Séance 4

Contrôle des erreurs : débogage, tests unitaires et exceptions

Cette séance est consacrée à trois mécanismes distincts permettant de détecter, gérer et/ou corriger des erreurs ; le débogage, les tests unitaires et les exceptions. Ces mécanismes s'inscrivent donc pleinement dans une démarche qualité.

Le débogage permet de suivre le déroulement pas-à-pas d'un programme de manière à s'assurer que le fonctionnement est conforme à celui attendu, au plus bas niveau. Il s'agit donc de vérifier que l'implémentation d'un algorithme est correcte.

Les tests unitaires effectuent un contrôle de plus haut niveau : un test unitaire permet de vérifier que la valeur de retour d'une méthode est valide pour un jeu de données de paramètres précis. On ne considère donc pas l'implémentation en elle-même, mais son comportement, en éprouvant la validité des résultats qu'elle retourne.

Ainsi les tests unitaires servent à détecter un dysfonctionnement éventuel, tandis que le débogage est une analyse du dysfonctionnement en lui-même.

Les exceptions permettent de traiter des déroulements "inattendus" : ce mécanisme des exceptions offre un moyen simple pour réagir à des événements qui risquent de survenir, comme par exemple, une erreur de lecture sur un fichier.

Sommaire

Le code de départ pour illustrer les notions de cours est fourni dans l'archive `erreurs.zip` (projet pour eclipse).

4.1 Débogage

Un bug est un défaut de conception d'un programme informatique à l'origine d'un dysfonctionnement. Ce nom vient d'un des premiers incidents informatique qui a été causé par un insecte.

Un débogueur est un logiciel qui aide un développeur à analyser les bugs d'un programme. Pour cela, il permet d'exécuter le programme pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme ...

En java, on peut déboguer un programme en utilisant le débogueur en ligne de commande fourni : `jdb`. Les outils de développement `Eclipse` et `NetBeans` ajoutent une interface graphique à celui-ci.

4.1.1 En ligne de commande : `jdb`

La documentation en ligne de l'utilitaire `jdb` est à : `http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/jdb.html`. Cet utilitaire contient également une description de ses commandes très complète, accessible via l'instruction `help`.

Dans cette section, nous allons faire un survol des principales fonctionnalités et instructions de `jdb`.

Lancement d'une session de débogage : `jdb MaClasse`

Cette commande démarre une machine virtuelle et charge dans celle-ci la classe `MaClasse`, puis le débogueur se met en attente au début de la méthode `main` de la classe `MaClasse`. L'invite de commande permet alors de contrôler le déroulement du programme, via les instructions décrites ci-après.

Lancement du déroulement du programme : `run`

Cette instruction provoque le déroulement du programme jusqu'au premier point d'arrêt qui a été défini.

Reprise du déroulement du programme : `cont`

Cette instruction permet de reprendre le déroulement du programme jusqu'au point d'arrêt suivant.

Définition d'un point d'arrêt : `stop`

Cette instruction dépose un point d'arrêt à l'endroit spécifié.

```
stop at MaClasse:xx // mettre un point d'arret a la ligne numero xx de MaClasse.java
stop in MaClasse.nomMethode // mettre un point d'arret au debut de la methode nomMethode de
    MaClasse
```

Déroulement pas-à-pas : `step`, `next`

L'instruction `step` avance à la ligne suivante (du code source) en suivant les appels de méthodes, tandis que l'instruction `next` redonne directement la main après l'appel de méthode (i.e. on passe à la ligne suivante de l'appel à la méthode).

Affichage de la valeur d'un attribut ou d'un objet : `print`, `dump`

L'instruction `print` affiche la valeur d'un attribut de type primitif et une description courte pour un objet. L'instruction `dump` donne une description plus complète pour un objet, en donnant les valeurs de chacun de ses attributs.

