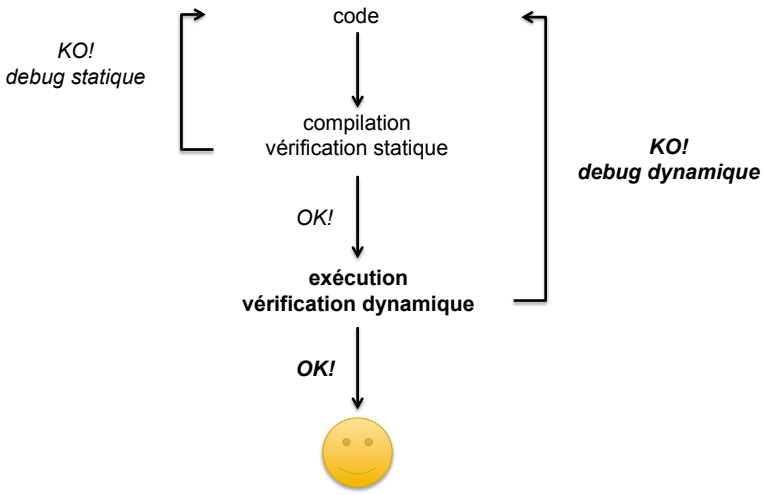
Programmation Par Objets

Tests unitaires
JUnit

Démarche de développement



Zoom sur la vérification dynamique

Vérifier que le logiciel :

A. s'exécute sans « planter » (`''exit on error''`)

   => debug des erreurs d'exécution détectées par le langage

B. produit les **résultats attendus**

- ❑ par le client en réponse à ses *spécifications fonctionnelles*
- ❑ indétectables par le langage
- ❑ d'où la responsabilité du concepteur/développeur de tester les cas d'erreurs possibles

=> notion de ***test*** étudiée ici.

debug vs. test : exemple

On reprend l'exemple des comptes bancaires.

Spécification:
« Réinitialiser » un compte consiste à reporter son solde en crédit ou en débit selon qu'il est positif (créditeur) ou négatif (débiteur).

Version 0 : le code vide …

```
public class Compte {
  public void reinit() {}
}
```

A. ne plante pas (le code vide ne plante jamais !)
B. mais ne produit rien.

C'est toute la différence, à l'extrême, entre debug et test.

debug vs. test : exemple

Version 1, un peu plus sérieuse

```
public class Compte {
  public void reinit() {
    if (solde()>0) // compte crediteur
      credits = solde();
    else // compte debiteur
      debits = solde();
  }
}
```

A. ne plante pas
B. mais ne produit pas les résultats attendus …

debug vs. test : exemple

- Situation 1 : que du crédit

```
Compte compte = new Compte();
compte.crediter(300.0);
```

état avant

```
compte.getCredit() -> 300.0
compte.getDebit() -> 0.0
compte.solde() -> 300.0
```

OK

- Action

```
compte.reinit();
```

état après

```
compte.getCredit() -> 300.0
compte.getDebit() -> 0.0
compte.solde() -> 300.0
```

OK

- Tests

1. le compte était créditeur et le solde a bien été reporté au crédit

OK

debug vs. test : exemple

- Situation 1 : que du crédit

```
Compte compte = new Compte();
compte.crediter(300.0);
```

état avant

```
compte.getCredit() -> 300.0
compte.getDebit() -> 0.0
compte.solde() -> 300.0
```

OK

- Action

```
compte.reinit();
```

état après

```
compte.getCredit() -> 300.0
compte.getDebit() -> 0.0
compte.solde() -> 300.0
```

OK

- Tests

1. le compte était créditeur et le solde a bien été reporté au crédit
2. le solde n'a pas changé

OK

debug vs. test : exemple

- Situation 1 : que du crédit

```
Compte compte = new Compte();
compte.crediter(300.0);
```

état avant

```
compte.getCredit() -> 300.0
compte.getDebit() -> 0.0
compte.solde() -> 300.0
```

