Programmation objet en Java

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://spiralconnect.univ-lyon1.fr` module "S2 - M2103 - Java et POO"
48H prévues $\approx$ 39H de cours-TDs + 6H de TPs, 1H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Plan

Introduction : anomalie dans le déroulement attendu

Lors de l'exécution d'un programme, il peut y arriver des circonstances exceptionnelles telles que :

- un **utilisateur ne saisisse pas des données valides** (e.g. un caractère à la place d'un nombre)
- **un argument effectif d'une fonction soit incorrect** : il ne vérifie pas les *préconditions* demandées par le concepteur de la fonction : par exemple on attend un argument positif, mais l'argument effectif est négatif, ou on attend un argument non nul et zéro est donné comme argument effectif...

Introduction : anomalie dans le déroulement attendu

Lors de l'exécution d'un programme, il peut y arriver des circonstances exceptionnelles telles que :

- un **utilisateur ne saisisse pas des données valides** (e.g. un caractère à la place d'un nombre)
- **un argument effectif d'une fonction soit incorrect** :
  il ne vérifie pas les *préconditions* demandées par le concepteur de la fonction :
  par exemple on attend un argument positif, mais l'argument effectif est négatif, ou on attend un argument non nul et zéro est donné comme argument effectif...

Introduction : anomalie dans le déroulement attendu

Lors de l'exécution d'un programme, il peut y arriver des circonstances exceptionnelles telles que :

- un **utilisateur ne saisisse pas des données valides** (e.g. un caractère à la place d'un nombre)
- **un argument effectif d'une fonction soit incorrect** :
  il ne vérifie pas les *préconditions* demandées par le concepteur de la fonction :
  par exemple on attend un argument positif, mais l'argument effectif est négatif, ou on attend un argument non nul et zéro est donné comme argument effectif...

Introduction : anomalie dans le déroulement attendu

Lors de l'exécution d'un programme, il peut y arriver des circonstances exceptionnelles telles que :

- un **utilisateur ne saisisse pas des données valides** (e.g. un caractère à la place d'un nombre)
- **un argument effectif d'une fonction soit incorrect** :
  il ne vérifie pas les *préconditions* demandées par le concepteur de la fonction :
  par exemple on attend un argument positif, mais l'argument effectif est négatif, ou on attend un argument non nul et zéro est donné comme argument effectif...

Introduction : anomalie dans le déroulement attendu

Lors de l'exécution d'un programme, il peut y arriver des circonstances exceptionnelles telles que :

- **il manque des informations dans un fichier texte que l'on lit** :
  on a atteint la fin du fichier, mais des informations sont manquantes et on ne peut pas continuer sans ces informations.
- etc.

Introduction : anomalie dans le déroulement attendu

Lors de l'exécution d'un programme, il peut y arriver des circonstances exceptionnelles telles que :

- **il manque des informations dans un fichier texte que l'on lit** :
  on a atteint la fin du fichier, mais des informations sont manquantes et on ne peut pas continuer sans ces informations.
- etc.

Quels sont les besoins des concepteurs de logiciels pour la gestion des anomalies ?

Une anomalie dans le déroulement normal :

- **doit être détectable dans toute méthode**, même dans les méthodes sans valeur de retour (constructeurs).
- **doit pouvoir être traitée soit dans la méthode dans laquelle l'anomalie a été détectée, soit dans une des méthodes appelantes**.

Cette gestion "souple" des anomalies permet d'obtenir des codes plus lisibles et faciles à maintenir, et surtout d'éviter que certaines anomalies possibles soient tout simplement oubliées.

Quels sont les besoins des concepteurs de logiciels pour la gestion des anomalies ?

Une anomalie dans le déroulement normal :

- **doit être détectable dans toute méthode**, même dans les méthodes sans valeur de retour (constructeurs).
- **doit pouvoir être traitée soit dans la méthode dans laquelle l'anomalie a été détectée, soit dans une des méthodes appelantes**.

Cette gestion "souple" des anomalies permet d'obtenir des codes plus lisibles et faciles à maintenir, et surtout d'éviter que certaines anomalies possibles soient tout simplement oubliées.

Quels sont les besoins des concepteurs de logiciels pour la gestion des anomalies ?

Une anomalie dans le déroulement normal :

- **doit être détectable dans toute méthode**, même dans les méthodes sans valeur de retour (constructeurs).
- **doit pouvoir être traitée soit dans la méthode dans laquelle l'anomalie a été détectée, soit dans une des méthodes appelantes**.

  Cette gestion "souple" des anomalies permet d'obtenir des codes plus lisibles et faciles à maintenir, et surtout d'éviter que certaines anomalies possibles soient tout simplement oubliées.

Java dispose d'un mécanisme très souple de gestion d'exception

Definition de la gestion d'exception

Mécanisme qui permet le traitement uniforme et structuré des anomalies survenant lors de l'exécution d'un programme.

