Java et OOP
Exceptions

Erreurs !

- C'est inévitable !
- Mais leur gestion ne doit pas transformer notre programme dans une usine à gaz !
- Classification du point de vue du cycle de vie du programme :
  - Erreurs qu'on peut détecter lors de la compilation – erreurs syntaxiques, les plus bénignes.
  - Erreurs qu'on détecte à l'exécution et qui on peut traiter à ce moment-la.
  - Erreurs détectées à l'exécution mais qu'on ne peut pas traiter.
  - Erreurs non-détectables à l'exécution.
- Origine des erreurs détectables à l'exécution :
  - Erreurs d'entrée-sortie : fautes de saisie, fichier corrompu ou non-existant.
  - Erreurs de matériels : imprimante déconnectée, page web temporairement inaccessible.
  - Contraintes physiques : disque dur saturé, mémoire insuffisante.
  - Erreurs de programmation : index hors tableau, référence non-initialisée, division par zéro.

Qu'est-ce que notre programme peut faire lors d'une erreur

- Revient à un état correcte/défini et poursuit son exécution – :-).
- Previent l'utilisateur, sauvegarde de l'état courant et sortie du programme – :-/.
- Sort un écran noir avec un message bêtâ `erreur at 0x56473` ou brouillé de couleurs :-(.
- Bien sûr, celà depend de la gravité de l'erreur !
- Mais peut-on rendre notre programme le plus "friendly" possible ?

Traitement des erreurs – solutions possibles

- Solution C : chaque programme/fonction/module renvoie un code d'erreur.
  - Ouais, mais comment faire si notre fonction devrait déjà renvoyer qqchose ? !
  - Et puis, cela reste une suggestion, les programmeurs ne sont pas tenus à s'en soumettre ! du coup, y en a plein qui ne le font pas !
  - Puisque ce n'est pas une **contrainte syntaxique** !
- Solution matérielle ou de bas niveau : interruptions/signaux.
  - Solution de trop bas niveau...
  - Mais elle mérite d'être étudiée et comprise !

Solution Java : exceptions

- Reprise de la solution matérielle :
  - Si un problème survient, si un comportement anormal est détecté, une exception est levée.
  - Toute exception peut et doit être levée.
- Le code des exceptions est à part, séparé du code de traitement normal.
- Un type d'exception est une classe Java.
- Les programmes Java sont assez sûrs... au moins plus surs que les programmes C !

Les exceptions

- Signaler une situation exceptionnelle = **lever une exception**.
- Les méthodes peuvent lever des exceptions.
- Traitement d'une exception = **attraper des exceptions**.
- On doit concevoir du code pour attraper une exception et la traiter.
- Une exception peut aussi être relancée (ou relevée), signalant ainsi l'ignorance du mode de traitement de la situation respective à l'endroit où elle apparait.

Attraper une exception

- On met dans un bloc `try` le code où se trouv(ent) le(s) méthodes qui peut(peuvent) lever l'exception (les exceptions).
- Suivant ce bloc, on place un ou plusieurs blocs `catch`, où l'exception sera "traitée".
  - Chaque bloc `catch` permet de "prendre la main" lors de la levée du type d'exception qu'il prend en argument.
- Le bloc `catch` = le **handler** de l'exception.
- Un bloc `finally` peut apparaître éventuellement, et ce bloc décrit le fonctionnement après la terminaison de la construction `try-catch`.

Exemple

```
class EssaiExceptions{
  public static void main(String[] argv){
     int[] tableau = new tableau[3];
     for(int i=0; i<4; i++)
         tableau[i] = i;
  }
  System.out.println("Programme fini");
}
```

- Lors de l'essai d'insérer le 4e élément dans le tableau, une exception de type `ArrayIndexOutOfBoundsException` sera levée.
- Et le programme se termine, en renvoyant un message d'erreur.
- L'instruction d'affichage du message "Programme fini" ne s'affiche pas.