- Action

```
compte.reinit();
```

état après

```
compte.getCredit() -> 300.0
compte.getDebit() -> 0.0
compte.solde() -> 300.0
```

- Tests
  - 1. le compte était créditeur et le solde a bien été reporté au crédit
  - 2. le solde n'a pas changé.
- Verdict : OK!

debug vs. test : exemple

- Situation 2 : crédits, débits, compte créditeur

```
// idem +
compte.debiter(50.0);
```

état avant

```
compte.getCredit() -> 300.0
compte.getDebit() -> 50.0
compte.solde() -> 250.0
```

OK

- Action

```
compte.reinit();
```

état après

```
compte.getCredit() -> 250.0
compte.getDebit() -> 50.0
compte.solde() -> 200.0
```

OK

- Tests

3. le compte était créditeur et le solde a bien été reporté au crédit

OK

debug vs. test : exemple

- Situation 2 : crédits, débits, compte créditeur

```
// idem +
compte.debiter(50.0);
```

état avant

```
compte.getCredit() -> 300.0
compte.getDebit() -> 50.0
compte.solde() -> 250.0
```

bug

- Action

```
compte.reinit();
```

état après

```
compte.getCredit() -> 250.0
compte.getDebit() -> 50.0
compte.solde() -> 200.0
```

bug

- Tests

3. le compte était créditeur et le solde a bien été reporté au crédit
4. mais le solde a changé

bug

- Verdict : KO!

debug vs. test : exemple

Correction :
On a oublié de réinitialiser le débit ou le crédit à 0 selon le cas (compte créditeur ou débiteur).

Version 2 :

```
public void reinit() {
  if (solde()>0) // compte crediteur
    {credits = solde(); debits = 0.0;}
  else // compte debiteur
    {debits = solde(); credits = 0.0;}
}
```

- Tests 1 et 2 : toujours OK (mais à confirmer)
- Tests 3 et 4 …

debug vs. test : exemple

- Situation 2 : crédits, débits, compte créditeur

```
// idem +
compte.debiter(50.0);
```

état avant

```
compte.getCredit() -> 300.0
compte.getDebit() -> 50.0
compte.solde() -> 250.0
```

OK

- Action

```
compte.reinit();
```

état après

```
compte.getCredit() -> 250.0
compte.getDebit() -> 0.0
compte.solde() -> 250.0
```

OK

- Tests

3. le compte était créditeur et le solde a bien été reporté au crédit

OK

debug vs. test : exemple

- Situation 2 : crédits, débits, compte créditeur

```
// idem +
compte.debiter(50.0);
```

état avant

```
compte.getCredit() -> 300.0
compte.getDebit() -> 50.0
compte.solde() -> 250.0
```

OK

- Action

```
compte.reinit();
```

état après

```
compte.getCredit() -> 250.0
compte.getDebit() -> 0.0 // ce qui a changé
compte.solde() -> 250.0
```

OK

- Tests

3. le compte était créditeur et le solde a bien été reporté au crédit
4. le solde n'a pas changé

OK

- Verdict : OK!

Comment faire ces tests ?

Manuellement

- exécuter
- en entrant les données à la main
- vérifier de tête ou sur un bout de papier (!) que les résultats sont ceux attendus
- recommencer après chaque modification du code: correction, ajout, évolution, …
- laborieux !

Comment faire ces tests ?

Mieux, par programme : notion d'**oracle**

- `main` de tests
- qui code « en dur » les situations à tester selon le schéma de code = *procédure de test* :

```
// situation
// action…
// verification = oracle
if( resultats = attendus )
   print(``OK!``)
else
   print(``KO! blabla de compte-rendu ``)
```


Comment faire ces tests ?

Mais …

- schéma de code **répétitif**
- **non-indépendance** des procédures de test
  - due à la **séquentialité** du programme de tests
  - rend chaque test dépendant des précédents
  - à l'exécution : risques d'effets de bord
  - ajout/suppression/modification de tests
    = modification du programme
    avec risque d'erreurs de programmation (on boucle!)

