Outils & BONNES PRATIQUES

Ce cours présente les différents outils mis en œuvre pour créer une application à partir du langage de programmation objet Java. Les programmes ‘Java’ sont exécutés (plus exactement le *byte code* créé lors de la phase de compilation) au travers d’un machine virtuelle (JVM : *Java Virtual Machine*).

La programmation d’une application est conduite avec un ensemble de bonnes pratiques, qu’il est d’usage de suivre afin qu’un code soit compréhensible et lisible.

Les exemples de ce cours sont dans le dossier ‘handson’.

# Rappels sur le langage JAVA

Java est langage de programmation orienté objet. Les programmes java se veulent indépendant de la plateforme d’exécution mais nécessite l’installation d’un environnement d’exécution la *JVM* (*Java Virtual Machine*).

## Le language

### Les mots clés

Les instructions suivantes sont réservées, elles ne peuvent donc pas être utilisées comme identifieurs.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| abstract | const | final | Int | public | throw |
| assert | continue | finally | interface | return | throws |
| boolean | default | float | long | short | transient |
| break | do | for | native | static | true |
| byte | double | goto | new | strictfp | try |
| case | else | if | null | super | void |
| catch | enum | implements | package | switch | volatile |
| char | extends | import | private | synchronized | while |
| class | false | instanceof | protected | this |  |

### Les Commentaires

|  |  |
| --- | --- |
| **commentaire sur une line** | **// commentaire sur une seule ligne** **int** N=1; // déclaration du compteur |
| **Commentaire multi-lignes** | **/\* commentaires ligne 1**  **commentaires ligne 2 \*/** |
| **Commentaire ‘java doc’** | **/\*\***  **\* commentaire de la methode**  **\* @param val la valeur à traiter**  **\* @since 1.0**  **\* @return la valeur de retour**  **\* @deprecated Utiliser la nouvelle methode XXX**  **\*/** |

### Types élementaires

|  |  |  |
| --- | --- | --- |
| **boolean** | valeur logique : true ou false | 1 bit |
| **byte** | octet signé | 8 bits |
| **short** | entier court signé | 16 bits |
| **char** | caractère Unicode | 16 bits |
| **int** | entier signé | 32 bits |
| **float** | virgule flottante simple précision | 32 bits |
| **double** | virgule flottante double précision | 64 bits |
| **long** | entier long | 64 bits |

### Opérateurs

|  |  |
| --- | --- |
| = | affectation |
| += | incrémente l’identifieur de gauche |
| ++ | incrément de 1 l’identifieur de gauche |
| -= | décrémente l’identifieur de gauche |
| -- | décrément de 1 l’identifieur de gauche |
| \*= | multiplie l’élément de gauche |
| /= | divise l’élément de gauche |
| %= | reste de la division |
| ^= | puissance |
| <<= | décalage de bits à gauche |
| >>= | décalage de bits à droite |
| >>>= | équivalent à : a = a >>> 10 décalage à gauche non signé |
| > | test, strictement supérieur |
| < | test, strictement inférieur |
| >= | test, supérieur ou égal |
| <= | test, inférieur ou égal |
| == | test, égalité |
| != | test, diffèrent de |
| & | ET binaire |
| ^ | OU exclusif binaire |
| | | OU binaire |
| && | ET logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient fausse |
| || | OU logique (pour expressions booléennes) : l'évaluation de l'expression cesse dès qu'elle devient vraie |
| ? : | opérateur conditionnel : renvoie la valeur b ou c selon l'évaluation de l'expression a (si a alors b sinon c) : b et c doivent retourner le même type |

## La programmation objet

### Les objets java

Une classe permet d’encapsuler un ensemble de données et de fonctions. En java une classe est instanciée avec l’opérateur **new.** Tous les objets Java sont instanciés par allocation dynamique. Les variables ainsi instanciées sont donc des références et les opérations telles que l’égalité, copie la référence et non le contenu. Autrement dit après une copie avec l’opérateur ‘=’ l’objet n’est pas dupliqué : toute modification de la copie entraîne une modification de l’objet initial.

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* Main program showing the instantiation of  \* object by using the "new" operator  \*  \*/  **public** **class** FirstJavaObject {  **public** **static** **void** main(String[] args) {    MyFirstObject myFirstObject1 = **new** MyFirstObject();  MyFirstObject myFirstObject2;  myFirstObject1.myFirstMethod(); // valid call  myFirstObject2.myFirstMethod(); // invalid call (non instantiate object)    }  } |
| Ici le programme ne compile pas car ***myFirstObject2*** n’est pas instancié.  (*Code accessible dans course01/FirstJavaObject*) |

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* Object used to illustrate the assignment  \* with '"' operator;  \*  \*/  **public** **class** MyFirstAssignment {    **protected** **int** value;    **public** MyFirstAssignment(){  value = 0;  }    **public** **void** setValue(**int** value)  {  **this**.value = value;  }  **public** **int** getValue() {  **return** value;  }  /\*\*  \* **@return** a string containing the 'value'  \*/  **public** String toString()  {  **return** "The value is '"+value+"'.";  }  } |
| L’objet MyFirstAssignment contient un entier avec ses accesseur  (*Code accessible dans course01/ObjectAssignment)* |
| **package** edu.iut;  /\*\*  \*  \* Main program : assignment  \*  \*/  **public** **class** ObjectAssignment {  **public** **static** **void** main(String[] args) {    MyFirstAssignment myFirstAssignment1 = **new** MyFirstAssignment();  MyFirstAssignment myFirstAssignment2 = **new** MyFirstAssignment();    myFirstAssignment1.setValue(1);  myFirstAssignment2.setValue(2);    // display the object  System.*err*.println("\*\* Display the objects newly created.");  System.*err*.println(myFirstAssignment1);  System.*err*.println(myFirstAssignment2);    // assignment  myFirstAssignment2 = myFirstAssignment1;  System.*err*.println("\*\* Display the objects after '=' assignment.");  System.*err*.println(myFirstAssignment1);  System.*err*.println(myFirstAssignment2);    // modification of assigned object  myFirstAssignment2.setValue(3);  System.*err*.println("\*\* Display the objects after a changment on assigned object.");  System.*err*.println(myFirstAssignment1);  System.*err*.println(myFirstAssignment2);  }  } |
| Programme principal d’illustration  (*Code accessible dans course01/ObjectAssignment)* |
| \*\* Display the objects newly created.  The value is '1'.  The value is '2'.  \*\* Display the objects after '=' assignment.  The value is '1'.  The value is '1'.  \*\* Display the objects after a changment on assigned object.  The value is '3'.  The value is '3'. |
| Sortie du programme principal d’illustration  (*Code accessible dans course01/ObjectAssignment)* |

Contrairement à des langages comme C/C++, Java détruit les objets ‘inutilisés’ automatiquement.

#### Définition d’une classe

Une classe est composée d’un ensemble d’attributs et de méthodes. L’exemple suivant montre les différentes formes de visibilité d’un attribut ou d’une méthode.

|  |  |
| --- | --- |
| **package** edu.iut;  /\*\*  \*  \* My FirstObject is used to illustrate  \* the course. This object do nothing  \*  \*/  **public** **class** MyFirstObject {    **public** **int** myPublicAttribute;  **private** **double** myPrivateDoubleAttribute;  **protected** String myPrivateStringAttribute;    /\*\*  \* Default constructor  \*/  **public** MyFirstObject()  {}    /\*\*  \* Display the string "My first method." on error  \*/  **public** **void** myFirstMethod() {  System.*err*.println("My first method.");  }    /\*\*  \* Method without argument displaying a string  \*  \*/  **private** **void** myPrivateMethod()  {  System.*out*.println("This method is not visible");  }  /\*\*  \* Method without argument displaying a string  \*  \*/  **protected** **void** myProtectedMethod()  {  System.*out*.println("This method is not visible but by the daughter classes");  }    } | |
| (*Code accessible dans course01/FirstJavaObject*) | |
| **public** | Une méthode ou un attribut déclaré comme ‘public’ est visible par tous les objets |
| **protected** | Une méthode ou un attribut déclaré comme ‘protected’ est visible par tous les objets du même package que la classe ou ses sous classes. |
| **private** | Une méthode ou un attribut déclaré comme ‘private’ n’est visible que par la classe le déclarant |
| **« rien »** | Une méthode ou un attribut est accessible pour les toutes les classes du package |

#### Les mots clés associés aux classes et à leur contenu

|  |  |
| --- | --- |
| final | un attribut ‘final’ ne peut pas être modifié une fois instancié. Une méthode ‘final’ ne peut-être redéfinie dans une classe fille |
| static | Un attribut ou une méthode ‘static’ est commun à toute les sous les classes. Il est inutile d’instancier une classe pour accéder à ses éléments ‘static’ |
| abstract | Une méthode ou une classe ‘abstract’ est un élément sans code. L’implémentation sera déléguée aux classes filles |
| volatile | En programmation concurrentielle (‘threads’), les modifications d’une variable déclarée comme ‘volatile’ sont accessible à tous les threads. |
| synchronized | Permet de gérer l’accès concurrent aux méthodes et variables |

### Notions d’héritage & d’interfaces

L’héritage permet de définir la relation entre deux classes : une classe mère et un classe fille. L’héritage permet, entre autre d’étendre les fonctionnalités d’une classe, par exemple, on peut s’intéresser à une classe ‘Employee’ qui hérite d’une classe ‘Person’ :

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* Object defining a Person  \* Here the person property is the name  \*  \*/  **public** **class** Person {  **private** String name;  /\*\*  \* Default constructor  \* set the name with empty string  \*/  **public** Person() {  name = "";  }    /\*\*  \* Set the name  \* **@param** name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the name of the person  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@return** a string containing the name of the person  \*/  **public** String toString()  {  **return** "The person name is '" + name +"'.";  }  } |
| Définition d’une classe ‘Person’  (*Code accessible dans course01/MyInheritance*) |

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* Employee object extends Person by adding the salary  \*  \*/  **public** **class** Employee **extends** Person {  **private** **double** salary;  /\*\*  \* Default constructor set salary to  \* 0  \*/  **public** Employee(){  salary = 0.0;  }  /\*\*  \* Set the salary  \* **@param** salary value to set  \*/  **public** **void** setSalary(**double** salary) {  **this**.salary = salary;  }  /\*\*  \* **@return** the salary of the employee  \*/  **public** **double** getSalary() {  **return** salary;  }  /\*\*  \* **@return** the name of the person and the salary  \* by accessing the super class 'Person'  \* and concat the value of the salary.  \*/  **public** String toString()  {  **return** **super**.toString() + " The salary is '"+salary+"'.";  }  } |
| Définition d’une classe ‘Employee’ héritant d’une classe ‘Person’  (*Code accessible dans course01/MyInheritance*) |

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* This program show an Inheritance case  \*  \*/  **public** **class** MyInheritance {  **public** **static** **void** main(String[] args) {    Employee myEmployee = **new** Employee();  myEmployee.setName("John Doe."); // from Person class  myEmployee.setSalary(5000.0); // from Employee class  System.*err*.println(myEmployee); // use implicilty 'toString' to display  }  } |
| The person name is 'John Doe.'. The salary is '5000.0'. |
| Programme principal |

La classe ‘*Employee*’ hérite bien des méthodes de la classe ‘*Person*’.

