =  35;

    private BadGuy badGuy = new BadGuy();
```

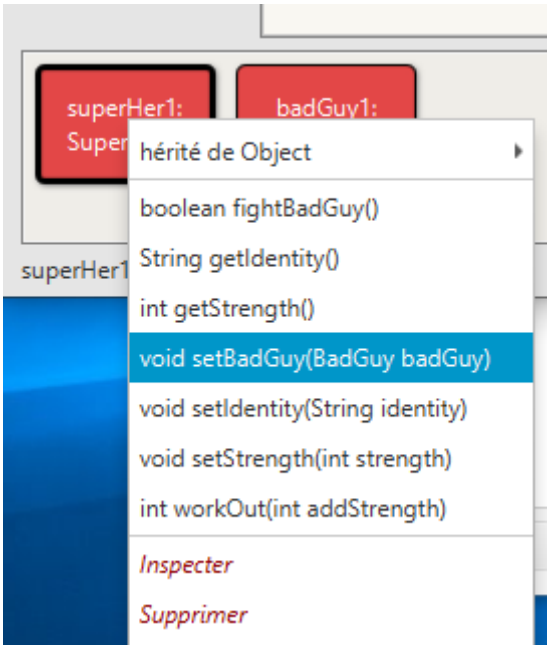
Par

```
private BadGuy badGuy;
```

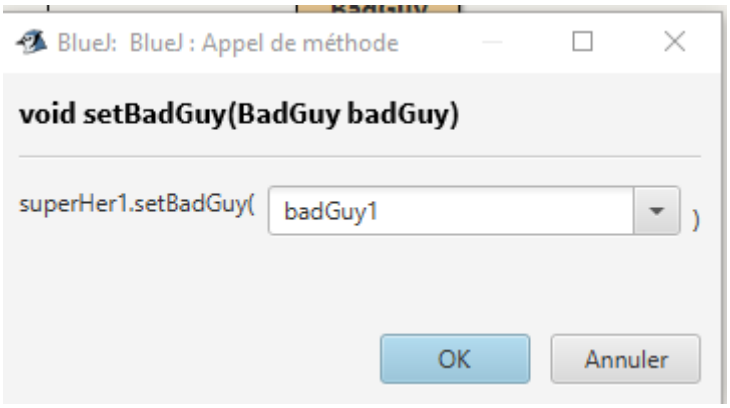
Et on ajouter le setter « *setBadGuy* » pour la classe « *SuperHero* »

