Cours JAVA : La fiabilité en Java

Version 1.02

Julien Sopena[1]

[1]julien.sopena@lip6.fr
Équipe RÉGAL - INRIA Rocquencourt
LIP6 - Université Pierre et Marie Curie

Licence professionnelle DANT - 2015/2016

Grandes lignes du cours

Outline

La fiabilité

Tout programme comporte des erreurs (**bugs**) ou est susceptible de générer des erreurs (*e.g* suite à une action de l'utilisateur, de l'environnement, etc ...).

La **fiabilité** d'un logiciel peut se décomposer en deux grandes propriétés :

1. la **robustesse** qui est la capacité du logiciel à continuer de fonctionner en présence d'événements exceptionnels tels que la saisie d'informations erronées par l'utilisateur ;
2. la **correction** qui est la capacité d'un logiciel à donner des résultats corrects lorsqu'il fonctionne normalement.

Assurer la fiabilité en Java

Le langage Java inclus plusieurs mécanismes permettant d'améliorer la fiabilité des programmes :

- les **exceptions** pour la robustesse ;
- les **assertions** pour la correction.

A ces mécanismes viennent s'ajouter des outils complémentaires tels :

- des outils de test unitaire comme **JUnit** ;
- des outils de debuggage comme **jdb**.

Outline

Comment traîter les erreurs ?

Comme en C, une solution consiste dans un premier temps à :

- prévoir les erreurs possibles
- définir un système de codes d'erreurs, c'est-à-dire un système permettant de retourner des valeurs spécifiques pour signaler un fonctionnement anormal de l'application.

Toutefois cette solution est loin d'être satisfaisante car elle rend difficile la maintenance du programme (à cause d'une trop grande imbrication de tests conditionnels (if .. else) par exemple).

La gestion d'erreurs en `Java`.

Le langage Java propose un mécanisme particulier pour gérer les erreurs : **les exceptions**. Ce mécanisme repose sur deux principes :

1. Les différents types d'erreurs sont modélisées par des classes.
2. Les instructions susceptibles de générer des erreurs sont séparées du traitement de ces erreurs : concept de **bloc d'essai** et de **bloc de traitement d'erreur**.

Les exceptions : définition.

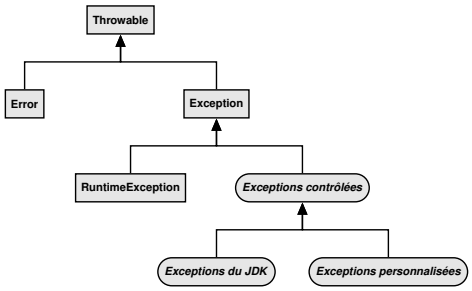
Définition

*Le terme **exception** désigne tout événement arrivant durant l'exécution d'un programme interrompant son fonctionnement normal.*

En java, les exceptions sont matérialisées par des instances de classes héritant de la classe **java.lang.Throwable**.

A chaque évènement correspond une sous-classe précise, ce qui peut permettre d'y associer un traitement approprié.

Arbre d'héritage des exceptions.



Outline

Les blocs : try.

Définition

*La clause **try** s'applique à un bloc d'instructions correspondant au fonctionnement normal mais pouvant générer des erreurs.*

```
try {
 ...
 ...
}
```

Attention

Un bloc **try** ne compile pas si aucune de ses instructions n'est susceptible de lancer une exception.

Les blocs : catch.

Définition

*La clause **catch** s'applique à un bloc d'instructions définissant le traitement d'un type d'erreur. Ce traitement sera lancé sur une instance de la classe d'exception passée en paramètre.*

```
try{
  ...
}
catch(TypeErreur1 e) {
  ...
}
catch(TypeErreur2 e) {
  ...
}
```


Règles sur les blocs : try-catch.

Si les blocs **try** et **catch** correspondent à deux types de traitement (resp. normal et erreur), ils n'en sont pas moins liés. Ainsi :

- Tout bloc **try** doit être suivit par au moins un bloc **catch** ou par un bloc **finally** (étudié plus loin).
- Tout bloc **catch** doit être précédé par un autre bloc **catch** ou par un bloc **try**.

Définition

*Un ensemble composé d'un bloc **try** et d'au moins un bloc **catch** est communément appelé bloc **try-catch**.*

Lancer et lever des exceptions : définitions.

