

Chapitre 18

Mécanismes avancés de Java

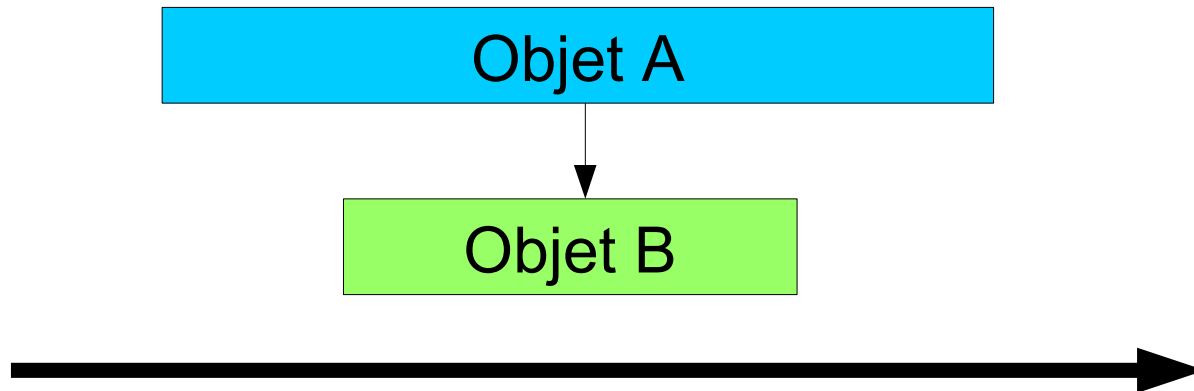
Garbage Collector (GC)

En C++, il est nécessaire de libérer la mémoire allouée lors de la création d'un objet au moyen d'un destructeur dans la classe et de l'opérateur delete.

En Java ce n'est pas la peine, c'est la JVM (Machine Virtuelle Java) qui va se charger de détruire automatiquement les objets qui ne sont plus utilisés grâce à un mécanisme de « **ramasse-miettes** » (« ***Garbage Collector*** »).

```
void fonction()  
{  
    Cercle cercle = new Cercle();  
}  
// A la fin de la fonction, l'objet cercle sera  
// automatiquement détruit et la mémoire occupée  
// par ses données libérée
```

La règle principale pour déterminer qu'un objet n'est plus utilisé est de vérifier qu'il n'existe plus aucun autre objet qui lui fait référence. Ainsi un objet est considéré comme libérable par le GC lorsqu'il n'existe plus aucune référence dans la JVM pointant vers cet objet.



Dans l'exemple ci-dessus, un objet A est créé. Au cours de sa vie, un objet B est instancié et l'objet A possède une référence sur l'objet B. Tant que cette référence existe, l'objet B ne sera pas supprimé par le GC même si l'objet B n'est plus considéré comme utile d'un point de vue fonctionnel.

Il existe plusieurs JVM. Celle d'Oracle s'appelle *Hotspot*. Chaque implémentation de JVM est libre de réaliser cette gestion de la mémoire à sa manière (pour peu que la mémoire soit finalement libérée).

→ il existe plusieurs algorithmes de Garbage collector :
serial collector, parallel collector, parallel compacting collector,
concurrent mark sweep collector, ...

https://fr.wikipedia.org/wiki/Ramasse-miettes_%28informatique%29

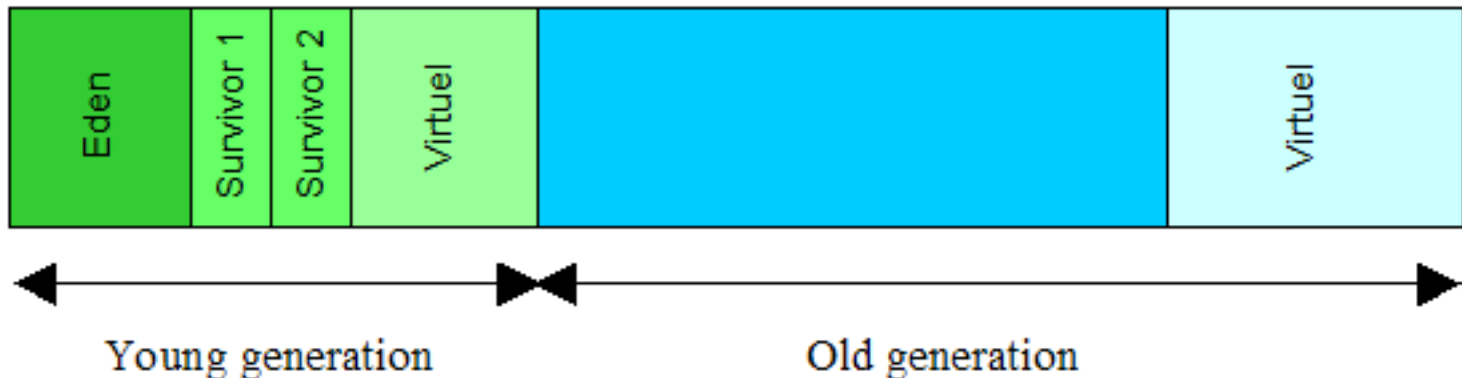
Fonctionnement de Hotspot

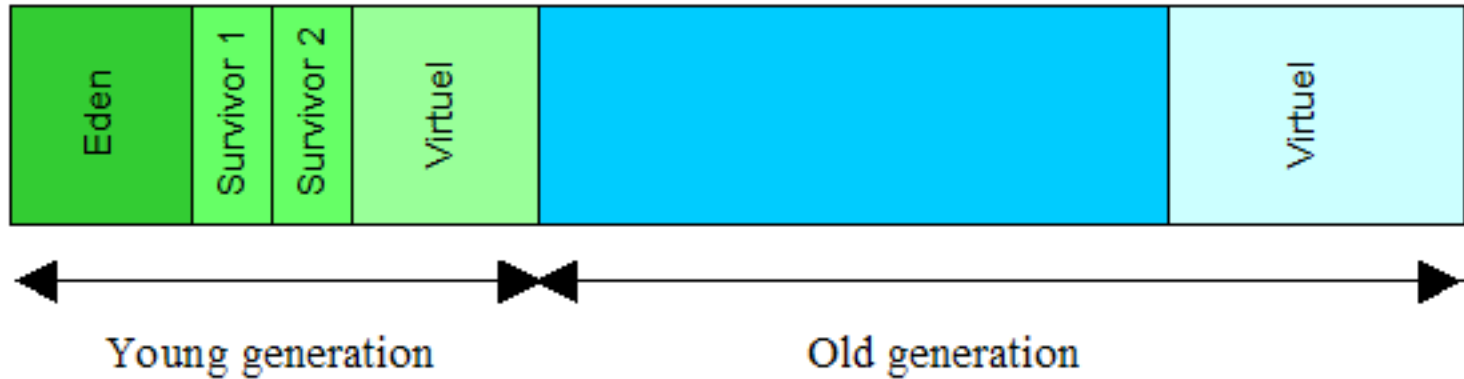
Dans la machine virtuelle, les objets naissent, vivent et meurent dans une zone mémoire appelée **tas** (« *heap* »).