Besoin d'automatisation …

Besoin d'automatisation

- Faciliter l'expression des tests
  - constructions dédiées
  - notamment pour l'expression des ***oracles*** :

```
// verification (oracle)
if( resultats = attendus )
    print(``OK!``)
else
    print(``KO! blabla de compte-rendu ``)
```

  - remplacée par des ***assertions*** (déclarations) du genre:

```
assert(resultat = attendus)
```

  - avec prise en charge automatique:
    - résultat : OK! / KO!
    - génération automatique de **compte-rendu.**

Besoins d'automatisation

- Indépendance
  - pouvoir les formuler et les exécuter isolément (sans effets de bord)
  - notion de ***test unitaire exécutable***
  - peu importe leur ordre d'exécution
  - pouvoir en ajouter/supprimer/ignorer unitairement et rejouer …
- Pouvoir facilement (re-)jouer les tests ... tout le temps !
  - à chaque modification/correction de code
  - lors d'ajout de fonctionnalités (classes, méthodes) : tests incrémentaux
  - lors de changement complet d'implémentation à fonctionnalités constantes
- Principe de ***non régression*** :
  « ce qui fonctionnait fonctionne toujours »

C'est l'apport de la technologie des XUnit, ***JUnit*** *en Java.*

JUnit

- www.junit.org
- Technologie d'automatisation de tests unitaires inventée en 1997 par K. Bleck (*Smalltalk/SUnit*) et E. Gamma après l'avènement de Java
- puis reprise dans de nombreux langages (famille logicielle des *XUnit*):
  CppUnit, *Cunit*, *PHPUnit*, ...
- et intégrée dans de nombreux outils de développemment intégré (IDE) ou de gestion de projets tels que Eclipse: www.eclipse.org

Assertions

- méthodes `static` de la classe **`org.junit.Assert`**
  - `assert[Not]Equals(expected value, actual value)`
  - `assertTrue(boolean condition) / assertFalse`
  - `assert[Not]Same(expected object, actual object)`
    `// par ==`
  - `assertArrayEquals(Type[] expecteds, Type[] actuals)`
    `// par equals itéré`
  - …
  - qui provoquent une exception `AssertionError` si l'assertion n'est pas vérifiée (KO!)
  - capturée par JUnit pour générer un compte-rendu automatique, fonction des paramètres.

Procédures de test

- à programmer dans de simples classes dites ***classes de test***
- sous la forme de ***méthodes annotées*** par **`@Test`**
- exploitant le jeu d'**assertions** précédentes.
- les classes et méthodes de test doivent être **public** pour être visibles de JUNIT, notamment de son moteur d'exécution (`org.junit.runner`)
- bonne pratique:
  les regrouper dans un package `test`

sur l'exemple…


```
package test;
import org.junit.*;
import banque.*;

public class CompteTest { // classe de test
   @Test //procedure de test 1: que du credit
   public void testReinit_1() {
      // situation
      Compte compte = new Compte(); compte.crediter(300.0);
      // action
      compte.reinit();
      // oracle
      Assert.assertEquals(300.0, compte.getCredit(), 0.0);
   }
   @Test public void testReinit_2() //test 2 …
   @Test public void testReinit_3() //test 3 …
   @Test //test 4: credits, debits, compte crediteur, solde?
   public void testReinit_4() {
      Compte compte = new Compte(); compte.crediter(300.0);
      compte.debiter(50.0);
      compte.reinit();
      Assert.assertEquals(250.0, compte.solde(), 0.0);
}}
```

Jouer les tests

- dans un terminal (quelque soit l'outillage utilisé) moyennant l'ajout de la lib **`junit.jar`**
  - compiler les classes de test