#### Méthodes & classes abstraites

Une classe abstraite permet de définir un ensemble de méthode non instancié. Ces méthodes devront être redéfinies et instanciées dans les classes filles.

L’exemple suivant illustre une utilisation possible d’une méthode abstraite :

|  |
| --- |
| **package** edu.iut;  **public** **abstract** **class** Figure {  /\*\*  \* scale factor use to transform  \* a figure  \*/  **private** **double** scaleFactor;    /\*\*  \* Default constructor  \* set scale factor to 1.0  \*/  **public** Figure() {  scaleFactor = 1.0;  }    /\*\*  \*  \* **@param** scaleFactor to set and apply scale factor  \*/  **public** **void** setScaleFactor(**double** scaleFactor) {  **this**.scaleFactor = scaleFactor;  **this**.applyScaleFactor();  }    /\*\*  \*  \* **@return** the scale factor  \*/  **public** **double** getScaleFactor() {  **return** scaleFactor;  }    /\*\*  \* return a string with scale factor  \*/  **public** String toString()  { **return** "Scale factor is '"+scaleFactor+"'."; }    /\*\*  \* abstract method specific to implemented  \* figure. This method should transform the  \* parameters of a Figure  \*/  **public** **abstract** **void** applyScaleFactor();  } |
| La fonction abstraite ‘*applyScaleFactor*’ est appelée dès que l’on modifie la valeur de ‘*scaleFactor*’  (*Code accessible dans course01/MyFigure*) |

|  |
| --- |
| **package** edu.iut;  **public** **class** Rectangle **extends** Figure {  **private** **double** height; // heigth of the rectangle  **private** **double** width; // width of the rectangle    /\*\*  \* Default constructor  \* set height and width to default value 1.0  \*/  **public** Rectangle() {  **super**();  **this**.height = 1.0;  **this**.width = 1.0;  }    /\*\*  \* Override function applyScaleFactor  \* modify the value width and height  \* are multiply by the scale factor.  \*/    @Override  **public** **void** applyScaleFactor() {  **this**.width \*= **super**.getScaleFactor();  **this**.height \*= **super**.getScaleFactor();  }    /\*\*  \* **@param** width of the rectangle  \*/  **public** **void** setWidth(**double** width) {  **this**.width = width;  }  /\*\*  \*  \* **@return** width of the rectangle  \*/  **public** **double** getWidth()  { **return** **this**.width; }    /\*\*  \* **@param** height of the rectangle  \*/  **public** **void** setHeight(**double** height) {  **this**.height = height;  }  /\*\*  \* **@return** height of the rectangle  \*/  **public** **double** getHeight() {  **return** **this**.height;  }    **public** String toString() {  **return** **super**.toString()  + " The rectangle properties are W:"  + **this**.width+" and H:"  + **this**.height;  }  } |
| La surchage de la méthode abstraite ‘*applyScaleFactor*’ hérité de la classe ‘*Figure*’ permet de multiplier la longueur et la largeur d’un rectangle par la value de ‘*scaleFactor*’.  (*Code accessible dans course01/MyFigure*) |

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* This program show how and abstract method work  \*  \*/  **public** **class** FigureScaling {  **public** **static** **void** main(String[] args) {  Rectangle rectangle = **new** Rectangle();  rectangle.setWidth(4);  rectangle.setHeight(2);  rectangle.setScaleFactor(2.5);  System.*err*.println(rectangle);  }  } |
| Scale factor is '2.5'. The rectangle properties are W:10.0 and H:5.0 |
| Le programme principal illustre l’utilisation de la classe ‘*Rectangle*’ |

#### Interface

En java les interfaces peuvent être vues comme des classes complétement abstraites. Cependant, contrairement aux classes abstraites, les objets peuvent hériter de plusieurs interfaces. Ainsi, toutes les méthodes des interfaces doivent être implémentées dans les classes filles.

|  |  |
| --- | --- |
| **package** edu.iut;  **public** **interface** Worker {  **public** **void** work();  **public** **void** makeMyBossFurious();  } | **package** edu.iut;  **public** **interface** Vacationer {  **public** **void** haveANap();  **public** **void** goSwimming();  } |

|  |  |
| --- | --- |
| **package** edu.iut;  **public** **class** Person **implements** Worker{  @Override  **public** **void** work() {  System.err.println("I'm so busy ...");  }  @Override  **public** **void** makeMyBossFurious() {  System.err.println("I browse the web during the meeting.");  }  } | **package** edu.iut;  **public** **class** Student **implements** Vacationer{  @Override  **public** **void** haveANap() {  System.err.println("Study, learn java and so on it's a bit much ! I want to sleep !");  }  @Override  **public** **void** goSwimming() {  System.err.println("No I can't go to the Java course, I have to go to swimming pool.");  }  } |

|  |
| --- |
| **package** edu.iut;  **public** **class** Teacher **implements** Vacationer, Worker {  @Override  **public** **void** haveANap() {  System.err.println("It's too difficult to sleep and make a course.");  }  @Override  **public** **void** goSwimming() {  System.err.println("Hum sunny day, make a course on the beach ...");  }  @Override  **public** **void** work() {  System.err.println("Please stop to talk during the course !");  }  @Override  **public** **void** makeMyBossFurious() {  System.err.println("This example of interface is a bit much, let us be serious !");  }  } |

|  |
| --- |
| **package** firststepinterface;  **public** **class** FirstStepInterface {  /\*\*  \* **@param** args the command line arguments  \*/  **public** **static** **void** main(String[] args) {  edu.iut.Vacationer vacationer = **new** edu.iut.Student();  edu.iut.Vacationer vacationer2 = **new** edu.iut.Teacher();  edu.iut.Worker person = **new** edu.iut.Person();  edu.iut.Worker person2 = **new** edu.iut.Teacher();    System.err.println("\*\*\*\*\*\* Vacationer is a student \*\*\*\*\*\*");  vacationer.haveANap();  vacationer.goSwimming();    System.err.println("\*\*\*\*\*\* Vacationer is a teacher \*\*\*\*\*\*");  vacationer2.haveANap();  vacationer2.goSwimming();    System.err.println("\*\*\*\*\*\* Worker is a person \*\*\*\*\*\*");  person.makeMyBossFurious();  person.work();    System.err.println("\*\*\*\*\*\* Worker is a teacher \*\*\*\*\*\*");  person2.makeMyBossFurious();  person2.work();    }  } |

|  |
| --- |
| **\*\*\*\*\*\* Vacationer is a student \*\*\*\*\*\***  **Study, learn java and so on it's a bit much ! I want to sleep !**  **No I can't go to the Java course, I have to go to swimming pool.**  **\*\*\*\*\*\* Vacationer is a teacher \*\*\*\*\*\***  **It's too difficult to sleep and make a course.**  **Hum sunny day, make a course on the beach ...**  **\*\*\*\*\*\* Worker is a person \*\*\*\*\*\***  **I browse the web during the meeting.**  **I'm so busy ...**  **\*\*\*\*\*\* Worker is a teacher \*\*\*\*\*\***  **This example of interface is a bit much, let us be serious !**  **Please stop to talk during the course** |

Dans cet exemple, la classe ‘*Person*’ instancie les méthodes de l’interface interface ‘*Worker*’, par la même occasion ‘*Person*’ ‘hérite’ du type ‘*Worker*’. La classe ‘*Student*’ hérite de la classe ‘*Vacationer*’. Pour finir la classe ‘*Teacher’* hérite des deux interfaces et implémente les méthodes des deux interfaces.

Les instantiations:

edu.iut.Vacationer vacationer = **new** edu.iut.Student();

edu.iut.Vacationer vacationer2 = **new** edu.iut.Teacher();

edu.iut.Worker person = **new** edu.iut.Person();

edu.iut.Worker person2 = **new** edu.iut.Teacher();

créent des objets du type des interfaces.

La particularité des objets vacationer2 et person2 étant qu’ils sont instanciés à partir de la même classe ‘*Teacher*’. Il est a noté que la définition de l’interface limite l’utilisation des méthodes de l’objet ‘*Teacher’*. Ainsi il ne sera **pas possible** de faire person2.goSwimming();

# Outils de développements

Les outils de développements sont à la base de la création d’une application. Les environnements de développement associés à Java sont généralement gratuit et offre une large gamme d’outils permettant de créer une application.

## Compilation

Un programme Java ne peut s’exécuter sans compilation. Cette phase ‘transforme’ un code intelligible en *byte code* (format compréhensible par la JVM). Cette phase de compilation est opérée par le programme ‘javac’.

### Premiers pas

#### Préparer son environnment

Sous Windows le kit de développement Java est généralement installé dans **c:\program file.** Sous Linux c’est un peu plus libre nous supposerons que ce kit de développement se trouve dans le dossier **/opt**.