Le GC de la JVM Hotspot utilise la notion de génération, en divisant la mémoire (le tas) de la JVM en différentes portions qui vont contenir des objets en fonction de leur âge :

Young Generation : Les objets à durée de vie courte (ex : des objets créés dans les traitements d'une méthode). Tous les objets sont créés dans cette generation.

Old Generation : Les objets à durée de vie longue.

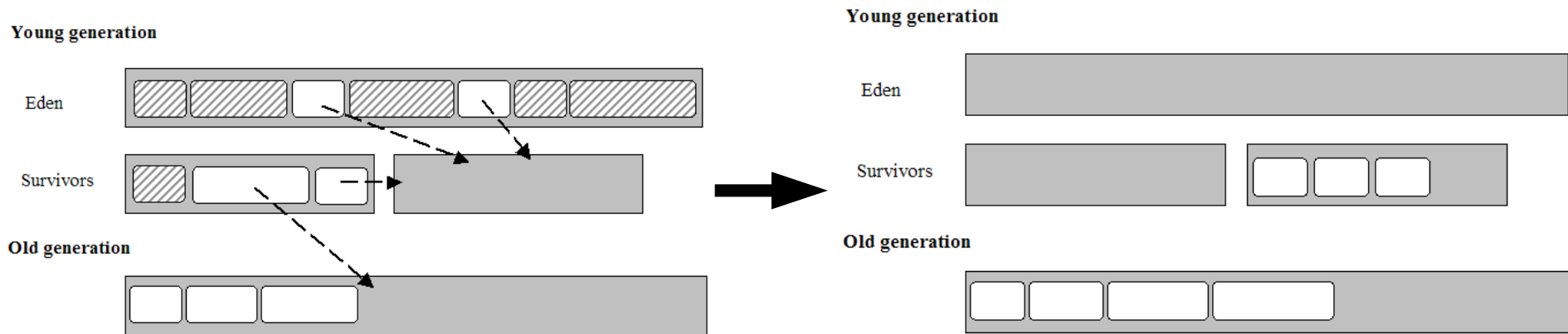




→ Permet d'appliquer des algorithmes différents, optimisés pour chaque génération.

Si un objet de la Young Generation survit à plusieurs phases de collections du GC, il peut être promu dans la Old Generation.

Lorsque la Young Generation est remplie, une « collection mineure » est exécutée par le GC.



Après le déplacement des objets en cours d'utilisation, tous les objets qui restent dans l'espace Eden et l'espace Survivor qui était rempli sont des objets inutilisés dont l'espace peut être récupéré.

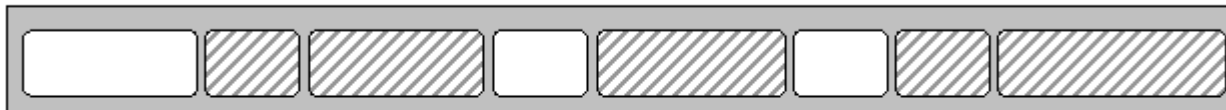
Si la Old Generation est remplie, une « collection majeure » est exécutée par le GC sur l'intégralité du tas (Young et Old Generation).

Généralement une collection majeure est beaucoup plus longue qu'une collection mineure puisqu'elle implique beaucoup plus d'objets à traiter.

3 étapes :

- **Mark** : le GC identifie les objets qui sont encore utilisés
- **Sweep** : le GC récupère la mémoire des objets qui n'ont pas été marqués lors de l'étape précédente.
- **Compact** : le GC compacte la mémoire en regroupant tous les objets utilisés au début de la mémoire de la Old Generation → la création d'objets dans la Old Generation sera ensuite plus rapide

Avant le compactage



Après le compactage



Limitations

Le ramasse-miettes est un processus complexe qui consomme des ressources et nécessite un temps d'exécution non négligeable pouvant être à l'origine de problèmes de performance.

On peut explicitement déclencher le GC avec :

System.gc();

