Exceptions, paquetage et accessibilité

Arnaud Labourel arnaud.labourel@univ-amu.fr

Section 1

Exceptions

Les exceptions

Un programme peut être confronté à une condition exceptionnelle (ou exception) durant son exécution.

Une exception est une situation qui empêche l'exécution normale du programme (elle ne doit pas toujours être considérée comme un bug). Quelques exemples de situations exceptionnelles :

- un fichier nécessaire à l'exécution du programme n'existe pas,
- une division par zéro,
- un débordement dans un tableau,
- un besoin de se connecter à un serveur et celui-ci est injoignable,
- un dépilement d'une pile vide,
- ...

Mécanisme de gestion des exceptions

Java propose un mécanisme de gestion des exceptions afin de distinguer l'exécution normale du traitement de celles-ci afin d'en faciliter leur gestion.

En Java, une exception est concrétisée par une instance d'une classe qui étend la classe `Exception`.

Pour lever (déclencher) une exception, on utilise le mot-clé `throw` :

```
if (problem) throw new MyException("error");
```

Pour capturer une exception, on utilise les mots-clés `try` et `catch` :

```
try { /* Problème possible */ }
catch (MyException e) { /* traiter l'exception. */ }
```

Exemple d'exceptions existantes en java

- `ArithmeticException` : opération arithmétique impossible comme la division par 0.
- `IndexOutOfBoundsException` : dépassement d'indice dans un tableau, un vecteur, ...
- `NullPointerException` : accès à un attribut/méthodes/case pour les tableaux d'une référence valant `null`, argument `null` alors que ce n'est pas autorisé.
- `FileNotFoundException` : échec de l'ouverture d'un fichier à partir d'un chemin.
- `IllegalArgumentException` : argument incorrect (en dehors des valeurs autorisées) lors de l'appel d'une méthode.
- `NoSuchElementException` : `next` alors que l'itération est finie, dépilement d'une pile vide, ...
- ...

Définir son exception

Il suffit d'étendre la classe `Exception` (ou une classe qui l'étend) :

```
public class MyException extends Exception {
  private int number;

  public MyException(int number) {
    this.number = number;
  }

  public String getMessage() {
    return "Error " + number;
  }
}
```

La syntaxe try/catch

Pour capturer une exception, on utilise la syntaxe `try/catch` :

```
public static void test(int value) {
  System.out.print("A ");
  try {
    System.out.println("B ");
    if (value > 12) throw new MyException(value);
    System.out.print("C ");
  } catch (MyException e) { System.out.println(e); }
  System.out.println("D");
}
```

`test(11)`	`test(13)`
A B	A B
C D	MyException: Error 13
	D

Exceptions et signatures des méthodes

Une méthode doit préciser dans sa signature toutes les exceptions qu'elle peut lever et qu'elle n'a pas traitées avec un bloc `try/catch` :

```
public static void testValue(int value)
                    throws MyException {
  if (value>12) throw new MyException(value);
}
public static void runTestValue(int value)
                    throws MyException {
  testValue(value);
}
```

La méthode `testValue` peut lever une exception de type `MyException`.

`runTestValue` doit indiquer qu'elle peut lever une exception car elle ne gère pas l'exception provoquée par l'appel `testValue(value)`.

Gestions des exceptions : règle

La méthode `runTestValue` peut lever une exception (de type `MyException`).

Lorsqu'on fait un appel à la méthode `runTestValue`, il est vérifié à la compilation que l'une des deux propriétés suivante est vraie :

- la méthode appelant `runTestValue` est indiquée comme pouvant lever l'exception `MyException` (en écrivant `throws MyException` à la signature de la méthode).
- l'exception potentielle est capturée par un loc `try/catch`.

Si aucune des deux propriétés est vérifiée alors il y a une erreur à la compilation.

```
Error:(YY, XX) java: unreported exception MyException;
      must be caught or declared to be thrown
```

Exceptions et signatures des méthodes

Une méthode doit donc préciser dans sa signature toutes les exceptions qu'elle peut lever et qu'elle n'a pas traitées avec un bloc `try/catch`

On doit donc écrire :

```
public class Main {
  public static void main(String[] args)  {
    try { Test.runTestValue(13); }
    catch (MyException e) { e.printStackTrace(); }
  }
}
```

Techniquement la méthode `main` pourrait indiquer qu'elle peut lever l'exception `MyException` mais cela n'aurait pas beaucoup de sens car cela voudrait dire qu'on ne gère pas vraiment l'exception.

