المدرسة الوطنية للعلوم التطبيقية –أكادير
ⵜⵉⵎⵙⵙⵓⵔⵜ ⵜⴰⵏⴰⵎⵓⵔⵜ ⵏ ⵜⵙⵙⵏⴰⵢⵉⵏ ⵜⵉⵜⵜⴰⵡⵉⵏ – ⴰⴳⴰⴷⵉⵔ.
ÉCOLE NATIONALE DES SCIENCES APPLIQUÉES - AGADIR

LA GESTION DES EXCEPTIONS

Abdellaziz Walid
a.walid@uiz.ac.ma
Département G-Info
2019-2020

Motivation 1

Question : Pourquoi tous les aéroports du monde recevant des avions de lignes doivent obligatoirement compter une équipe de pompiers qui s'entraîne très régulièrement, malgré la rareté des incendies ?

Réponse : Selon les normes de protection contres les incendies, l'intervention à l'endroit le plus éloigné de la piste d'un aéroport doit être faite en moins de 3 minutes après le déclenchement de l'alerte afin d'évacuer tous les passagers à bord de l'avion.

Motivation 2 : Contexte de programmation

Class Operation

```
public final class Operation{
public static double division( double p,
double q)
{
 return p / q;
}
public static double somme( double p,
double q)
{
 return p+q;
}
…… }
```

Quoi faire si q=0;

Programme principale

```
import java.util.Scanner;
class Main {
public static void main(String args[]) {
double p, q, r;
System.out.println("Enter two real
numbers for Division: ");
Scanner sc = new Scanner(System.in);
p = sc.nextInt();
q = sc.nextInt();
r = Operation.division(p,q);
System.out.println("The result of the
Division = "+r);} }
```

Questions : - Y-a-t'il une exception dans ce programme ?
- Ce programme prévoit-t-il et gère-t-il les exceptions ?

Exception : contexte de programmation

- Une exception est un évènement qui arrive durant l'exécution d'un programme qui interrompt le déroulement normal des instructions.

- Plusieurs erreurs très différentes peuvent provoquer des exceptions, d'un problème matériel, manque de mémoire, ,des actions imprévues de l'utilisateur, …etc.

- Lorsque ce genre de chose arrive, il donne l'impression que le logiciel n'est pas robuste. D'où :
  - Nécessité de prévoir que le logiciel sera soumis à des défaillances dues à certains paramètres externes ou internes(exceptions).
  - Prévoir une réponse adaptée à chaque type de situation(exception).

- L'objectif principal est d'améliorer la qualité de " ***robustesse*** " d'un logiciel.

Application sans gestion d'exceptions

Class Operation

```
public final class Operation{
public static double division( double p, double q)
{
 return p / q;
}
public static double somme( double p, double q)
{
 return p+q;
}
…… }
```

Erreur si q=0 qui provoquera l'arrêt du programme

Programme principale

```
import java.util.Scanner;
class Main {
public static void main(String args[]) {
double p, q, r;
System.out.println("Enter two real numbers for Division: ");
Scanner sc = new Scanner(System.in);
p = sc.nextInt();
q = sc.nextInt();
r = Operation.division(p,q);
System.out.println("The result of the Division = "+r);} }
```

Si l'utilisateur tape 0

La Java machine arrêterait le programme immédiatement à cet endroit parce qu'elle n'aurait pas trouvé de code d'interception de cette exception qui serait automatiquement levée.

Comment gérer l'exception ?

Programme principale

```
import java.util.Scanner;
class Main {
public static void main(String args[]) {
double p, q, r;
System.out.println("Enter two real numbers for Division: ");
Scanner sc = new Scanner(System.in);
p = sc.nextInt();
q = sc.nextInt();
if q!=0 {
r = Operation.division(p,q);
System.out.println("The result of the Division = "+r);}
else
 System.out.println("Error : Division by zero = ");}}
```

- Solution 1 : Détecter et gérer l'exception au niveau du programme principale.

- Problème : la réutilisation de la classe Operation dans un autre programme ou application provoquera un arrêt immédiat du programme.

Comment gérer l'exception ?

Classe Operation

```
public final class Operation{
public static double division( double p, double q) {
if q!=0
 return p / q;
else {
System.out.println("Error : Division by zero = ");
return ???????????
}
public static double somme( double p, double q)
{
 return p+q;
}
…… }
```

- Solution 2 : Détecter et gérer l'exception au niveau local.
- Problème : quoi retourner effectivement en cas d'erreur.
- Très mauvaise solution car il produit des effets de bord : Affichage dans le terminal alors que ce n'est pas du tout le rôle de méthode division.

