Exception en Java.

Ce document n'est pas un polycopié, c'est seulement un résumé des principales notions abordées en cours, il vient en complément du cours d'amphi et ne contient pas toutes les notions et les illustrations qui ont été présentées.

Le principe général des exceptions

Le mécanisme d'exception est un mécanisme de déroutement du programme, c'est à dire qu'elle brise la séquentialité du programme (comme le fait return, break, continue). Le mécanisme qui brise la séquentialité est la **levée d'exception.** Lorsqu'une exception est levée, le programme est dérouté vers les **premiers gestionnaires d'exceptions**.

Les gestionnaires d'exception sont associés à **des blocs de code qui ont été sensibilisés** à certains types d'exception.

L'exception se propage depuis l'endroit où elle est levée, elle remonte d'appelant en appelant en s'arrêtant à chaque gestionnaire d'exceptions qu'elle rencontre dans la remontée. Chaque gestionnaire d'exception est passée en revue, en commençant par le premier qui est déclaré. Si celui ci est de type compatible avec le type de l'exception, alors le code associé à la récupération de l'exception est exécuté et le programme reprend normalement son exécution après le dernier gestionnaire d'exception . Si aucun gestionnaire d'exception ne convient, le même processus est réitéré en remontant, jusqu'à qu'un gestionnaire d'exception la capture ou que la fonction main est atteinte.

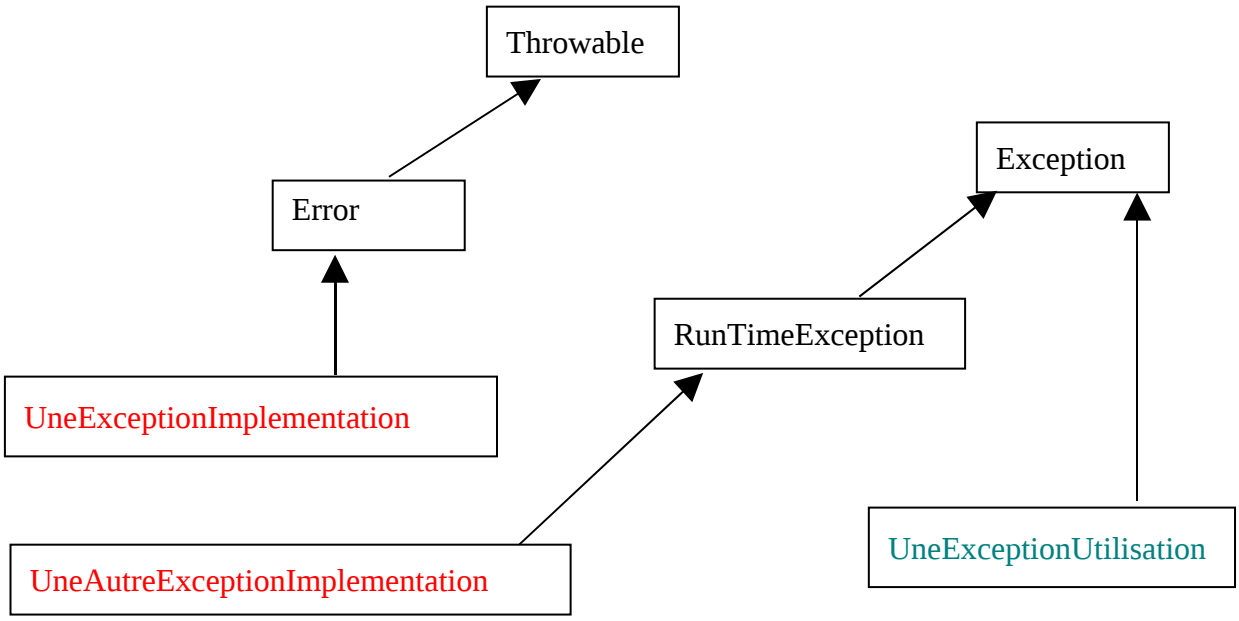
La syntaxe des exceptions.