La détection d'une anomalie y est dissociée de son traitement.

Le code pour la gestion des anomalies est séparé du code "normal" (donc lisibilité accrue).

Java impose que toutes les exceptions soient des objets d'une classe dérivée de la classe Exception (dans java.lang).

Java dispose d'un mécanisme très souple de gestion d'exception

Definition de la gestion d'exception

Mécanisme qui permet le traitement uniforme et structuré des anomalies survenant lors de l'exécution d'un programme.
La détection d'une anomalie y est dissociée de son traitement.
Le code pour la gestion des anomalies est séparé du code "normal" (donc lisibilité accrue).

Java impose que toutes les exceptions soient des objets d'une classe dérivée de la classe Exception (dans java.lang).

Java dispose d'un mécanisme très souple de gestion d'exception

Definition de la gestion d'exception

Mécanisme qui permet le traitement uniforme et structuré des anomalies survenant lors de l'exécution d'un programme.
La détection d'une anomalie y est dissociée de son traitement.
Le code pour la gestion des anomalies est séparé du code "normal" (donc lisibilité accrue).

Java impose que toutes les exceptions soient des objets d'une classe dérivée de la classe Exception (dans java.lang).

Java dispose d'un mécanisme très souple de gestion d'exception

Definition de la gestion d'exception

Mécanisme qui permet le traitement uniforme et structuré des anomalies survenant lors de l'exécution d'un programme.
La détection d'une anomalie y est dissociée de son traitement.
Le code pour la gestion des anomalies est séparé du code "normal" (donc lisibilité accrue).

Java impose que toutes les exceptions soient des objets d'une classe dérivée de la classe Exception (dans java.lang).

Traitement des exceptions avec try-catch

Principe

- Certaines méthodes lancent une exception lorsqu'une erreur est détectée via une instruction throw.
- Un bloc try{...} entoure le code pour lequel on souhaite détecter les exceptions lancées. On peut placer ce bloc autour de l'appel d'une méthode au sein d'une méthode appelante.
- A la suite de ce bloc try {...}, on place 1 ou plusieurs blocs catch(TypeErreur e){...} : un bloc catch est appelé un **gestionnaire d'exception** et ne gère qu'un seul type d'exception spécifié en argument.
- **Optionnel** : à la suite des blocs catch, un bloc finally {...}, toujours exécuté (même si pas d'erreur), peut être placé.

Traitement des exceptions avec try-catch

Principe

- Certaines méthodes lancent une exception lorsqu'une erreur est détectée via une instruction throw.
- Un bloc try{...} entoure le code pour lequel on souhaite détecter les exceptions lancées. On peut placer ce bloc autour de l'appel d'une méthode au sein d'une méthode appelante.
- A la suite de ce bloc try {...}, on place 1 ou plusieurs blocs catch(TypeErreur e){...} : un bloc catch est appelé un **gestionnaire d'exception** et ne gère qu'un seul type d'exception spécifié en argument.
- **Optionnel** : à la suite des blocs catch, un bloc finally {...}, toujours exécuté (même si pas d'erreur), peut être placé.

Traitement des exceptions avec try-catch

Principe

- Certaines méthodes lancent une exception lorsqu'une erreur est détectée via une instruction throw.
- Un bloc try{...} entoure le code pour lequel on souhaite détecter les exceptions lancées. On peut placer ce bloc autour de l'appel d'une méthode au sein d'une méthode appelante.
- A la suite de ce bloc try {...}, on place 1 ou plusieurs blocs catch(TypeErreur e){...} : un bloc catch est appelé un **gestionnaire d'exception** et ne gère qu'un seul type d'exception spécifié en argument.
- **Optionnel** : à la suite des blocs catch, un bloc finally {...}, toujours exécuté (même si pas d'erreur), peut être placé.

Traitement des exceptions avec try-catch

Principe

- Certaines méthodes lancent une exception lorsqu'une erreur est détectée via une instruction throw.
- Un bloc try{...} entoure le code pour lequel on souhaite détecter les exceptions lancées. On peut placer ce bloc autour de l'appel d'une méthode au sein d'une méthode appelante.
- A la suite de ce bloc try {...}, on place 1 ou plusieurs blocs catch(TypeErreur e){...} : un bloc catch est appelé un **gestionnaire d'exception** et ne gère qu'un seul type d'exception spécifié en argument.
- **Optionnel** : à la suite des blocs catch, un bloc finally {...}, toujours exécuté (même si pas d'erreur), peut être placé.