Comment gérer l'exception ?

Classe Operation

```
public final class Operation{
public static Boolean division( double p, double q, double r) {
if q!=0 {
r=p/q;
 return true
}
else
return false
}
public static double somme( double p, double q)
{
 return p+q;
}
…… }
```

Solution3 : déjà meilleur car elle laisse à la fonction qui appelle la méthode division quoi faire en cas d'erreur.

Mais elle présente des inconvénients :

- Cas de l'appel d'appel d'appel …. d'appel de fonction.
- Écriture peu intuitive division(x,y,z) au lieu de z=division(x,y) : division est un opérateur binaire

Une autre solution mais efficace

- Il existe une solution efficace permettant de généraliser et d'assouplir cette dernière solution :
  - déclencher une exception.
- Il se base sur un mécanisme permettant de :
  - Prévoir une erreur à un endroit;
  - De la gérer à un autre endroit;
  - De ne pas mettre fin au programme.

Mécanise de gestion des exceptions

Principe :

- Lorsque une erreur a été détectée à un endroit, on la signale en « lançant » un objet contenant toutes les informations que l'on souhaite donner sur l'erreur.
- À l'endroit où l'on souhaite gérer l'erreur (au moins partiellement), on peut « attraper » l'objet « lancé ».
- Si un objet « lancé » n'est pas attrapé du tout, cela provoque l'arrêt du programme : toute erreur non gérée provoque l'arrêt.
- Un tel mécanisme s'appelle « gestion des exceptions ».

Illustration du mécanisme de gestion des exceptions

Class Operation

```
public final class Operation{
public static double division( double p, double q)
{
 return p / q;
}
public static double somme( double p, double q)
{
 return p+q;
}
…… }
```

Lancement d'une exception (objet)

Cette fois pas d'arrêt de l'exécution du programme

Programme principale

```
import java.util.Scanner;
class Main {
public static void main(String args[])
{
double p, q, r;
System.out.println("Enter two real numbers for Division: ");
Scanner sc = new Scanner(System.in);
p = sc.nextInt();
q = sc.nextInt();
r = Operation.division(p,q);
System.out.println("The result of the Division =
```

Si l'utilisateur tape 0

Rattraper et traiter l'objet

Gestion des exceptions : contexte Java

Dans Java, on cherche à remplir 4 tâches élémentaires :

A. Signaler une erreur (exception).

B. Marquer les endroits réceptifs aux erreurs (exceptions).

C. Leur associer (à chaque endroit réceptif) un moyen de gérer les erreurs(exceptions) qui se présentent.

D. Éventuellement, « faire le ménage » après un bloc réceptif aux erreurs.

Syntaxe de gestion des exceptions

On a donc 4 mots du langage Java dédiés à la gestion des exceptions :

1. throw: indique l'erreur (c-à-d lance l'exception)
2. try : indique un bloc réceptif aux erreurs
3. catch : gère les erreurs associées sou exceptions (i.e. les « attrape » pour les traiter)
4. Finally : (optionnel) indique ce qu'il faut faire après un bloc réceptif

Illustration du mécanisme de gestion des exception par java

Class Operation

```
public final class Operation{
public static double division( double p, double q)
{
if (q==0) {
throw …….. // lancer l'exception(objet)
}
return p / q;
}
public static double somme( double p, double q) {
 return p+q;}
…… }
```

Cette fois pas d'arrêt de l'exécution du programme

Programme principale

```
import java.util.Scanner;
class Main {
public static void main(String args[]) {
double p, q, r;
System.out.println("Enter two real numbers for Division:
");
try { // bloc pour attraper l'objet (bloc réceptif d'erreurs)
Scanner sc = new Scanner(System.in);
p = sc.nextInt();
q = sc.nextInt();
r = Operation.division(p,q);
}
Catch (…….) { //bloc pour rattraper et traiter l'erreur }
Finally {……..} // bloc pour faire ménage
System.out.println("The result of the Division = "+r);}}
```

Si l'utilisateur tape 0

Synthèse

Une exception est un moyen de signaler un événement nécessitant une attention spéciale au sein d'un programme, comme :

- Une erreur grave
- Une situation inhabituelle devant être traitées de façon particulière.
- But : améliorer la robustesse des programmes en :
  - Séparant le code du traitement de l'exception du code effectif.
  - Fournissant le moyen de forcer une réponse à des erreurs particulières.