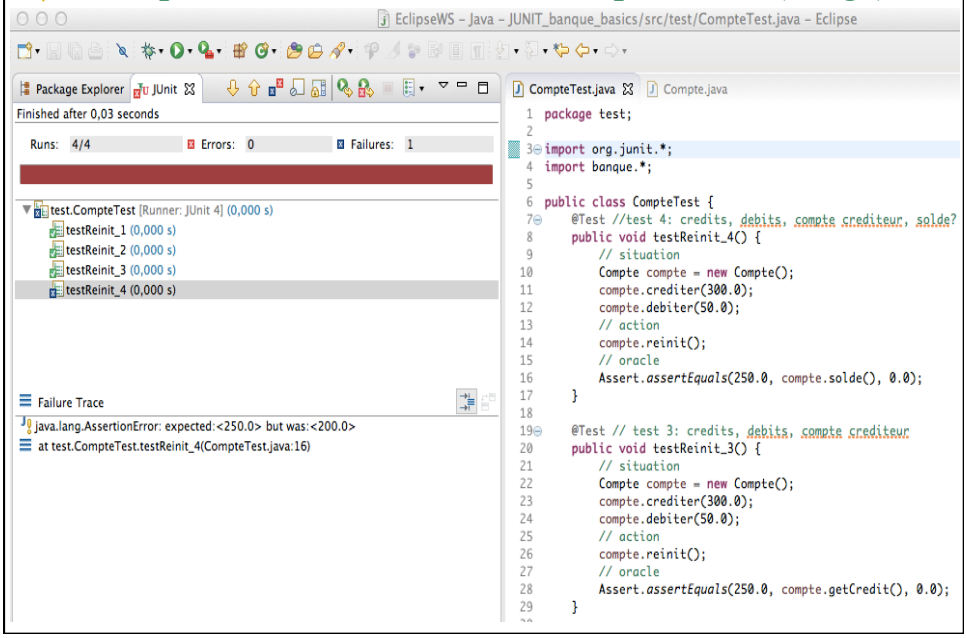
```
javac
    –cp <path to junit.jar>
    test.CompteTest.java
```

  - exécuter

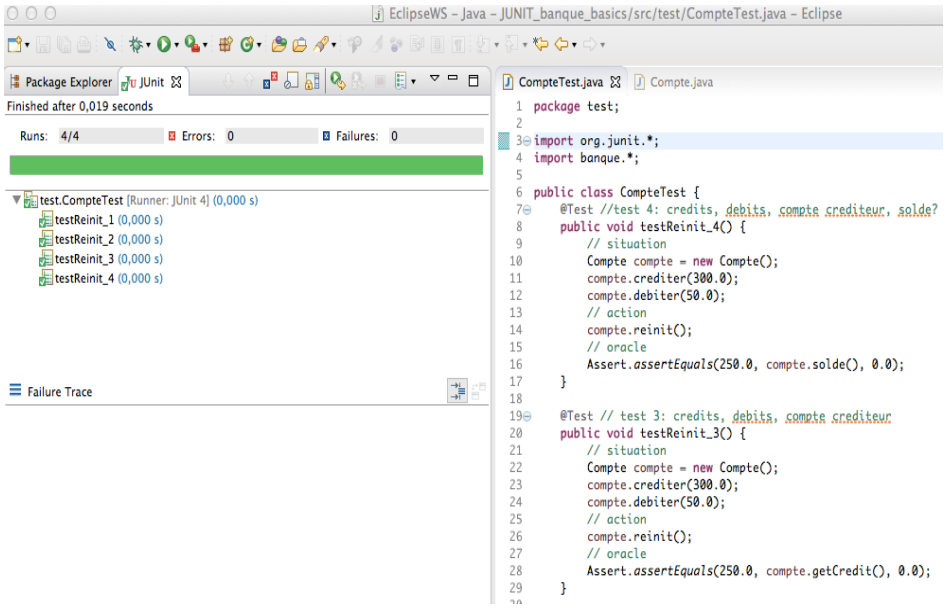
```
java
    –cp <path to junit.jar>
    org.junit.runner.JUnitCore
    test.CompteTest
```

- ou dans les outils (IDE) intégrant JUnit tels qu'Eclipse …

Exemple sur V1 de la classe Compte : KO (rouge)!