Surveiller l'activité du Garbage Collector

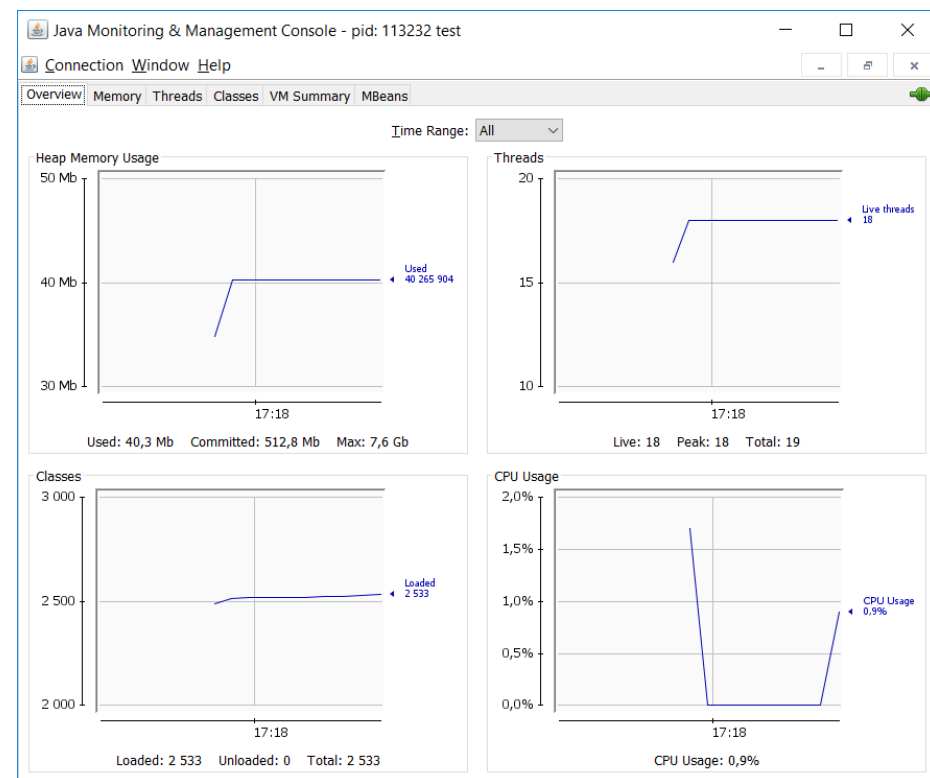
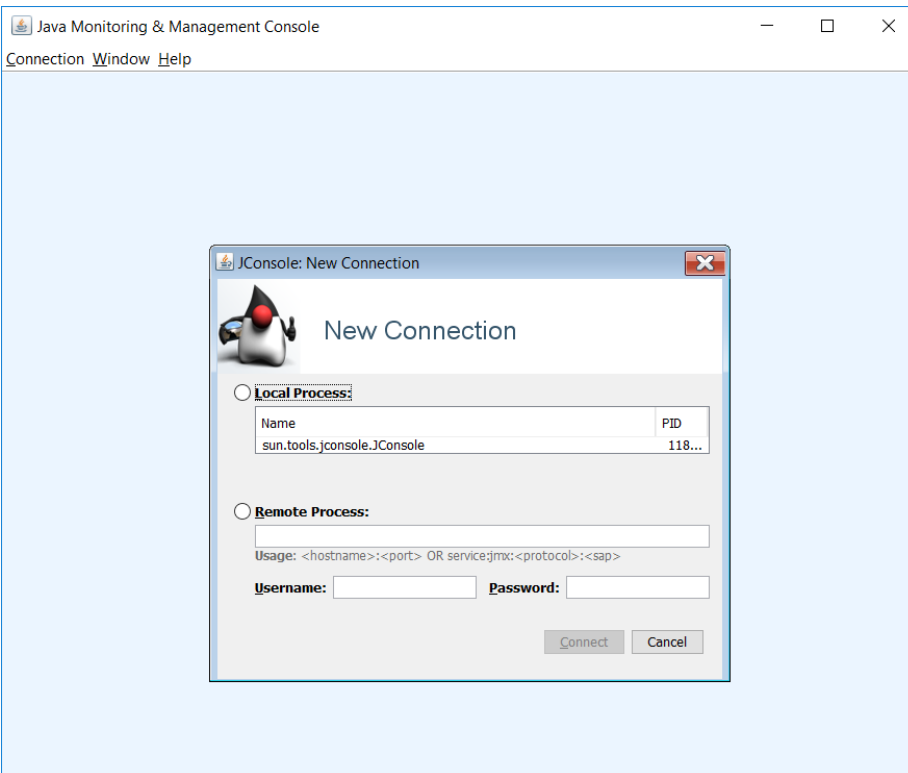
Il est possible de surveiller l'activité du Garbage Collector en rajoutant les options suivantes sur la ligne de commande **java** lors de l'exécution :

-verbose:gc : permet d'afficher l'activité du GC sur la sortie standard.

-Xloggc:filename : permet de stocker des logs sur l'activité du GC dans le fichier spécifié.

Surveiller l'activité du Garbage Collector en mode graphique

Il est possible de surveiller l'activité du Garbage Collector et d'obtenir bien d'autres informations encore sur un programme Java dans un environnement graphique comme **JConsole** (disponible dans le JDK), **VisualGC** ou **GCViewer**.



Autres commandes de terminal

jps

C'est l'équivalent de la commande **ps** donnant la liste des processus tournant sur une machine, mais dédiée aux JVM. Elle présente sur deux colonnes l'identifiant de processus et le nom de l'application.

```
99312 testJava  
126812 Jps
```

Note : jps est elle-même listée car elle est elle aussi une application Java.

jinfo

Affiche la configuration de la JVM, c'est-à-dire l'intégralité des propriétés systèmes ainsi que les paramètres de démarrage de la JVM

jstack

Affiche les données portant sur l'intégralité des threads présents dans la JVM : leur état, la pile d'appels complète, ...

jmap

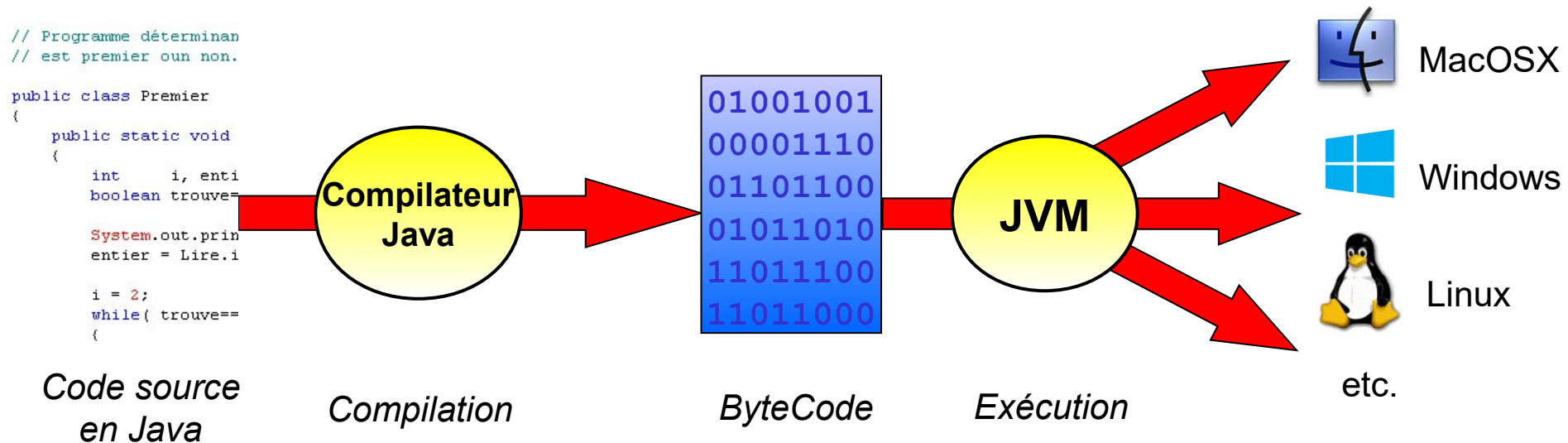
Permet d'obtenir des informations sur la mémoire.

jstat

Donne de nombreuses statistiques sous différents formats concernant le comportement de la JVM : l'état de la mémoire, l'activité du GC, du JIT, ...