throw

- throw est l'instruction qui signale l'erreur au reste du programme.
- Syntaxe java : throw exception;

exception : est un objet de type de la classe Exception qui est lancé au reste du programme pour être « attrapé »

Exemple :

```
Throw new Exception("my first exception");
```

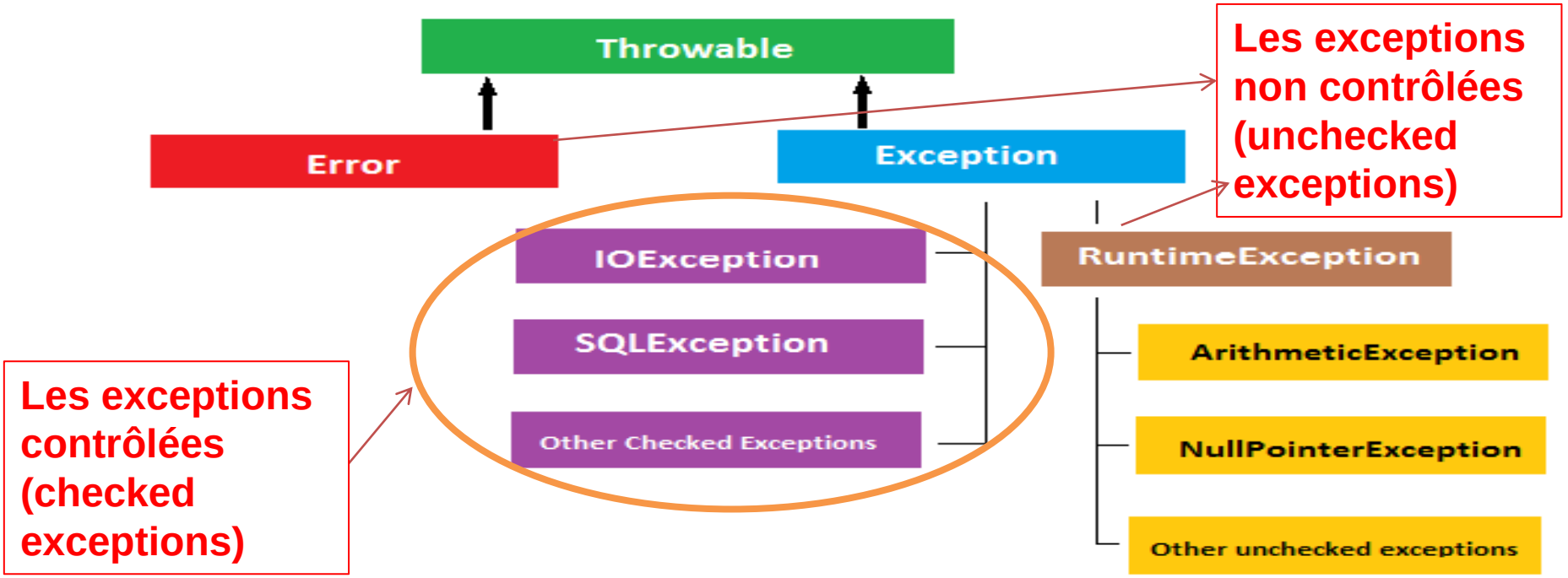
Remarque:

Exception est une classe de java.lang qui possède plusieurs sous classes et qui hérite de la classe Throwable.