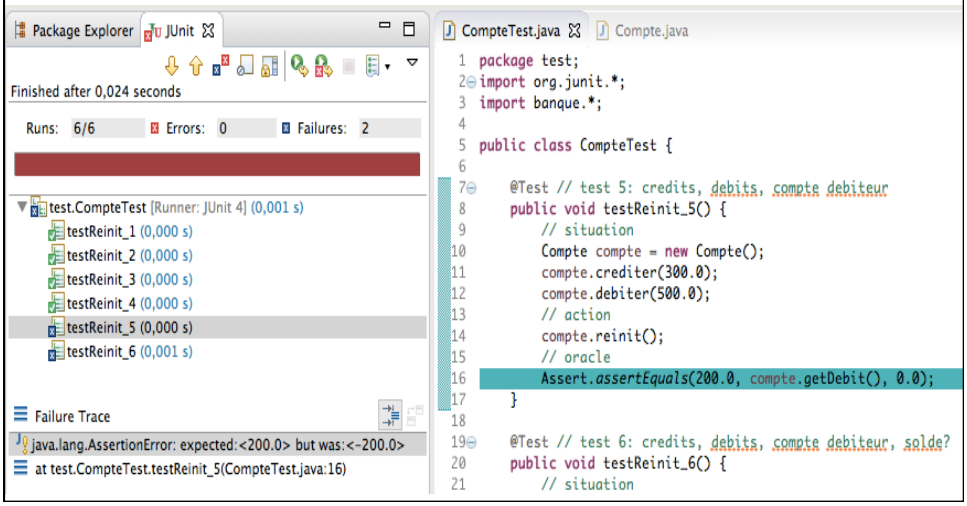


Rejouer après correction (V2) : OK (vert)!



JUnit : indépendance des tests

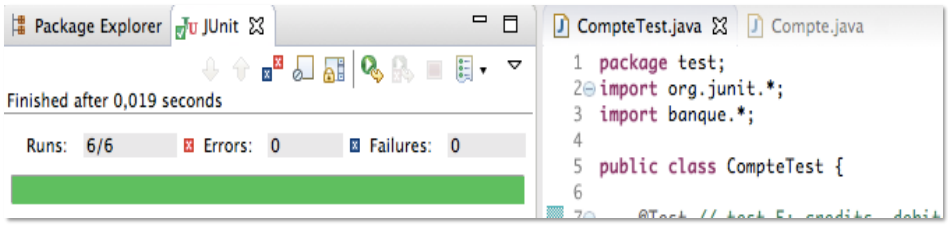
- ajouter/supprimer/modifier des tests unitairement
- exemple : tester aussi l'état débiteur



OOPS! Corriger …

```
public void reinit() { // Version 3
  if (solde()>0) // compte crediteur
    {credits = solde(); debits = 0.0;}
  else // compte debiteur
    {debits = -solde(); credits = 0.0;}
}
```

… et rejouer.



Bien remarquer la **non-régression**.

Pour aller plus loin avec JUnit

- Factoriser les situations
  notion de « fixture » : **`@Before`**
- Test d'exception
- Test et héritage
- Regrouper des classes de test
  notion de ***suite de tests***

Factoriser les situations

- on constate que les tests partagent souvent des conditions de mise en situation :

```
// situation
Compte compte = new Compte();
compte.crediter(300.0);
```

- il est possible de les factoriser grâce à la notion de « fixture » de JUnit (« installation ») :
  - méthode dédiée annotée **`@Before`**
  - convention de nommage : **`setUp()`**
  - exécutée **avant chaque méthode de test**

sur l'exemple…


```
public class CompteTest {
   Compte compte;
   @Before // mise en situation partagee
   public void setUp() {
      compte = new Compte();
      compte.crediter(300.0);
   }
   @Test //test 1 : que du credit
   public void testReinit_1() {
      // situation = celle de @Before
      // action puis oracle … }
   @Test // test 2 : que du credit, vérif solde
   public void testReinit_2() {
      // situation = celle de @Before
      // action puis oracle … }
   @Test // test 3: credits, debits, compte crediteur
   public void testReinit_3() {
      // situation = celle de @Before + complement :
      compte.debiter(50.0);
      // action puis oracle … }
...}
```

Test d'exception

- Il est possible de tester qu'une exception doit être provoquée
- Compléter l'annotation `@Test` de la méthode de test par:

  `@Test(expected=ExceptionName.class)`
- Le test sera alors :
  - valide si l'exception `ExceptionName` est provoquée
  - invalide sinon.