#### Cas d’utilisation

La création d’un premier programme passe par le fameux *Hello World*.

|  |
| --- |
| /\*\*  \* Hello World is a simple program which display a string  \*  \*/  **class** HelloWorld  {  /\*\*  \* Main program display the string 'Hello world.'  \* **@param** args program parameters  \*/  **public** **static** **void** main(String[] args)  {  System.out.println("Hello world.");  }  } |

La compilation de ce code est effectuée à l’aide de la commande javac

Dossier: **handson/FirstSteps1**

javac src/HelloWorld.java –d build

Cette commande permet de compiler le programme ‘HelloWorld.java’. Le résultat (le *byte code*) généré est placé dans de dossier ‘build’.

En allant dans le dossier ‘build’, on trouve un fichier ‘HelloWorld.class’. Pour exécuter il suffit d’écrire la commande suivante :

java HelloWorld

>>Hello world.

Lors de développements complexes, il est d’usage de séparer les codes métier et de créer des *packages*

L’exemple FirstStep2 illustre cette notion en créant un *package* ***edu.iut***

|  |
| --- |
| **package** edu.iut;  /\*\*  \* Hello World is a simple program which display a string  \*/  **class** HelloWorld  {  /\*\*  \* Main program display the string 'Hello world.'  \* **@param** args program parameters  \*/  **public** **static** **void** main(String[] args)  {  HelloWorldDisplayer helloWorld = **new** HelloWorldDisplayer(3);  System.out.println(helloWorld);  }  } |

|  |
| --- |
| **package** edu.iut;  **import** javax.swing.event.\*;  /\*\*  \* Hello World is a simple class which repeat  \* a string  \*/  **public** **class** HelloWorldDisplayer  {  **private** String[] arrayOfHelloWorld = **null**;  **public** HelloWorldDisplayer(**int** repeatN)  {  arrayOfHelloWorld = **new** String[repeatN];  **for** (**int** ri = 0;ri<repeatN;ri++)  {  arrayOfHelloWorld[ri] = "Hello World "+ri;  }  }  /\*\*  \* **@return** a string contaiing a list of 'Hello World'  \*/  **public** String toString()  {  String concatenedStrings = "";  **if** (arrayOfHelloWorld != **null**)  {  **for** (**int** si = 0;si<arrayOfHelloWorld.length;si++)  {  concatenedStrings+=(arrayOfHelloWorld[si] + "\n");  }  }  **return** concatenedStrings;  }  **public** **void** doWhile()  {  **boolean** condition = **true**;  **int** i = 0;  **do**  {  condition = (i++ < 10);  /\* do something until condition is true \*/  }**while**(condition);  }  } |

Dossier: **handson/FirstSteps2/src**

javac edu/iut/HelloWorld.java edu/iut/HelloWorldDisplayer.java –d build

Cette commande permet de compiler le programme ‘HelloWorld.java’ et la classe ‘HelloWorldDisplayer.java’. Le résultat (le *byte code*) généré est placé dans de dossier ‘build’.

En allant dans le dossier ‘build’, on trouve un dossier ‘edu/iut’. Pour exécuter il suffit d’écrire la commande suivante :

java edu.iut.HelloWorld

>>Hello world 1

>>Hello world 2

>>Hello world 3

Un *package* peut être vu comme un ensemble de codes métiers. Ces codes peuvent être isolés dans une bibliothèque réutilisable

### Créer et utiliser une bibliothèque

L’outil jar permet de créer une bibliothèque. Communément, une bibliothèque regroupe un ensemble de fonctionnalités.

L’exemple *FirstStepJar* illustre la création d’une bibliothèque offrant un service permettant de dire « bonjour le monde » dans plusieurs langues.

L’archive est composée

* d’une interface

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* Interface of HelloWorld  \* sayHellWorld should return "Hello world" translate in different languages  \*  \*/  **public** **interface** IHelloWorld {  /\*\*  \*  \* **@return** the string "Hello world" translated in different language  \*/  **public** String sayHelloWorld();  } |

* d’une Usine à objet (*Design Pattern*) simplifiée :

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* HelloWorldFactory create Object of type IHelloWorld (an interface)  \*  \*  \*/  **public** **class** HelloWorldFactory {  /\*\*  \* Default constructor  \*/  **public** HelloWorldFactory()  {}    /\*\*  \*  \* **@param** language is the language for the translation of "Hello World"  \* **@return** an IHelloWorld instanciate with HelloWorldEN or HelloWorldFR  \* **@throws** LanguageException when the language is unknown  \*/  IHelloWorld getHelloWorld(String language) **throws** LanguageException  {  **if** (language == "en")  {  **return** **new** HelloWorldEN();  }  **if** (language == "fr")  {  **return** **new** HelloWorldFR();  }  **throw** **new** LanguageException("Language '"+language+"' is unknown.");  }  } |

* de deux instanciations de l’interface *IHelloWorld*:

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* HelloWorldFR implements HelloWorldInsterface (IHelloWorld)  \* This object implement the function "sayHelloWorld" to  \* return "Hello World"  \*  \*/  **public** **class** HelloWorldEN **implements** IHelloWorld {  /\*\*  \* Default contructor  \*/  **public** HelloWorldEN()  {}    /\*\*  \* Implementation of the interface IHelloWorld  \* **@return** the string "Hello World"  \*/  @Override  **public** String sayHelloWorld() {  **return** "Hello World";  }  } |
| **package** edu.iut;  /\*\*  \*  \* HelloWorldFR implements HelloWorldInsterface (IHelloWorld)  \* This object implement the function "sayHelloWorld" to  \* return "Hello World" translated in French  \*  \*/  **public** **class** HelloWorldFR **implements** IHelloWorld {  /\*\*  \* Default contructor  \*/  **public** HelloWorldFR()  {}    /\*\*  \* Implementation of the interface IHelloWorld  \* **@return** the string "Bonjour le monde" tranlation of "Hello World" in french  \*/  @Override  **public** String sayHelloWorld() {  **return** "Bonjour le monde";  }  } |

* d’une classe d’exception permettant de soulever des exceptions lorsqu’un langage n’est pas géré :

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* Exception Class managing or unknwon language  \* This Exception is raised when a language is not managed by  \* the HelloWorldFactor object  \*  \*/  **public** **class** LanguageException **extends** Exception {  /\*\*  \* Default constructor  \*/  **public** LanguageException()  { **super**(); }    /\*\*  \* Construct an exception with a message  \* **@param** message recorded by the exception  \*/  **public** LanguageException(String message)  { **super**(message); }  /\*\*  \* Construct a exception from another exception  \* **@param** cpy is a the exception to copy  \*/  **public** LanguageException(LanguageException cpy)  { **super**(cpy); }  } |

* d’un programme principal :

|  |
| --- |
| **package** edu.iut;  /\*\*  \*  \* Main program  \*  \*/  **public** **class** FamousHelloWorld {  **public** **static** **void** main(String[] args) {  HelloWorldFactory helloWorldFactory = **new** HelloWorldFactory();  **try**  {  System.*err*.println(helloWorldFactory.getHelloWorld("fr").sayHelloWorld());  System.*err*.println(helloWorldFactory.getHelloWorld("en").sayHelloWorld());  System.*err*.println(helloWorldFactory.getHelloWorld("gr").sayHelloWorld());  }  **catch**(LanguageException langEx)  {  System.*err*.print(langEx);  }  }  } |

Par exemple dans le dossier *‘handson/FirstStepJar’*, sous linux, Pour créer une archive il faut procéder comme suit :

Afin de construire une archive il faut dans un premier temps ‘compiler’ ces objets :

* Aller dans dans le dossier src/edu/iut puis taper la commande

***javac FamousHelloWorld.java HelloWorldEN.java HelloWorldFactory.java HelloWorldFR.java IHelloWorld.java LanguageException.java -d ../../../build/***

* Aller dans le dossier build puis taper la commande

***jar cf helloworld.jar edu***

* Pour finir, pour exécuter le programme taper

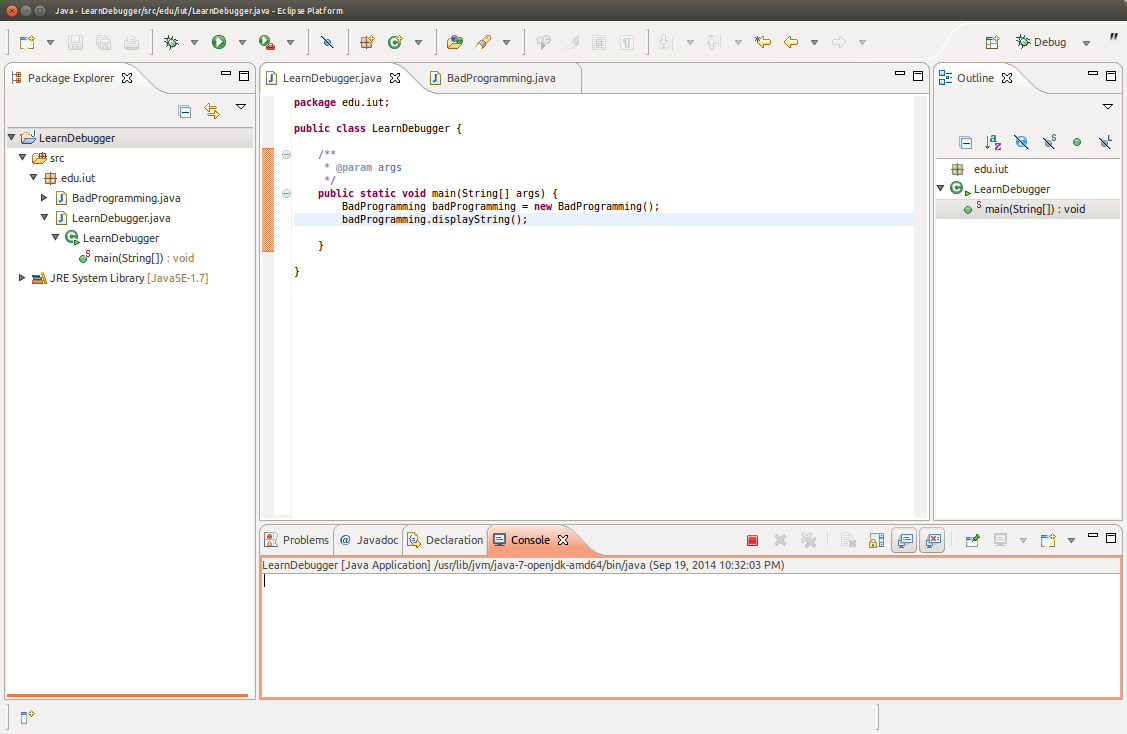
***java edu.iut.FamousHelloWorld -jar helloworld.jar***

