Tests et Junit 4

Bruno Mermet
Licence 3

Dernière mise à jour : Junit 4.12, 11/2014

But et principes de Junit 4

- But :

  Intégrer les tests unitaires au code Java

- Principe

  Chaque classe à tester est doublée d'une classe la testant

  Utilisation des annotations

  Intégration dans Eclipse

  Intégrer junit-4.4.jar dans son CLASSPATH

- Référence

  www.junit.org

Schéma général d'une classe de test

```
import org.junit.Test;
public class NomClasseTestéeTest {
 déclaration var. D'instance
 @Test public void testM1() {...}
 @Test public void testM2() {...}
}
```

La classe Assert

- Utilisation

```
import static org.junit.Assert.*;
```

- Méthodes disponibles
  - Caractéristiques
    - Commencent par "assert"
    - Existent toutes en 2 versions (avec ou sans message)
  - Liste des noms de méthode
    - assertEquals
    - assertTrue/assertFalse
    - assertNull/assertNotNull
    - assertSame/assertNotSame
    - AssertArrayEquals
    - Fail
  - Principe
    - Ces méthodes renvoient une exception (AssertionError) en cas d'échec

Méthodes d'assertion : exemples

- assertEquals

  static void assertEquals(Object attendu, Object effectif)
  static void assertEquals(String message, Object obtenu, Object effectif)
  => la méthode equals doit avoir été surchargée

  static void assertEquals(double expected, double actual, double delta)

- assertNull

  static void assertNull(Object objet)

- assertSame : égalité sur les références

  static void assertSame(Object attendu, Object effectif)

- assertArrayEquals : pour plusieurs types

  static void assertArrayEquals(int[] attendu, int[] effectif)

  static void assertArrayEquals(long[] attendu, long[] effectif)

  static void assertArrayEquals(Object[] attendu, Object[] effectif)

Initialisation avant chaque test

- But

  Faire précéder systématiquement les méthodes de test par une initialisation de certaines données

- Principe
  - Définir une méthode (en général, *setUp*) annotée par `@org.junit.Before` qui effectue cette initialisation
  - Chaque appel à une méthode de test sera précédé par l'appel à la méthode annotée par `@Before`
- Corollaire

  Il existe également une annotation `@org.junit.After` (pour une méthode appelée en général *tearDown*)

- Remarques

  Les méthodes setUp et tearDown doivent être publiques, ne prendre aucun paramètre et ne rien renvoyer

Initialisation avant une série de tests

- But

  Effectuer une initialisation globale (une seule fois) avant tous les tests définis dans une classe

- Principe
  - Définir une méthode annotée par

    `@org.junit.BeforeClass`

  - Cette méthode doit être de classe, publique, ne prendre aucun paramètre et ne rien renvoyer
- Corollaire
  - Il existe aussi l'annotation @org.junit.AfterClass

Test sur la durée d'exécution

- But

  Vérifier qu'une méthode s'exécute dans un délai donné

- Principe

  Spécifier ce délai (en millisecondes) en paramètre de l'annotation @Test

- Exemple

```
@Test(timeout=100)
public void monTest() {…}
```

- Voir aussi
  - Règle Timeout

Spécifier un fonctionnement spécifique de test

- But

  Remplacer le processus de test par défaut par un autre processus de test

- Principe

  Utilisation de l'annotation de classe

  `@org.junit.runner.RunWith`

- Utilisations
  - Définition de suites de test
  - Définition de tests d'une méthode pour différentes valeurs
  - Vérification de « théories »

Suite de tests

- But

  Permettre d'exécuter en une fois des tests spécifiés dans différentes classes de test

- Principe

```
Import org.junit.runner.RunWith;
Import org.junit.runners.Suite;
Import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class)
@SuiteClasses({classe1.class, class2.class, ...})
public class TestGeneral() {}
```

Tests paramétrés (1)

- But
  - tester une méthode pour différentes valeurs
- Principe de base
  - Définir ce test dans une classe annotée avec @RunWith(Parameterized.class)
  - Définir une méthode de classe annotée avec @Parameters renvoyant une Collection de tableaux de paramètres pour le constructeur
  - La/les méthodes de tests seront appelées sur chacune des instances de la classe créées à partir de la collection évoquée ci-dessus