Méthode printStackTrace

La méthode printStackTrace permet d'afficher la pile d'appels :

```
public class Main {
  public static void main(String[] args)  {
    try { Test.runTestValue(13); }
    catch (MyException e) { e.printStackTrace(); }
  }
}
```

```
MyException: Error 13
    at Test.testValue(Test.java:23)
    at Test.runTestValue(Test.java:20)
    at Main.main(Main.java:6)
```

La classe `RuntimeException`

Java définit une classe étendant `Exception` nommée `RuntimeException`.

Les `RuntimeException` correspondent généralement à des bugs pouvant arriver très fréquemment (références non initialisées, divisions par zéro, mauvaises valeur d'arguments, ...) qu'on n'a pas obligation de gérer :

- `ArithmeticException`
- `ClassCastException`
- `IllegalArgumentException`
- `IndexOutOfBoundsException`
- `NegativeArraySizeException`
- `NullPointerException`
- `NoSuchElementException`

Règle générale : ajout cas `RuntimeException`

Lorsqu'on fait un appel à une méthode `canThrow` pouvant lever une exception `MyException` qui n'étend pas `RuntimeException`, il est vérifié à la compilation que l'une des deux propriétés suivantes est vraie :

- la méthode appelant `canThrow` dans son code est indiquée comme pouvant lever une exception de type `MyException` ou une de ces super-classes (en écrivant `throws MyException` à la signature de la méthode).
- l'exception potentielle est capturée par un bloc `try/catch`.

$\Rightarrow$ On n'a pas obligation de gérer les `RuntimeException` mais on peut le faire si on le souhaite.

Capturer une exception en fonction de son type

```
public static int divide(Integer a, Integer b) {
  try { return a/b; }
  catch (ArithmeticException exception) {
    exception.printStackTrace();
    return Integer.MAX_VALUE;
  } catch (NullPointerException exception) {
    exception.printStackTrace();
    return 0;
  }
}
```

```
divide(12,0) :
java.lang.ArithmeticException: / by zero
  at Test.diviser(Test.java:17)
  at Test.main(Test.java:28)
```

Le mot-clé finally

On rajouter un bloc finally après des blocs try. Le bloc associé au mot-clé finally est toujours exécuté :

```
public static void readFile(String fileName) {
  try {
    FileReader fileReader = new FileReader(fileName);
      /* peut déclencher une FileNotFoundException. */
    try {
      int character = fileReader.read();// IOException ?
      while (character!=-1) {
        System.out.println(character);
        character = fileReader.read(); // IOException ?
      }
    } finally { fileReader.close();/*dans tous les cas*/
  } catch (IOException exception) {
    exception.printStackTrace();
```

Gestion de différents types d'exceptions

```
try { FileReader fileReader = new FileReader(fileName);
  try {
    int character = fileReader.read(); // IOException ?
    while (character!=-1) {
      System.out.println(character);
      character = fileReader.read(); // IOException ?
    }
  } finally { /* à faire dans tous les cas. */
    fileReader.close();
  }
} catch (FileNotFoundException exception) {
  System.out.println("File "+fileName+" not found.");
} catch (IOException exception) {
  exception.printStackTrace();
}
```

Exceptions pour des piles (1/2)

```
public class Stack<T> {
  private Object[] stack;
  private int size;

  public Stack(int capacity) {
      stack = new Object[capacity];
      size = 0;
  }
}
```

Exceptions pour des piles (2/2)

```
public class Stack<T> {
  public void push(T object) throws FullStackException {
    if (size == stack.length)
      throw new FullStackException();
    stack[size] = object;
    size++;
  }
  public T pop() throws EmptyStackException {
    if (size == 0) throw new EmptyStackException();
    size--;
    T object = (T)stack[size];
    stack[size]=null;
    return object;
  }
}
```

Définition des exceptions pour les piles

```
public class StackException extends Exception {
  public StackException(String msg) {
    super(msg);
  }
}
public class FullStackException extends StackException {
  public FullStackException() {
    super("Full stack.");
  }
}
public class EmptyStackException extends StackException {
    public EmptyStackException() {
        super("Empty stack.");
    }
}
```

Exemple d'utilisation (1/2)

```
Stack<Integer> stack = new Stack<Integer>(2);

try {
  stack.push(1);
  stack.push(2);
  stack.push(3);
} catch (StackException e) {
  e.printStackTrace();
}
```

```
FullStackException: Full stack.
    at Stack.push(Stack.java:13)
    at Main.main(Main.java:10)
```

Exemple d'utilisation (2/2)

```
Stack<Integer> stack = new Stack<Integer>(2);

try {
  stack.push(1);
  stack.pop();
  stack.pop();
} catch (StackException e) {
  e.printStackTrace();
}
```

```
EmptyStackException: Empty stack.
    at Stack.pop(Stack.java:18)
    at Main.main(Main.java:10)
```

Section 2

Structure d'un projet et paquetages

Structure d'un projet

En Java, un projet peut être découpé en paquetages (`package`).