JIT (Just In Time)

On l'a vu, le compilateur Java génère un fichier de **bytecode** qui est **interprété** par la JVM lors de l'exécution.



Pour améliorer les performances, la JVM utilise une compilation "***Just In Time***" (JIT) pendant l'exécution.

Au fur et à mesure que la machine virtuelle interprète le code elle le compile en **langage machine** (Code natif) avec le *JIT Compiler*.

Principe

Au fur et à mesure que la machine virtuelle interprète le code elle le compile en **langage machine** (Code natif) avec le *JIT Compiler*.

Le code ainsi compilé est gardé en mémoire afin d'être exécuté plus rapidement (car il n'est plus interprété) lors de la prochaine passe.

Tout le code n'est pas compilé à la première passe (Les blocs de code qui ne sont pas exécutés ne sont pas compilés).

Intérêt

Permet d'avoir des temps d'exécution plus performant que si le bytecode restait interprété.

Avantage par rapport à une compilation classique (par ex avec C++) : le code peut être optimisé à la volée au processeur et à l'OS de la machine.

Graal VM

<https://www.graalvm.org/>

Machine Virtuelle universelle développée par Oracle permettant d'exécuter des programmes écrits en Java (mais aussi en JavaScript, Python, Ruby, R, Scala, Groovy, Kotlin, Clojure, C, C++) avec une technique JIT.

<https://www.baeldung.com/graal-java-jit-compiler>

IDE (Integrated Development Environment)

= Environnement de développement intégré.

Permet de gérer des projets comportant plusieurs fichiers, d'éditer du code, de compiler, de déboguer, de construire visuellement une interface graphique, etc.