```
public void setBadGuy(BadGuy badGuy){
    this.badGuy = badGuy;
}
```

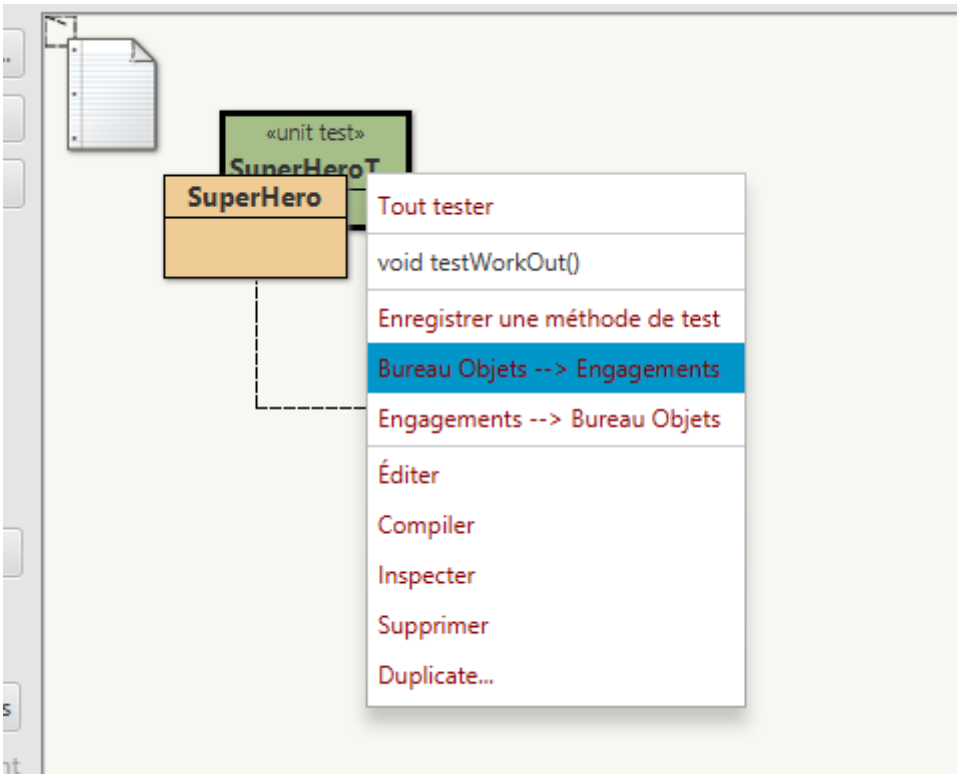
Maintenant, nous pouvons dire à notre SuperHero « *superHer1* » que « *badGuy1* » est son méchant via la méthode « *setBadGuy* ».



On précise en paramètre le nom de la variable du méchant (ici *badGuy1*)



Et, pour insérer ces deux instances dans le « *setUp* » de notre fichier test, nous faisons **ClicDroit** > « **Bureau Objets → Engagements** »



Tadam, Test-Man a encore frappé. Allons voir le code de « *SuperHeroTest* » en double cliquant dessus.

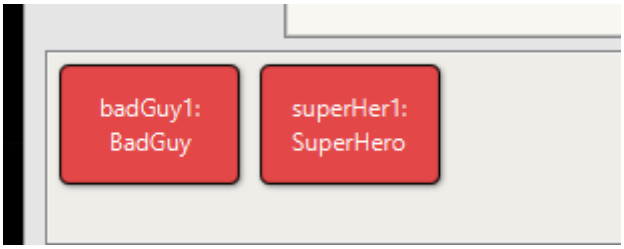
```
public class SuperHeroTest
{
    private SuperHero superHer1;
    private BadGuy badGuy1;
```

Deux champs sont apparus. Notre super-héros et son méchant. Et ci-dessous on a leur initialisation dans la méthode « *setUp* »

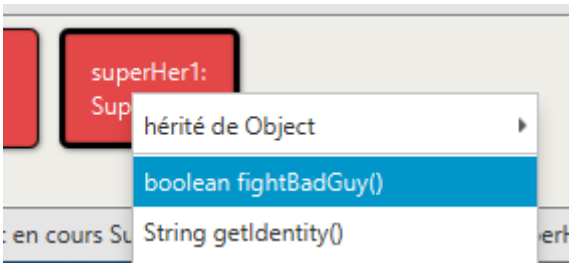
```
 */
@Before
public void setUp() // throws java.lang.Exception
{
    superHer1 = new SuperHero();
    badGuy1 = new BadGuy();
    superHer1.setBadGuy(badGuy1);
}
```

Enfin, la bataille finale !!! Pour tout combat, il y a des règles. Vérifions que les nôtres soient bien respectées. Nous allons tester la méthode « *fightBadGuy* ».

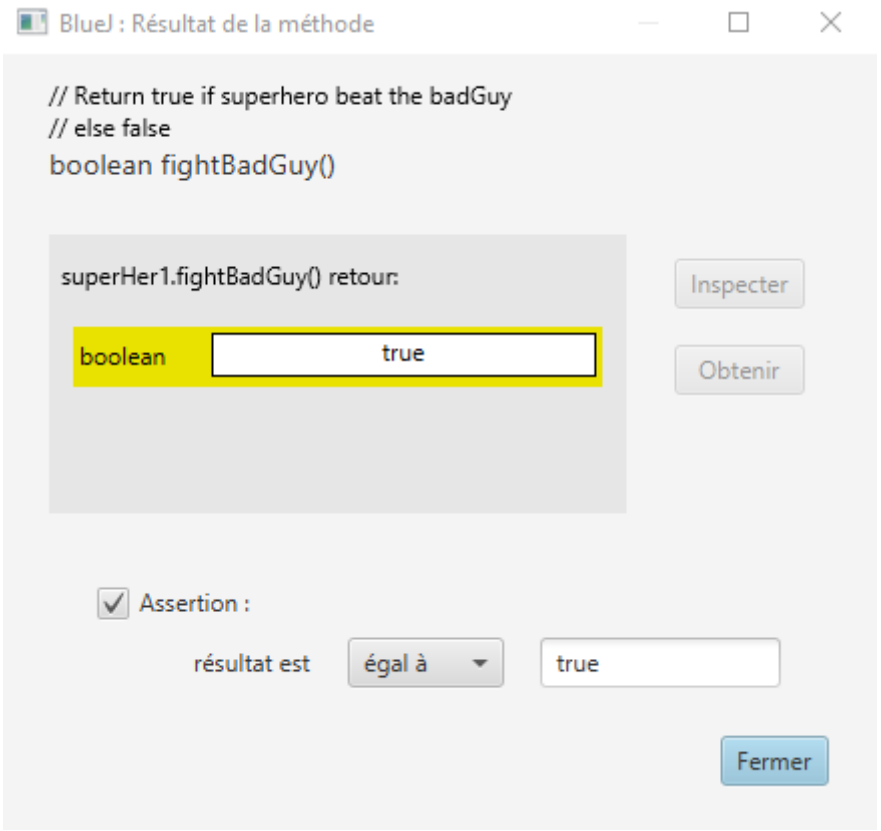