Exemples :

```
print MaClasse.unAttributStatique
print monObj.unAttribut
print i + j + k // i,j,k : attributs
print monObj.uneMethode() // affiche la valeur retournee par la methode
```

Exemple de session de débogage : détection d'une erreur dans la classe `Professeur`

Considérons la classe `Professeur` légèrement modifiée ci-dessous :

```
package scolarite;

public class Professeur extends Personne{
   private static int nbTotalProfesseurs=0;

   public Professeur( String nom){
      setNom(nom);
      nbTotalProfesseurs+=2;
   }

   public static int getNbTotal() {
      return nbTotalProfesseurs;
   }

}
```

Lorsque l'on crée un `Professeur`, le nombre de professeur `nbTotalProfesseurs` est incrémenté de deux unités (au lieu d'une). Si bien que l'exécution du programme principal ci-dessous :

```
1 package scolarite;
2
3 /**
4  *
5  * @author professors team
6  */
7 public class Main {
8
9
10    Main(){
11        Etudiant a = new Etudiant("Arsene Lupin");
12        Etudiant b = new Etudiant("Dark Vador");
13        Etudiant c = new Etudiant("Saurion");
14        Professeur d = new Professeur("Jojo Lapin");
15        Professeur e = new Professeur("Barbapapa");
16        System.out.println(Personne.getNbTotal()+" personnes dont "+Etudiant.getNbTotal()+"
    etudiants "+" et "+Professeur.getNbTotal()+" professeurs");
17    }
18
19
20    /**
21     * @param args the command line arguments
22     */
23    public static void main(String[] args) {
24
25        new Main();
26    }
27 }
```

affiche :

```
5 personnes dont 3 etudiants  et 4 professeurs
```

Nous allons maintenant effectuer une session de débogage pour identifier l'endroit fautif.

Lancement de la session :

```
$ jdb -classpath  bin -sourcepath src scolarite.Main
```

Retranscription de la session :

```
> stop at scolarite.Main:14 // mise en place d'un point d'arret au debut de la ligne 14
Deferring breakpoint scolarite.Main:14.

> run // lancement de l'execution du programme jusqu'au point d'arret
run scolarite.Main
VM Started: Set deferred breakpoint scolarite.Main:14

Breakpoint hit: "thread=main", scolarite.Main.<init>(), line=14 bci=34
14            Professeur d = new Professeur("Jojo Lapin");

// point d'arret atteint
> print scolarite.Personne.nbTotalPersonnes
 scolarite.Personne.nbTotalPersonnes = 3 // 3 personnes : ok (les 3 etudiants)

> print scolarite.Etudiant.nbTotalEtudiants
 scolarite.Etudiant.nbTotalEtudiants = 3 // 3 etudiants : ok

> print scolarite.Professeur.nbTotalProfesseurs
scolarite.Professeur.nbTotalProfesseurs = null // ok: pas encore de professeurs

> next // on execute la ligne 14 : creation du premier professeur
Step completed: "thread=main", scolarite.Main.<init>(), line=15 bci=45
15            Professeur e = new Professeur("Barbapapa");

> print scolarite.Professeur.nbTotalProfesseurs
scolarite.Professeur.nbTotalProfesseurs = 2 // !!! 2 professeurs : valeur incorrecte !!!

> print scolarite.Personne.nbTotalPersonnes
scolarite.Personne.nbTotalPersonnes = 4 // ok (3 etudiants + 1 professeur)
```

Cette session a donc mis en évidence que la mise à jour de la variable `nbTotalProfesseurs` dans la classe `Professeur` est incorrecte puisque la création du premier professeur donne déjà une valeur incorrecte.

Exercice 56. *(⋆) Reproduisez cette session de débogage. Identifiez ensuite la ligne précise qui provoque le dysfonctionnement à l'aide d'une autre session de débogage en utilisant l'instruction `step` pour suivre ce qui se passe lorsque l'on crée un `Professeur`.*

4.1.2 Dans Eclipse

Pour utiliser les fonctionnalités d'aide au débogage d'`Eclipse`, il faut activer la perspective `Debug`. La mise en place de point d'arrêts se fait simplement avec un clic droit sur la ligne souhaitée. La vue `Breakpoints` donne la liste des points d'arrêts. La vue `Variables` permet de visualiser les valeurs des attributs.