## Debugger

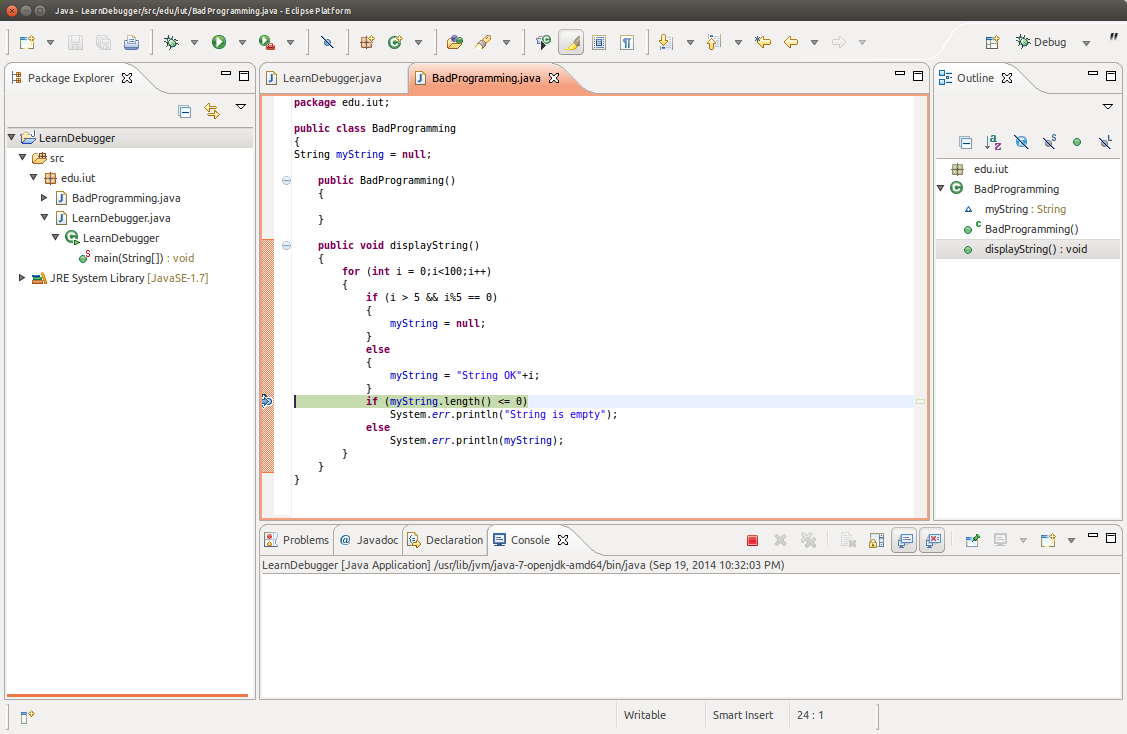
Le debugger permet de suivre l’exécution d’un programme pas à pas. Dès lors, il permet l’accès aux structures de données et à leur contenu au moment de l’exécution.

Afin d’illustrer l’utilisation du debugger intégré à Eclipse, la classe ‘BadProgramming’ comportant des erreurs a été créée. Plus précisément cette classe instancie une méthode dans laquelle on essaie d’accéder à une méthode d’un objet ‘null’.

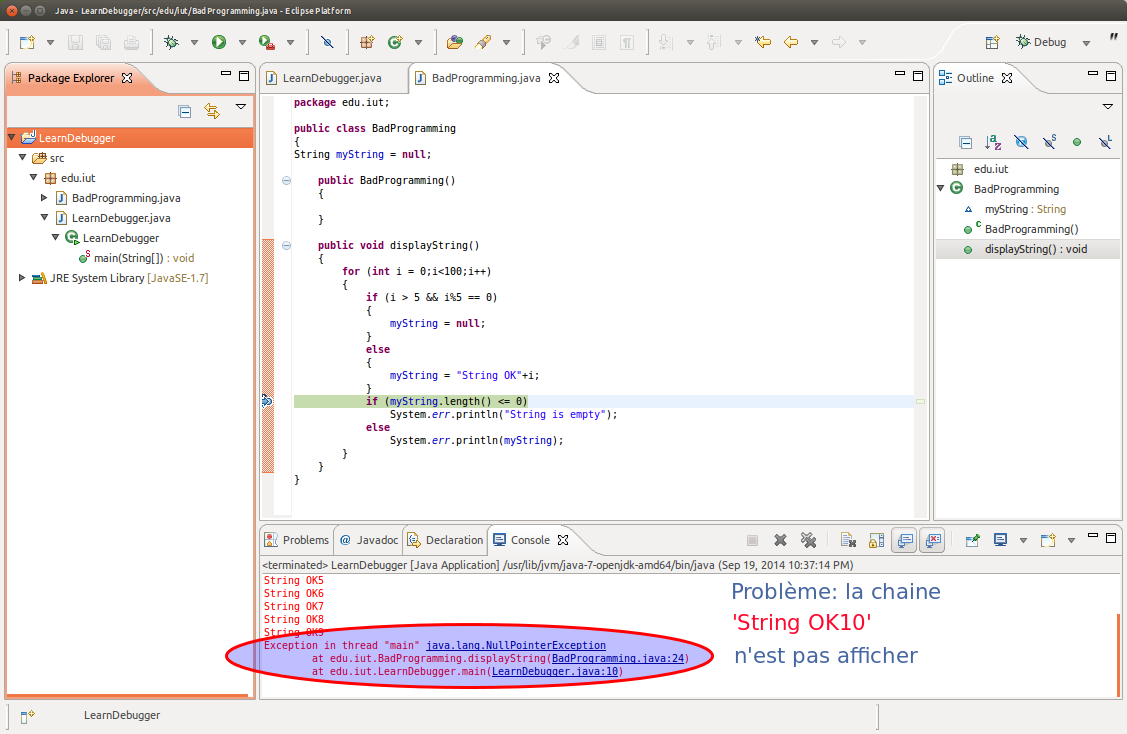
Le fichier *LearningDebugger.java* contient le programme principal qui ne fait rien d’autre qu’appeler la classe d’étude *BadProgramming****.*** La méthode posant un problème étant *displayString()*.



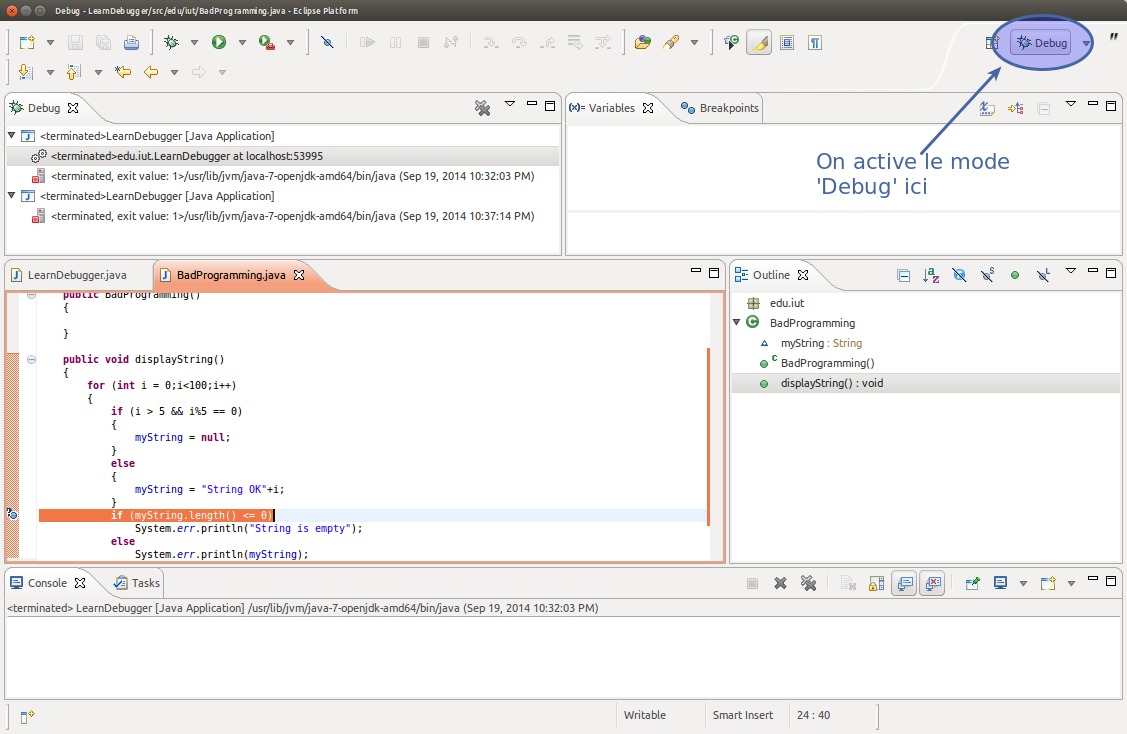
On essaie d’appeler la méthode *length()* de l’objet *myString*  de type String. Or il arrive que cet objet soit *null*. Cela introduit un problème.