Pour cela, sur notre classe « *SuperHeroTest* », **Clic droit** > « **Enregistrer une méthode de test** ».



Comme par magie, nos deux objets sont réapparus. Testons la méthode « *fightBadGuy* »

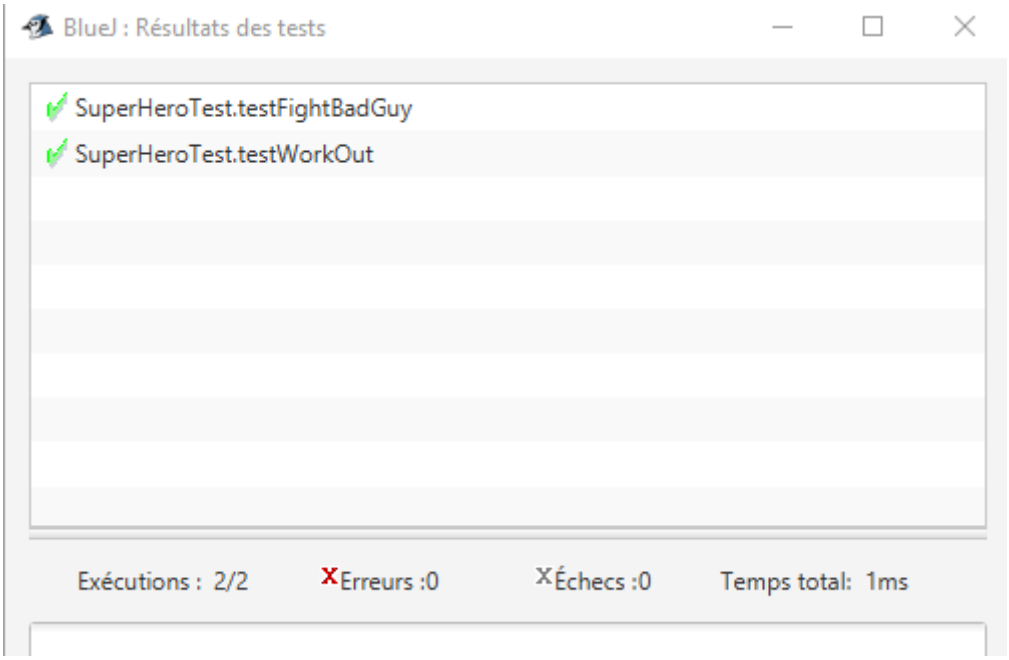


Notre super-héros a une force de 35 et le méchant une force de 10 donc on s'attend à ce que le résultat soit « *True* ».



On fait ce que Compile-man sait faire le mieux : on compile.

Et on fait ce que Test-man fait le mieux : On teste en exécutant les tests (tout simplement)



Bravo ! Grâce à toi la ville est enfin débarrassée du grand méchant ! Et notre super-héros peut continuer à veiller sur nos vies et nos espoirs.

4. Implémentation d'un attribut de relation 0..1 - * - La bi-directionnalité

Nous avons fini de travailler avec **BlueJ**, maintenant place aux outils de ceux qui jouent dans la cours des grands ; nous allons travailler avec **IntelliJ**.

Donc, ou est ce qu'on s'était arrêtés ? Ah oui. J'ai remarqué quelque chose : on a un superhéros contre un seul méchant ? Cela ne tient pas trop la route. Notre super-héros est très fort et il peut se permettre de dérouiller plusieurs méchants.

Nous allons donc transformer l'attribut de type « *BadGuy* » dans « *SuperHero* » en Collection<BadGuy>