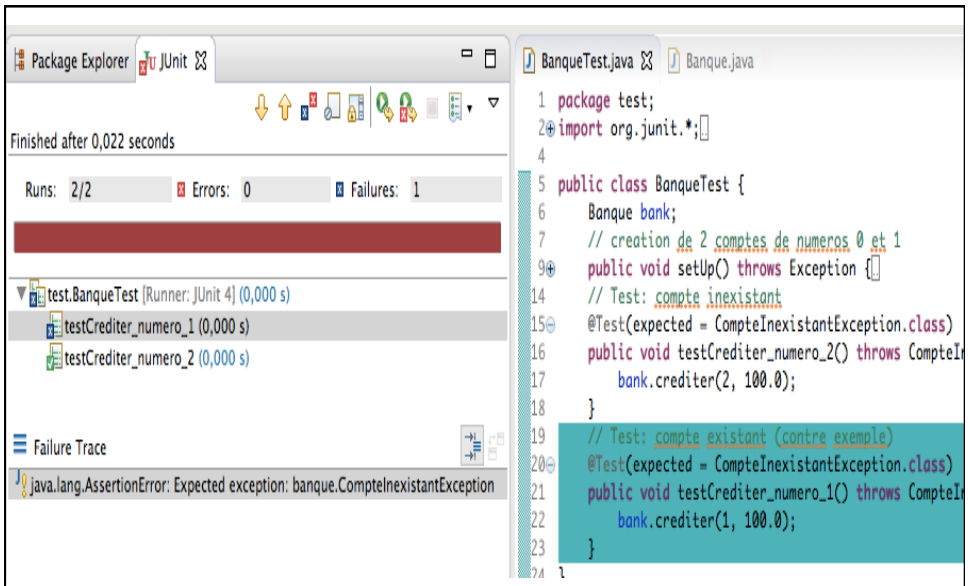
Exemple…


```
public class BanqueTest {
 Banque bank = new Banque();

 // creation de 2 comptes de numeros 0 et 1
 @Before public void setUp() {
    bank.creerCompte(); bank.creerCompte();}

 // Test: compte numero 2 inexistant
 @Test(expected = CompteInexistantException.class)
 public void testCrediter_numero_2()
   throws CompteInexistantException {
     bank.crediter(2, 100.0);
 }

 // Test: compte numero 1 existant (contre exemple)
 @Test(expected = CompteInexistantException.class)
 public void testCrediter_numero_1()
   throws CompteInexistantException {
     bank.crediter(1, 100.0);
 }
...}
```



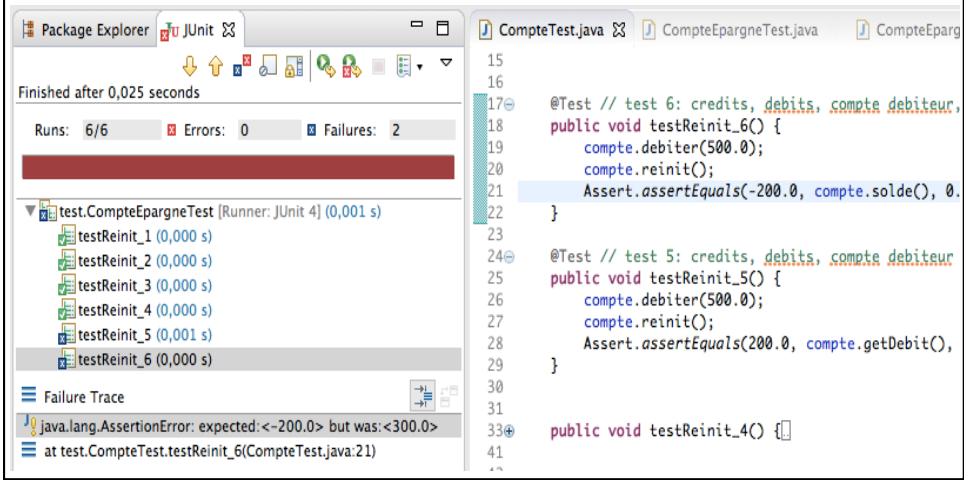
Test et héritage

- les tests étant formulés dans des classes de plein droit il est possible d'en hériter
- typiquement: en parallèle à une hiérarchie de classes du logiciel
- prendre soin des redéfinitions.