Définition

*Lorsqu'une instruction du bloc d'essai génère une erreur et y associe une exception, on dit qu'elle **lève** (**lance**) cette exception.*

Définition

*Lorsqu'un bloc de traitement d'erreur est déclenché par une exception, on dit qu'il **traite** (**capture**) cette exception.*

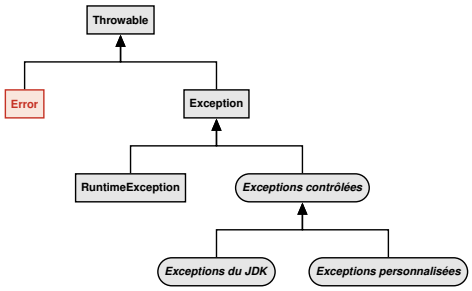
Les bloc try-catch : fonctionnement.

Le fonctionnement d'un bloc **try-catch** est le suivant :

1. si aucune des instructions du bloc d'essai ne lance d'exception, il est entièrement exécuté et les blocs de traitement d'erreur sont ignorés.
2. si une des instructions du bloc d'essai lance une exception, alors toutes les instructions du bloc d'essai après elle sont ignorées et le premier bloc de traitement d'erreur correspondant au type d'exception lancée. Tous les autres blocs de traitement d'erreur sont ignorés.

Outline

Les exceptions de type Error.



Les bloc try-catch : fonctionnement.

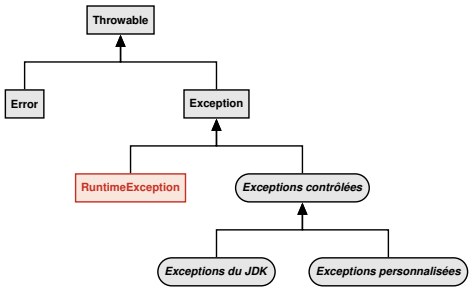
Définition

*Les exceptions de type **Error** sont réservées aux erreurs qui surviennent dans le fonctionnement de la JVM. Elles peuvent survenir dans toutes les portions du codes.*

`Java` définit de nombreuses sous-classes de `Error` :

- `OutOfMemoryError` : survient lorsque la machine virtuelle n'a plus de place pour faire une allocation et que le *GC* ne peut en libérer.
- `NoSuchMethodError` : survient lorsque la machine virtuelle ne peut trouver l'implémentation de la méthode appelée.
- `StackOverflowError` : survient lorsque la pile déborde après une série d'appel récursif trop profond.
- etc...

Les exceptions de type RuntimeException.



Les exceptions de type RuntimeException.

Définition

*Les exceptions de type **RuntimeException** correspondent à des erreurs qui peuvent survenir dans toutes les portions du codes.*

`Javadéfinit` de nombreuses sous-classes de `RuntimeException` :

- `ArithmeticException` : division par zéro (entiers), etc ....
- `IndexOutOfBoundsException` : dépassement d'indice dans un tableau.
- `NullPointerException` : référence **null** alors qu'on attendait une référence vers une instance.
- etc...

Remarque(s)

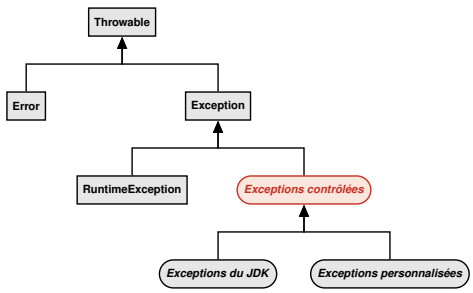
Attention `1.0d/0.0d` renvoie `Infinity` !

RuntimeException : exemple.

```
int a = 0;
try {
  int x = 1 /a;
  System.out.println(" x = " + x);
}
catch (ArithmeticException e) {
  System.out.println("division par 0 : 1 / " + a);
}
```

```
division par 0 : 1 / 0
```


Les exceptions contrôlées.



Les exceptions contrôlées.

Définition

*On appelle **exception contrôlée**, toute exception qui hérite de la classe `Exception` et qui n'est pas une `RuntimeException`. Elle est dite contrôlée car le compilateur vérifie que toutes les méthodes l'utilisent correctement.*

Le JDK définit de nombreuses exceptions :

- `EOFException` : fin de fichier.
- `FileNotFoundException` : erreur dans l'ouverture d'un fichier.
- `ClassNotFoundException` : erreur dans le chargement d'une classe.
- etc...