Tests paramétrés (2) : Exemple

```
Import org.junit.runner.RunWith; import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
@RunWith(Parameterized.class)
public class MonTest {
        @Parameters public static Collection<Object []> donnees() {
            // méthode renvoyant une collection de n-uplets
            List<Object[]> retour = new ArrayList<>() ;
            TypeDonnee dp = … ; TypeRes1 r1p = … ; TypeRes2 r2p = … ;
            Object[] cas1 = {dp, r1P, r2p} ; retour.add(cas1) ; …
            return retour ;
        }
        //définition des variables d'instance
        private TypeDonnee donnee ; private TypeRes1 res1 ; private TypeRes2 res2 ;
        public MonTest(TypeDonnee d, TypeRes1 r1, TypeRes2 r2){
           // constructeur à n arguments
           Donnee = d ; res1 = r1 ; res2 = r2 ;
        }
        @Test public void test1() {
          TypeRes1 resEffectif1 = donnee.methode() ;
          assertEquals(res1, resEffectif1) ;
        }
}
```

Tests paramétrés (3)
Utilisation de @Parameter

- Version sans passer par un constructeur
  - Version plus récente
  - Utiliser l'annotation @Parameter avec un paramètre optionnel, value (vaut 0 par défaut)
- Les variables d'instance de la classe de test dans lesquelles doivent être stockées les valeurs de chaque cas de test sont annotées avec @Parameter(value=i) où i précise l'indice de la donnée du cas de test devant être stockée dans la variable en question

Tests paramétrés (4)
Exemple revu et corrigé

```
import ...
@RunWith(Parameterized.class)
public class MonTest {
        @Parameters public static Collection<Object []> donnees() {
           // méthode renvoyant une collection de n-uplets
           List<Object[]> retour = new ArrayList<>() ;
           TypeDonnee dp = … ; TypeRes1 r1p = … ; TypeRes2 r2p = … ;
           Object[] cas1 = {dp, r1P, r2p} ;
           retour.add(cas1) ;
           …
           return retour ;
        }
        //définition des variables d'instance
        @Parameter public TypeDonnee donnee ; // value vaut 0 par défaut
        @Parameter(value = 1) public TypeRes1 res1 ;
        @Parameter(value = 2) public TypeRes2 res2 ;

        @Test public void test1() {
          TypeRes1 resEffectif1 = donnee.methode() ;
          assertEquals(res1, resEffectif1) ;
        }
}
```

Test sur la génération d'exception

- Problème
  On veut vérifier que dans un cas précis, une méthode renverra bien une exception donnée.
- Principe
  Spécifier cette exception en paramètre de l'annotation `@Test`
- Exemple

```
@Test(expected=ArrayOutOfBoundsException.class)
 public void monTest() {...}
```

- Conséquence
  le test `monTest` générera une AssertionError si `monTest` **ne renvoie pas d'exception** de type `ArrayOutOfBoundsException`.
- Voir aussi
  Règle ExpectedException

Assert That (1)
à partir de Junit 4.4

- But

  Permet de pratiquer des tests plus généraux, avec des messages d'erreur plus explicites

- Mise en oeuvre

```
import static org.junit.Assert.*;
import org.hamcrest.core.*;
import static org.hamcrest.CoreMatchers.*;
```

- Méthode

```
static <T> void assertThat(java.lang.String
reason, T actual, org.hamcrest.Matcher<T>
matcher)
```

Assert That Exemples

- AssertThat("text", donneeEffective, is(equalTo(resultatAttendu)))
- AssertThat("text", donneeEffective,
  is(
  either(
  equalTo(res1)
).and(
  equalTo(res2)
)
)
)
- AssertThat("text", monIterable, hasItem(donneeDevantEtrePresente))
- ...

