Les exceptions

NFA032

Présentation n°4

La gestion des erreurs sans les exceptions

- Un exemple basique
  - Écrire un programme :
    - Demandant un réel x
    - Affichant la valeur de $f(x) = \frac{3}{1-x}$
  - Exemples d'exécution :
    - Premier exemple :

```
x = 2↵
f(2) = 3.0
```

    - Deuxième exemple :

```
x = 1↵
f(1) = ???
```

Solutions

- Cinq solutions étudiées :
  - Solution sans aucun contrôle
  - Solution avec contrôle à l'appel (coté *appelant* = *client* = fonction *main*)
  - Deux solutions avec contrôle dans la fonction (coté *appelé* = *fournisseur du service* = fonction *f*)
  - Solution avec exception : idéale ☺

Solution 1 : aucun contrôle

```
class solution_1 {

  static double f(double x) { return 3/(1-x); }

  public static void main (String[] args) {
    Terminal.ecrireString("x = ");
    double x = Terminal.lireDouble();
    Terminal.ecrireStringln("f("+x+") = "+f(x));
  }
}
```

```
x = 2↵
f(2.0) = 3.0
```

```
x = 1↵
f(1.0) = Infinity
```

Solution 1 : analyse

- Avantages
  - Code simple, aussi bien du coté appelé (*f*) que appelant (*main*)
    - Sans filet
- Inconvénients
  - Si une erreur est rencontrée, aucun traitement associé
    - Plantage du programme principal probable

Solution 2 : contrôle avant appel

```
class solution_2 {

  static double f(double x) { return 3/(1-x); }

  public static void main (String[] args) {
    Terminal.ecrireString("x = ");
    double x = Terminal.lireDouble();
    if (x==1)
      Terminal.ecrireStringln("f("+x+") est indéfini");
    else
      Terminal.ecrireStringln("f("+x+") = "+f(x));
  }
}
```

```
x = 2↵
f(2.0) = 3.0
```

```
x = 1↵
f(1.0) est indéfini
```

Solution 2 : analyse

- Avantages
  - coté appelé (*f*) :
    - Code simple, mais sans filet
  - coté appelant (*main*) :
    - Appels sûrs
- Inconvénients
  - coté appelant (*main*) :
    - pas obligé de contrôler (on retrouve la solution 1)
    - doit systématiquement vérifier que les paramètres ne poseront pas de problème avant chaque appel
    - doit avoir connaissance de tous les cas qui posent problème

Solution 3 : contrôle au niveau de *f*

- Version astucieuse : ne change pas le type de résultat
  - exploite une valeur spéciale du résultat : ici 0 n'est pas une valeur possible pour *f*(*x*) ; pas toujours possible.

```
class solution_3 {

  static double f(double x) { return x==1 ? 0 : 3/(1-x); }

  public static void main (String[] args) {
    Terminal.ecrireString("x = ");
    double x = Terminal.lireDouble();
    double y = f(x);
    if (y==0)
      Terminal.ecrireStringln("f("+x+") est indéfini");
    else
      Terminal.ecrireStringln("f("+x+") = "+y);
  }
}
```

Solution 3 : analyse

- Avantages
  - coté appelé (*f*) :
    - A priori, c'est l'appelé qui est le plus à même de savoir quels jeux de paramètres poseront des problèmes
    - La signature de la fonction est naturelle (ici `double` vers `double`)
  - coté appelant (*main*) :
    - Appels sûrs
- Inconvénients
  - coté appelant (*main*) :
    - Ce n'est plus lui qui vérifie si les appels sont licites
    - Il doit systématiquement vérifier que le résultat n'est pas une valeur spéciale
    - Doit systématiquement stocker le résultat dans une variable temporaire (y)
  - coté appelé (*f*) :
    - Il n'existe pas toujours de valeurs spéciales

Solution 4 : contrôle au niveau de *f*

- Solution générale si pas d'exception
  - Le résultat est un objet

```
class solution_4 {

  static unResultatEventuel f(double x) {
    unResultatEventuel résultat = new unResultatEventuel();
    résultat.défini = (x != 1); // vrai si x différent de 1, faux sinon
    if (résultat.défini) résultat.valeur = 3/(1-x);
    return résultat;
  }

  public static void main (String[] _) {
    Terminal.ecrireString("x = ");
    double x = Terminal.lireDouble();
    unResultatEventuel r = f(x);
    if (r.défini) Terminal.ecrireStringln("f("+x+") = "+r.valeur);
    else          Terminal.ecrireStringln("f("+x+") est indéfini");
  }
}

class unResultatEventuel {
  boolean défini; // vrai si résultat défini (valeur dans .valeur), faux sinon
  double  valeur;
}
```