Exemple de throw

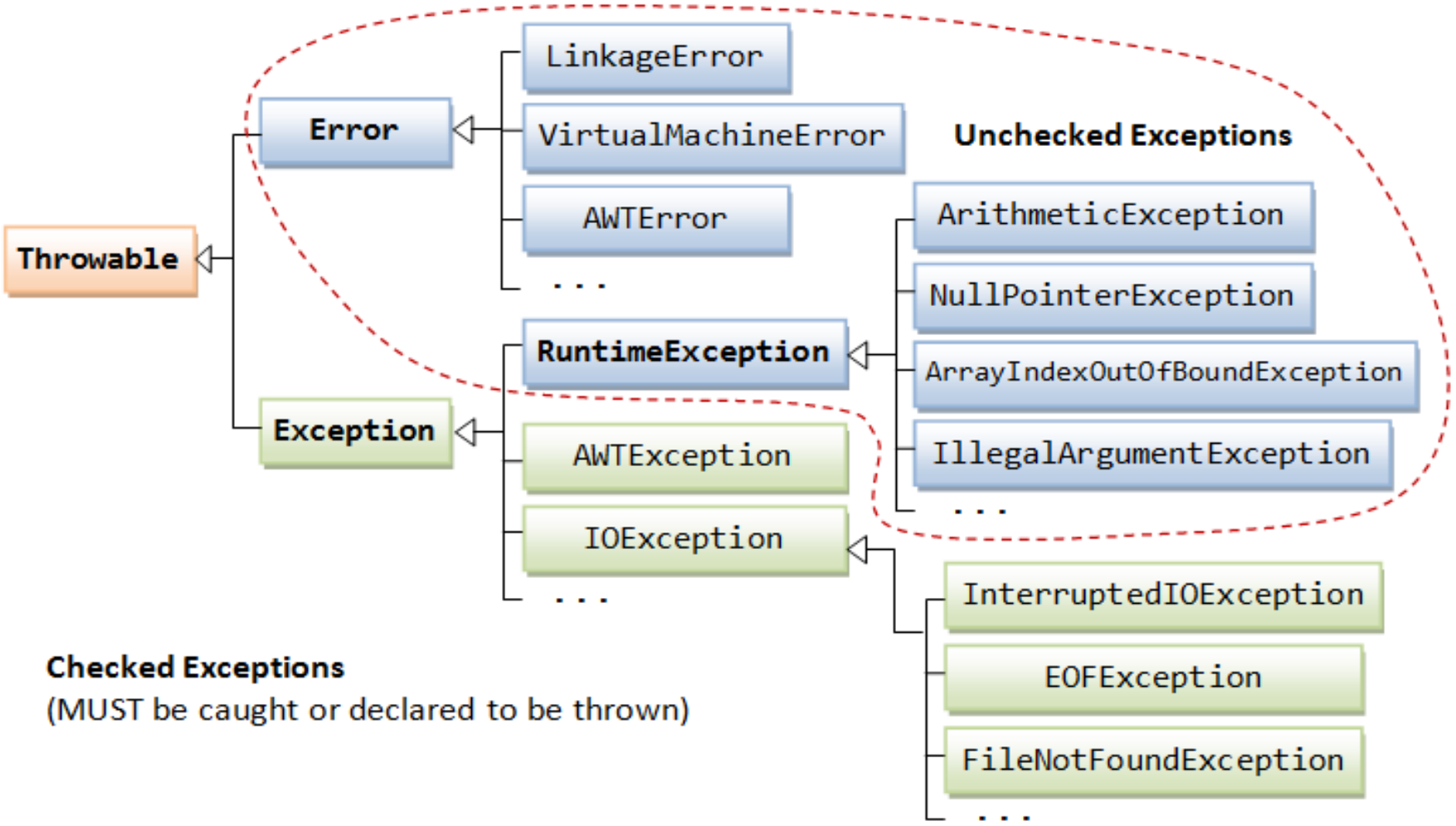

```
public final class Operation{
public static double division( double p, double q)
{
if (q==0) {
throw new Exception("Division par zéro");
}
return p / q;
}
public static double somme( double p, double q)
{
 return p+q;
}
…… }
```

Hiérarchie partielle de Throwable



- Error correspondent à des exceptions ou erreurs fatales car elles sont engendrées par la JVM. Le développeur n'est pas censé d'attraper et traiter (try/catch) ce genre d'erreurs.
- Les RuntimeException que l'on peut rattraper et traiter mais que l'on n'est pas obligé.
- Les exceptions contrôlées que l'on est obligé d'attraper (try/catch) ou de dire que la méthode appelante devra s'en occuper (throws).

Hiérarchie partielle de Throwable



Exemple de throw avec un objet plus spécifique


```
public final class Operation{
public static double division( double p, double q)
{
if (q==0) {
throw new ArithmeticException("Division par zéro");
}
return p / q;
}
public static double somme( double p, double q)
{
 return p+q;
}
…… }
```

La classe java.lang.Throwable

public class Throwable extends Object

- Posséde deux constructeurs (avec et sans message) :
  - public Throwable()
  - public Throwable (String message)
- Deux méthodes (parmi d'autres) :
  - Accès au message d'erreur :

    public String getMessage()
  - Affichage du chemin vers l'erreur :

    public void printStackTrace()

try

try :

Introduit un bloc réceptifs aux exceptions lancées par des instructions, ou des méthodes appelées par des méthodes ……..appelées à l'intérieur du bloc.