Eclipse

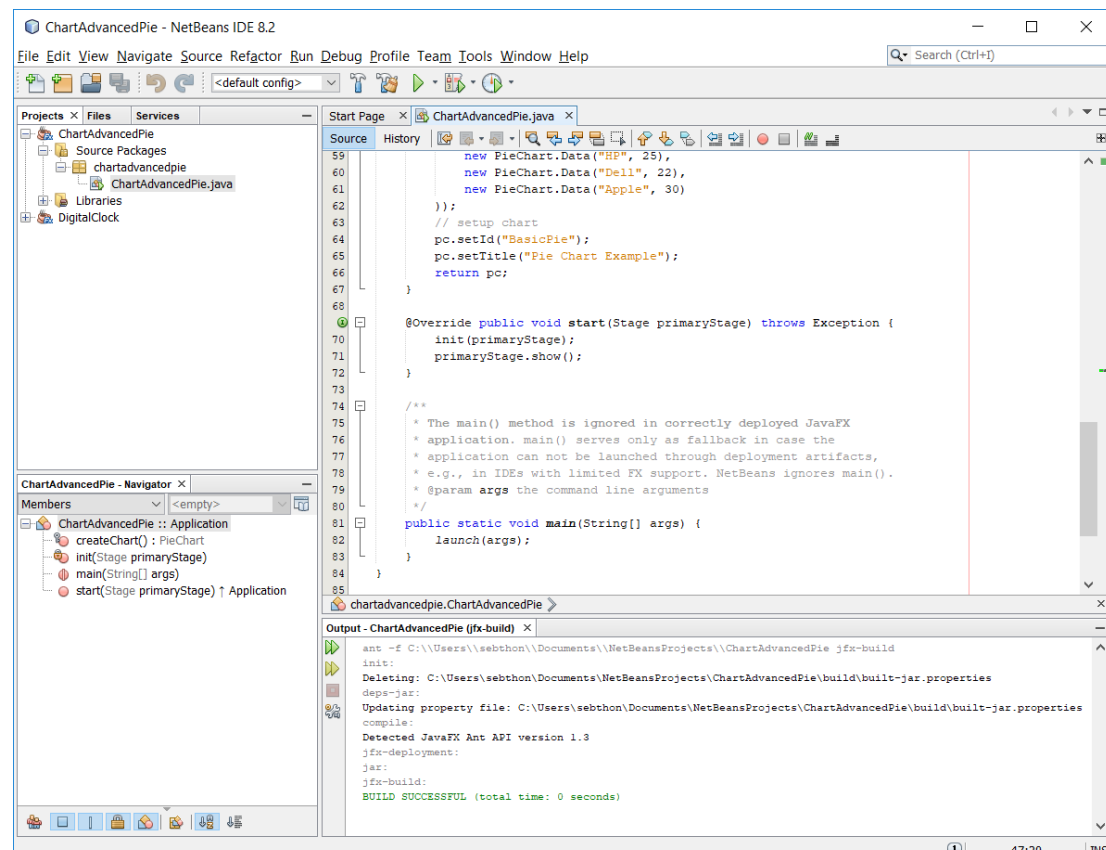
<https://www.eclipse.org/ide/>

IntelliJ

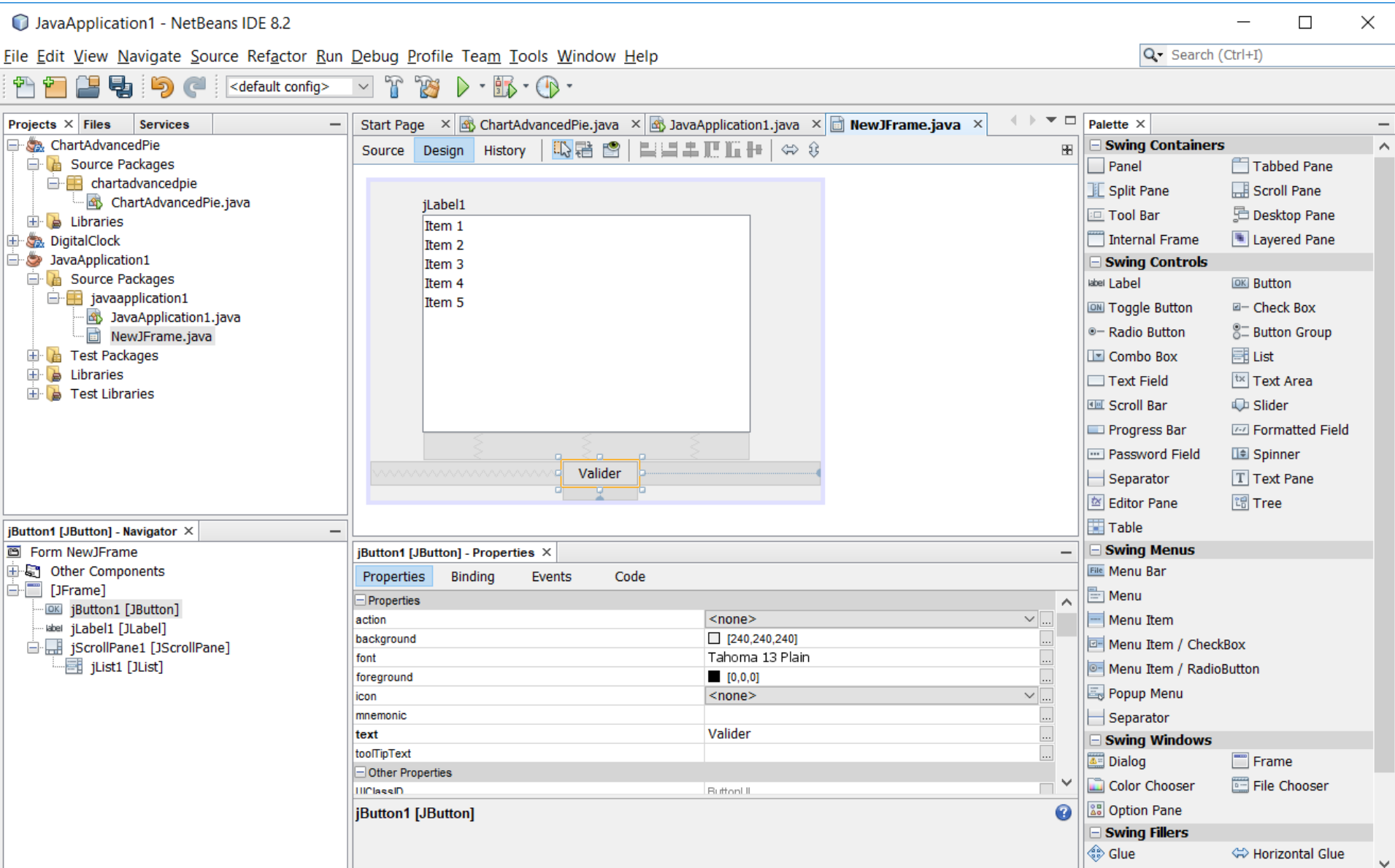
<https://www.jetbrains.com/idea/>

Netbeans

<https://netbeans.org/>



Un IDE permet de construire visuellement une interface graphique.
<http://netbeans.apache.org/kb/docs/java/gui-functionality.html>



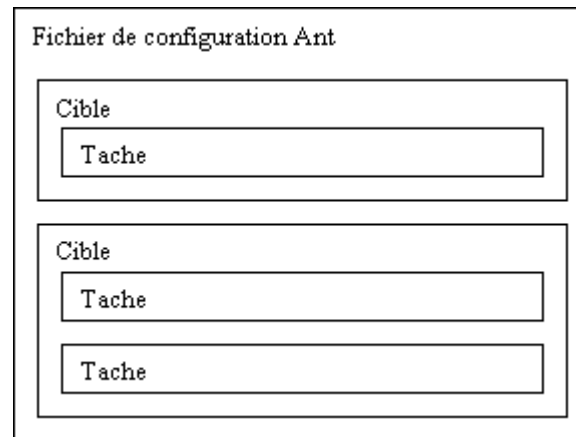
Ant

<http://ant.apache.org/>

Comparable à la commande **make** et ses makefile, **Ant** permet la construction d'applications (compilation, exécution de tâches post et pré compilation, ...).

Permet d'automatiser des opérations répétitives tout au long du cycle de développement de l'application (développement, tests, recettes, mises en production, ...)

Ant repose sur un fichier de configuration XML qui décrit les différentes tâches qui devront être exécutées. Il contient un ensemble de cibles (targets). Chaque cible contient une ou plusieurs tâches. Chaque cible peut avoir une dépendance envers une ou plusieurs autres cibles pour pouvoir être exécutée.



Exécution :

```
ant -buildfile fichierbuild.xml nomDeLaCible
```

build.xml

```
<project name="compiltation des classes" default="compile" basedir=". ">
  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir" value="src"/>
  <property name="projet.bin.dir" value="bin"/>
  <property name="projet.lib.dir" value="lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
  </path>

  <!-- Compilation des classes du projet -->
  <target name="compile" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
      destdir="${projet.bin.dir}"
      debug="on"
      optimize="off"
      deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>
</project>
```

Exécution :

ant -buildfile build.xml compile

Maven



<http://maven.apache.org/>

Maven est un outil permettant de construire un projet en prenant en charge :

- La compilation
- Le packaging
- La gestion des dépendances
- Les tests
- La génération de la documentation
- L'accès aux dépôts ou aux gestionnaires de dépendances
- Le déploiement
- ...

Pour créer un projet, tapez la commande :

```
mvn archetype:generate
```

De nombreux « artifacts » (JAR) sont alors téléchargés, puis Maven vous pose des questions pour configurer le projet :