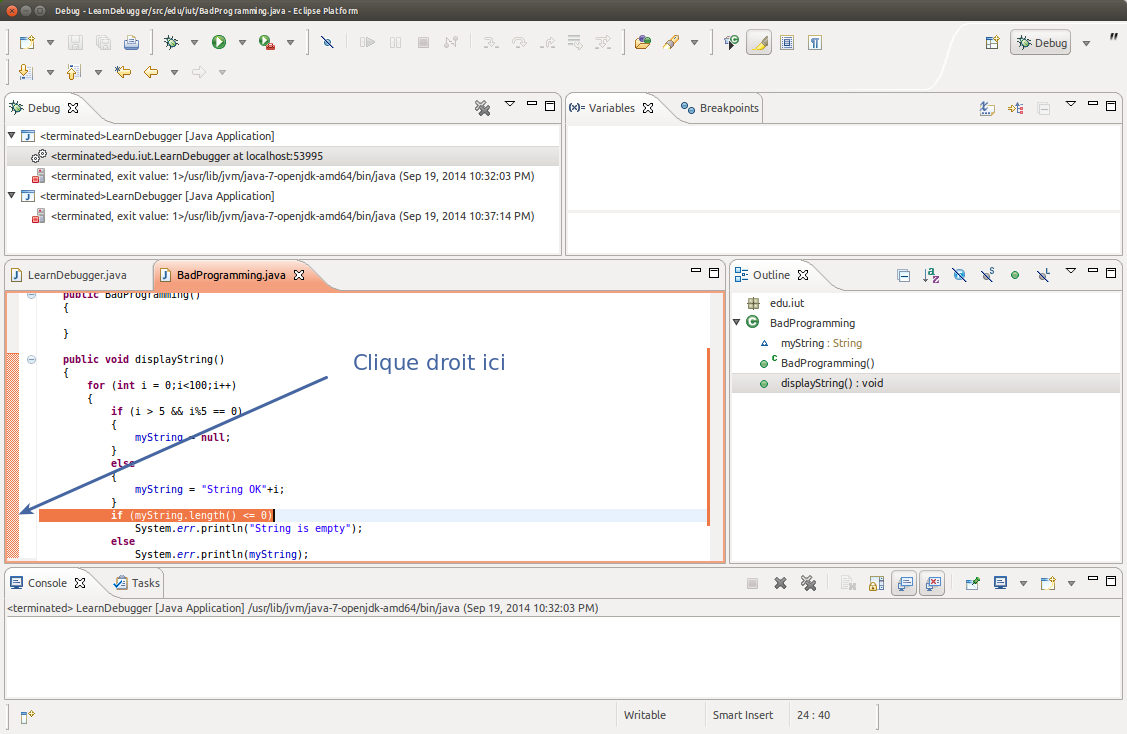
Ainsi lors de l’exécution du programme une exception *NullPointerException* est levée et le programme s’arrête.

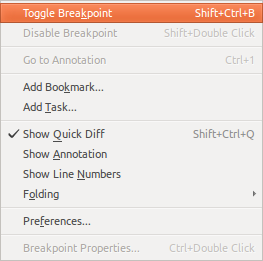


Lorsque l’exception est levée on dispose d’une partie de la pile d’appel et de la ligne où le problème à surgit. En utilisant le débugger, il est possible de stopper le cours de l’exécution du programme afin de ‘regarder’ ce qui se produit

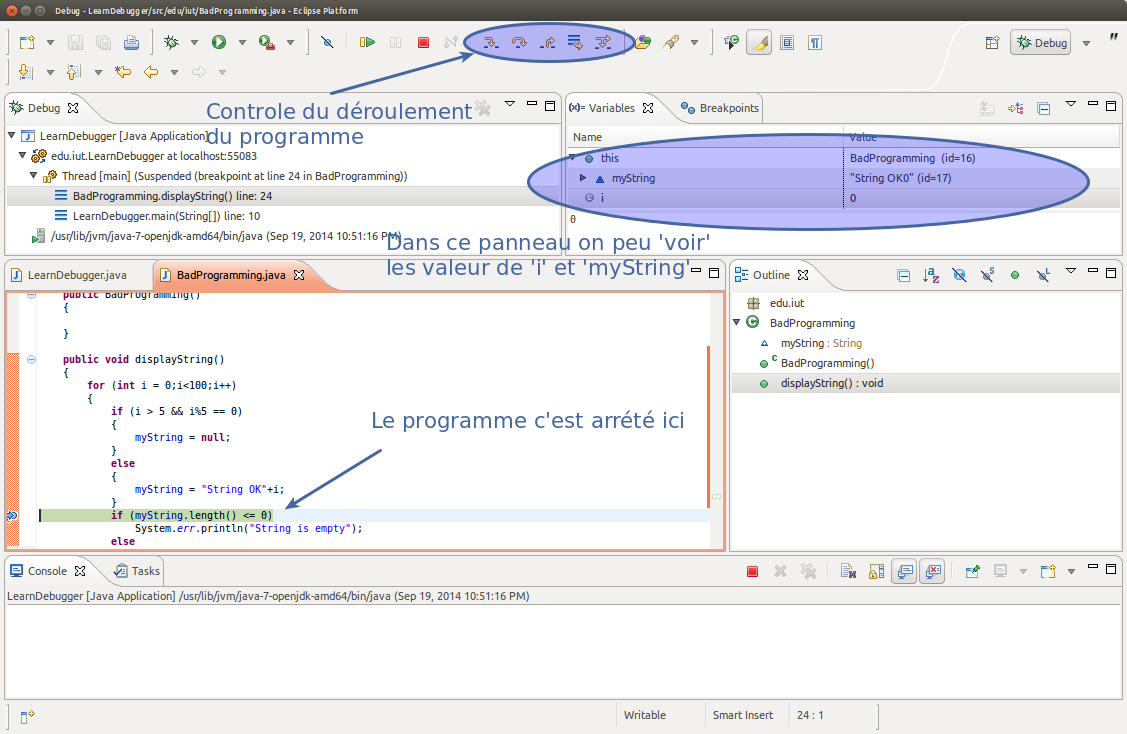


Pour ce faire, il suffit de positionner un point d’arrêt sur la ligne concernée :

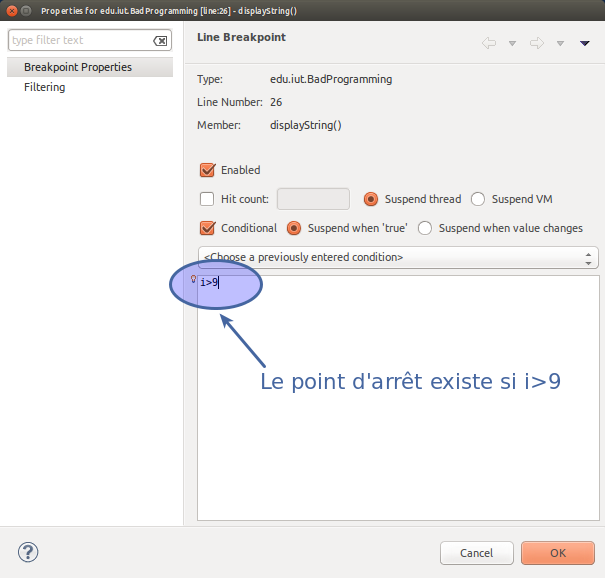




Le programme sera stoppé à chaque fois qu’il atteindra cette ligne. Il sera alors possible d’étudier le problème en observant l’évolution des variables.

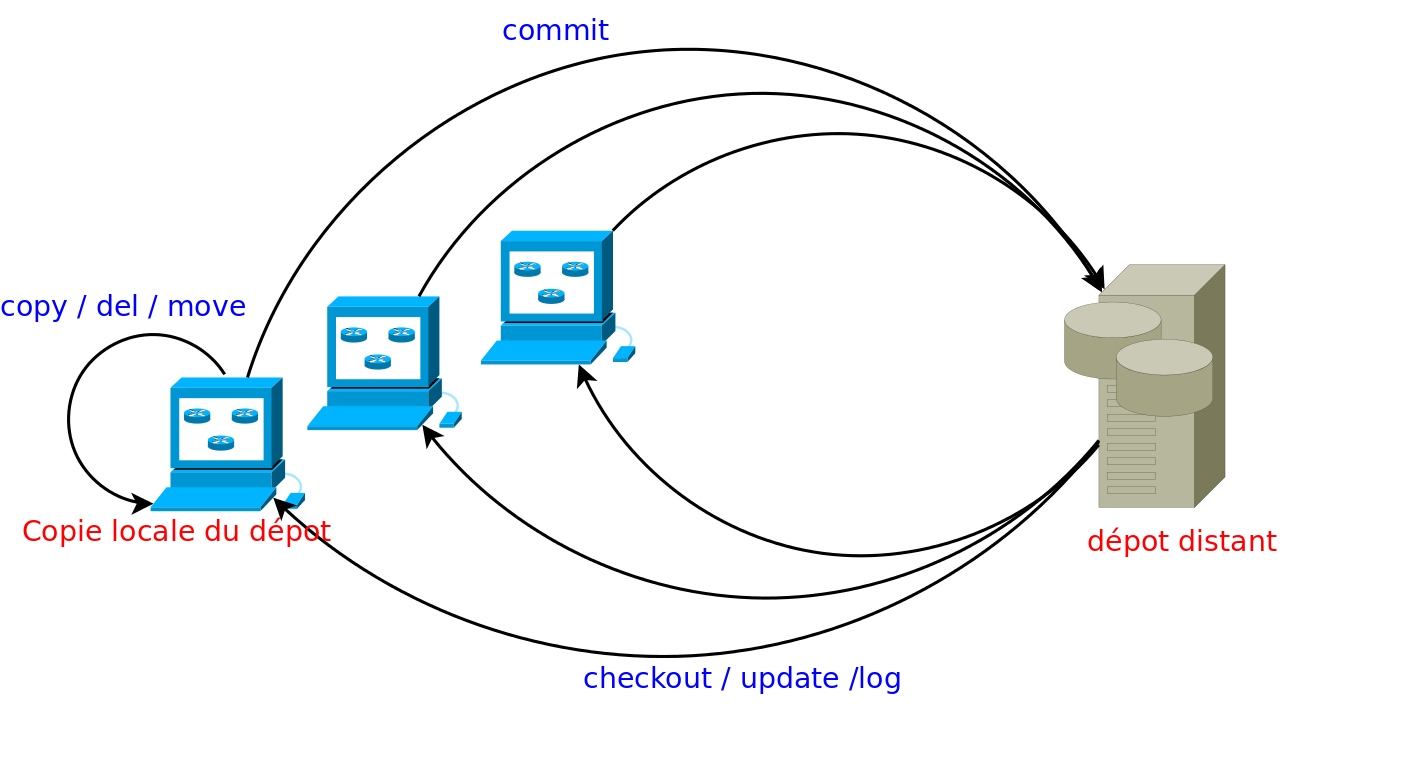


Dans le problème présenté, l’erreur est située dans une boucle et n’intervient qu’à l’itération 10. Pour éviter l’arrêt à chaque itération, il est possible d’ajouter des conditions sur le point d’arrêt. Typiquement ici, il suffira de stopper le programme dès lors que 9 itérations ont été effectuées.