Exemple :

```
public static void main(String args[])
{
double p, q, r;
System.out.println("Enter two real numbers for Division: ");
Scanner sc = new Scanner(System.in);
try {
p = sc.nextInt();
q = sc.nextInt();
r = Operation.division(p,q);
}
System.out.println("The result of the Division = "+r);}
 }
```

catch

- Le mot clé catch introduit un bloc dédié à la gestion d'une ou plusieurs exceptions.
- Tout bloc try doit toujours être suivi d'au moins un bloc catch gérant les exceptions pouvant être lancées dans ce bloc try.
- Si une exception est lancée mais n'est pas interceptée par le catch correspondant, le programme s'arrête (message « Exception in thread… » et affichage de la trace d'exécution).
- Syntaxe :

```
catch (type nom) {……..}
```

  - Intercepte toutes les exceptions de type type lancées depuis le bloc try précédent.
  - type peut être une classe prédéfinie de la hiérarchie d'exceptions de Java ou une classe d'exception créer par le programmeur.

Exemple de catch


```
try {…….
}
catch (IOException ex)  {
logger.log(ex);
throw ex;
}
catch (SQLException ex) {
logger.log(ex);
throw ex;
}
```

À partir de java 7 : multi-catch : traiter plusieurs types d' exceptions par un seul bloc catch

```
catch (IOException | SQLException ex | ……)
{
logger.log(ex);
throw ex;
}
```

Syntaxe : try and catch


```
try {
 < bloc de code à protéger>
}
catch ( TypeException1 E ) { <Traitement TypeException1 > }
catch ( TypeException2 E ) { <Traitement TypeException2 > }
.....
catch ( TypeExceptionk E ) { <Traitement TypeExceptionk > }
```

Où TypeException1, TypeException12, … , TypeExceptionk sont des classes d'exceptions obligatoirement toutes **distinctes.**

Seule une seule clause `catch ( TypeException E ) {...}` est exécutée (celle qui correspond au bon type de l'objet d'exception instancié).

N.B :

S'il y a plusieurs blocs catch toujours les indiquer du plus spécifique au plus général (sinon erreur signalée par le compilateur).

Exemple : try and catch


```
try {
 // ...
 if (age >= 150)
 { throw new Exception("valeur trop grande"); }
 // ...
 if (x == 0.0)
 { throw new ArithmeticException("Division par zero"); }
 // ...
}

catch (ArithmeticException e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
}

catch (Exception e) {
    System.out.println("Qui peut vivre si vieux?");
}
```

Exception spécifique

Exception générale

try\throw\catch dans la même méthode


```
int lireEntier(int maxEssais) throws Exception
{
  int nbEssais = 1;
  do {
    System.out.println("Donnez un entier : ");
    try {
      int i = clavier.nextInt();
      return i;
    }
    catch (InputMismatchException e) {
      System.out.println("Il faut un nombre entier. Recommencez !");
      clavier.nextLine();
      ++nbEssais;
    }
   } while(nbEssais <= maxEssais);

  throw new Exception ("Saisie échouée");
}
```

finally

- Le bloc finally est optionnel, il suit les blocs catch.
- finally est optionnel.
- Il contient du code destiné à être exécuté qu'une exception ait été lancée ou pas par le bloc try.
- Le but de finally est de faire le ménage (fermer des fichiers, des connexions etc…).

Déclaration d'une exception

- Une méthode lançant une exception sans la traiter localement doit généralement informer qu'il le fait.
- La syntaxe pour déclarer une exception est :

Type methode (….) throws Exception1, Exception 2, ……

Exemple :

```
public static double division( double p, double q) throws  ArithmeticException
{
if (q==0) {
throw new ArithmeticException("Division par zéro");
}
return p / q;
}
}
```

Règle : déclarer ou traiter

Toute les exceptions contrôlées (cheked exceptions) en dehors des RunTimeException et des Error (uncheked exceptions) :

- Soit être interceptées dans la méthode où elles sont lancées.
- Soit être déclarées par la méthode.
- Sinon : le compilateur émettra un message d'erreur.