Define value for property 'groupId': **iut.exemples.demo1**

Define value for property 'artifactId': **mon-appli**

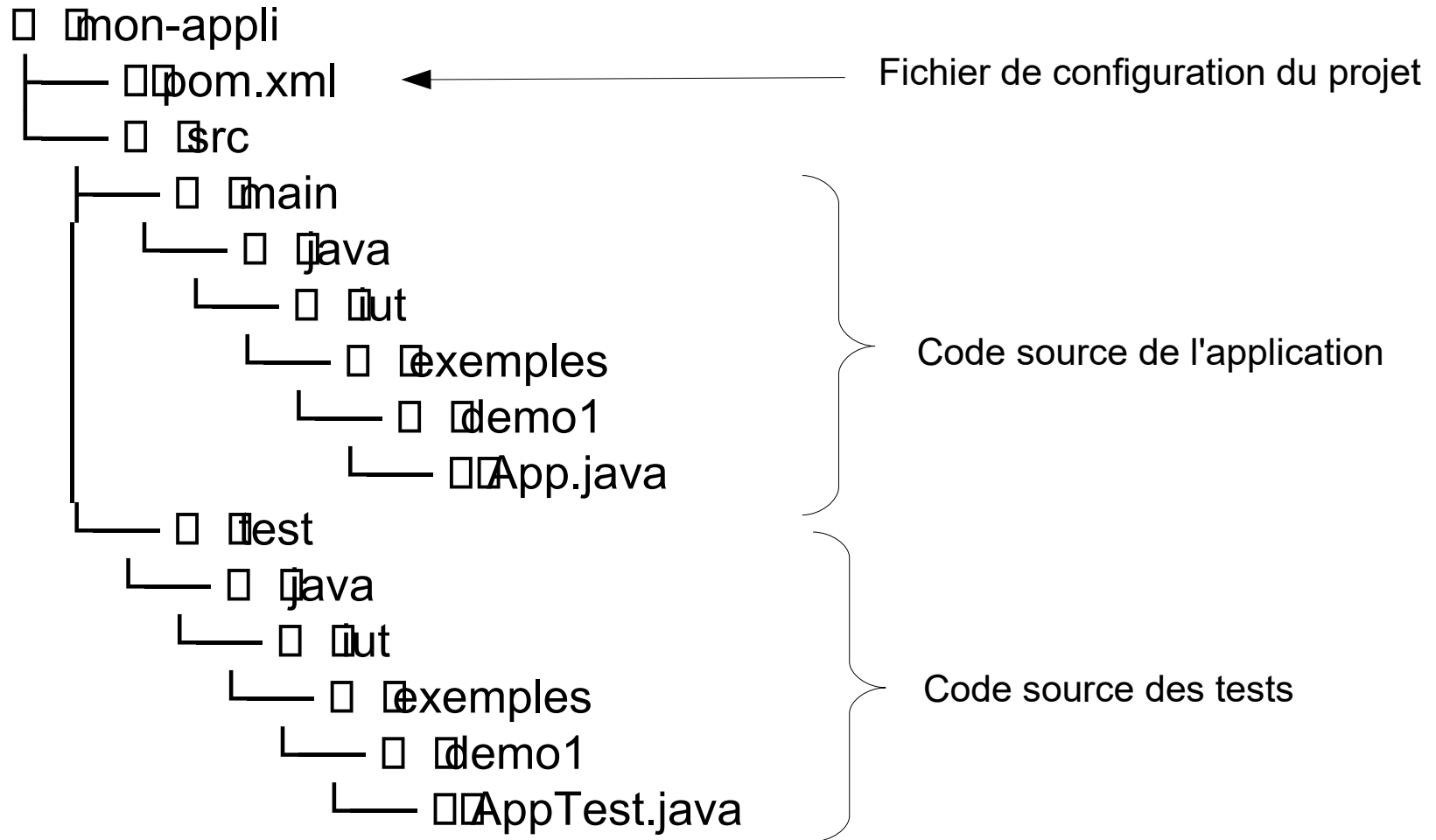
Define value for property 'version' 1.0-SNAPSHOT: : laisser vide

Define value for property 'package' iut.exemples: : laisser vide

Ensuite Maven vous demande de confirmer les paramètres, presser la touche Entrée.

Maven crée alors le squelette du projet.

Maven a crée le répertoire **mon-appli** contenant l'arborescence suivante :



pom.xml

Le POM (Project Object Model) est le fichier de configuration du projet Maven.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
  <groupId>iut.exemples</groupId>
  <artifactId>mon-appli</artifactId>
  <version>1.0-SNAPSHOT</version>
```

```
  <name>mon-appli</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>
```

```
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
```

```
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

App.java

```
package iut.exemples.demo1;
```

```
/**  
 * Hello world!  
 */
```

```
public class App  
{  
    public static void main( String[] args )  
    {  
        System.out.println( "Hello World!" );  
    }  
}
```

Compilation d'un projet Maven

Dans un terminal, placez-vous à la racine du projet (répertoire **mon-appli**), là où se trouve le fichier **pom.xml**, et tapez la commande :

mvn package

→ Création d'un JAR de l'application dans le répertoire **target**, nommé **mon-appli-1.0-SNAPSHOT.jar**

Pour l'exécuter :

java -cp mon-appli-1.0-SNAPSHOT.jar iut.exemples.demo1.App

Maven a aussi compilé et exécuté les tests.

Dépendances

Une des grandes forces de Maven est sa gestion des dépendances, pour ajouter facilement des bibliothèques tierces à une application. Pour cela, il suffit d'ajouter dans le fichier **pom.xml** une balise **<dependency>** dans la section **<dependencies>**.

Par exemple, il y a déjà de base une dépendance à JUnit :

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Pour récupérer les dépendances, Maven les télécharge un **repository**.
<https://search.maven.org/>

Pour récupérer les dépendances, Maven les télécharge sur un **repository** central et les stocke dans votre repository local.

<http://repo1.maven.apache.org/maven2/>

Vous pouvez chercher des dépendances ici :

<https://search.maven.org/>

JUnit

<http://junit.org/>



C'est un framework open source pour le développement et l'exécution de tests unitaires automatisables. Le principal intérêt est de s'assurer que le code répond toujours aux besoins même après d'éventuelles modifications (tests unitaires de non-régression).

JUnit propose :

- Un framework pour le développement des tests unitaires reposant sur des assertions qui testent les résultats attendus.
- Des applications pour permettre l'exécution des tests et afficher les résultats.

Le but est d'automatiser les tests. Ceux-ci sont écrits dans des classes sous la forme de cas de tests avec leurs résultats attendus. JUnit exécute ces tests et les compare avec ces résultats.

Cela permet de séparer le code de la classe du code qui permet de la tester. Souvent pour tester une classe, il est facile de créer une méthode `main()` qui va contenir les traitements de tests. L'inconvénient est que ce code "superflu" est inclus dans la classe. De plus, son exécution doit se faire manuellement.

Avec JUnit les cas de tests sont regroupés dans des classes Java qui contiennent une ou plusieurs méthodes de tests précédées d'**annotations** (`@before`, `@after`, `@test`, ...). Les cas de tests peuvent être exécutés individuellement ou sous la forme de suites de tests.

