Le mécanisme d'exception du langage Java

Plan

Qu'est-ce qu'une exception en Java ?

Définition

- Une exception est un événement, se produisant lors de l'exécution d'un programme, qui interrompt l'enchaînement normal des instructions

L'objet Exception

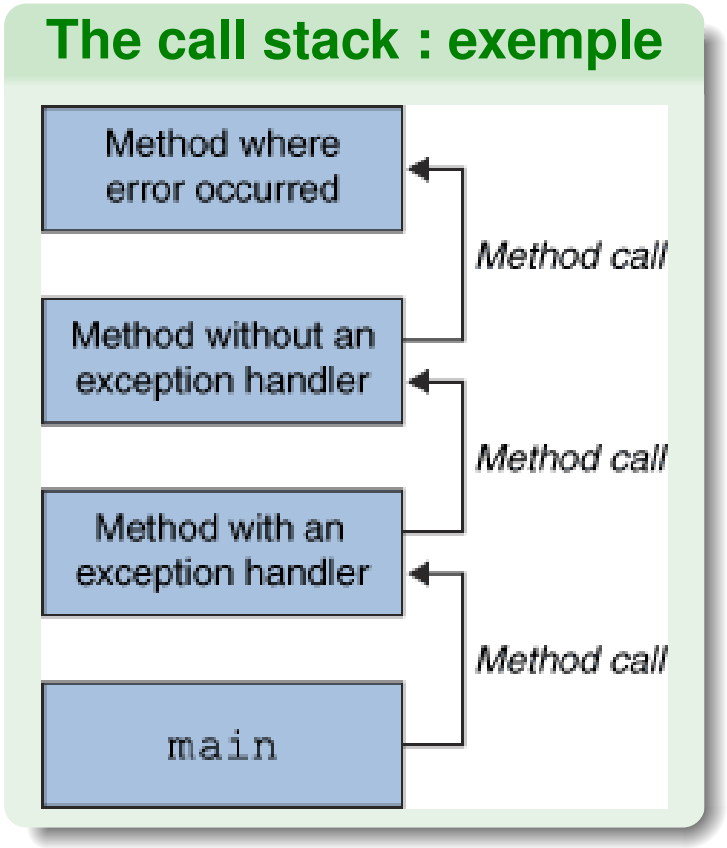
- Lorsqu'une erreur se produit dans une méthode, celle-ci crée un **objet Exception** qui contient des informations sur l'erreur survenue :
  - le type de l'erreur : objet null, dépassement de tableau, etc.
  - la trace d'exécution du programme : la *stack trace*
- L'action de créer un objet Exception et de le passer au système d'exécution est appelée **lever/lancer une exception** (throwing an exception)

Gestion d'une exception

Récupération de l'exception

- Lorsqu'une exception est levée, le système d'exécution tente de trouver un *handler* pour cette exception : c-à-d. une méthode pouvant la gérer.
- Pour cela, il cherchera dans les méthodes qui ont été appelées jusqu'à l'exception : the *call stack* (la pile d'appels)