Problème : manque d'informations


```
public static void main(String[] args) {
  int nbEssais = 0;
  final int MAX_ESSAIS = 2;
  ArrayList<Double> mesures = new ArrayList<Double>();

  do {
    nbEssais++;

    acquerirTemp(mesures);      // remplit le tableau

    try {
      plotTempInverse(mesures);
     }
   // ...
    catch (ArithmeticException e) {
      if (nbEssais < MAX_ESSAIS) {
        System.out.println("Ressaisir les valeurs !");
      } else {
        System.out.println("Il y a déjà eu au moins "
           + MAX_ESSAIS + " essais.");
        System.out.println(" -> abandon");
      }
    }
  } while (nbEssais < MAX_ESSAIS);
}
```

```
private static void plotTempInverse(ArrayList<Double> t)

{
  for(int i = 0; i < t.size(); i++) {

     plot(inverse(t.get(i)));
   }

}
```

```
private static void plot(double x) {
  // fait le dessin
}
private static double inverse(double x)
  throws ArithmeticException // PAS NECESSAIRE
                             //RunTimeException
{
 if (x == 0.0) {
  throw new ArithmeticException("Division par 0 !");
 }
 return 1.0/x;
}
```

Manque d'informations

Lever une exception qui porte comme information « Division par zéro» sans préciser l'élément de la liste où se trouve le zéro.

Solution : Relancement

- Une exception peut être partiellement traitée par un bloc catch et attendre un traitement plus complet à un niveau supérieur.
- Il suffit pour cela de relancer l'exception au niveau du bloc n'effectuant que le traitement partiel.
- Il faudra pour cela que l'appel à ce bloc de catch soit lui-même dans un autre bloc try à un niveau supérieur.

Exemple de relancement


```
public static void main(String[] args) {
  int nbEssais = 0;
  final int MAX_ESSAIS = 2;
  ArrayList<Double> mesures = new ArrayList<Double>();

  do {
    nbEssais++;

    acquerirTemp(mesures);      // remplit le tableau

    try {
      plotTempInverse(mesures);
     }
   // ...
    catch (ArithmeticException e) {
      if (nbEssais < MAX_ESSAIS) {
        System.out.println("Ressaisir les valeurs !");
      } else {
        System.out.println("Il y a déjà eu au moins "
           + MAX_ESSAIS + " essais.");
        System.out.println(" -> abandon");
      }
    }
  } while (nbEssais < MAX_ESSAIS);
}
```

```
private static void plotTempInverse(ArrayList<Double> t)
throws ArithmeticException
{
  for(int i = 0; i < t.size(); i++) {
    try {
      plot(inverse(t.get(i)));
    } catch (ArithmeticException e) {
      System.out.println("Problème à l'indice :" + i);
      // RELANCEMENT
      throw e;
    }
  }
}
```

Relancement de l'exception pour capturer l'indice de l'élément qui a levé l'exception

```
private static void plot(double x) {
  // fait le dessin
}

private static double inverse(double x)
   throws ArithmeticException // PAS NECESSAIRE
                              //RunTimeException
{
  if (x == 0.0) {
    throw new ArithmeticException("Division par 0 !");
  }
  return 1.0/x;
}
```

Lever une exception qui porte comme information « Division par zéro»

Exceptions personnalisées

Dans Java, il est possible de programmer ses propres classes d'exception soit :

- Comme sous classe de la classe Exception
- Comme sous classe d'une sous classe existante de Exception

Contenu minimal :

```
Classe MonException extends Exception
{
public MonException() {
Super("mon message par défaut");
}
Public MonException(String message) {
Super(message);
}
}
```

Exceptions personnalisées

Il est possible de définir dans une sous classe d'exception personnalisée tout membre jugé utile :

- Code d'erreur.
- Informations sur le contexte de détection de l'exception.
- etc…

Exemple : Exception personnalisée


```
class TropChaudException extends Exception
{
 private double temperatureAnormale;
 private String consigne;

 public TropChaudException(double uneTemperature, String uneConsigne) {
  super("Température trop élevée");
  temperatureAnormale = uneTemperature;
  consigne = uneConsigne;
 }
 public double getTemperature() {
  return temperatureAnormale;
 }
 public String getConsigne() {
  return consigne;
 }
}
```

Exemple : Exception personnalisée (try … catch)


```
try {
  //...
    if (temperature > TEMP_MAX){
      throw new TropChaudException(temperature,
                "Vérification de l'appareil de mesure");
    }
  }

  catch(TropChaudException e){
    System.out.print(e.getMessage() + " : " );
    System.out.println(e.getTemperature());
    System.out.println("Consigne -> " + e.getConsigne());
  }
```

Conclusion

- Lancer des exceptions plus informatif et utile.
- Si l'erreur peut être levée et traiter là où elle est découverte, il faut le faire sans la traiter à un niveau supérieur.
- La gestion d'une exception coûte beaucoup plus en temps qu'une simple structure de contrôle conditionnelle (if … then … else)