Le contrôle des exceptions : deux solutions.

Toute exception contrôlée, du JDK ou personnalisée, pouvant être émise dans une méthode doit être :

- soit levée dans cette méthode. Elle est alors lancée dans un bloc try auquel est associé un catch lui correspondant.
- soit être indiquées dans le prototype de la méthode à l'aide du mot clé **throws**.

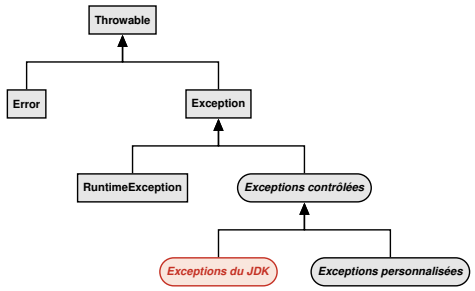
Le contrôle des exceptions : traitement local.

```
public void f() {
  try {
    FileInputStream monFichier ;
    // Ouvrir un fichier peut générer une exception
    monFichier = new FileInputStream("./essai.txt");
  } catch (FileNotFoundException e) {
    System.out.println(e);
  }
}

public static void main(String[] args) {
    f();
}
```


Le contrôle des exceptions : throws.

```
public void f() throws FileNotFoundException {
  FileInputStream monFichier ;
  // Ouvrir un fichier peut générer une exception
  monFichier = new FileInputStream("./essai.txt");
}

public static void main(String[] args) {
  try {
    // Un appel a f() peut générer une exception
    f();
  } catch (FileNotFoundException e) {
    System.out.println(e);
  }
}
```


Les exceptions contrôlées du JDK



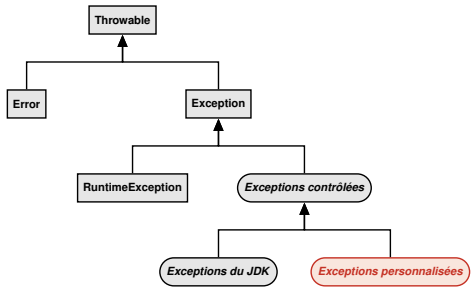
Le JDK et les exceptions contrôlées

Le JDK contient des API qui abusent des **exceptions contrôlées**. En effet, elles utilisent ces exceptions alors que la méthode appelante ne pourra pas résoudre le problème. Par exemple, *JDBC* (liée au langage *SQL*) et les *entrées-sorties*.

Dans ce cas, une solution est d'attraper l'**exception contrôlée** et de renvoyer une **exception non contrôlée**.

Mais attention à ne pas utiliser des **exceptions non contrôlées** uniquement pour ne pas être gêné dans l'écriture des méthodes qui utilisent la méthode qui peut lancer l'exception (pour éviter d'avoir à écrire des **throws** ou des blocs **try/catch**).

Les exceptions contrôlées personnalisées.



Les exceptions contrôlées personnalisées.

On peut définir ses propres exceptions en définissant une sous-classe de la classe **Exception**.

```
public class MonException extends Exception {
  private int x;
  public MonException(int x) {
    this.x = x;
  }
  public String toString() {
    return "Valeur incorrecte : " + x;
  }
}
```


Lancer une exception

Définition

*Pour lancer une exception, on peut utiliser la clause **throw**.*

```
class TestTableau {
  public void main (String args) {
    try {
      int size = args[0];
      if (size < 0) {
        throw new MonException(size);
      }
      int[] tab = new int[size] ;
      System.out.println("Taille du tableau :" + tab.length);
    } catch (MonException e) {
      System.err.println(e);
    }
  }
}
```

```
java TestTableau 3
  Taille du tableau : 3
```

```
java TestTableau -1
  Valeur incorrecte : -1
```

Outline

La clause finally : définition.

Définition

*La clause **finally** définit un bloc d'instruction qui sera exécuté même si une exception est lancée dans le bloc d'essai. Elle permet de forcer la bonne terminaison d'un traitement en présence d'erreur, par exemple : la fermeture des fichiers ouverts.*

```
try {
  ...
}
catch (...) {
  ...
}
finally {
  ...
}
```


La clause finally : quand ?

Le code de la clause **finally** sera toujours exécuté :

- si la clause *try* ne lance pas d'exception : exécution après le *try* (même s'il contient un *return*).
- si la clause *try* lance une exception traitée par un *catch* : exécution après le *catch* (même s'il contient un *return*).
- si la clause *try* lance une exception non traitée par un *catch* : exécution après le lancement de l'exception.