Pour une description plus complète du débogueur d'`Eclipse`, vous pouvez consulter le cours en ligne de Jean-Michel Doudoux : `http://jmdoudoux.developpez.com/cours/developpons/eclipse/chap-eclipse-debug.php`

Exercice 57. *(⋆) Reproduisez avec `Eclipse` la session de débogage permettant de mettre en évidence l'erreur dans la classe `Professeur` de la section précédente. Pour cela, il est nécessaire de rajouter les attributs statiques dans la vue `Variables`, de manière à visualiser la valeur de `nbTotalProfesseurs` de la classe `Professeur`.*

4.2 Tests unitaires

Selon Wikipedia,

> le test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »).
>
> On écrit un test pour confronter une réalisation à sa spécification. Le test définit un critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification. Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie. Le test permet de vérifier que la relation d'entrée / sortie donnée par la spécification est bel et bien réalisée.

JUnit est une bibliothèque dédiée au test d'une application Java. Avec JUnit, on peut créer un ensemble de classes dans le *même* projet, pour tester les classes propres au projet. JUnit sert donc à séparer le code pour réaliser les tests du code de l'application.

Un développeur peut définir un ensemble de contraintes que doit satisfaire une classe. Si un autre développeur change le code de la classe, il n'a besoin que de quelques clics (sous Eclipse) pour s'assurer que la classe est toujours valide.

Considérons à nouveau la classe `Professeur` de la section précédente, avec son erreur dans le constructeur sur la mise à jour de l'attribut `nbTotalProfesseurs`.

Pour tester le "bon" fonctionnement de la classe `Professeur`, on peut écrire la classe de test suivante :

```
package tests;
import static org.junit.Assert.*;
import org.junit.Test;
import scolarite.Professeur;

public class ProfesseurTest {

   @Test
   public void testProfesseurNom() {
          Professeur p = new Professeur("test");
          // on verifie si le nom du professeur est bien egal a "test"
          assertEquals(p.getNom(),"test");
   }

   @Test
   public void testProfesseurNb(){
      int nbTotalProfesseurs = Professeur.getNbTotal();
      Professeur p = new Professeur("test");
      // on verifie si le nombre de professeurs a bien augmente d'un
      assertEquals(Professeur.getNbTotal(),nbTotalProfesseurs+1);
   }
}
```

Les méthodes qui constituent des tests à réaliser doivent être marquées par l'annotation `@Test`.

A l'exécution, le test `testProfesseurNom` passe tandis que celui `testProfesseurNb` *ne* passe *pas* : ces tests unitaires permettent de détecter la mauvaise implémentation de la classe `Professeur`.

JUnit procure les méthodes de test suivantes : `assertEquals(a,b)`, `assertFalse(a)`, `assertNotNull(a)`, `assertNotSame(a,b)`, `assertNull(a)`, `assertSame(a,b)`, `assertTrue(a)`.

Exercice 58. *(⋆) Tester ce code de test dans Eclipse. Ecrire des tests similaires pour les classes `Etudiant` et `Personne`, puis lancer les tests.*

Exercice 59. *(⋆⋆) Ecrire une classe `Groupe` avec des méthodes `void add(Personne p)`, `int size()` et `String toString()` qui permet de créer des groupes de personnes, dont les membres ont des noms tous différents.*

Ecrire une classe de tests `GroupeTest` qui permet de contrôler le bon fonctionnement de chacune des trois méthodes de la classe `Groupe`.

Pour aller plus loin, ce tutoriel en ligne http://www.vogella.com/articles/JUnit/article.html présente de manière plus complète les fonctionnalités de la bibliothèque `JUnit`.