La gestion des exceptions en Java se fait par l'utilisation de trois mots clefs **try, catch, throw.** Le mot clef **try** sert à définir un bloc dans lequel les exceptions sont susceptibles d'être capturées. On parle aussi de sensibilisation d'un bloc à un ensemble d'exceptions. La capture des exceptions est confiée à des gestionnaires d'exception. Après la définition de ce bloc se trouve la liste des gestionnaires d'exception. Un gestionnaire d'exception est défini en utilisant le mot réservé **catch**. La liste des gestionnaires d'exception ressemble à un ensemble de fonctions surchargées. Une exception est levée en utilisant le mot clef **throw**. L'identification de l'exception, se fait en utilisant le type d'un objet.

La hiérarchie des exceptions:

En java, toutes les exceptions héritent de Throwable. Il y a deux types d'exceptions :

- celles qui héritent soient de Error soient de RuntimeException, ces exceptions sont des exceptions d'implémentation et elles n'ont pas besoin d'être déclarée dans la signature de la fonction. En Rouge sur la figure.
- Les autres qui sont alors considérées comme des exceptions d'utilisation et qui doivent être impérativement déclarées dans la signature de la fonction si elles ne sont pas capturées par la fonction. En bleu sur la figure.



Les services de la classe Throwable sont :

Throwable	**fillInStackTrace()**	Fills in the execution stack trace.
Throwable	**getCause()**	Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String	**getLocalizedMessage()**	Creates a localized description of this throwable.
String	**getMessage()**	Returns the detail message string of this throwable.
StackTraceElement[]	**getStackTrace()**	Provides programmatic access to the stack trace information printed by printStackTrace().
Throwable	**initCause(Throwable cause)**	Initializes the cause of this throwable to the specified value.
void	**printStackTrace()**	Prints this throwable and its backtrace to the standard error stream.
void	**printStackTrace(PrintStream s)**	Prints this throwable and its backtrace to the specified print stream.
void	**printStackTrace(PrintWriter s)**	Prints this throwable and its backtrace to the specified print writer.
void	**setStackTrace(StackTraceElement[] stackTrace)**	Sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.
String	**toString()**	Returns a short description of this throwable.

Exemple d'exception.

Ces méthodes permettent de capturer le contexte d'une exception. Mais on peut aussi créer une nouvelle catégorie d'exception en créant sa propre classe et en la rattachant à la hiérarchie des exceptions de Java.

```
class MonException extends Exception{
        private String mesDonnées;
        public MonException(String donnees){ MesDonnées = new String(donnees);}
        public String getInfo(){ return mesDonnées;}
}
class MonAutreException extends MonException{.....}
class MonError extends Error{.....}
```

Sur cet exemple, on vient de définir trois nouveaux types d'exception qui permettront de définir les types de saut de notre programme.

```
public class Ex1{
        static public void f(int i) throws MonException, MonAutreException{
        try {
                if(i==0) throw new MonError(.....); // Lancement d'une exception
                if(i % 2 == 0) throw new MonAutreException(.......);
                else throw new MonException(....);
                System.out.println(« La FIN du Bloc »);
        }
        catch(MonError e) {System.out.println(« Exception Traitée »);}
        System.out.prinln(« La fonction est finie »);
}
```

Sur cette exemple, la ligne

public void f(int i) throws MonException, MonAutreException

complète la signature d'une fonction en indiquant quelles sont les exceptions que le service f peut lever. En java toutes les exceptions d'utilisation qui peuvent être levée d'une fonction doivent être déclarées dans la signature de cette fonction. Les exceptions ne sont pas pris en compte dans la distinction de la surcharge. Par contre, les exceptions d'implémentation n'ont pas être obligatoirement déclarées dans la signature de la fonction.