Remarque(s)

Attention : Un appel à la méthode *System.exit()* dans le bloc *try* ou dans un bloc *catch* arrête l'application sans passer par la clause *finally*.

La clause finally : exemple.

```
class TestTableau b {
  public void main (String args) {
    try {
      int x = -1;
      if (x < 0) {throw new MonException(x);}
      System.out.println(x);
    }
    catch (MonException e) {
      System.err.println(e);
    }
    finally {
      System.out.println("Tout est bien qui ...");
    }
  }
}
```

```
java TestTableau -1
  Valeur incorrecte : -1
  Tout est bien qui ...
```


Outline

Pourquoi propager les exceptions ?

Plus une méthode est éloignée de la méthode **main** dans la pile d'exécution, moins elle a de vision globale de l'application. Une méthode peut donc avoir du mal à savoir traiter une exception.

Il est souvent difficile pour une méthode effectuant un petit traitement de décider si une exception est critique :

> *"Il vaut mieux laisser remonter une exception que de décider localement d'interrompre l'ensemble du programme."*

Comment faire remonter une exceptions ?

Pour traiter une exception, on distingue finalement trois méthodes :

1. **Traitement local :** l'exception est levée dans la méthode .
2. **Traitement délégué :** l'exception, qui n'est pas levée dans la méthode, est transmise à la méthode appelante.
3. **Traitement local partiel :** l'exception est levée dans la méthode, puis re-lancée pour être transmise à la méthode appelante.

Traitement délégué : exemple

```
public void f() throws FileNotFoundException {
  monFichier = new FileInputStream("./essai.txt");
}

public void g() {
  try {
   f();
  } catch (FileNotFoundException e) {
    System.err.println("Traitement délégué à g");
  }
}
```

```
Traitement délégué à h
```

Traitement local partiel : exemple

```
public void f() throws FileNotFoundException {
  try {
    monFichier = new FileInputStream("./essai.txt");
  } catch (FileNotFoundException e) {
    System.err.println("Traitement local par f");
    throw e;
  }
}

public void g() {
  try {
   f();
 } catch (FileNotFoundException e) {
    System.err.println("Traitement délégué à g");
  }
}
```

```
Traitement local par f
Traitement délégué à g
```


Traitement local partiel : exemple

```
public void f() throws FileNotFoundException {
  try {
    monFichier = new FileInputStream("./essai.txt");
  } catch (FileNotFoundException e) {
    System.err.println("Traitement local par f");
    throw new MonException();
  }
}

public void g() {
  try {
   f();
 } catch (MonException e) {
    System.err.println("Traitement délégué à g");
  }
}
```

```
Traitement local par f
Traitement délégué à g
```


Et si aucune méthode ne traite l'exception ?

Si une exception remonte jusqu'à la méthode **main** sans être traitée par cette méthode :

- l'exécution du programme est stoppée,
- le message associé à l'exception est affiché, avec une description de la pile des méthodes traversées par l'exception

Remarque(s)

Seul le thread qui a généré l'exception non traitée meurt ; les autres threads continuent leur exécution.

Outline

Outline

Principe de la programmation par contrats

Pour vérifier la correction d'un programme lors de son développement, il est utile de vérifier certains **invariants/conditions** à des endroits bien choisis du programme.

Cette approche a été formalisée et généralisée dans la **programmation par contrats**.

En Java, elle peut s'appuyer sur le mécanisme des **assertions** qui permet au programmeur de vérifier dynamiquement des conditions.

Localisation des points critiques

Lors du développement d'une application, plusieurs points critiques peuvent être à l'origine de bug. On s'attachera à contrôler :

- les **pré-conditions** au début d'une méthode.
- les **post-conditions** à la fin d'une méthode.
- les **invariants de classe** permettant de vérifier une propriété de l'état d'une instance
- les **invariants de boucle**

Le mot clé assert

Java 1.4 introduit un nouveau mot-clé **assert** qui permet d'insérer dans le code des assertions qui :

1. exécute une condition et vérifier qu'elle retourne retourne **true**.
2. lance une erreur `AssertionError` (classe du paquetage `java.lang` et fille de de la classe `Error`),si la condition est **false**.

```
assert x < y ;
```

Une expression peut suivre l'assertion. Facultative, elle ne sert qu'a définir le message d'erreur lié à l'`AssertionError`.

```
assert x == 0 : "x n'est pas nul" ;
```

Compilation des assertions.