## Gestionnaire de version (subversion)

Subversion est un logiciel de contrôle de versions. Il permet d’enregistrer les évolutions du code. Ainsi, il est possible, par exemple, de revenir en arrière.



### Notion de dépots

Subervsion met en place une architecture client/serveur. Le dépôt (‘repository’) et placé sur un serveur distant et permet de garder une trace de toutes les modifications. Un utilisateur de subversion peut récupérer la version courante du dépôt (ou choisir une version) par le biais d’un client subversion.

La création d’un dépôt (côté serveur) se fait comme suit :

>>svn create myrepository

### Copie locale

La copie locale permet de récupérer le contenu d’un dépôt. La récupération d’un dépôt peut être effectuée par le biais de plusieurs protocoles réseaux (http, ssh, fichier local, serveur svn). Le choix du (ou des) protocole est à la discrétion de l’administrateur de dépôt.

>>svn checkout http://mywebhost/myrepository

ou

>>svn checkout svn+ssh://mysshhost/myrepository

ou

>>svn checkout file://mypath/myrepository

ou

>>svn checkout svn://mysvnhost/myrepository

Il est possible de récupérer une version spécifique du code. Autrement dit, il est possible de remonter le temps du développement

svn checkout –r 2 svn://mysvnhost/myrepository

### Mise à jour

La mise à jour de la copie locale permet de récupérer les dernières modifications effectuées. Typiquement lorsque plusieurs personnes travaillent sur un même projet elles sont amenées à le modifier en ajoutant de nouveaux fichiers ou en modifiant les fichiers existant. Pour profiter des mises à jour d’autres utilisateurs, il suffit de déplacer dans la copie locale et de taper la commande suivante :

>>svn update

ou

>>svn update theDirectoryIwantToUpdate

Ici aussi on peut faire un update d’un revision particulière du code

svn update –r 2 myOldGoodFile

### Versionner son travail

Pour sauvegarder le travail effectuer, dû à des modifications de fichiers ou des ajouts. Il suffit d’utiliser la commande

>>svn commit

Lorsqu’un fichier n’a jamais était ajouté il faut l’ajouter

>>svn add myFileOrDirectory

>> svn commit

### Supprimer, copier, déplacer

Il est parfois nécessaire de supprimer un fichier. Dans ce cas on utilisera la commande :

>> svn del myFileOrDirectory

>> svn commit

Il est également possible de renommer un fichier

>> svn move myFileOrDirectory myNewFileOrDirectory

>> svn commit

Pour finir un fichier peut-être dupliqué :

>> svn copy myFileOrDirectory myNewFileOrDirectory

>> svn commit

### Statuts, conflits, blocage de fichier de travail

Il est parfois nécessaire de connaitre le statut de la copie locale, à savoir si un fichier n’est pas versionné ou a été modifié :

>> svn status

Lorsque plusieurs personnes travaillent sur le même fichier, il se peut que lors de l’update (ou du commit) des conflits apparaissent. Dans ce cas subversion prévient l’utilisateur et propose de mélanger les codes. Ce mélange est fait automatiquement par subversion dès lors qu’il n y a pas d’ambiguïté. Sinon, un fichier de ‘merge’ est proposé et l’utilisateur doit effectuer le mélange lui-même.

Pour éviter les désagréments d’un ‘merge’ il est possible de bloquer les modifications sur un fichier de travail avec la commande

>> svn lock myWorkingFile

Pour débloquer ce fichier il suffit de taper la commande :

>> svn unlock myWorkingFile

# Bonnes pratiques

## Ecriture de code

La mise en place de quelques règles d’écritures participe au bon développement d’une application et surtout à la lisibilité du code. Cette section s’attache à **proposer** quelques règles, en aucun cas ce sont des règles issues d’un « standard », elles reflètent plus une expérience.

### Nommage

|  |  |
| --- | --- |
| Classes & Objets | Les classes devraient être composées de nom ou de phrases nominales. Il faut éviter d’utiliser un verbe. Un classe devrait toujours commencée par une majuscule est suivre la syntaxe suivante : **MaSuperbeClasse** |
| Méthodes & Variables | Les méthodes et les propriétés d’une classe devraient être construites avec un verbe signifiant une action. Les méthodes devraient commencée par une minuscule et être composé d’un verbe : par exemple une méthode pourrait avoir le nom **ajouterUnEntier** |
| Les constantes | Les constantes devraient être écrites en majuscules. Par exemple  **final int AVOGADRO\_NUMBER = 6.0221413e23** |

Les noms utilisés devraient avoir un sens. Par exemple

|  |
| --- |
| **int** i; // compteur général  **int** studentCounter; // compteur ayant une signification |

### Déclarations

Il est préférable de faire une déclaration par ligne :

|  |  |
| --- | --- |
| Ne pas faire | **double** resultOfFunction, resultOfProcess; |
| Préférer | **double** resultOfFunction;  **double** resultOfProcess; |

Les initialisations des variables sont à faire dès que possible.

### Les accolades

Dans les branchements ou les boucles il est préférable d’utiliser les accolades ainsi le code :

|  |
| --- |
| **if** (myCondition) myVariable = 3; |

n’est pas recommandé, il faudrait écrite

|  |
| --- |
| **if** (myCondition) {  myVariable = 3;  } |

### Les espaces et l’indentation

La clarté du code passe également par une bonne indentation du code et des règles concernant les espaces. Par exemple, il faudrait préférer les espaces aux tabulations (les tabulations n’ont pas toujours la même tailles), mettre des espaces autour des opérateurs, aligner les déclarations …

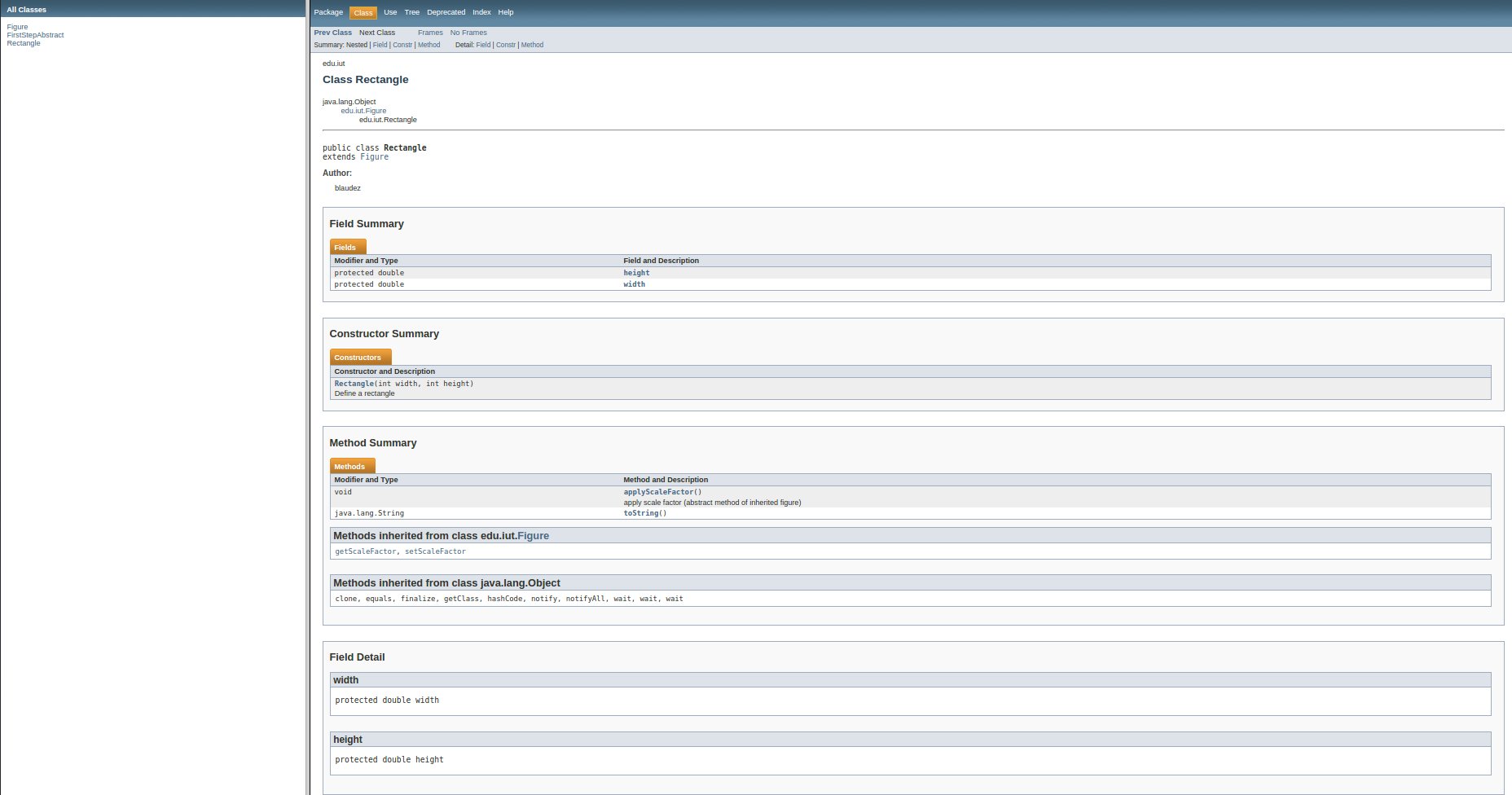
## Documentation (JAVADOC) & commentaires

Javadoc est un outil et une présentation des commentaires permettant de générer une documentation html. Cette documentation est calquée sur les informations données par le développeur au travers de commentaire et de labelles spécifique à JavaDoc.

|  |  |
| --- | --- |
| **@param** | Paramètre d’une méthode |
| **@return** | La valeur de retour d’une fonction |
| **@throws** | Les exceptions jetées par une méthode |
| **@author** | L’auteur |
| **@version** | La version |
| **@see** | Faire une référence à une autre méthode |
| **@since** | Dater la création d’une méthode |
| **@deprecated** | Marque une méthode comme dépréciée |

Le résultat de la commande :

Javadoc –d doc edu.iut -sourcepath src –version -author



## Gestion des erreurs

En java la gestion des erreurs est faite par le biais d’un système d’exceptions.