4.3 Exceptions

Considérons la classe suivante décrivant une `Promotion` d'étudiants :

```
public class Promotion {
   int annee;
   String intitule;
   private int nbEtudiants;

   public Promotion( String intitule, int annee, int nbEtudiants){
      this.intitule=intitule;
      this.annee=annee;
      setNbEtudiants(nbEtudiants);
   }

   public int getNbEtudiants() {
      return nbEtudiants;
   }

   public void setNbEtudiants(int nbEtudiants) {
      if (nbEtudiants<0)
         erreur("Le nombre d'etudiants doit etre positif ou nul");
      this.nbEtudiants = nbEtudiants;
   }

   protected void erreur(String message){
      System.err.println("ERREUR dans la classe " + getClass().getName() + " : " + message);
      System.exit(1);
   }

}
```

Que se passe-t-il lorsque l'on crée une promotion avec la ligne ci-dessous ?

```
Promotion as2012 = new Promotion("DUT Informatique - Annee Speciale", 2012,-1);
```

Lors de l'appel à `setNbEtudiants` dans le constructeur avec une valeur du paramètre incorrecte :
— la méthode `erreur` est appelée,
— un message décrivant l'erreur est affiché,
— le programme s'interrompt (brutalement).

Dans le cas présent, l'arrêt brutal n'est pas justifié ; nous aimerions refaire un appel avec un paramètre correct et continuer l'exécution du programme, sans surcharger le code avec des instructions rendant les algorithmes illisibles. Vous avez certainement constaté la multiplication de clauses `if(...)...` afin de gérer les cas limites qui peuvent intervenir dans vos programmes : division par zéro,...

Le plus souvent, la méthode constatant l'erreur n'est pas habilitée à prendre la meilleure décision pour la traiter :
— on redemande une saisie ?
— on arrête ?
— on affiche un message et on continue (vaille que vaille) ?

Il est donc nécessaire d'avoir un mécanisme de gestion des erreurs, prenant en charge la *détection* une erreur, le *traitement* de celle-ci, ainsi que la reprise de l'exécution du programme.

Revenons à notre exemple. Considérons la classe `nbEtudiantsException` suivante :

```
public class NbEtudiantsException extends Exception{
   private int nbErr;

   public NbEtudiantsException(int nbErr){
      this.nbErr = nbErr;
   }

   public String toString(){
          return "Nb etudiants errone : " + nbErr ;
   }
}
```

et modifions notre classe `Promotion` ainsi :

```
public class Promotion {

...

  public void setNbEtudiants(int nbEtudiants) throws NbEtudiantsException{
     if (nbEtudiants<0)
        throw new NbEtudiantsException(nbEtudiants);
     this.nbEtudiants = nbEtudiants;
  }

...
}
```

Quels sont les changements ?

— On détecte l'erreur (clause `if`).
— On prévient l'environnement (via la création d'une exception `NbEtudiantsException`).

La méthode ayant détectée l'erreur ne la gère pas : il y a *séparation* entre la détection et la gestion de l'erreur. Plus précisément, que se passe-t'il lors de l'exécution des instructions suivantes ?

```
throw new NbEtudiantsException(nbEtudiants);
```

Ces instructions déclenchent successivement ces différentes opérations :

i. Un objet de la classe `NbEtudiantsException` est instancié.

ii. L'instruction `throw` *lève* une exception.

iii. La méthode courante (`setNbEtudiants`) est stoppée.

iv. Le gestionnaire d'exceptions prend la main : `setNbEtudiants` a levé une exception. Il enlève l'environnement associé à la méthode `setNbEtudiants`. de la pile d'exécution

v. Il rend (éventuellement) la main à la méthode ayant appelé `setNbEtudiants`. En fait, le gestionnaire d'exceptions va remonter la pile d'appels jusqu'à trouver une méthode capable de gérer l'exception.

Regardons de plus près, la déclaration de la méthode `setNbEtudiants` :

```
public void setNbEtudiants(int nbEtudiants) throws NbEtudiantsException
```

Sa signature stipule que la méthode est susceptible de lever une exception du type `NbEtudiantsException` et elle précise au compilateur que toute méthode appelant `setNbEtudiants` devra se préoccuper de cette exception : soit *en la traitant*, soit *en la propageant*.