Les paquetages permettent de :

- associer des classes afin de mieux organiser le code
- de créer des parties indépendantes réutilisables
- d'avoir plusieurs classes qui possèdent le même nom (du moment qu'elles ne sont pas dans le même paquetage)

Un paquetage (`package`) :

- est une collection de classes
- peut contenir des sous-paquetages

Lors de l'exécution...

Java utilise l'arborescence de fichier pour retrouver les fichiers `.class`

- Une classe (ou une interface) correspond à un fichier `.class`
- Un dossier correspond à un paquetage

Les `.class` du paquetage `com.univ_amu` doivent :

- être dans le sous-dossier com/univ_amu
- le dossier com doit être à la racine d'un des dossiers du `ClassPath`

Le `ClassPath` inclut :

- le répertoire courant
- les dossiers de la variable d'environnement `CLASSPATH`
- des archives JAR
- des dossiers précisés sur le ligne de commande de java (`-classpath path`)

Lors de la compilation... (1/2)

Le mot-clé `package` permet de préciser le paquetage des classes ou interfaces définies dans le fichier :

```
package com.univ_amu;
public class MyClass { /* ... */ }
```

Java utilise l'arborescence pour retrouver le code des classes ou interfaces :

- Une classe (ou une interface) `MyClass` est cherchée dans le fichier `MyClass.java`
- Le fichier `MyClass.java` est cherché dans le dossier associé au paquetage de `MyClass`

Lors de la compilation... (2/2)

```java
package com.univ_amu;
public class MyClass { /* ... */ }
```

Dans l'exemple précédent, il est donc conseillé que le fichier :

- se nomme `MyClass.java`
- se trouve dans le dossier `com/univ_amu` (Par défaut, la compilation crée `MyClass.class` dans `com/univ_amu`)

Utilisation d'une classe à partir d'un autre paquetage

Accessibilité :

- Seules les classes publiques sont utilisable à partir d'un autre paquetage
- Un fichier ne peut contenir qu'une seule classe publique

On peut désigner une classe qui se trouve dans un autre paquetage :

```
package com.my_project;
public class Main {
  public static void main(String[] args) {
    com.univ_amu.MyClass myClass =
      new com.univ_amu.MyClass();
  }
}
```

Importer une classe

Vous pouvez également importer une classe :