Exemple

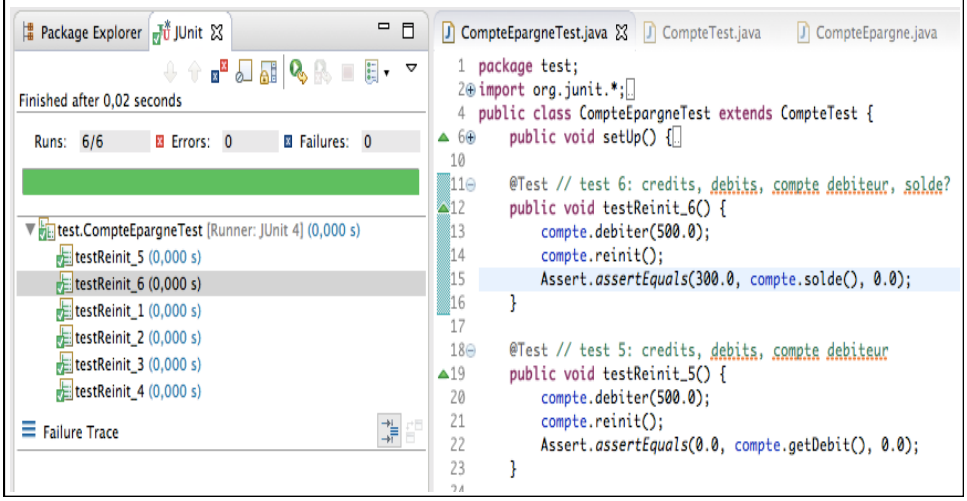
- on considère la sous-classe `CompteEpargne` de `Compte`, sur laquelle effectuer les tests précédents
- pour rappel
  - un compte épargne a des intérêts
  - il ne peut être débiteur.

```
public class CompteEpargneTest extends CompteTest {
   @Before
   public void setUp() {
      compte = new CompteEpargne();
      compte.crediter(300.0);
}}
```

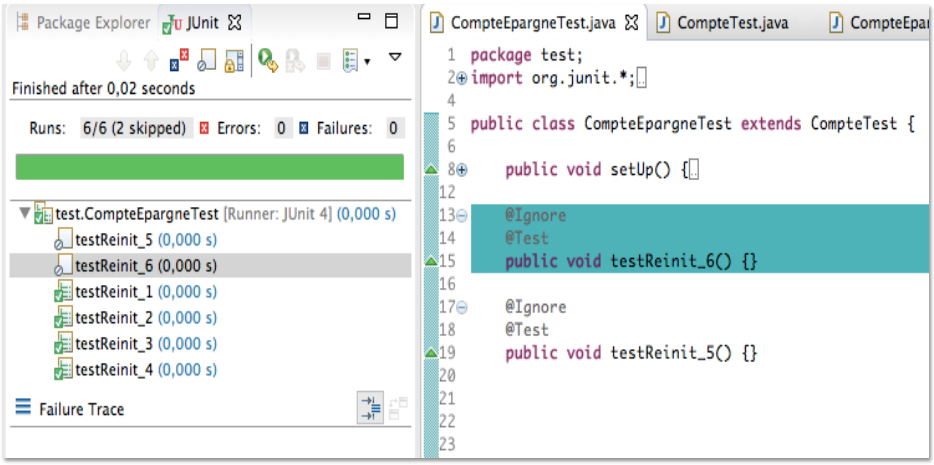


1ère solution : ***redéfinir*** ces tests en respect des spécificités de la sous-classe

Sur l'exemple: le débit ne doit pas se faire, devant laisser le compte épargne en l'état (tests 5 et 6).