Exercice 60. *(⋆) Ecrire la classe `Promotion` et observer le message d'erreur du compilateur.*

Le compilateur indique une erreur dans le constructeur de la classe `Promotion` :

```
public Promotion( String intitule, int annee, int nbEtudiants){
   this.intitule=intitule;
   this.annee=annee;
   setNbEtudiants(nbEtudiants); // cette ligne ne compile pas
}
```

Le compilateur affiche le message d'erreur : "Unhandled exception type nbEtudiantException".

En effet, le constructeur `Promotion` appelant `setNbEtudiants` ne gère pas l'exception `NbEtudiantsException` susceptible d'être levée. Le constructeur doit soit *traiter* l'exception, soit la *propager*.

Modifions l'entête du constructeur `Promotion` afin de propager l'exception :

```
public Promotion( String intitule, int annee, int nbEtudiants) throws NbEtudiantsException{
   this.intitule=intitule;
   this.annee=annee;
   setNbEtudiants(nbEtudiants);
}
```

Bien, nous propageons l'exception *mais qui la traite et comment ?*

Cela se fait via le bloc d'instructions `try... catch... finally` :

```
try{
   Promotion as2012 = new Promotion("DUT Informatique - Annee Speciale", 2012,-1);
   // si le constructeur n'a pas leve d'exception, poursuite de l'execution ici
}
catch (NbEtudiantsException nbe){
   // exception capturee: on la traite (ici affichage)
   System.err.println(nbe);
   }
finally{
  // bloc optionnel, execute quoi qu'il advienne
  System.out.println("finally");
}
// poursuite normale du programme ici
System.out.println("yo");
```

Le déroulement étape par étape est donc le suivant :

i. appel du constructeur

ii. appel de `setNbEtudiants(-1)`, qui lève une exception

iii. dépilement de l'environnement de `setNbEtudiants` et propagation de l'exception au constructeur

iv. le constructeur ne capturant pas l'exception, son environnement est dépilé (aucun objet instancié) ; l'exception est propagée à la méthode appelante (ici dans la classe `Main`)

v. la clause `catch` de capture l'exception et provoque l'exécution du bloc qui lui est associé.

Exercice 61. *(⋆) Ecrire le bloc `try ... catch` décrit ci-dessus dans une classe `Main` et tester le traitement de l'exception par celle-ci.*

Exercice 62. *(⋆⋆) (adaptation d'un exercice conçu par Irène Charon)*
http://perso.telecom-paristech.fr/~charon/coursJava/exercices/fact+Exc.html Considérons le programme ci-dessous permettant de calculer la factorielle d'un nombre passé en argument. Que se passe-t'il si :
— *on ne passe pas d'argument ?*
— *on passe en paramètre un argument non-entier ?*
— *on passe en argument un entier négatif ?*
— *on passe en argument l'entier 20 ?*

Dans les deux premiers cas, une exception est signalée. Dans les deux derners cas, le résultat est faux.

Modifier le programme pour que, dans chacun de ces cas, l'erreur soit gérée par le programme et signalée à l'utilisateur :
— *s'il n'y a pas de paramètre sur la ligne de commande, il s'agit d'une exception de type `ArrayIndexOutOfBoundsException` ; dans ce cas, le programme devra afficher "Nombre entier positif manquant" puis se terminer.*
— *Si le paramètre passé ne représente pas un entier, il s'agit d'une exception de type `NumberFormatException` ; dans ce cas, le programme devra afficher "L'argument doit être entier" puis se terminer.*
— *si le paramètre passé est négatif, le programme devra afficher "Le paramètre est négatif" puis se terminer ; on utilisera une exception personnalisée de type `ExceptionNegatif`.*
— *si le paramètre est trop grand, le programme devra afficher "Le paramètre est trop grand pour ce programme", puis se terminer ; on utilisera une exception personnalisée de type `ExceptionTropGrand`. En Java, `Integer.MAX_VALUE` désigne le plus grand entier.*

```
public class Factorielle {

        public static int calcul(int n){
                int res = 1;
                for (int i = 2; i <= n; i++) res *= i;
                return res;
        }

         public static void main(String[] arg) {
                    int n;
                    n = Integer.parseInt(arg[0]);
                    System.out.println("Factorielle de " + n + " : " + calcul(n));
                  }

}
```