Utilisation en ligne de commande

1) Télécharger les JAR de JUnit et les installer dans votre classpath :
<https://github.com/junit-team/junit4/wiki/Download-and-Install>

2) Ecrire la classe à tester (ex : **MaClasse**), la compiler avec javac.

3) Ecrire une classe de test (ex : **MaClasseTest**) qui va contenir les différents tests à réaliser par JUnit, la compiler avec javac :

```
javac -cp .;junit-4.13-rc-1.jar;hamcrest-core-1.3.jar MaClasseTest.java
```

4) Appeler JUnit pour qu'il exécute la séquence de tests :

```
java -cp .;junit-4.13-rc-1.jar;hamcrest-core-1.3.jar org.junit.runner.JUnitCore  
MaClasseTest
```

org.junit.runner.JUnitCore est une classe du framework de JUnit qui va exécuter le code de la classe de test.


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

Structure d'une
classe de test

```
public class MaClasseDeTest {
```

```
    /** Pre et post conditions */
```

@BeforeClass

```
public static void setUpBeforeClass() throws Exception {
    // Le contenu de cette méthode ne sera exécuté qu'une fois avant toutes
    // les autres méthodes avec annotations
    // (y compris celles ayant une annotation @Before)
}
```

@AfterClass

```
public static void tearDownClass() throws Exception {
    // Le contenu de cette méthode ne sera exécuté qu'une fois après toutes
    // les autres méthodes avec annotations
    // (y compris celles ayant une annotation @After)
}
```

@Before

```
public void setUp() throws Exception {  
    // Le contenu de cette méthode sera exécuté avant chaque test  
    // (méthode avec l'annotation @Test)  
}
```

@After

```
public void tearDown() throws Exception {  
    // Le contenu de cette méthode sera exécuté après chaque test  
    // (méthode avec l'annotation @Test)  
}
```

```
/** Cas de tests */
```

```
@Test
```

```
public void testCas1() {
```

```
    // Code contenant l'exécution du premier scénario avec
```

```
    // les assertions associées
```

```
}
```

```
@Test
```

```
public void testCas2() {
```

```
    // Code contenant l'exécution du second scénario avec
```

```
    // les assertions associées
```

```
}
```

```
}
```

Ordre d'exécution des méthodes de la classe MaClasseDeTest :

SetUpBeforeClass()

SetUp()
testCas1()
tearDown()

SetUp()
testCas2()
TearDown()

tearDownClass()

Assertions

En informatique, une assertion est une expression qui doit être évaluée vrai ou faire échouer le programme ou le test en cas d'échec (en levant une exception ou en mettant fin au programme par exemple).

Dans le cas de JUnit on peut assimiler une assertion à une vérification. En cas d'échec, l'exception **java.lang.AssertionError** est levée.

Voici une liste non exhaustive des fonctions d'assertions de JUnit :

assertEquals : vérifier l'égalité de deux expressions ;

assertNotNull : vérifier la non-nullité d'un objet ;

assertNull : vérifier la nullité d'un objet ;

assertTrue : vérifier qu'une expression booléenne est vraie ;

assertFalse : vérifier qu'une expression booléenne est fausse ;

fail : échouer le test si cette assertion est exécutée.

Import static

Les imports statiques permettent de référencer directement un objet ou une méthode statique.

Par exemple au lieu d'écrire :

```
System.out.println("affiche du texte");
```

Avec l'import statique il suffit de rajouter l'instruction :

```
import static java.lang.System.out;
```

Ensuite on peut référencer directement le flux de sortie `out` depuis n'importe quelle méthode de la classe :

```
out.println("affiche du texte");
```

Ainsi, un import statique de **org.junit.Assert.*** nous permet de simplifier l'écriture de l'appel aux méthodes de la classe **Assert**, comme **assertEquals**, **assertNotNull**, **assertNull**, etc.

Au lieu d'écrire :

```
import org.junit.Assert;  
...  
Assert.assertEquals(expResult, result);
```

On peut écrire :

```
import static org.junit.Assert.assertEquals;  
// ou :  
// import static org.junit.Assert.*;  
...  
assertEquals(expResult, result);
```

Exemple : on souhaite tester les méthodes de la classe Addition :

```
public class Addition
{
    int nb1;
    int nb2;

    public Addition(int nb1, int nb2) {
        this.nb1 = nb1;
        this.nb2 = nb2;
    }

    public int somme() {
        return nb1 + nb2;
    }

    public int getNb1() {
        return nb1;
    }
    public int getNb2() {
        return nb2;
    }
}
```


On écrit une classe de cas de test AdditionTest :

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

public class AdditionTest
{
    private Addition add;

    @Before
    public void initialiser() throws Exception {
        add = new Addition(1,2);
    }

    @After
    public void nettoyer() throws Exception {
        add = null;
    }
}
```

```
@Test
public void testNb1() {
    // On vérifie si l'attribut nb1 a bien été initialisé à 1
    assertEquals("L'affectation est incorrecte", 1, add.getNb1());
}
```

```
@Test
public void testNb2() {
    // On vérifie si l'attribut nb2 a bien été initialisé à 2
    assertEquals("L'affectation est incorrecte", 2, add.getNb2());
}
```

```
@Test
public void testAddition() {
    // On vérifie si la méthode somme() donne bien 3
    assertEquals("L'addition est incorrecte", 3, add.somme());
}
}
```

Pour lancer les tests en ligne de commande, taper :

```
java -cp junit.jar;. junit.textui.TestRunner AdditionTest
```

Pour compiler :

```
javac -cp .;junit-4.13-rc-1.jar;hamcrest-core-1.3.jar AdditionTest.java
```

Pour lancer les tests :