Assume That (Junit 4.4)

- But

  n'effectuer un test que dans certaines conditions (sinon, il n'a pas de sens)

- Mise en oeuvre

```
import static org.junit.Assume.*;
```

  au début de la méthode :

```
assumeThat(Object, Matcher)
```

- Intérêt

  Éviter certains tests non significatifs dans les tests paramétrés et la vérification de théories

Théories (1)

- But

  Spécifier des axiomes (= Theory) et laisser Junit les vérifier sur toute entrée potentielle (= toute variable de classe du bon type) mentionnée dans la classe de test et annotée avec @DataPoint ou générée grâce à une méthode annotée avec @DataPoints

- Exemple

```
@DataPoint public static Point p1 = new Point(1,2) ;
@DataPoint public static Point p2 = new Point(3,5) ;
@Datapoint public static int v1 = 3 ;
@DataPoint public static int v2 = 5 ;
@Theory public static void suffixe(Point p, int x) {
        assertEquals(p, p.translaterX(x).p.translaterX(-x));
}
```

- Annoter la classe avec @RunWith(Theories.class)

Théories
Génération de données avec fonctions

- Principe

  Utiliser l'annotation @DataPoints

- Exemples

```
@DataPoints public static int[] genereInt() {
        return {1, 3, 5} ;
}
@DataPoints public static Couleur[] genereCoul() {
        // Couleur est un type énuméré
        return Couleur.values() ;
}
```

Catégories

- But

  Étiqueter les tests pour filtrer ceux qu'on lance

- Exemples de motif d'utilisation
  - Ne pas toujours lancer les tests prenant du temps
  - Ne pas toujours lancer les tests nécessitant une connexion particulière à un serveur
- Mise en œuvre
  - Définir des interfaces « marqueur » pour chaque catégories de test
  - Étiqueter les méthodes de test (ou classe, notamment pour les tests paramétrés) avec l'annotation @Category({Cat1.class, Cat2.class})
  - Définir une classe de test annotée avec
    - @RunWith(Categories.class)
    - @IncludeCategory(categoriesAInclure.class)
    - @ExcludeCategory(categoriesAExclure.class)

Règles (1) (Junit ≥ 4.7)

- Késako ?

  Variables d'instance avec une annotation particulière permettant de paramétrer/enrichir le fonctionnement du moteur Junit

- Utilisation

  @Rule public TypeRegle nom = valeur

- Premier exemple
  - Contrôler le temps d'exécution de chacun des tests
    - @Rule public Timeout limite = new Timeout(500, TimeUnit.MILLISECONDS) ;
    - @Rule public Timeout limite = new Timeout(2, TimeUnit.SECONDS)
  - Pour désactiver le contrôle en mode « debug »

    @Rule public TestRule timeout = new DisableOnDebug(new Timeout(20));

Règles (2) : Contrôler finement les exceptions levées

- Problème

  L'utilisation du paramètre *expected* de l'annotation @*Test* permet juste de vérifier qu'une exception a bien été levée, mais pas d'autre contrôle possible

- Solution
  - Définir dans la classe de test une règle

    ```
    @Rule public ExpectedException levee =
      ExpectedException.none() ;
    ```

  - Dans une méthode devant générer une exception, avant le code levant l'exception, paramétrer l'objet levee :

    ```
    levee.expect(ArraysIndexOutOfBounds.class)
    levee.expectMessage(containsString("toto"))
    ...
    ```

Règles (3) : Accéder au nom d'une méthode de test

- Problème

  Avoir accès de manière sûre dans une méthode de test au nom de la méthode

- Solution
  - Définir dans la classe de test une règle

    ```
    @Rule public TestName nom = new TestName() ;
    ```
  - Dans une méthode de test, pour afficher son nom, faire :

    ```
    System.out.println(nom.getMethodName()) ;
    ```

Règles (4) : Créer un répertoire temporaire pour les tests

- Problème

  Pendant les tests, on a besoin de générer des fichiers, mais ceux-ci doivent être effacer ensuite

- Solution
  - Définir dans la classe de test une des règles :

    ```
    @Rule public TemporaryFolder repTest = new
      TemporaryFolder() ;
    ```

    ```
    @Rule public TemporaryFolder repTest = new
      TemporaryFolder(repParent) ;
    ```

  - Dans une méthode de test, pour créer un fichier/répertoire, faire :

    ```
    repTest.newFile() ; repTest.newFile(nomFic)
    ```

    ```
    repTest.newFolder() ; repTest.newFolder(nomRep)
    ```

Règles (5) : Utiliser des ressources externes pour les tests

- Problème

  Pendant les tests, on a besoin d'allouer des ressources, à libérer à la fin (flux, connexions à des bases de données, etc.)