Le bloc **try** sensibilise le bloc d'instruction de la fonction à une seule nature d'exception qui est représentée par le type **MonError**. La clausse **catch** représente le gestionnaire d'exception qui est traitera les exceptions de type MonError. Si une exception de type compatible avec le type MonError est levée dans ce bloc, alors le code associé à ce catch sera exécuté et le programme continuera après la dernière classe catch du code de ce service.

Suivant la valeur de i le code sera :

- **Si i est égal à zéro**, l'exception est levée, le programme sort du bloc try et commence à parcourir les clauses catch les unes après les autres en commençant par la première. Ici, il n'y a qu'une clause catch. Le type réel de l'exception est MonError ce qui correspond au type déclaré de la clause catch. Le code de la clause catch est exécuté, le message « **Exception Traitée »** est affiché, puis le programme reprend séquentiellement et le message « **La fonction est finie »** est ensuite affiché**.**
- **Si i est différent de zéro, u**ne exception de type MonException ou de type MonAutreException sera levée dans le bloc d'instruction. Puis la clause catch sera consultée, comme les types des exceptions ne sont pas compatibles avec MonError cette clause catch ne sera pas exécutée. Le programme continuera en se déroutant sur l'appelant et le message « **La fonction est finie »** ne sera pas affiché.

Exception et signature de service.

Si maintenant on regarde un client possible de Ex1, par exemple

```
public class Client {
        public g(int i) throws  MonException
        {  Ex1.f(i); }
}
```

Dans ce cas, la fonction **g** appelle la fonction **f** qui a été définie précédemment. La fonction **f** lève deux natures d'exceptions, comme la fonction g ne capture aucune exception, elle doit les déclarer dans sa signature.

En Java, les exceptions qui peuvent être levées par une fonction doivent appartenir à la signature de la fonction. Mais, il n'est pas obligatoire de les identifier précisement, il suffit que toutes les exceptions levées par la fonction soient compatibles avec les exceptions déclarées dans la signature. La déclaration des exceptions dans la signature de la fonction permet au compilateur Java, de vérifier que toutes les fonctions déclarent bien les exceptions qu'elles lèvent. Il suffit de regarder le code de la fonction, de collecter les exceptions qui peuvent être levées par le code (ELLES sont nécessairement dans la signature), d'enlever pour chaque bloc try celles qui sont capturées, puis de vérifier que les exceptions restantes sont bien déclarées dans la signature de la fonction.

RAPPEL, seules les exceptions d'utilisation sont vérifiées par le compilateur.

Sur l'exemple précédent, la fonction g leve deux types différents d'exception qui sont MonAutreException et MonException, comme le type MonAutreException est compatible avec le type MonException. Il est suffisant de déclarer seulement :

```
public g(int i) throws  MonException
```

Propagation des Exceptions.

```
public class Client1
```

```
{
        public void h(int i) throws MonException
        {   Tmp  e = new Tmp();
            try {
                    double j = 1/i;
                     System.out.prinln(« La valeur de i est non nulle »);
                    e.fun(i);
                    System.out.printl(«  Ligne jamais écrite »);
                  }
             catch(MonAutreException e){ System.out.printl(« J'ai traité mon AutreException »);}
            catch (Error e) {System.out.println(« J'ai divisé par zero »)}
            System.out.println(« Je continue »);
        }
public class Tmp
{
        public void fun(int i) throws MonException, MonAutreException
        {
          Ex1.f(i);
          System.out.println(« La fonction se termine correctement »);
        }
}
```

Sur cet exemple, le service f de la classe Client1 a sensibilisé une partie du code à des exceptions. Elle capture les sous type de Error et elle capture les exceptions de type MonAutreException.