2ème solution : ignorer ces tests par l'annotation `@Ignore`



- peu satisfaisant (« anti-héritage »)
- toujours préférer de **maximiser les tests** (et non de les contourner)

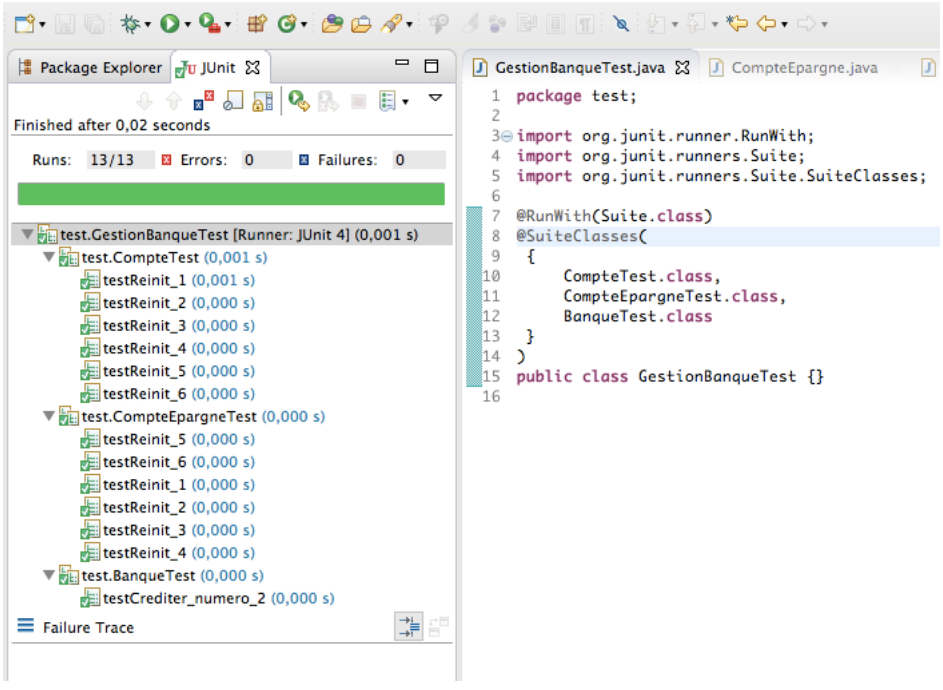
Un mot de plus sur `@Ignore`

- au delà de son utilisation dans la situation précédente (pour l'« anti-héritage » de méthodes de test)
- cette technique est plus généralement utile en phase intermédiaire de développement
  - quand on prévoit les tests à effectuer vis à vis des spécifications de fonctionnalités
  - mais que ces fonctionnalités ne sont que partiellement réalisées, certaines n'étant donc pas testables.
  - méthodologie « write tests first »

Regrouper des classes de test

- un logiciel comporte en général un nombre important de classes/méthodes à tester
- et donc un nombre important de classes de test
- JUnit permet de les exécuter ensemble en les regroupant sous la forme de ***suite de tests***
  - classe annotée `@SuiteClasses`
  - recensant les classes de test concernées
  - attention: il est nécessaire de changer le `runner` de Junit (cf. `RunWith`).

Exemple :
regrouper `CompteTest, CompteEpargneTest, BanqueTest`en une seule ***suite de test*** `GestionBanqueTest`



Conclusion

- démarche itérative
  - écrire les tests progressivement et en même temps que le code (allers-retours)
- « maximiser » les tests
  - peut paraître fastidieux dans un premier temps
  - mais investissement : les tests écrits sont rejouables indéfiniment et sans effort après toute modification du code : correction, ajout de fonctionnalités, changement de choix d'implem (versions) …
  - *Scénario*: Remplacer dans la hiérarchie de classes la classe `Compte` primitive par une version plus élaborée (historisation des opérations de crédits/débits par exemple). La suite de classes de tests est écrite, il suffit de les rejouer.