Le mot clé **assert** n'existant pas avant Java 1.4, il pouvait être librement utilisé comme nom de méthodes. Le code suivant devient alors ambigüe :

```
assert (x < y) ; // assertion ou this.assert(x<y) ;
```

Afin de permettre une rétro-compatibilité, le mot clé **assert** n'est pas reconnu par défaut (les vieux codes marchent toujours !).

Le programmeur doit donc activer la nouvelle syntaxe lors de la compilation de son application à l'aide de l'option `source` du compilateur `javac`.

```
javac -source 1.4 MaClasseAvecAssert.java
```


Exécution avec vérification des assertions.

Les vérifications liée aux assertions, utiles en période de test, ralentissent l'exécution de l'application. Elles sont donc désactivées par défaut.

Pour activer les assertions (sauf dans les classes système) :

```
java -ea MonProgrammeAvecAssert
```

Pour activer seulement pour une classe :

```
java -ea<nom-classe> MonProgrammeAvecAssert
```

Pour activer seulement pour un paquetage et ses sous-paquetages (les **...** sont à mettre tels quels) :

```
java -ea<nom-paquetage>... MonProgrammeAvecAssert
```

Pour désactiver seulement pour un paquetage (ou une classe) :

```
java -ea -da<nom-paquetage>... MonProgrammeAvecAssert
```


Outline

Les invariants logiques

Les **invariants logiques** sont tous les invariants induits naturellement par la logique sémantique du code. Sous forme d'assertions, ils permettent de se prémunir contre toute erreur dans les futures modifications.

```
coef = (x * x) / (1 + (x * x));
assert coef < 1  : "coef est compris entre 0 et 1";
assert 0 <= coef : "coef est compris entre 0 et 1";
```

```
if (i % 3 == 0) {
  ...
} else if (i % 3 == 1) {
  ...
} else {
  assert i % 3 == 2 : i;
  ...
}
```


Les invariants sur les contrôle de flux

Les **invariants de contrôle de flux** sont essentiels au bon déroulement des programmes. Ils assurent le bon enchainement des instructions. Pour en vérifier le respect, il suffit de placer des `assert` sur des lignes qui ne devraient pas être atteintes.

```
switch(x) {
  case -1 : ...
  case 0 :  ...
  case 1 :  ...
  case default : assert false : x;
}
```

```
void foo() {
  for (...) {
    if (...) return;
  }
  assert false : "On devrait être sortie de foo";
}
```


Les préconditions

Lorsque l'on code une méthode privée, on peut faire la supposition que toutes ses invocations respecteront certaines conditions implicites appelées **préconditions**.

Cette supposition est possible car on garde le contrôle sur tous les appels à cette méthode. L'ajout d'**assert** offrent une méthode simple pour vérifier le respect des préconditions.

```
private int updateEtu(Etudiant etu, int note) {
  assert 0 <= note && note <= 20 ;
  assert Thread.holdsLock(etu);
  ...
}
```


Les postconditions

Les **postconditions** expriment des propriétés qui doivent être satisfaites en fin d'exécution d'une méthode. Elles permettent de vérifier que le traitement de la méthode s'est bien déroulé et que la valeur de retour est conforme aux spécifications.

```
public float inverseAuCarre (int x) {
  float tmp = 1/x ;
  float result = tmp * tmp ;
  assert result * x * x == 1 : "Impossible !";
}
```


Les invariants de classe

Pour tester un invariant de classe Exemple : écriture d'une classe permettant d'insérer des objets dans une liste, tout en s'assurant que cette liste est toujours triée

```
public void insert(Object o) {
  ...
  assert this.isSorted() : "Il y a du désordre ici";
}
```

```
public void add(Object o) {
  int _old_size = maCollection.size();
  this.maCollection.add(o);
  assert this.maCollection.size() == _old_size + 1;
}
```

Outline

Précondition dans une méthode public ?

Une mauvaise pratique avec les assertions, peut être la plus courante, consiste à les utiliser pour vérifier une précondition sur une méthode **public**.