Certaines erreurs sont nativement gérées par Java. Par exemple une erreur due à une division par 0 ou encore les tentatives d’accès à des méthodes d’un objet non instancié.

On distingue 3 types d’erreurs :

* Les erreurs critiques, fatales pour le programme (classe **java.lang.Error**).
* Les erreurs à traiter (classe **java.lang.Exception**).
* Les erreurs ne peuvent pas être traitées (classe **java.lang.RuntimeException**).

Il est également possible de gérer des erreurs ‘prévues’ par le programmeur, auquel cas il faudra écrire des exceptions en adéquation avec le besoin. Par exemple, on peut imaginer une classe *Employee* disposant d’une fonction *setSalary(int salary)* où le salaire doit être supérieur ou égale à 0 ; dans le cas contraire une exception sera jetée.

La classe contenant la programme principal récupérant les exceptions :

|  |
| --- |
| **package** edu.iut;  **public** **class** FirstStepExcpetion {  /\*\*  \* **@param** args the command line arguments  \*/  **public** **static** **void** main(String[] args) {  Employee employee = **new** Employee();  **try**  {  employee.setSalary(-5);  }  **catch** (edu.iut.exceptions.EmployeeException ex)  {  System.err.println(ex);  }  **try**  {  employee.applyTheTaxes(0);  }  **catch**(java.lang.ArithmeticException aex)  {  System.err.println(aex);  }  }  } |

L’objet *Employee* d’où proviennent les exceptions ***:***

* ***ArithmeticException*** : une exception jetée automatiquement par java lorsqu’on tente de diviser le salaire par 0 dans la fonction *applyTheTaxe***s**
* **EmployeeException :** une exception personnalisée qui est jetée lorsque le salaire est négatif.

|  |
| --- |
| **package** edu.iut;  **import** edu.iut.exceptions.EmployeeException;  **public** **class** Employee {  **private** **int** salary;    **public** Employee() {  salary=0;  }    **public** **void** applyTheTaxes(**int** divide)  {  **this**.salary/=divide;  }    **public** **void** setSalary(**int** salary) **throws** EmployeeException{  **if** (salary < 0)  **throw** **new** EmployeeException("Salary should be greater or equal to 0.");  **this**.salary = salary;  }    **public** **int** getSalary() {  **return** **this**.salary;  }  } |

Pour finir le gestionnaire d’exception personnalisé :

|  |
| --- |
| **package** edu.iut.exceptions;  **public** **class** EmployeeException **extends** Exception {  **public** EmployeeException(){  **super**();  }  **public** EmployeeException(String exceptionMessage) {  **super**("Personalized:" + exceptionMessage);  }  } |

## Tests unitaires avec JUNIT

Les tests permettent de valider un code et d’éviter les régressions. JUnit est un framework de rédaction et d'exécutions de tests unitaires. Il offre un cadre de développement très simple. Un test est souvent associé à une classe. Dès lors, un test pourra combiner plusieurs tests unitaires pour valider les différentes méthodes d’une classe.

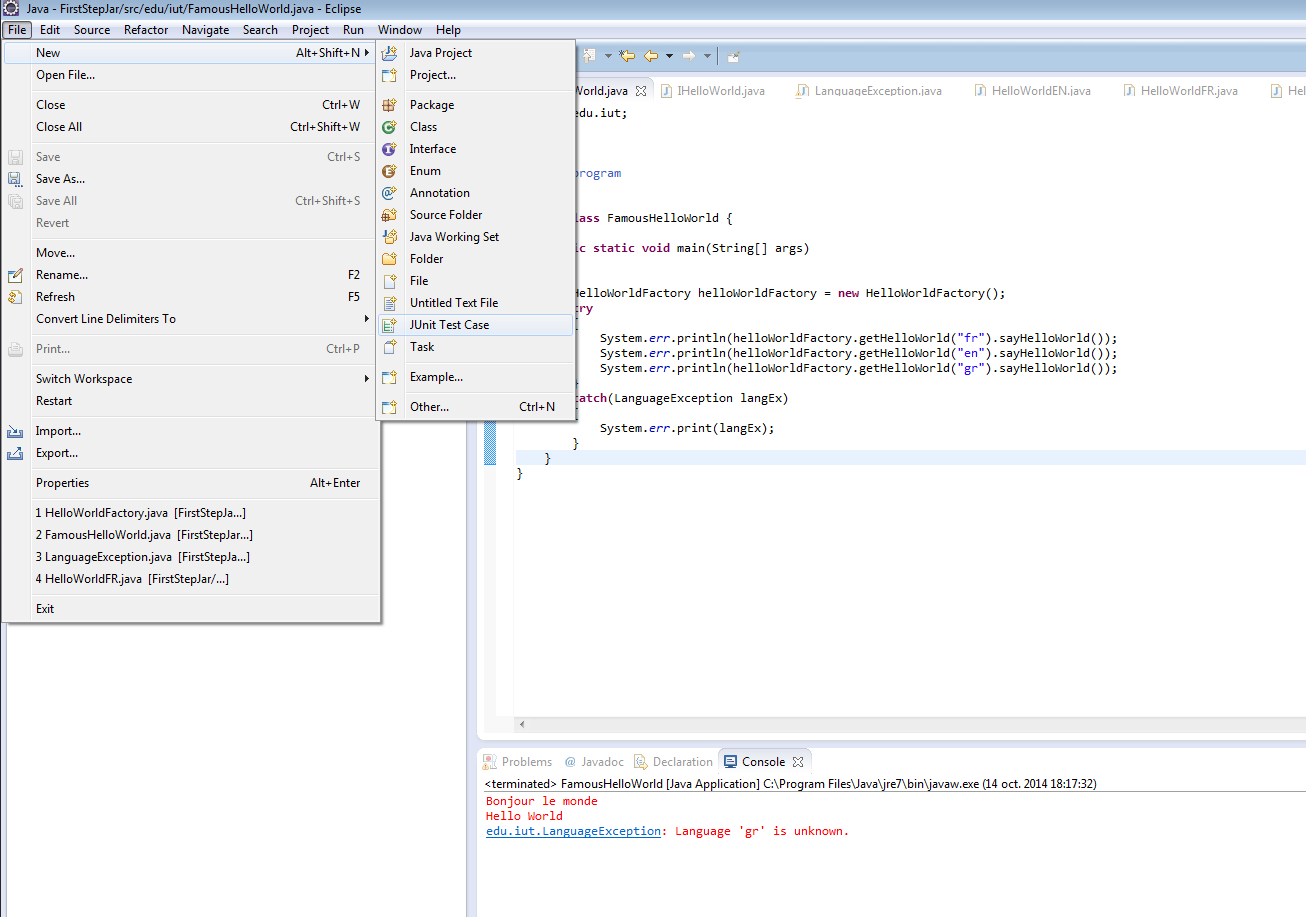
JUnit propose deux classes

* **TestCase** : le test
* **TestSuite** : un ensemble de test.