Gestion d'une exception



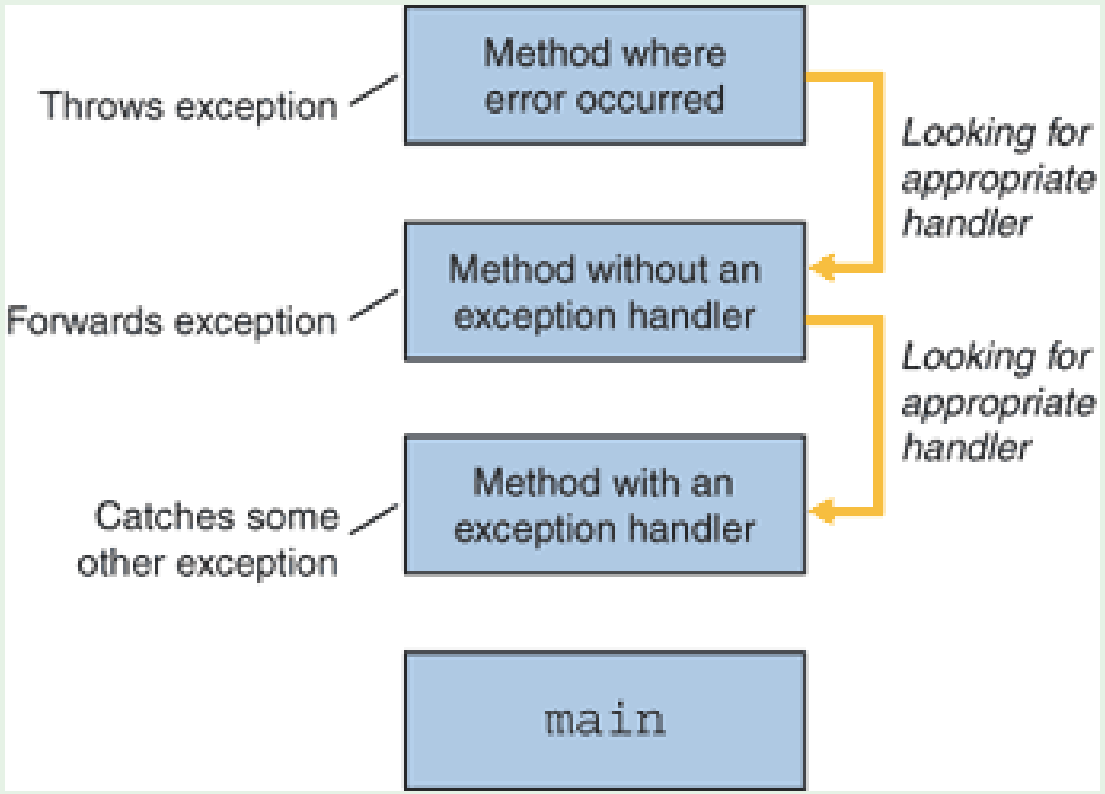
Gestion d'une exception

Recherche dans la call stack

- Le système d'exécution va donc chercher dans la call stack une méthode contenant un bout de code capable de gérer l'exception levée.
- Ce bout de code est appelé **exception handler**.
- La recherche commence par la méthode où l'erreur est apparue.
- Cette recherche continue dans l'ordre inverse des méthodes contenues dans la call stack jusqu'à trouver un exception handler approprié.
- Un exception handler est approprié s'il est **capable de gérer le type d'exception renvoyé**.
- Si un exception handler approprié est trouvé, on dit qu'il attrape l'exception : **catch the exception**. Sinon, le programme s'arrête en affichant les informations associées à l'exception : type et stack trace (enchaînement des appels de méthode).

Gestion d'une exception

Recherche d'un exception handler dans la call stack



Le "Catch or specify requirement"

Du code Java valide doit **nécessairement** inclure la gestion des exceptions et respecter ainsi le **Catch or specify requirement** :

Tout code susceptible de renvoyer une exception doit être contenu soit :

- dans une instruction **try/catch** qui définit ainsi le handler approprié.
- dans une méthode spécifiant explicitement qu'elle peut renvoyer une exception : il faut inclure le mot clé **throws**, suivi du type d'exception renvoyée, dans la signature de la méthode.
  Exemple : public void setAge(int age) **throws NegativeAgeException**

DU CODE NE RESPECTANT PAS CETTE RÈGLE NE COMPILE PAS !

Exemples

Le constructeur de la classe java.io.PrintWriter est susceptible, par définition (cf. api), de renvoyer une exception du type **java.io.IOException**. C'est pourquoi le code suivant **ne peut pas être compilé**. Il ne respecte pas le catch or specify requirement !

Exemple de code erroné

```
package prog;

import java.io.FileWriter; import java.io.PrintWriter;

public class TestingExceptions {

  public void writeFileTesting(){
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    out.print("test");
  }
}
```

Solution 1 : gestion active avec try/catch

Solution 1 : try/catch

```
package prog;

import java.io.FileWriter; import java.io.PrintWriter;
import java.io.IOException; // Il faut importer la classe de l'exception

public class TestingExceptions {

  public void writeFileTesting(){
    PrintWriter out = null;
    try {
      out = new PrintWriter(new FileWriter("OutFile.txt"));
    } catch (IOException e) {
      System.out.println("erreur ouverture fichier");
      e.printStackTrace(); //affichage de la call stack pour debugger
    }
    if (out != null) { // il est preferable de verifier
      out.print("test");
    }
  }
}
```

Solution 2 : gestion passive avec throws

Solution 2 : throws dans la signature

```
package prog;

import java.io.FileWriter; import java.io.PrintWriter;
import java.io.IOException;

public class TestingExceptions {

  public void writeFileTesting() throws IOException{
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    out.print("test");
  }
```

Remarque

- Préférez la solution 1. Il est toujours plus sûr de gérer soi-même les exceptions. Ici la gestion est déléguée à la méthode appelante.