En effet le contrat ne sera pas vérifié quand les assertions seront désactivées en production, or le programmeur n'a pas le contrôle sur toutes les invocations de la méthode. Dans ce cas il vaut mieux utiliser les exceptions à la place des assertions.

```
public float inverseAuCarre (int x) {
  assert x == 0 : "Le paramètre est nul";
  ...
}
```


Précondition dans une méthode public ?

Une mauvaise pratique avec les assertions, peut être la plus courante, consiste à les utiliser pour vérifier une précondition sur une méthode **public**.

En effet le contrat ne sera pas vérifié quand les assertions seront désactivées en production, or le programmeur n'a pas le contrôle sur toutes les invocations de la méthode. Dans ce cas il vaut mieux utiliser les exceptions à la place des assertions.

```
public float inverseAuCarre (int x) {
  if (x == 0) {
    throw new ArithmeticException("Le paramètre est nul");
  }
  ...
}
```


Pas de modification de l'objet dans un assert

Une autre pratique très mauvaise, serait d'appeler une méthode ayant un effet sur l'état de votre classe. En effet, il est possible une fois la phase de développement terminée, de débrayer les assertions pour des raisons évidentes d'optimisation des performances. Donc, si vous modifiez l'état de votre système pendant une assertion, au moment du déploiement, votre programme ne fonctionnera plus !

```
assert connect() : "La connection est impossible";
```


Comparaison de deux réels

Enfin, une pratique qui n'est pas mauvaise que dans les assertions, mais qui est particulièrement redoutable à cet endroit : ne considérez jamais l'égalité entre deux réels ou entre deux dates comme possible !

En effet, il est pratiquement impossible d'obtenir l'égalité parfaite entre deux réels ou entre deux dates. Si vous devez vérifier l'égalité de deux réels, vérifiez plutôt que la distance entre les deux est faible :

```
assert r1 == r2 : "r1 et r2 devraient être égaux";
```

```
assert abs(r1 - r2) < 0.0001 : "r1 et r2 devraient être égaux";
```


Limitations des assertions Java

Les assertions n'implémentent qu'une très petite partie de la programmation par contrat.

Elles présentent notamment plusieurs limitations :

- Les assertions Java ne sont pas suffisamment expressives pour vérifier simplement une assertion du type : $\forall i, t[i] > 0$
- Les assertions portant sur les invariants de classe ne se transmettent pas aux classes filles.

Outline

De l'importance des tests

Dans les applications non critiques, l'écriture des tests a longtemps été considérée comme une tâche secondaire.

La méthode **Extreme programming** (XP) a remis les tests unitaires au centre de l'activité de programmation. Elle préconise d'écrire les tests en même temps, voire même avant la méthode à tester (**Test Driven Development**).

Il existe différentes façons de classer les tests informatiques :

1. Selon le niveau de détail ;
2. Selon le niveau d'accessibilité ;
3. Selon certaines propriétés ou caractéristiques recherchées.

Classification des tests : niveau de détail

On peut classer les tests informatiques selon le niveau de détail :

- **tests unitaires** : vérification des fonctions une par une ;
- **tests d'intégration** : vérification du bon enchaînement des fonctions et des programmes ;
- **tests de non-régression** : vérification qu'il n'y a pas eu de dégradation par rapport à la version précédente (robots).

Classification des tests : niveau d'accessibilité

On peut aussi classer les tests informatiques selon le niveau d'accessibilité :

- **boîte noire** : à partir d'entrées définies on vérifie que le résultat final correspond au résultat attendu, donné par un objet de référence ;
- **boîte blanche** : on a accès à l'état complet du système que l'on vérifie (à chaque ligne).

Classification des tests : propriétés recherchées

Enfin, on peut aussi classer les tests informatiques selon certaines propriétés ou caractéristiques recherchées :

- **tests de performance** : vérification des performances annoncées dans les spécifications ;
- **test fonctionnel** : vérification que les fonctions sont bien réalisées ;
- **test de vulnérabilité** : vérification de la sécurité du logiciel.

Couverture du code par les tests

Définition

*La **couverture de code** consiste à s'assurer que le test conduit à exécuter l'ensemble des instructions présentes dans le code à tester.*

Il est cependant souvent illusoire de tester l'ensemble des instructions présentes dans le code. On se limite plutôt à une couverture évaluée de trois façons différentes :

1. **Statement Coverage** : est ce que chaque ligne du code a été exécutée et testée ?
2. **Condition Coverage** : est ce que chaque point d'évaluation (expression booléennes dans un test par exemple) a été exécuté et testé ?
3. **Path Coverage** : est ce que chaque route possible à travers un chemin donné a été exécuté et testé ?