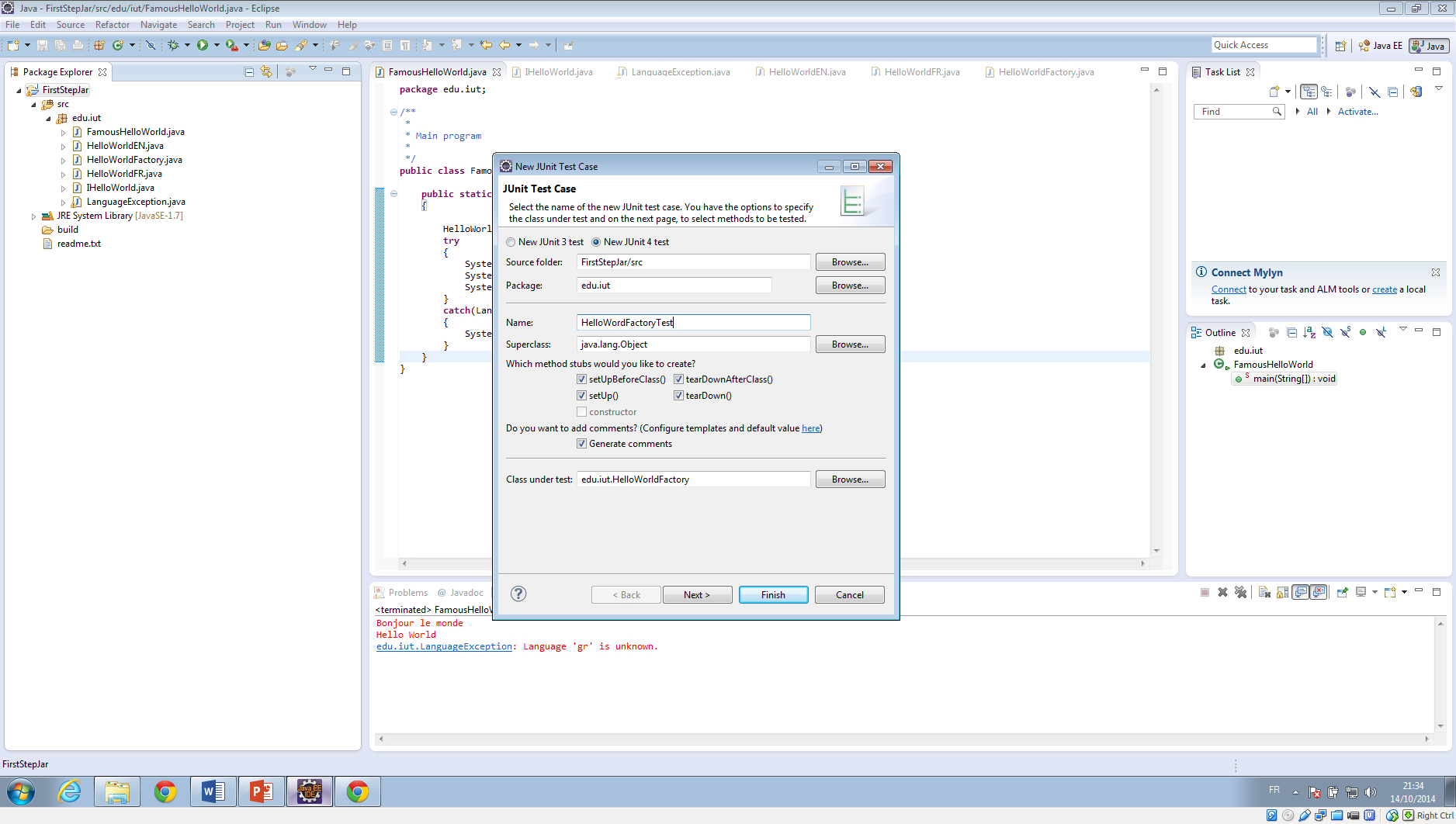
Pour faciliter la création des tests, les IDE modernes proposent généralement une mise en place automatique d’un patron de test. Il faudra « combler les trous » en écrivant le code du test.

Par Exemple, sous Eclipse, la création d’un test.

Tout d’abord créer une nouvel JUnit Test Case sur la classe « **HelloWorldFactory** » étudiée dans le chapitre « Créer et utiliser une bibliothèque »:



Puis renseigner les champs comme suit



Une nouvelle classe « **HelloWorldFactoryTest** » est créée. :

|  |
| --- |
| **package** edu.iut;  **import** **static** org.junit.Assert.\*;  **import** org.junit.After;  **import** org.junit.AfterClass;  **import** org.junit.Before;  **import** org.junit.BeforeClass;  **import** org.junit.Test;  **public** **class** HelloWorldFactoryTest {  @BeforeClass  **public** **static** **void** setUpBeforeClass() **throws** Exception {  }  @AfterClass  **public** **static** **void** tearDownAfterClass() **throws** Exception {  }  @Before  **public** **void** setUp() **throws** Exception {  }  @After  **public** **void** tearDown() **throws** Exception {  }  @Test  **public** **void** test() {  *fail*("Not yet implemented");  }  } |

Junit offre un ensemble de méthodes d’assertion permettant de contrôler les erreurs :

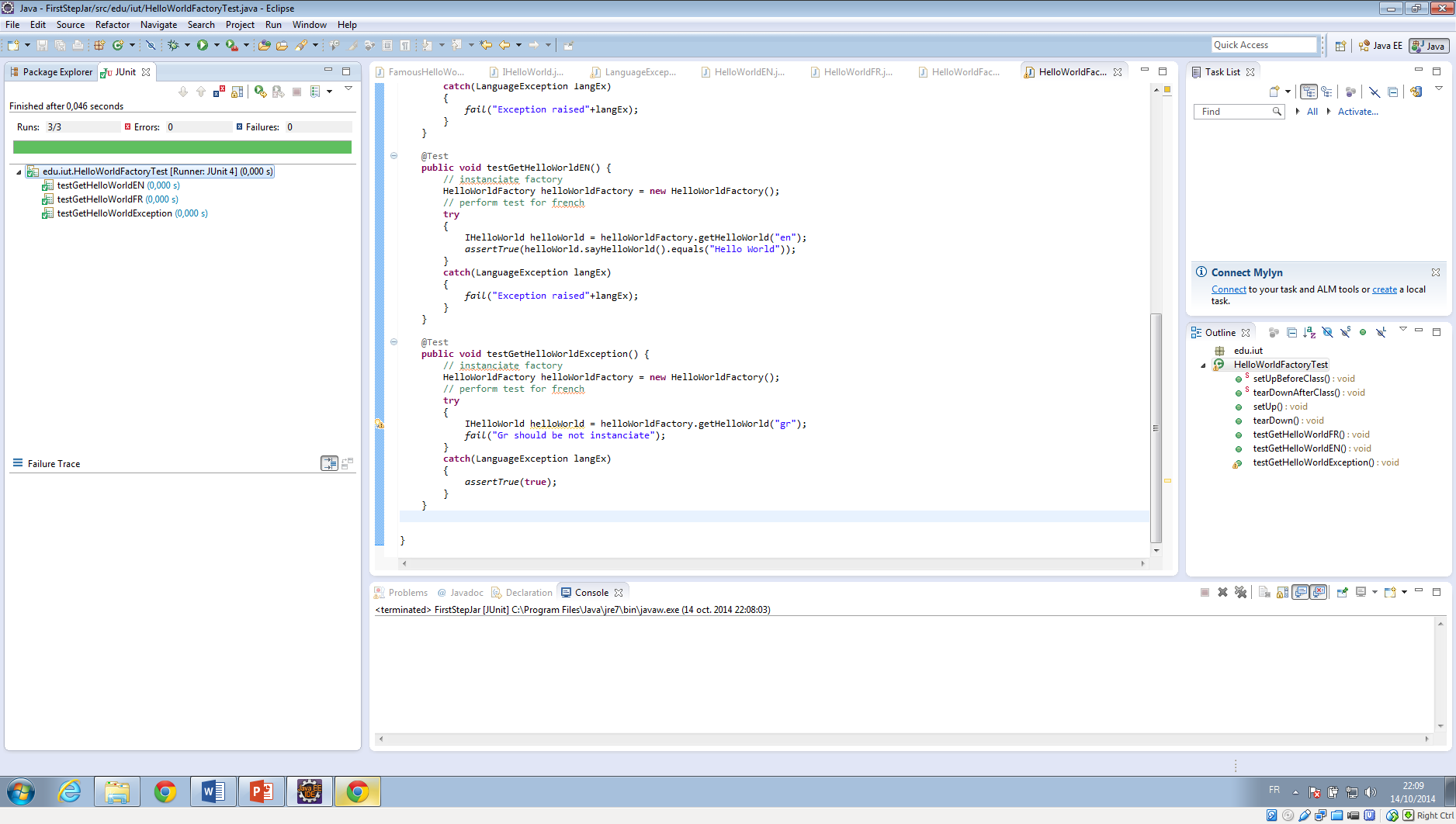
|  |  |
| --- | --- |
| assertEquals | Vérifie que deux objets sont égaux |
| assertFalse | Vérifie que l'expression est fausse |
| assertNotNull | Vérifie que l'objet n'est pas nul |
| assertNotSame | Vérifie que deux références ne sont pas les mêmes |
| assertNull | Vérifie qu'un objet est nul |
| assertSame | Vérifie que deux références sont les mêmes |
| assertTrue | Vérifie que l'expression est vraie |
| fail | Provoque l'échec du test |

Le code de test final est le suivant :

|  |
| --- |
| **package** edu.iut;  **import** **static** org.junit.Assert.\*;  **import** org.junit.After;  **import** org.junit.AfterClass;  **import** org.junit.Before;  **import** org.junit.BeforeClass;  **import** org.junit.Test;  **public** **class** HelloWorldFactoryTest {  @BeforeClass  **public** **static** **void** setUpBeforeClass() **throws** Exception {  }  @AfterClass  **public** **static** **void** tearDownAfterClass() **throws** Exception {  }  @Before  **public** **void** setUp() **throws** Exception {  }  @After  **public** **void** tearDown() **throws** Exception {  }  @Test  **public** **void** testGetHelloWorldFR() {  // instanciate factory  HelloWorldFactory helloWorldFactory = **new** HelloWorldFactory();  // perform test for french  **try**  {  IHelloWorld helloWorld = helloWorldFactory.getHelloWorld("fr");  *assertTrue*(helloWorld.sayHelloWorld().equals("Bonjour le monde"));  }  **catch**(LanguageException langEx)  {  *fail*("Exception raised"+langEx);  }  }    @Test  **public** **void** testGetHelloWorldEN() {  // instanciate factory  HelloWorldFactory helloWorldFactory = **new** HelloWorldFactory();  // perform test for french  **try**  {  IHelloWorld helloWorld = helloWorldFactory.getHelloWorld("en");  *assertTrue*(helloWorld.sayHelloWorld().equals("Hello World"));  }  **catch**(LanguageException langEx)  {  *fail*("Exception raised"+langEx);  }  }    @Test  **public** **void** testGetHelloWorldException() {  // instanciate factory  HelloWorldFactory helloWorldFactory = **new** HelloWorldFactory();  // perform test for french  **try**  {  IHelloWorld helloWorld = helloWorldFactory.getHelloWorld("gr");  *fail*("Gr should be not instanciate");  }  **catch**(LanguageException langEx)  {  *assertTrue*(**true**);  }  }  } |

Eclipse permet l’exécution du test case en choisissant dans le menu du bouton « **run**», « **JU HelloWorldFactoryTest** ».

Le résultat du test est le suivant :



Ici tous les tests sont passés avec succès.