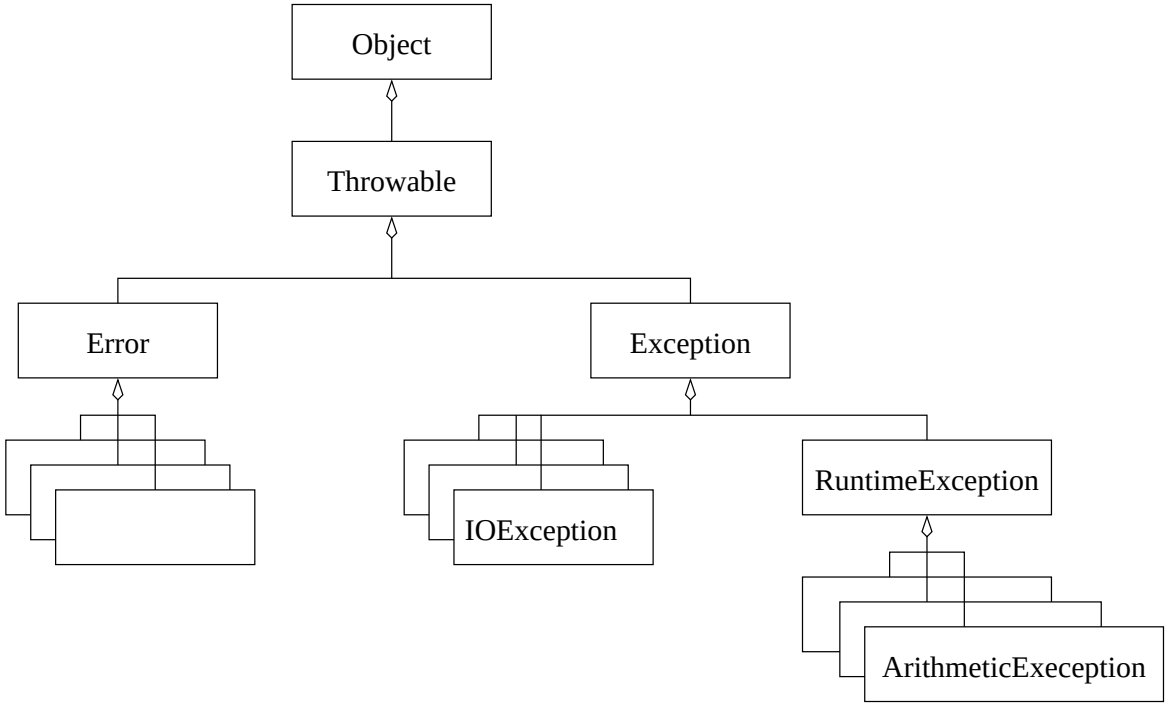
Exemple

♦ Mais on veut ne pas se faire tuer par la JVM lors de cette exception ! On veut traiter cette erreur et continuer à travailler !

```
class EssaiExceptions{
  public static void main(String[] argv){
     int[] tableau = new tableau[3];
     try{
         for(int i=0; i<4; i++)
             tableau[i] = i;
     }
     catch (ArrayIndexOutOfBoundsException e){
             System.Out.println("Pourquoi veux tu m'embeter ?");
     }
     System.out.println("Programme fini");
  }
}
```

♦ Lors de l'essai d'affecter une valeur à `tableau[3]`, l'exception levée sera traitée dans le corps `catch` et l'exécution reprendra après ce corps.

Hiérarchie des types d'exceptions



- Classe `Error` : erreurs sérieux (`FactoryConfigurationError`, `LinkageError`, `VirtualMachineError` qu'une application ne devrait pas être en mesure d'attraper.
- Classe `Exception` : exceptions qu'une application raisonnable pourrait attraper et traiter sans pour cela détruire l'ordinateur.
- Classe `RuntimeException` : exceptions qu'on n'a pas besoin d'attraper explicitement.

Exemples d'exceptions

- `ArithmeticException` : par exemple, division par zéro.
- `NegativeArraySizeException`.
- `ArrayIndexOutOfBoundsException`.
- `StringIndexOutOfBoundsException`.
- `IndexOutOfBoundsException` – classe mère pour les deux.
- `ArrayStoreException`.
- `ClassCastException`.
- `IllegalArgumentException`.
- `NullPointerException`.
- `IOException`

Toutes, sauf `IOException`, se trouvent dans `RuntimeException`.

Classe `Throwable`

- Constructeurs : `Throwable()` et `Throwable(String message)`.
  Le dernier nous permet de définir un message spécifique à afficher lors de l'attrapage de l'exception.
- `getMessage()` – renvoie le message définit lors de la création de l'objet.
- `toString()` – donne une description courte de l'objet qu'on peut afficher en tant que `String`.
  `toString` est la méthode héritée de la classe `Object` !
- Il est possible de définir des *causes* pour la levée d'une exception.

Gestion de plusieurs exceptions

```
import java.io.*;

class Fichier{
  public static void main(String[] argv){
     try{
        FileInputStream f = new FileInputStream("toto.txt");
        byte[] tmp = new byte[10];
        f.read(tmp);
        String s = new String(tmp);
        System.out.println("Ce que j'ai lu: "+s+" fin");
     }
     catch(FileNotFoundException e){
        System.out.println("Pas de fichier !");
     }
     catch(IOException e){
        System.out.println("J'ai pas pu lire !");
     }
     catch(Exception e){
        System.out.println("Je ne sais pas quelle autre exception :" + e);
     }
  }
}
```

- `FileNotFoundException` est une héritière de `IOException` !
- Son handler doit se trouver **avant** celui de sa mère !