Exercice 63. *(⋆⋆) Ecrire deux tests unitaires pour l'exercice précédent permettant de tester la levée des exceptions `ExceptionNegatif` et `ExceptionTropGrand` par la méthode `calcul`.*

En résumé

— Une exception est un objet qui est instancié lors d'un incident : *une exception est levée*.
— Le traitement du code de la méthode est interrompu et l'exception est *propagée* à travers la pile d'exécution de méthode appelée en méthode appelante.
— Si aucune méthode ne *capture* l'exception : celle-ci remonte l'ensemble de la pile d'exécution ; l'exécution se termine avec une indication d'erreur.
— La *capture* est effectuée avec les clauses *try... catch*.
— La clause *try* définit un bloc d'instructions pour lequel on souhaite capturer les exceptions éventuellement levées. Si plusieurs exceptions peuvent se produire, l'exécution du bloc est interrompue lorsque la première est levée. Le contrôle est alors passé à la clause *catch*.
— La clause *catch* définit l'exception à capturer, en référençant l'objet de cette exception par un paramètre puis le bloc à exécuter en cas de capture.

Compléments sur la bibliothèque des exceptions

Toutes les exceptions héritent de la classe `Throwable` :

— `Throwable(String)` : la chaîne passée en paramètre sert à décrire l'incident
— `getMessage()` : obtenir l'information associée à l'incident
— `printStackTrace()` : exception + état de la pile d'exécution

`Throwable` possède deux classes filles `Error` et `Exception` :

— `Error` : erreurs graves de la machine virtuelle (état instable, récursivité infinie, classe en chargement non trouvée,...).
— `Exception` : ensemble des erreurs pouvant être gérées par le programmeur (`RunTimeExeption` (`NullPointerException`, `IndexOutOfBoundsException`, `ArithmeticException`), `IOException`, ...).

Fiche de synthèse

Exceptions

- `try` : pour spécifier une section de code pouvent lever une exception (une erreur) ;
- `catch` sert à spécifier le code à exécuter lors de l'interception d'une exception ;
- `finally` (optionnel) ; pour le code à toujours exécuter ;
- `throws` : pour ne pas traiter une exception, et la relayer à la méthode appelante.
- `throw` : pour lever une exception ;
- création d'exceptions personnalisées par héritage de la classe `Exception`.

```
try { // des lignes de code susceptible de lever une exception }
catch (IOException e) { // traitement de l'exception de type IOException }
finally { // sera toujours execute }

void fonction throws IOException { ...}
```

DeplacementInterditException

```
package application;
public class DeplacementInterditException extends Exception {
   private Salle d; private Salle pos;
        public DeplacementInterditException(String text) { super(text); }
   public DeplacementInterditException(Salle destination, Salle position) {
      this.d=destination; this.pos=position;
   }
   public String toString() {
      return "DeplacementInterditException: Destination = "+d+" - Position "+pos;
   }
}
```

JUnit

- `JUnit` est une librairie dédiée aux tests unitaires d'une application Java. ;
- permet de *séparer le code dédié aux tests* du code de l'application ;
- permet de *définir des contraintes que doit satisfaire une classe* ;
- en cas de modification du code d'une classe, relancer les tests pour s'assurer que la classe est toujours valide.

TestCreationLabyrinthe

```
package tests;
import static org.junit.Assert.*;
import org.junit.Test;
import application.*;
public class TestCreationLabyrinthe {
        @Test
        public void testLargeurLabyrinthe() {
                LabyrintheDefaut labyrinthe = new LabyrintheDefaut();
                labyrinthe.creerLabyrinthe("labys/level9.txt");
                assertTrue( labyrinthe.getLargeur() == 40);
        }
}
```