Nature des exceptions en Java

Hiérachie des exceptions

- Les exceptions sont des objets comme les autres en Java (leur gestion mise à part)
- Il est donc possible de créer ses propres exceptions et de les lancer (throw) lors de l'exécution d'un programme.
- une exception est un objet qui hérite de la classe **java.lang.Throwable**
  ▸ java.lang.Throwable
- Il existe deux sous-classes prédéfinies dans le JDK : **java.lang.Error** et **java.lang.Exception**
- les exceptions "classiques" sont des sous-classes de java.lang.Exception
- les exceptions sous-classes de java.lang.Error stoppent un programme lorsqu'elles sont lancées. (on ne s'en sert que très rarement).

Créer ses propres exceptions

Création d'une nouvelle exception

- Pour créer sa propre classe d'exception, il suffit donc de créer une classe qui hérite de **java.lang.Exception**
- La plupart du temps, on crée une nouvelle exception pour
  1. gérer, par programmation (et non par message d'erreur), les utilisations incorrectes d'une méthode.
  2. et surtout *prévenir* l'utilisateur de la méthode que certaines utilisations sont incorrectes et qu'elles doivent être gérées explicitement.

Créer ses propres exceptions

Soit la classe suivante :

Personne.java

```
package prog;

public class Personne {

  private int age; //must be >= 0

  public Personne(int age) {
    this.age = age;
  }

  public void setAge(int age) {
    this.age = age;
  }
}
```

Création d'une nouvelle classe d'exception

NegativeAgeException.java

```
package prog;

public class NegativeAgeException extends Exception {
}
```

Lancer une exception

Pour lancer une exception lors de l'exécution il faut utiliser le mot clé **throw** avec l'exception créée pour la circonstance :

Personne.java

```
  . . .

public void setAge(int age) throws NegativeAgeException {
  if(age < 0){
    throw(new NegativeAgeException()); //sort de la methode
  }
  this.age = age; //execute seulement si age >= 0
}

  . . .
```

Nouvelle utilisation de la méthode

L'utilisation de la méthode est maintenant sécurisée : elle implique le respect du *catch or specify requirement* :

TestPersonne.java

```
package prog;

public class TestPersonne {

  public static void main(String[] args) {

    Personne pers = new Personne(10);

    try {
      pers.setAge(-1);
    } catch (NegativeAgeException e) {
      e.printStackTrace();
    }
  }
}
```

Exécution

TestPersonne.java

```
package prog;

public class TestPersonne {

  public static void main(String[] args) {
    Personne pers = new Personne(10);
    try {
      pers.setAge(-1);
    } catch (NegativeAgeException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat : (l'âge n'a pas été modifié !)

```
prog.NegativeAgeException
   at prog.Personne.setAge(Personne.java:13)
   at prog.TestPersonne.main(TestPersonne.java:8)
```

La classe java.lang.Exception en détail

Il est possible de personnaliser très simplement une exception grâce aux méthodes qui existent déjà dans la classe *java.lang.Exception* :

▸ java.lang.Exception

java.lang.Exception

Constructor Summary

Exception() Constructs a new exception with `null` as its detail message.
Exception(String message) Constructs a new exception with the specified detail message.
Exception(String message, Throwable cause) Constructs a new exception with the specified detail message and cause.
Exception(Throwable cause) Constructs a new exception with the specified cause and a detail message of `(cause==null ? null : cause.toString())` (which typically contains the class and detail message of `cause`).

Personnalisation du message d'erreur

La classe java.lang.Exception possède un constructeur qui permet d'initialiser le message qui sera obtenu :

NegativeAgeException.java

```
package prog;

public class NegativeAgeException extends Exception {

  public NegativeAgeException(String message) {
    super(message);
  }
}
```

Création d'un message d'erreur personnalisé

Nouvelle version de Personne.java

```
. . .
  public void setAge(int age) throws NegativeAgeException {
    if(age < 0){
      throw(new NegativeAgeException("erreur :
        l'age doit etre > 0, valeur entree : "+age));
    }
    this.age = age;
  }
. . .
```

Exécution

TestPersonne.java