```
java -cp .;junit-4.13-rc-1.jar;hamcrest-core-1.3.jar org.junit.runner.JUnitCore  
AdditionTest
```

Utilisation avec Netbeans

Netbeans vient avec un module JUnit, pas la peine de l'installer.
Netbeans facilite la création de classes de tests (création automatique du squelette) et leur lancement.

<https://netbeans.org/kb/docs/java/junit-intro.html>

The screenshot displays the NetBeans IDE 8.2 interface. The main window shows the 'Run' menu open, with 'Test Project (EssaiJUnit)' selected. The 'Projects' pane on the left shows the project structure, including 'EssaiJUnit', 'Source Packages', 'Test Packages', and 'Libraries'. The 'AdditionTest.java' file is highlighted in the 'Test Packages' section. The 'Run' menu options include: Run Project (EssaiJUnit) (F6), Test Project (EssaiJUnit) (Alt+F6), Build Project (EssaiJUnit) (F11), Clean and Build Project (EssaiJUnit) (Maj+F11), Set Project Configuration, Set Main Project, Generate Javadoc (EssaiJUnit), Run File (Maj+F6), Test File (Ctrl+F6), Compile File (F9), Check File (Alt+F9), Validate File (Alt+Maj+F9), Repeat Build/Run: EssaiJUnit (test) (Ctrl+F11), and Stop Build/Run.

In the bottom-left corner, the 'New File' dialog is open, showing the 'Choose File Type' step. The 'Project' is set to 'EssaiJUnit'. The 'Filter' is empty. The 'Categories' list includes JavaFX, Swing GUI Forms, JavaBeans Objects, AWT GUI Forms, Unit Tests, Persistence, Hibernate, XML, and Other. The 'File Types' list includes JUnit Test, Test for Existing Class, Test Suite, TestNG Test Case, and TestNG Test Suite. The 'Description' field states: 'Creates a simple JUnit test case for testing methods of a single class.'

The 'AdditionTest.java' file is open in the editor, showing the following code:

```
public class AdditionTest {  
  
    public AdditionTest() {  
    }  
  
    @BeforeClass  
    public static void setUpClass() {  
    }  
  
    @AfterClass  
    public static void tearDownClass() {  
    }  
  
    @Before  
    public void setUp() {  
    }  
  
    @After  
    public void tearDown() {  
    }  
}
```

The 'Test Results' pane at the bottom right shows the test results for 'essaijunit.AdditionTest'. It indicates that all tests passed, with a success rate of 100.00%.

Test Results x
essaijunit.AdditionTest x
Tests passed: 100,00 %
All 3 tests passed. (0,051 s)

Génération de documentation

<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>

JavaDoc est une commande permettant de générer une documentation au format HTML à partir du code source Java, en analysant les commentaires qui se trouvent entre `/**` et `*/`.

`//` Commentaire standard sur une seule ligne

```
/*  
    Commentaire standard  
    sur plusieurs lignes  
*/
```

```
/**  
    Commentaire Javadoc  
*/
```

Les commentaires JavaDoc se placent au-dessus d'une classe, d'une méthode ou d'une donnée membre que l'on veut documenter. On peut y mettre des balises HTML.

```
/**
 * <p>Classe décrivant un étudiant.
 * </p>
 * @author Sébastien Thon
 * @version 2.0
 */
public class Etudiant
{
    /**
     * <p>Constructeur de la classe Etudiant
     * </p>
     * @param nom Le nom de l'étudiant
     * @param prenom Le prenom de l'étudiant
     * @since 1.0
     */
    public Etudiant (String nom, String prenom)
    {
        this.nom = nom;
        this.prenom = prenom;
    }
}
```

Des mots clés précédés de '@' sont des « *tags* » qui permettent d'ajouter des champs dans la documentation :
@param, @return, @see, @since, @author, @deprecated, ...

Documentation d'une méthode :

```
/**
 * <p>Méthode retournant la moyenne d'un module du
 * <a href="https://www.iut.fr/">PPN</a>
 * </p>
 * @param module Le module dont on veut la moyenne
 * @return La moyenne du module
 * @see <a href="https://www.iut.fr/">PPN</a>
 * @since 1.0
 * @deprecated Utiliser la nouvelle methode moyennes
 */
public float moyenneModule( int module )
{
    return moyennes[module];
}
```

Method Detail

moyenneModule

```
public float moyenneModule(int module)
```

Deprecated. *Utiliser la nouvelle methode moyennes*

Méthode retournant la moyenne d'un module du PPN

Parameters:

`module` - Le module dont on veut la moyenne

Returns:

La moyenne du module

Since:

1.0

See Also:

PPN

















Des mots clés précédés de '@' sont des « *tags* » qui permettent d'ajouter des champs dans la documentation :

@param, @return, @see, @since, @author, @deprecated, ...

Création de la documentation :

```
javadoc -d doc src\*
```

Génère la documentation dans le répertoire **doc** des fichiers sources se trouvant dans le répertoire **src**.

	allclasses-frame.html	1 Ko	02/11/2019 21:22
	allclasses-noframe.html	1 Ko	02/11/2019 21:22
	constant-values.html	4 Ko	02/11/2019 21:22
	deprecated-list.html	5 Ko	02/11/2019 21:22
	Etudiant.html	10 Ko	02/11/2019 21:22
	help-doc.html	8 Ko	02/11/2019 21:22
	index.html	3 Ko	02/11/2019 21:22
	index-all.html	6 Ko	02/11/2019 21:22
	Main.html	9 Ko	02/11/2019 21:22
	overview-tree.html	4 Ko	02/11/2019 21:22
	package-frame.html	1 Ko	02/11/2019 21:22
	package-list	1 Ko	02/11/2019 21:22
	package-summary.html	5 Ko	02/11/2019 21:22
	package-tree.html	4 Ko	02/11/2019 21:22
	script.js	1 Ko	02/11/2019 21:22
	stylesheet.css	14 Ko	02/11/2019 21:17

PACKAGE **CLASS** TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METH

Class Etudiant

java.lang.Object
Etudiant

```
public class Etudiant
extends java.lang.Object
```

Classe décrivant un étudiant.

Author:

Sébastien Thon

Constructor Summary

Constructors

Constructor and Description

Etudiant(java.lang.String nom, java.lang.String prenom)
Constructeur de la classe Etudiant

Method Summary

All Methods

Instance Methods

Concrete Methods

Deprecated Methods

Modifier and Type Method and Description

float **moyenneModule**(int module)

Deprecated

Reverse engineering

Décompilation

Des outils permettent à partir de fichiers de bytecode (.class) de retrouver le code source (.java)

<http://jode.sourceforge.net/>

<http://www.kpdus.com/jad.html>

Obsfucation

L'obsfucation permet de modifier le bytecode de manière à rendre difficile la décompilation.

<https://www.yworks.com/products/yguard>

<https://www.guardsquare.com/en/products/proguard>

<http://www.zelix.com/klassmaster/>