Laisser passer une exception

- Il est parfois intéressant de laisser passer une exception dans une méthode `A` et de la traiter dans la méthode qui appelle `A`, soit `B`.

```
class Essai{
  static int lecture() throws IOException{
     return System.in.read();
  }
}
```

- Il est impératif de déclarer les exceptions laissées passer quand il s'agit des exceptions dérivant de la classe `Exception` !

  Raison : les exceptions font partie de l'"interface" des méthodes !

- Par contre, les héritières de `RuntimeException` peuvent être laissées passer sans le déclarer explicitement !

```
class Essai2{
  static int[] tableau = {1,2,3,4};
  static int renvoi_stupide(){
     return tableau[6];
  }
}
```

Bloc `finally`

- Contient du code à exécuter systématiquement
  - que le bloc `try` lève l'exception ou non.
  - que l'exception est attrapée ou non.
  - donc, si l'exception est laissée passer, le bloc `finally` s'exécute toujours, même avant de traiter la situation exceptionnelle !
- Permet de "faire le ménage" – ex. libérer certaines ressources qui ont été spécifiquement requises lors du traitement des exceptions.

```
int fonction(){
   try{ // code à surveiller
   }
   catch{ // traitement des erreurs
   }
   finally{ // nettoyage, toujours exécuté
   }
   // reste de la fonction, exécutée seulement lors du
   // fonctionnement correcte ou d'un attrapage par catch
   // mais pas lors d'un "laisser passer"
}
```

Lever explicitement une exception

- Parfois on veut signaler explicitement une situation exceptionnelle, même avant qu'elle se produise.
- Clause `throw` avec l'exception qu'on lève.

```
int LireDenomiateur() throws ArithmeticException{
    int x=Lecteur.readInt();
    if (x==0) throw new ArithmeticException("zero lu au clavier !");
    System.out.println("x="+x);
    return x;
}
```

- On peut lever explicitement même l'exception qu'on est en train de traiter dans une clause `catch` !
- `Lecteur` est une classe intermédiaire qui aide à lire des entiers.
- Il faut faire attention, lors de l'attrapage de l'exception, au fait que `Lecteur` n'a rien renvoyé lors de la levée d'une exception !
- La semaine prochaine on verra aussi comment implémenter le `Lecteur`...

Sommaire du traitement des exceptions

Lors d'une exception générée dans le `try` :

- La JVM cherche immédiatement le premier handle `catch` compatible avec l'exception – c'est à dire, le handle qui prend en paramètre la classe de l'exception levée ou une classe ancêtre de celle-ci.
- Si trouvé, la JVM exécute ce handle.
- Sinon, la JVM exécute d'abord le `finally` (si présent) et ensuite remonte l'exception au `try` englobant le plus proche, dans la méthode qui a appelé la fonction qui a déclenché l'exception.
- Si pas de niveau englobant, le programme s'arrête et un message indiquant l'exception est affiché.
- Cela (se rendre compte qu'il n'y a pas de niveau englobant) peut se passer **que** pour les `RuntimeException`s !

Créer des nouvelles exceptions

- Utile lorsqu'on rencontre un problème qui ne correspond pas aux types d'exceptions standards.
- On peut alors créer nous-même une nouvelle classe `MonException`, héritant de la classe `Exception` ou de la classe `RuntimeException`.
- On crée deux constructeurs :
  - un constructeurs sans arguments qui fait appel au `super()`.
  - un constructeur avec un argument `String qqchose` – le message d'erreur – qui est passé au constructeur de la classe de base avec `super(qqchose)`.

Créer des nouvelles exceptions

Exemple : classe spéciale pour la division par zéro

```
class DivisionByZeroException extends ArithmeticException{
   DivisionByZeroException(){ super("Division par zéro); }
   DivisionByZeroException(String mesg){ super(mesg); }
}
class Main{
  int LireDenomiateur() throws ArithmeticException{
    int x=Lecteur.readInt();
    if (x==0) throw new ArithmeticException("zero lu au clavier !");
    System.out.println("x="+x);
    return x;
  }
}
```

Quand et comment utiliser des exceptions

- Exception = **situation exceptionnelle !**
- Donc, ne les mettez pas dans la sauce n'importe où !
- Pour une lecture d'une chaîne d'entiers du clavier, il convient de faire des tests sur les indices que de lever des exceptions !
- Le mécanisme de levée/attrapage des exceptions est très gourmand, il consomme du temps précieux pour votre programme !