```
package prog;
public class TestPersonne {

  public static void main(String[] args) {
    Personne pers = new Personne(10);
    try {
      pers.setAge(-1);
    } catch (NegativeAgeException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
prog.NegativeAgeException: erreur : l'age doit etre > 0,
valeur entree : -1
  at prog.Personne.setAge(Personne.java:13)
  at prog.TestPersonne.main(TestPersonne.java:8)
```

Instruction try/catch/finally

Le bloc try

```
try {
    code
}
catch and finally blocks . . .
```

Instruction try/catch/finally

Les blocs catch

```
try {

} catch (ExceptionType name) {

} catch (AnotherExceptionType name) {

}
```

Exemple

```
try {
    . . .
} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: "
                       + e.getMessage());

} catch (IOException e) {
    System.err.println("Caught IOException: "
                       + e.getMessage());
}
```

Instruction try/catch/finally

Le block *finally* est un block optionnel contenant des instructions qui seront toujours exécutées quelque soit le résultat du try :

Exemple pour une ouverture de fichier

```
try {
    code
}
catch {
   code
}
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

Instruction try/catch/finally

Il permet donc d'éviter d'écrire plusieurs fois le même code.
Notamment, il permet de s'assurer que les ressources éventuellement utilisées sont toujours libérées :

Exemple sans finally (code redondant)

```
try {
   ...
     out.close();        //Don't do this; it duplicates code.
} catch (FileNotFoundException e) {
     out.close();        //Don't do this; it duplicates code.
     System.err.println("Caught: FileNotFoundException: "
                        + e.getMessage());
} catch (IOException e) {
     out.close();        //Don't do this; it duplicates code.
     System.err.println("Caught "+e.getMessage());
}
```

Java 7 : le *try-with-resources*

On peut maintenant utiliser la syntaxe *try-with-resources* avec les objets qui implémentent l'interface `java.lang.AutoCloseable` :

Le bloc *try-with-resources*

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

Remarque

- *java.io.BufferedReader* implémente *java.lang.AutoCloseable*
- On voit ici que le bloc catch est optionnel

Java 7 : un seul bloc catch / plusieurs exceptions

Intérêt

- Permet de limiter la redondance dans le code
- Réduit la *tentation* d'utiliser `java.lang.Exception`

un seul bloc catch peut gérer plusieurs types d'exceptions

```
try {
. . .
}catch (IOException | SQLException ex) {
    ex.printStackTrace();
}
```

Classes d'exceptions particulières

La documentation de la classe *java.lang.Integer* indique que la méthode *parseInt* peut renvoyer l'exception **java.lang.NumberFormatException** (cf. api).

Pourtant le code suivant, qui semble ne pas respecter le **catch or specify requirement**, compile sans problème :

Testing.java

```
package prog;
public class Testing { //pas de throws

  public static void main(String[] args) {
    Integer.parseInt("10"); //pas de try/catch
  }
}
```

Classes d'exceptions particulières

En fait, il existe trois types d'exceptions différents :

1. Les exceptions classiques

- Les exceptions "classiques" (checked exceptions) qui **nécessitent de respecter le try or specify requirement**
- I.e **Toutes les exceptions** sauf les exceptions de type *java.lang.Error*, *java.lang.RuntimeException* et leurs sous-classes.

Classes d'exceptions particulières

2. Les exceptions de type java.lang.Error

- Elles se produisent dans des conditions exceptionnelles qui ne sont pas directement liées à l'application (ex : *java.lang.OutOfMemoryError* , *java.lang.VirtualMachineError*)
- Elles ne sont généralement pas prises en comptent dans le code : il est normal que le programme quitte dans ces cas là.

Classes d'exceptions particulières

3. Les exceptions de type `java.lang.RuntimeException`

- Elles se produisent dans des conditions exceptionnelles qui sont liées à l'application
- ex : mauvaise utilisation d'une API, bug de programmation, etc.
- ex : **java.lang.NumberFormatException**, **java.lang.ArrayIndexOutOfBoundsException**

Remarque

- **`NegativeAgeException`** aurait du être définie comme une sous classe de **`java.lang.RuntimeException`** !

Avantages liés aux exceptions

Le principal avantage lié à l'utilisation du mécanisme d'exception est de permettre une séparation explicite entre la logique du programme et la gestion d'évenements qui sortent de cette logique.

Soit le programme suivant :

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Toutes ces instructions sont potentiellement source de problème ; il faut pouvoir les gérer.

Programmation sans gestion d'exceptions

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Avantages liés aux exceptions

Programmation avec gestion des exceptions

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```