Solution 4 : analyse

- Avantages
  - coté appelé (*f*) :
    - Code propre, spécifiant clairement tous les cas de figure
  - coté appelant (*main*) :
    - Appels sûrs
- Inconvénients
  - coté appelant (*main*) :
    - Doit systématiquement stocker le résultat dans une variable temporaire (y)
    - Doit systématiquement vérifier que le résultat n'est pas une valeur spéciale
  - coté appelé (*f*) :
    - Un objet résultat créé à chaque appel
      - Possibilité de mutualiser si objet passé en paramètre
    - Signature de la fonction plus complexe

Solution 5 : les exceptions

```
class erreur_DivisionParZéro extends RuntimeException {}

class exemple {

  static double f(double x) throws erreur_DivisionParZéro {
      if (x != 1) return 3/(1-x);
      else        throw  new erreur_DivisionParZéro();
  }

  public static void main (String[] _) {
    Terminal.ecrireString("x = ");
    double x = Terminal.lireDouble();

    try {
      double y = f(x);
      Terminal.ecrireStringln("f("+x+") = "+y);
    } catch (erreur_DivisionParZéro e) {
      Terminal.ecrireStringln("f("+x+") est indéfini");
    }
  }
}
```

signalement

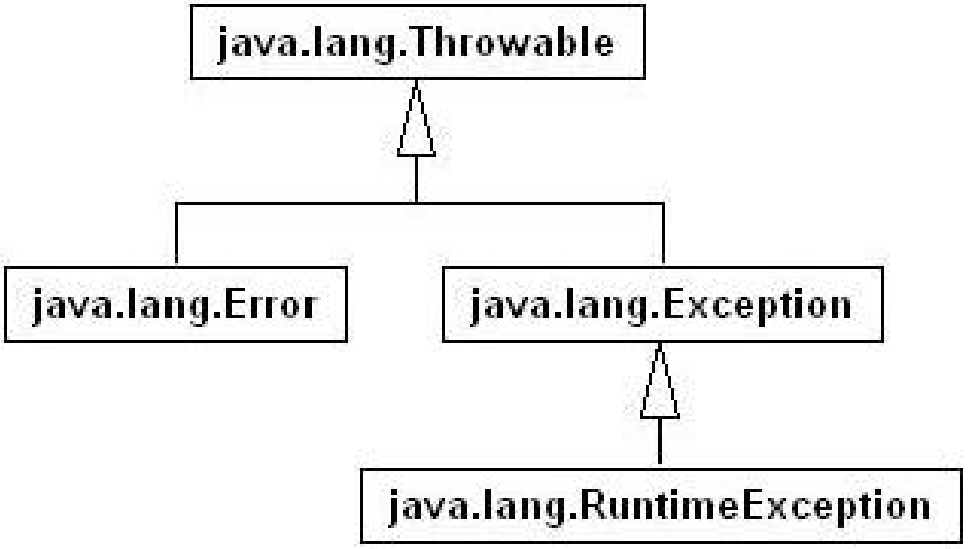
déclenchement

traitement

Les exceptions en Java

- Une exception
  - Est un objet dont la classe implémente `Throwable`
    - Instruction **`throw`**
    - Une exception est levée / déclenchée / jetée
  - Peut être récupérée ou non
    - Instruction **`try`** … **`catch`** … **`finally`**
  - Peut être signalée ou non par la méthode qui la lève
    - Clause **`throws`** intégrée à la signature

Organisation générale



- Trois sortes d'erreurs :
  - L'erreur `Error`
    - Non récupérable
    - Signalement proscrit
    - Signification : erreur fatale
  - L'exception `Exception`
    - Récupérable obligatoirement
    - Signalement obligatoire
    - Signification : erreur à traiter impérativement
  - L'exception d'exécution `RuntimeException`
    - Récupérable éventuellement
    - Signalement optionnel
    - Signification : erreur éventuellement gérable

Principales propriétés

- Si `e` est une exception `Trowable`
  - **`new`** `RuntimeException(String)`
    - Créer un objet avec message d'erreur associé
  - `e.getMessage()`
    - Retourne le message d'erreur associé
  - `e.printStackTrace()`
    - Affiche la pile système des appels en cours

Clause de signalement de levée d'exception : **throws**

- Méthode qui lève une exception:
  - clause de signalement :
    - liste des exceptions susceptibles d'être levées
    - juste après la parenthèse fermante qui clôt la liste des paramètres
    - juste avant l'accolade ouvrante de définition de son corps
    - s'écrit : **throws** *exception*
      - *exception* est un nom de classe d'exception
      - si plusieurs noms, les séparer par une virgule.
    - Fait partie intégrante de la signature

Instruction de levée d'exception : **throw**

- **Syntaxe :**

  throw *expression* ;

  – où *expression* est une expression de calcul dont le résultat doit être un objet exception

- **Sémantique :**

  1. calculer la valeur de *expression* : nécessairement un objet exception.
  2. arrêter le bloc en cours d'exécution ; transférer la gestion de l'erreur au premier bloc englobant piégé par un **`try catch finally`**