```
private Collection<BadGuy> badGuys;


/**
 * Constructeur d'objets de classe SuperHero
 */
public SuperHero()
{
    badGuys = new ArrayList<BadGuy>();
}
```

Sans oublier d'initialiser la collection dans le constructeur de « *SuperHero* » (sinon NullReference-Man se moquera de toi).

Voilà. Désormais on peut enfin combattre le crime comme il faut. Mais … KESAKO ? Il y a des erreurs de partout.

```java
public void setBadGuy(BadGuy badGuy){
    this.badGuy = badGuy;
}
```

```java
public boolean fightBadGuy()
{

    return this.strength >= badGuys.getStrength();
}
```

Il va falloir corriger tout cela pour que notre super-héros puisse se battre correctement. Nous devons modifier la méthode « *setBadGuy* » pour qu'elle ajoute le méchant.

```java
public void setBadGuy(BadGuy badGuy){
    badGuys.add(badGuy);
}
```

Et nous allons modifier notre méthode « *fightBadGuy* » pour que notre héro combatte toute sa liste de méchants.

```java
/**
 * The hero fight every bad guy. If he have been defeat one time, the m
 * If he beats all his badGuy
 */
public boolean fightBadGuy()
{
    boolean win = true;
    for(BadGuy badguy : badGuys)
    {
        if(badguy.getStrength() > this.getStrength())
        {
            win = false;
            break;
        }
    }
    return win;
```

Ah ouf ! Il n'y a plus d'erreurs. Mais nous sommes confrontés à un problème … tu ne vois pas de quoi je parle ? Aller tu me fais marcher ! Nous n'avons pas de relation bidirectionnelle. Le héro connait ses méchants mais le méchant ne connait pas le héro contre qui il se bat. Rectifions cela !

```
private SuperHero nemesis;
/**
 * Constructeur d'objets de classe BadGuy
 */
public BadGuy()
{
    // initialisation des variables d'instance
    nemesis = new SuperHero();
}
```

Il faut pouvoir affecter le bon super-héros au méchant. Créons les méthodes d'accès dans la classe « *BadGuy* ».

```
public void setSuperHero(SuperHero nemesis)
{
    this.nemesis = nemesis;
}
public SuperHero getSuperHero()
{
    return this.nemesis;
}
```

Et maintenant on écrit le test.