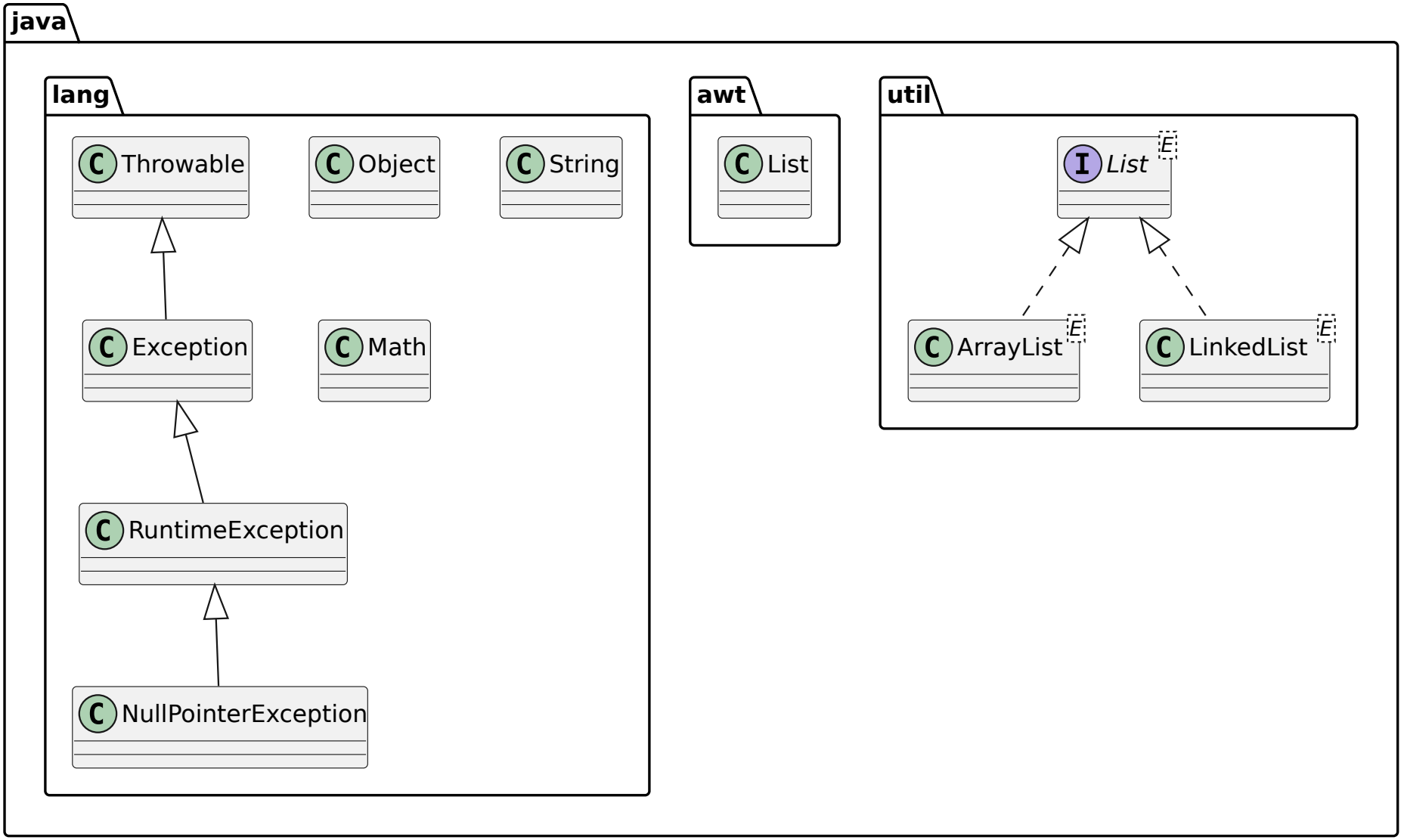
```
package com.my_project;
import com.univ_amu.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass myClass = new MyClass();
    }
}
```

Deux classes de deux paquetages différents peuvent avoir le même nom :

- Exemple : `java.util.List` et `java.awt.List`
- Attention de choisir le bon import

Diagramme avec paquetages



Utiliser une classe d'un autre paquetage sans import

En fait, il est possible d'utiliser des classes d'un autre paquetage sans import.

Il suffit de mettre le chemin complet à chaque utilisation de la classe.

```
public class AppPackage {
  static public void main(String[] args){
    java.util.List<Integer> list1 =
      new java.util.ArrayList<>();
    java.awt.List list2 =
      new java.awt.List();
  }
}
```

⇒ On peut même utiliser dans ce cas deux classes ayant le même nom mais des paquetages différents.

Importer un paquetage

Vous pouvez également importer toutes les classes d'un paquetage :

```
package com.my_project;
import com.univ_amu.*;

public class Main {
    public static void main(String[] args) {
        MyClass myClass = new MyClass();
    }
}
```

Remarques :

- Les classes des sous-paquetages ne sont pas importées
- Le paquetage `java.lang` est importé par défaut

Importer des membres statiques

Depuis Java 5, il est possible d'importer directement des méthodes ou attributs de classes (`static`).

La syntaxe est la suivante :

```
import static my_package.my_class.myStaticMember;
```

Exemple :

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
```

Exemple sans import statique

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
    System.out.println(
      "A circle with a diameter of 5 cm has");
    System.out.println("a circumference of "
                       + (Math.PI * 5) + " cm");
    System.out.println("and an area of "
            + (Math.PI * Math.pow(2.5, 2))
            + " sq. cm");
  }
}
```

Exemple d'import statique

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
import static java.lang.System.out;

public class HelloWorld {
    public static void main(String[] args) {
        out.println("Hello World!");
        out.println(
            "A circle with a diameter of 5 cm has");
        out.println("a circumference of "
                + (PI * 5) + " cm");
        out.println("and an area of "
            + (PI * pow(2.5, 2)) + " sq. cm");
    }
}
```

Un exemple

Le fichier com/univ_amu/HelloWorld.java :