JUnit

On la vu, les assertions (**assert**) n'implémentent qu'une très petite partie de la programmation par contrat.

Pour la mettre vraiment en œuvre, on peut utiliser plusieurs le frameworks de test **JUnit** qui appartient à la série des **xUnit** (intitiée en 1994 par Kent Beck).

JUnit présente de nombreux avantages :

- simple (non intrusif) ;
- permet de construire des suites de tests unitaire de manière incrémentale ;
- permet de lancer les tests de non régression de manière automatique et répétée ;
- permet la composition des suites de tests en hierarchies ;
- logiciel libre écrit totalement en Java ;
- très bien intégré dans *eclipse*.

La version 4 de JUnit

Faisant suite à la version 3.8, la version 4 est une évolution majeure de JUnit. Si elle maintient une compatibilité descendante, elle introduit de grosses différences dans la syntaxe :

- un des grands bénéfices résulte de l'utilisation des annotations pour remplacer les conventions de nommage sur les méthodes ;
- JUnit 4 requiert une version 5 ou ultérieure de Java ;
- le nom du package des classes de JUnit est différent entre la version 3 et 4 :
  - les classes de Junit 3 sont dans le package `junit.framework` ;
  - les classes de Junit 4 sont dans le package `org.junit` ;
- une classe de tests n'a plus l'obligation d'étendre la classe **TestCase** sous réserve d'utiliser les annotations définies par JUnit et d'utiliser des imports static sur les méthodes de la classe `org.junit.Assert`.

Les annotations JUnit 4

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

public class TestFoobar {
  @BeforeClass
  public void setUpClass() {
    // Code exécuté avant l'exécution du premier test
    // et de la première exécution de la  méthode @Before
  }

  @AfterClass
  public void tearDownClass() {
    // Code exécuté après l'exécution des tous les tests
  }

  @Before
  public void setUp() {
    // Code exécuté avant chaque test
  }

  @After
  public void tearDown() {
    // Code exécuté après chaque test
  }

  @Test
  public void test() {
    assertTrue(true);
  }
}
```


Les assertions JUnit

Avec JUnit, la plus petite unité de tests est l'assertion dont le résultat de l'expression booléenne indique un succès ou une erreur. Il définit une série d'assertion nommées **assertXXX()** qui sont héritées de la classe `junit.framework.Assert` :

- **assertTrue()** : vérifie que la valeur fournie en paramètre est vraie ;
- **assertFalse()** : vérifie que la valeur fournie en paramètre est fausse ;
- **assertNull()** : vérifie que l'objet fourni en paramètre soit null ;
- **assertNotNull()** : vérifie que l'objet fourni en paramètre ne soit pas null ;
- **assertSame()** : vérifie que les deux objets fournis en paramètre font référence à la même entité ;
- **assertNotSame()** : vérifie que les deux objets fournis en paramètre ne font pas référence à la même entité ;
- **assertEquals()** : vérifie l'égalité de deux valeurs de type primitif ou objet (en utilisant la méthode equals()).

Exemple simple

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

public class SampleTest {

  private java.util.List emptyList;

  @Before
  public void setUp() {
    emptyList = new java.util.ArrayList();
  }

  @After
  public void tearDown() {
    emptyList = null;
  }

  @Test
  public void testSomeBehavior() {
    assertEquals("Empty list should have 0 elements", 0, emptyList.size());
  }

  @Test(expected=IndexOutOfBoundsException.class)
  public void testForException() {
    Object o = emptyList.get(0);
  }

}
```


La limitation du temps d'exécution d'un cas de test

JUnit 4 propose une fonctionnalité rudimentaire pour vérifier qu'un cas de tests s'exécute dans un temps maximum donné. Elle utilise l'attribut timeout de l'annotation **@Test** fixant un délai maximum en millisecondes :

```
public class UniversiteTest {
  @Test(timeout=100)
  public void compteurEtu() {
    for(long i = 0 ; i < 999999999; i++)
      this.getEtu(i).check() ;
  }
  public static void main(String[] args) {
    org.junit.runner.JUnitCore.main("UniversiteTest");
  }
}
```

```
JUnit version 4.3.1
  Time: 0,112
   ....
  FAILURES!!!
  Tests run: 3,  Failures: 1
```