Lorsque la valeur de i est nulle, Java lance une Exception qui est division par zéro, cette exception est un sous-type de Error. Le programme sera donc dérouté vers les clauses catch, la première clause catch traitée correspond au type MonAutreException. Le type Error n'est pas compatible avec le type MonAutreException, cette clause catch ne convient donc pas. On passe alors à la suivante. La seconde clause est bien compatible avec l'exception, le code de la clause catch est exécuté, le message

« J'ai divisé par zéro » apparaît et le programme continue normalement après la dernière clause catch en affichant le message « **Je continue** »

Si la valeur de i est différente de zéro, le programme appelle le service fun de l'objet e qui est de type Tmp. Le code de la fonction fun appelle le code la fonction f qui est un service de classe de Ex1. La fonction fun ne fait que laisser passer les exceptions de la fonction f. La fonction fun sera appelée avec une valeur de i différente de zéro. Dans ce cas, la fonction fun ne se terminera jamais correctement car la fonction f lève systématiquement une exception si la valeur de i est non nulle. Donc le message, « **La fonction se termine correctement** » ne sera jamais affiché. La fonction fun appelle donc f. Comme nous l'avons vu précédemment, cette fonction lève une exception de

type MonException ou de type MonAutreException. La séquentialité de f est arrêtée et le programme revient dans la fonction fun. Comme il n'y a pas de bloc try dans cette fonction, la séquentialité de la fonction est aussi arrêtée et l'exception se propage à l'appelant qui est h. Le code qui a engendré la levée d'exception était sensibilisé à des types d'exception. Donc maintenant, les clauses catch peuvent être vérifiées. Si la valeur de i est :

- **Paire:** la fonction f a levée une exception qui est de type MonAutreException. Cette exception est donc traitée par la clause catch, le code de la clause catch associé est exécuté. Le message **« J'ai traité mon AutreException »** apparaît et le programme reprend son exécution en affichant le message « Je continue ».
- **Impaire:** la fonction f a levée une exception qui est de type MonException. Cette exception n'est pas compatible avec les exceptions traitées par les clauses catch du programme. Donc la séquentialité est interrompue et le programme est dérouté vers l'appelant de h en propageant l'exception MonException. En effet, le type MonException n'est pas compatible avec le type MonAutreException.

Ordre de déclaration des clauses catch.

Comme déjà vu, les clauses catch d'un bloc try sont traitées les une après les autres dans l'ordre de leur déclaration. La première clause catch qui est compatible avec le type de l'exception est exécuté et le programme continue son exécution après la dernière clause catch du bloc try. **Il faut donc déclarer les clauses catch de la plus précise à la plus générale.** En effet supposons l'autre des clauses catch suivante

```
try{....
}
catch(MonException e) {System.out.println(« Je traite MonException et ses soustypes »);}
catch(MonAutreException e){System.out.println(« Code Jamais Exécuté »);}
```

Sur cet exemple, le type MonAutreException est un sous type de MonException, il est donc compatible avec le type MonException. Lorsque des exceptions de type MonException ou mon AutreException sont levées, elles seront capturées par la première clause catch qui convient. Et donc le code associé à **catch(MonAutreException e)** ne sera jamais exécutée, il faut donc inverser l'ordre de déclaration des clauses catch.

```
try{....
}
catch(MonAutreException e){System.out.println(« Code Associé à MonAutreException»);}
catch(MonException e) {
        System.out.println(« Je traite MonException mais pas MonAutreException »);
}
```

Redéfinition et déclaration des exceptions.

Lorsqu'une fonction est redéfinie dans un sous-type, elle peut être appelée avec comme type déclaré celui du sur-type. Comme Java, doit vérifier les signatures des fonctions par rapport aux exceptions, il faut que les exceptions levées par la fonction redéfinie soient compatibles avec les exceptions déclarées par la fonction au niveau du surtype.

```
public interface UneInterface {
        public void f(….)  throws E1,…..EN ;
}
public class UneClasse implement UneInterface{
        public void f(…..) throws C1,…..CM{……};
}
```

Pour que le code de UneClasse soit compilable, il faut que pour une exception de type Ci, il existe un exception de type Ej, telle que Ci soit compatible avec Ej. Comme ça nous sommes sur que le code suivant ne lèvera par d'exception qui ne seront pas capturées par les clauses catch.

```
UneIntercace uneinterface = new UneClasse()
try {
        uneinterface.f(….)
}
catch(E1 e) {….}
……
catch(EN e){…..}
```

Exception d'implémentation et Exception d'utilisation.