- Solution

  Créer une règle définissant une classe héritant de *ExternalResource* et redéfinissant les méthodes *before*() et *after*() ainsi :

```
@Rule public ExternalResource resource= new
ExternalResource() {
    @Override
    protected void before() throws Throwable {
        // Allocation ressource
    };
    @Override
    protected void after() {
        // Désallocation ressource
    };
};
```

Règles (6) : Collecter les erreurs

- Problème

  Dans une même méthode de test, on souhaite effectuer plusieurs vérification. Mais la première erreur empêchera les autres vérifications d'être exécutées

- Solution
  - Créer une règle définissant ErrorCollector
  - Dans les méthodes de test, utiliser la méthode checkThat du collecteur d'erreur (fonctionnement type assertThat)
  - Si des vérifications génères des erreurs, les erreurs sont stockées dans le collecteur ; le test sera considéré comme échoué et les différents problèmes seront reportés

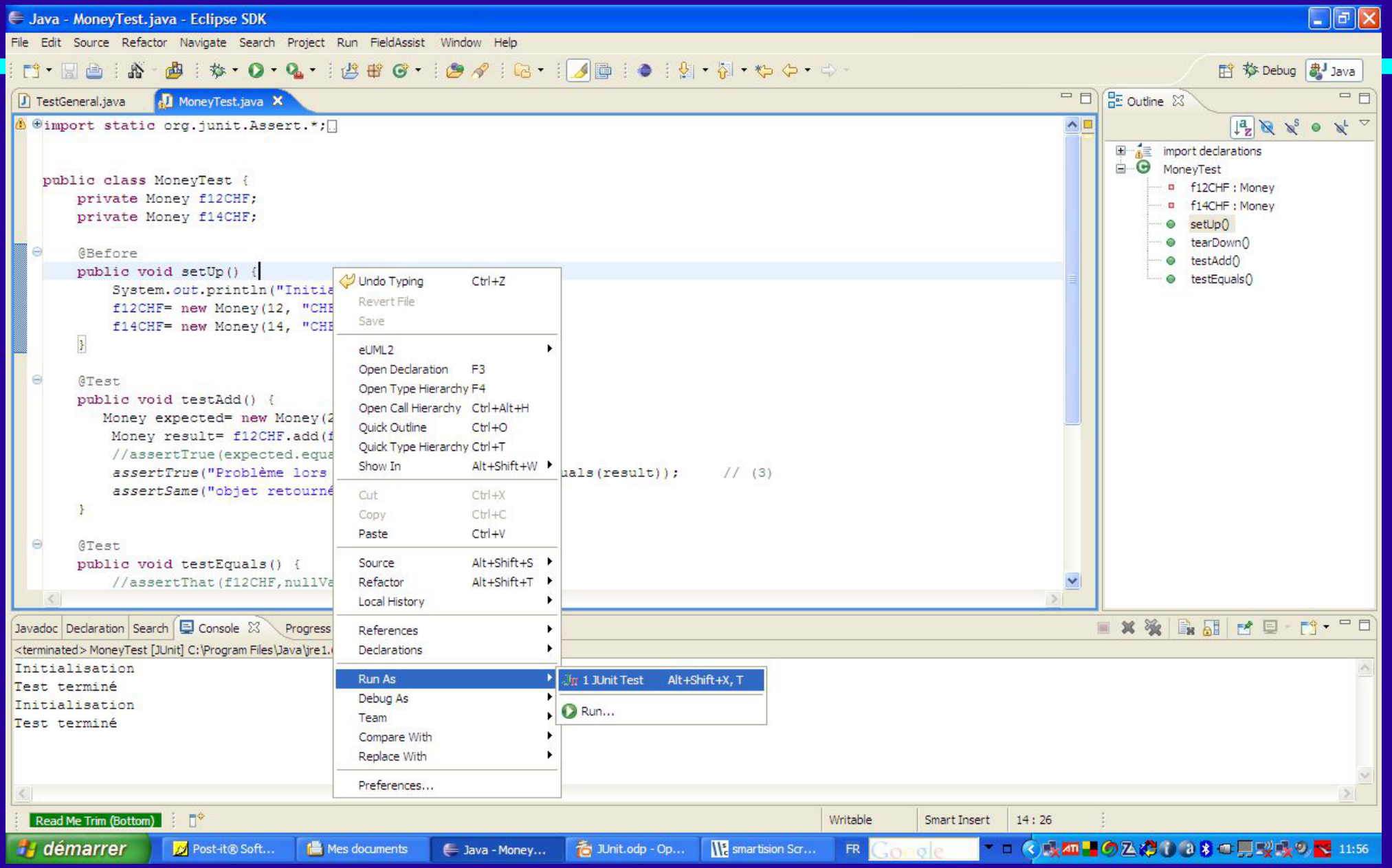
Règles (7) : Extensibilité

Définir ses propres règles

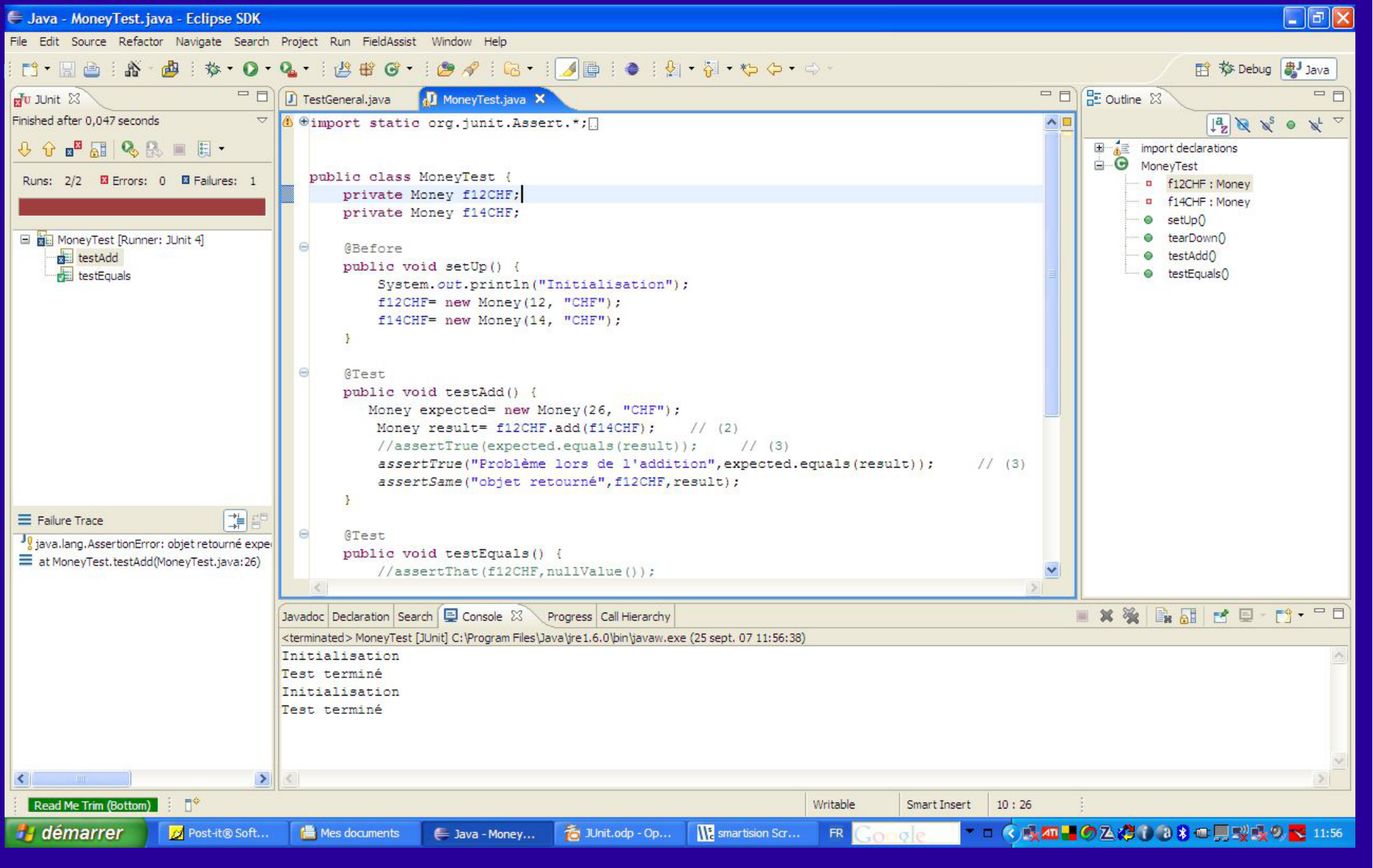
Il existe plusieurs classes de base dont on peut hériter :