```java
@BeforeEach
public void setUp()
{

    badGuy = new BadGuy();
    superhero = new SuperHero();
}
@Test
public void testGetStrength(){
    assertEquals( expected: 10, badGuy.getStrength());
}
@Test
public void testGetSuperHero()
{
    assertEquals(superhero, badGuy.getSuperHero());
}
@Test
public void testSetSuperHero()
{
    badGuy.setSuperHero(superhero);
    assertEquals(superhero, badGuy.getSuperHero());
}
```

Et on lance les tests et …. Mais « Qué ça dire que ceci » ?

```
Tests failed: 1 of 1 test – 18 ms
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" ...


java.lang.AssertionError: expected:<model.SuperHero@442675e1> but was:<model.SuperHero@6166e06f>
Expected :model.SuperHero@442675e1
Actual   :model.SuperHero@6166e06f
<Click to see difference>
```

« *testGetSuperHero* » a échoué : pourtant quand on crée un nouveau super héros il s'appelle « Spider-man » et possède 35 de force alors pourquoi ?

C'est parce qu'un objet créé, même s'il possède les mêmes valeurs dans ses attributs qu'un autre objet ne possède pas forcément la même adresse en mémoire. Donc si deux objets possèdent les mêmes caractéristiques mais pas la même adresse mémoire, est ce que pour autant on doit les considérer comme différents ?

Nous allons donc utiliser et définir la méthode « *equals* » afin de rendre deux objets « *SuperHero* », possédant les mêmes données, égaux.

```java
@Override
public boolean equals(Object o)
{
    /**
     * si l'objet passé en parametre est null ou n'est pas un objet SuperHero alors ils ne
     * peuvent pas être égaux
    **/
    if(o == null || !(o instanceof SuperHero)) return false;
    /**
     * Si la référence (l'adresse mémoire est la même) ils sont forcément égaux
     */
    if(o == this) return true;
    /**
     * Sinon on transforme o en objet SuperHero et on compare les champs idendity et strength
     */
    SuperHero superhero = (SuperHero)o;
    return superhero.getStrength() == this.strength && superhero.getIdentity() == this.identity;
}
```

On peut relancer notre test et …. C'est un succès !!!

Je vous laisse ajouter la méthode « *getSuperHero* » et son test 😉

5. Refactoring

Le refactoring est utilisé lorsque nous modifions la structure d'un objet, du programme et qui va impliquer de devoir modifier la logique visuelle et de compréhension du code. Nous allons chercher à adapter le code que nous avons écrit aux modifications qui ont été faites. Et je vais te parler de deux méthodes aujourd'hui : le **renommage** et **l'extraction**.

1. Rename Method

Cette méthode consiste à renommer les variables, attributs ou méthodes pour les rendre cohérents avec leur fonction. Par exemple, avant d'implémenter la Collection de « *BadGuy* », nous avions ceci :

```java
public class SuperHero
{
    // variables d'instance - remplacez l'exemple
    private String identity = "Spiderman";
    private int strength =  35;

    private BadGuy badGuy = new BadGuy();
```

Et, subtilement, nous l'avons modifié de la manière suivante :

```java
    private Collection<BadGuy> badGuys;
```

En effet, « *badGuy* » au singulier est devenu « *badGuys* ». Cet exemple est subtil, mais nous allons faire une modification qui n'a pas été encore faite : la méthode « *fightBadGuy* ». En effet, son nom

implique que notre héro va se battre contre seulement un seul méchant. Or notre méthode le fait combattre tous les méchants. On va devoir la renommer :

```
public boolean fightAllBadGuys()
```

Comme ça, toute personne qui verra ton code va tout comprendre, d'un seul coup.

2. Extract Method

La méthode d'extraction permet d'atomiser le code suivant la responsabilité. Imaginons que nous ajoutons se code dans la méthode « *workOut* » de SuperHero :

```
public int workOut(int addStrength)
{
    this.strength += addStrength;
    //Print details about the hero after the work out
    log.info( msg: "name : "+identity);
    log.info( msg: "force : " + strength);
    return this.strength;
}
```

L'affichage des détails est effectué grâce à un champ de type **Logger** nommé « *log* » dans la classe SuperHero.

L'affichage des détails n'est pas vraiment lié à la méthode « *workOut* » on va dont l'extraire et en faire une méthode :