Instruction de capture d'exceptions : **try catch finally**

- syntaxe :

```
try bloc0
catch (type1 nom1) bloc1
...
catch (typek nomk) block
finally block+1 ;
```

où :

- $bloc_i$ est un bloc d'instructions appelé *bloc piégé*
- $type_i$ est un nom de classe d'exception
- nom_i est un nom de paramètre capturant l'exception traitée
- il peut y avoir plusieurs clauses de capture **catch**
- il ne peut y avoir au mieux qu'une clause finale **finally**
- il peut ne pas y avoir de clause de capture **catch** ou de clause finale **finally** ; mais il doit nécessairement y avoir l'une ou l'autre
- ne pas écrire de clause finale équivaut à : **finally** {}

Scenario 0 : exception avant ou après
try catch finally

- Situation :

```
avant ;
try bloc0
catch (type1 nom1) bloc1
...
catch (typek nomk) block
finally blocK+1 ;
après ;
```

- Une exception levée dans *avant* ou *après* est gérée dans le bloc englobant **try**

Scenario 1 : aucune exception levée

1. $bloc_0$ s'exécute sans levée d'exception
2. la clause finale $bloc_{k+1}$ est exécutée, sans erreur
3. l'instruction **try** est terminée ; l'exécution se poursuit normalement

```
try
  bloc0
catch (type1 nom1)
  bloc1
...
catch (typek nomk)
  blocк
finally
  bloc k+1
```

Scenario 2 : exception levée dans le bloc piégé, et traitée sans erreur

1. $bloc_0$ s'exécute, mais exception *e* de type T levée :
   - $bloc_0$ est arrêté et fermé.
2. il existe une clause de capture **catch** ($type_i$ nom_i) traitant cette exception T :
   - T hérite de $type_i$
   - clauses examinées dans l'ordre; le bloc de traitement d'exception $bloc_i$ est exécuté, l'environnement d'exécution étant enrichi d'une variable :
     - son nom est nom_i
     - son type est $type_i$
     - sa valeur est *e*
   - Le bloc de traitement se termine sans erreur
3. la clause finale $bloc_{k+1}$ est exécutée, sans erreur
4. l'instruction **try** est terminée ; l'exécution se poursuit normalement

```
try
  bloc0
catch (type1 nom1)
  bloc1
...
catch (typek nomk)
  blocк
finally
  bloc k+1
```

Scenario 3 : exception levée, mais non gérée dans une clause de capture

1. $bloc_0$ s'exécute, mais exception *e* de type T levée :
   – $bloc_0$ est arrêté et fermé.
2. aucune clause de capture **catch** ($type_i$ nom_i) ne traite cette exception T :
   a. la clause finale $bloc_{k+1}$ est exécutée, sans erreur
   b. L'exception *e* est re-levée dans le bloc englobant **try**

```
try
  bloc0
catch (type1 nom1)
  bloc1
...
catch (typek nomk)
  blocк
finally
  blocк+1
```

Scenario 4 : exception levée et traitée mais avec erreur (scenario 2 + erreur)

1. $bloc_0$ s'exécute, mais exception *e* de type T levée :
   - $bloc_0$ est arrêté et fermé.
2. il existe une clause de capture **catch** ($type_i$ nom_i) traitant cette exception T :
   - T hérite de $type_i$
   - clauses examinées dans l'ordre; le bloc de traitement d'exception $bloc_i$ est exécuté, l'environnement d'exécution étant enrichi d'une variable :
     - son nom est nom_i
     - son type est $type_i$
     - sa valeur est *e*
   - Une exception *x* se produit durant le traitement $bloc_i$:
     1. Le bloc $bloc_i$ est arrêté et fermé
     2. la clause finale $bloc_{k+1}$ est exécutée, sans erreur
     3. L'exception *x* est levée dans le bloc englobant **try**

```
try
  bloc0
catch (type1 nom1)
  bloc1
...
catch (typek nomk)
  blocк
finally
  bloc k+1
```

Scenario 5 : erreur durant la clause finale

- Quel que soit le scenario d'origine, une exception *x* est levée durant l'exécution le bloc $bloc_{k+1}$ de la clause finale
  1. Le bloc $bloc_{k+1}$ est arrêté et fermé
  2. L'exception *x* est levée dans le bloc englobant **try**

```
try
  bloc0
catch (type1 nom1)
  bloc1
...
catch (typek nomk)
  blocк
finally
  block+1
```

Quid des exceptions non gérées

- Toute exception non capturée par le programmeur est capturée au niveau de `main`

```
static void extra_main(String[] a)
{
   try {
      main(a);
   } catch (Throwable e) {
      // Affichage sur le terminal de l'erreur
   }
}
```

Un exemple complet

- Source `plusieurs_exceptions`