- Stopwatch
  - Pour réagir par rapport à une violation ou non de la durée d'exécution
- TestWatcher
  - Pour réagir après chaque test en fonction de son déroulement
- Verifier
  - Pour rajouter une vérification supplémentaire à la fin de chaque test

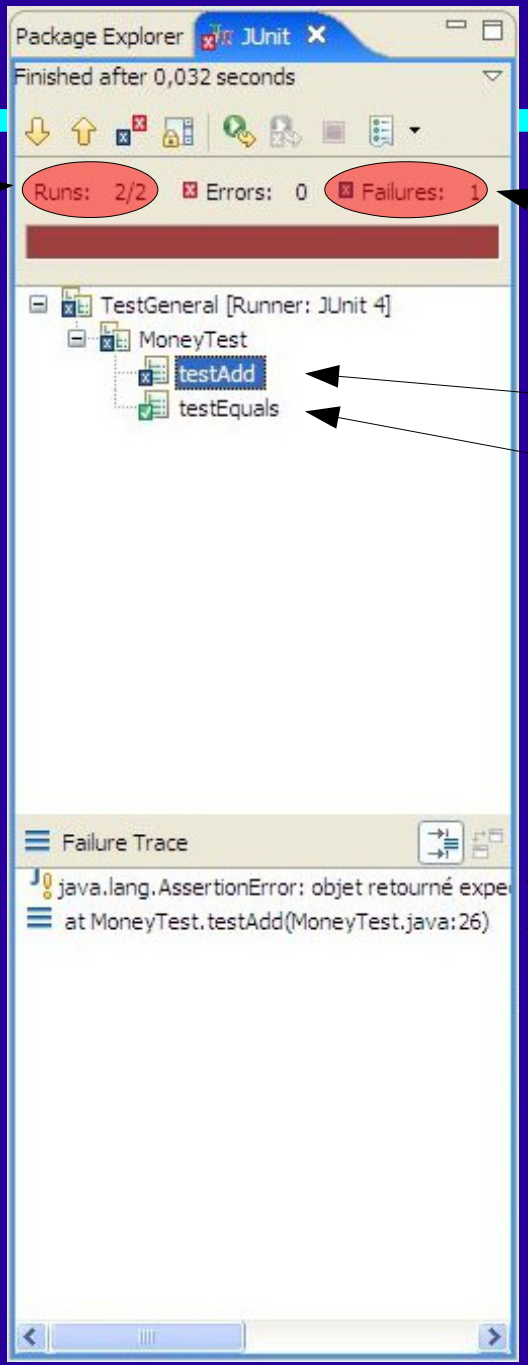
Intégration dans *Eclipse* (1)



Intégration dans *Eclipse* (2)



Intégration dans *Eclipse* (3)



Tests effectués, que ce soit sur succès ou échec

Hiérarchie des tests

Détail sur le test raté sélectionné

Nombre de tests avec échec

Test raté

Test réussi

Description de l'erreur

Ligne ayant généré l'erreur

Lancement automatique des tests nécessaires

- Installer le plugin éclipse Infinitest
  - http://infinitest.github.io
- Utilisation
  - À chaque sauvegarde, les tests susceptibles d'être impliqués par les modifications sont ré-exécutés
  - Les lignes des tests ayant échoué (ainsi que les classes contenant ces lignes) sont étiquetées comme erronée (croix rouge) dans les différentes fenêtres d'Eclipse
- Éviter de lancer automatiquement certains tests
  - Créer un fichier infinitest.filters (voir doc.) à la racine du projet
- Remarque : gestion moyenne des « ErrorCollector »