```
public int workOut(int addStrength)
{
    this.strength += addStrength;
    printDetailsHero();
    return this.strength;
}

private void printDetailsHero()
{
    //Print details about the hero after the work out
    log.info( msg: "name : "+identity);
    log.info( msg: "force : " + strength);
}
```

Voilà, nous avons séparé la responsabilité et cela permet de réutiliser l'affichage des détails d'un héros dans le journal de log.

Tu as vraiment beaucoup appris aujourd'hui ! Tu as pu créer ta propre BD à toi avec tes propres héros et méchants. Et sans t'en rendre compte, tu as appris à coder correctement en Java.

Et maintenant je vais te poser une question. Pourquoi on a commencé à faire des tests tout à l'heure avec JUnit ? Ce n'est pas plus simple de vérifier si nos méthodes fonctionnent en faisant des petits tests nous-même rapidement ? Beaucoup de programmeurs, (comme toi désormais), pensent que c'est une perte de temps. Ils veulent créer le meilleur jeu le plus vite possible. Et du coup quand ils doivent vérifier si quelque chose marche, ils vont juste créer un super-héros, l'entrainer un peu, des fois le faire combattre et ainsi de suite...

Mais à chaque ligne de code que tu rajoutes, il y a un risque que ton code ne marche plus. C'est pour ça, qu'en fait, il est plus rapide au final de rédiger des petites fonctions de test, « d'automatiser » nos tests, comme nous l'avons fait, pour nous assurer que chaque méthode, chaque champ est vérifié. Une seule ligne de code peut tout faire planter !

Alors la prochaine fois que tu diras ça « Oh mais ce n'est qu'une petite méthode, je suis sur qu'elle va marcher, pas besoin de tester », rappelle-toi cela : J'ai rencontré un jour Splinter, un grand personnage qui pour vocation d'entrainer des tortues à devenir des ninjas... mais aussi, à coder très, très bien pendant son temps libre. Et c'est lui qui m'a expliqué ce que je viens de te raconter : « Prend toujours le temps de faire des tests, sinon tu te retrouveras noyé par des ' Pointers Exception ' ».

```
D:\testAgile>javac -cp junit-4.13.jar;. BadGuy.java SuperHero.java testBadGuy.java testSuperHero.java testJUNIT.java

D:\testAgile>java -cp junit-4.13.jar;hamcrest-core-1.3.jar;. org.junit.runner.JUnitCore testSuperHero testJUNIT testBadG
uy
JUnit version 4.13
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
avr. 29, 2020 9:07:21 PM testSuperHero testFightBadGuy
INFO: Test FightBadGuy
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
avr. 29, 2020 9:07:21 PM testSuperHero testWorkOut
INFO: Test WorkOut Start
avr. 29, 2020 9:07:21 PM SuperHero printDetailsHero
INFO: name : Spiderman
avr. 29, 2020 9:07:21 PM SuperHero printDetailsHero
INFO: force : 50
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.avr. 29, 2020 9:07:21 PM testSuperHero setUp
INFO: Before statement
.....
Time: 0,046

OK (12 tests)
```

Pour finir, je vais te citer un adage : « *Si tout semble bien marcher, vous avez forcément négligé quelque chose* ». C'est une variante de la **loi de Murphy.**

En effet, si vos tests sont valides, est-ce que cela signifie pour autant qu'ils contrôlent tous les aspects de votre code ? ou bien qu'une seule partie ?

Parfois les tests de méthodes complexes cachent des fonctionnalités non testées mais qui fonctionnent à un instant t. Cela peut être le constructeur d'une classe qui oublie d'instancier un de ces champs de type Collection par exemple, et vos tests valident cet oubli quand soudain. POUF, un NullPointerException sauvage apparait dans les hautes herbes.