Nous avons vu dans les parties précédentes qu'il y avait deux types d'exception :

- **Les exceptions d'utilisation** : Ces exceptions doivent obligatoirement être déclarées dans la signature de la fonction. En effet, elles documentent l'utilisation de la fonction. Elles peuvent survenir lorsque le service ne peut être effectué correctement. Par exemple, les valeurs de paramètres ne sont pas correctes. Ou bien l'objet qui doit effectuer le service n'est pas dans un état correct. Ces exceptions font partie du service, et même si l'implémentation est modifiée, elles ne devraientt pas être remise en cause. On parle aussi parfois d'**exception d'interface**.
- **Les exceptions d'implémentation**. Ces exceptions sont liées à une implémentation particulière d'un service. Si l'implémentation change, alors l'exception peut-être caduque. Ces exceptions n'ont pas à être déclarées dans la signature de la fonction. Certaines sont générées automatiquement par java pendant l'exécution du programme, il s'agit par exemple des débordements de tableaux, des divisions par zéro. Toutes les exceptions qui sont liées à des ressources physiques, mémoire, descripteur de fichiers, sont des exceptions d'implémentation

```
public class Pile {
```

```
        int [] t = new t[10] ;
        int indice = -1;
        public void empiler(int e) { p[++indice] = e} ;
        public int depiler() throws PileVideException {
                if (pileVide()) throw new PileVideException();
                else return p[indice--];
        }
        public Boolean pileVide() { return indice == -1;}
}
```

Sur cet exemple, on voit bien les deux types d'exception, la fonction dépiler quelque soit son implémentation peut lever une exception si l'utilisateur essaye de dépiler alors que la pile est vide. Il s'agit bien d'une exception d'utilisation ou d'interface.

Par contre, l'exception associée à la pile pleine n'est qu'un choix d'implémentation. Si avec cette implémentation la pile est pleine, une exception d'implémentation liée au débordement de tableau sera levée par Java. Si on change, l'implémentation en utilisant maintenant une liste chaînée cette limitation n'aura plus de raison d'être. Il ne faut donc pas quelle face partie de la signature publique de la fonction empiler.

L'utilisation des exceptions.

Le but principale des exceptions est de signaler un dysfonctionnement du programme, soit du à des erreurs d'utilisation soit du à des pénuries en ressources physiques. Les traitements associés aux exceptions permettent soit de réparer le dysfonctionnement, soit de sauvegarder le contexte.

La possibilité de créer ses propres exceptions en créant une nouvelle classe permet de capturer les informations sur le dysfonctionnement pour pouvoir essayer de leur associée un traitement.

Par exemple, imaginons la classe TemperatureException

```
public class TemperatureException
{
        private double temperature ;
        public TemperatureException(double t) { temperature = t ;}
        public double getTemperature() { return temperature ;}
}
```

Une classe CapteurTemperature peut-être la suivante :

```
public class CapteurTemperature
{
        final static double TEMPERATURE_LIMITE = 50 ;
```

```
        public void run() throws TemperatureException
        {  …..
           double t = getTemperature();
           if (t > TEMPERATURE_LIMITE) throw new TemperatureException(t);
          ……
        }
}
```

Lorsque l'exception est levée par le capteur on peut maintenant accéder à l'information du dysfonctionnement qui est la température acquise. On peut par exemple, écrire une clause catch qui en fonction de cette valeur arrêterait ou non le programme.

```
catch(TemperatureException e) {
        double temp = e.getTemperature() ;
         if(temp > 100.0) System.exit();
          else continuer le programme;
}
```

Une autre utilisation d'une exception est de la considérer comme un signal. Par exemple, dans un jeu lorsque Pacman a été mangé par un fantôme, on peut lancer une exception qui serait PacmanMortException et qui permettrait par une clause catch de relancer le tableau de jeu si il existe encore des vies pour Pacman.