```
package com.univ_amu;
    public class HelloWorld {
    public static void main(String[] arg) {
        System.out.println("Hello world ! ");
    }
}
```

```
$ javac com/univ_amu/HelloWorld.java
$ ls com/univ_amu
HelloWorld.java HelloWorld.class
$ java com.univ_amu.HelloWorld
Hello world !
```

Quelques remarques

Nommage des paquetages :

- Les noms de paquetages sont écrits en minuscules
- Pour éviter les collisions, on utilise le nom du domaine à l'envers ⇒ `com.google.gson`, `com.oracle.jdbc`
- Si le nom n'est pas valide, on utilise des *underscores* : ⇒ `com.univ_amu`

Fichier JAR (Java Archive) :

- est une archive ZIP pour distribuer un ensemble de classes Java
- contient un *manifest* (qui peut préciser la classe qui contient le main)
- peut également faire partie du `ClassPath`
- peut être généré en ligne de commande (`jar`) ou avec un IDE

Section 3

Extension et accessibilité

Accessibilité : modificateur `public` et `default`

Une classe ou un membre (attribut ou méthode) est accessible :

- de n'importe où s'il est précédé du modificateur `public` ;
- des classes de son paquet si rien n'est précisé (`default` ou *package private*).

```
public class MyClass {
  public int myPublicField;
  int myPackagPrivateField;
  public void doPublicAction() { }
  void doPackagePrivateAction() { }
}
```

Remarques :

- Seules les classes publiques sont utilisable à partir d'un autre paquet
- Un fichier ne peut contenir qu'une seule classe publique

Accessibilité : modificateur `private`

Un membre (attribut ou méthode) est accessible :

- de n'importe où s'il est précédé du modificateur `public`
- des classes de son paquet si rien n'est précisé (`default`)
- des méthodes de la classe s'il est précédé de `private`

```
public class MyClass {
  private int privateField;
  private void doPrivateAction() { }
}
```

Afin de limiter les conséquences d'une modification :

- Les méthodes ou attributs définies pour rendre lisible l'implémentation des fonctionnalités doivent être privées;
- Seule l'interface de la classe doit être publique.

Accessibilité : modificateur `protected`

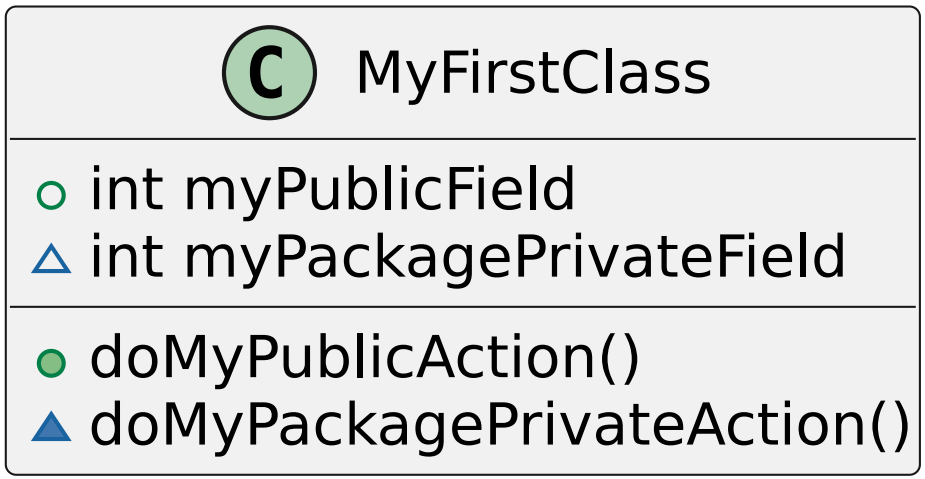
Un membre (attribut ou méthode) `protected` n'est accessible que par les méthodes des classes du paquet et par les classes qui l'étendent.

Le modificateur protected permet de protéger un membre :

```
public class MyClass {
  protected int protectedField;
  protected void doProtectedAction() { }
}
```

Modificateur	Classe	Paquet	Extension	Extérieur
`private`	✓			
`default`	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

Diagramme avec modificateurs d'accès



Règles

○ = `public` (autre symbole possible **+**)

◇ = `protected` (autre symbole possible **#**)

△ = `default` (autre symbole possible **~**)

□ = `private` (autre symbole possible **-**)