Traitement des exceptions avec try-catch

Principe

- Certaines méthodes lancent une exception lorsqu'une erreur est détectée via une instruction throw.
- Un bloc try{...} entoure le code pour lequel on souhaite détecter les exceptions lancées. On peut placer ce bloc autour de l'appel d'une méthode au sein d'une méthode appelante.
- A la suite de ce bloc try {...}, on place 1 ou plusieurs blocs catch(TypeErreur e){...} : un bloc catch est appelé un **gestionnaire d'exception** et ne gère qu'un seul type d'exception spécifié en argument.
- **Optionnel** : à la suite des blocs catch, un bloc finally {...}, toujours exécuté (même si pas d'erreur), peut être placé.

Traitement des exceptions avec try-catch

Principe

- Certaines méthodes lancent une exception lorsqu'une erreur est détectée via une instruction throw.
- Un bloc try{...} entoure le code pour lequel on souhaite détecter les exceptions lancées. On peut placer ce bloc autour de l'appel d'une méthode au sein d'une méthode appelante.
- A la suite de ce bloc try {...}, on place 1 ou plusieurs blocs catch(TypeErreur e){...} : un bloc catch est appelé un **gestionnaire d'exception** et ne gère qu'un seul type d'exception spécifié en argument.
- **Optionnel** : à la suite des blocs catch, un bloc finally {...}, toujours exécuté (même si pas d'erreur), peut être placé.

Traitement des exceptions avec try-catch

```
void methode() throws MonException // on spécifie le type d'exception jetable pour pouvoir
{                                  // traiter l'exception de ce type à l'extérieur
   // détection des anomalies dès le début de la méthode
   if(anomalie) throw new MonException(...) ; // le type de l'exception sera utilisé
                                              // plus tard pour identifier le type
   ... // exécution "normale"                 // d'exception attrapée
}
...// au sein d'une méthode (e.g. main)
try
{
   o.methode() ;
}
catch(MonException e)
{
   // traitement approprié de ce type d'exception
}
catch(AutreException e)
{
   // traitement approprié de ce type d'exception
}
... // Autres gestionnaires d'exception
finally
{
   // Toujours exécuté
}
...
```

Déclaration des exceptions

- Une exception lançable est soit
  - traitée direment dans la méthode qui la lance ;
  - sinon il faut obligatoirement spécifier son type (ou un type parent, i.e. Exception) à la suite du mot-clé throws.

Ainsi, **l'en-tête d'une méthode est complétée par la liste des types des exceptions pouvant "sortir"** :

```
public void methode() throws Type1, Type2, ..., TypeN { ... }
```

- Le fait de spécifier throws Exception suffit pour n'importe quel type et nombre d'exception. *Quelles conséquences ?* Code plus compact, mais un programmeur extérieur ne pourra pas connaître les types d'exception spécifiques sans "rentrer" dans le code.

Déclaration des exceptions

- Une exception lançable est soit
  - traitée direment dans la méthode qui la lance ;
  - sinon il faut obligatoirement spécifier son type (ou un type parent, i.e. Exception) à la suite du mot-clé throws.

Ainsi, **l'en-tête d'une méthode est complétée par la liste des types des exceptions pouvant "sortir"** :

```
public void methode() throws Type1, Type2, ..., TypeN { ... }
```

- Le fait de spécifier throws Exception suffit pour n'importe quel type et nombre d'exception. *Quelles conséquences ?* Code plus compact, mais un programmeur extérieur ne pourra pas connaître les types d'exception spécifiques sans "rentrer" dans le code.

Déclaration des exceptions

- Une exception lançable est soit
  - traitée direment dans la méthode qui la lance ;
  - sinon il faut obligatoirement spécifier son type (ou un type parent, i.e. Exception) à la suite du mot-clé throws.

  Ainsi, **l'en-tête d'une méthode est complétée par la liste des types des exceptions pouvant "sortir"** :
  public void methode() throws Type1, Type2, ..., TypeN { ... }

- Le fait de spécifier throws Exception suffit pour n'importe quel type et nombre d'exception. *Quelles conséquences ?* Code plus compact, mais un programmeur extérieur ne pourra pas connaître les types d'exception spécifiques sans "rentrer" dans le code.

Déclaration des exceptions

- Une exception lançable est soit
  - traitée direment dans la méthode qui la lance ;
  - sinon il faut obligatoirement spécifier son type (ou un type parent, i.e. Exception) à la suite du mot-clé throws.

  Ainsi, **l'en-tête d'une méthode est complétée par la liste des types des exceptions pouvant "sortir"** :
  public void methode() throws Type1, Type2, ..., TypeN { ... }

- Le fait de spécifier throws Exception suffit pour n'importe quel type et nombre d'exception. *Quelles conséquences ?* Code plus compact, mais un programmeur extérieur ne pourra pas connaître les types d'exception spécifiques sans "rentrer" dans le code.

Déclaration des exceptions

- Une exception lançable est soit
  - traitée direment dans la méthode qui la lance ;
  - sinon il faut obligatoirement spécifier son type (ou un type parent, i.e. Exception) à la suite du mot-clé throws.

  Ainsi, **l'en-tête d'une méthode est complétée par la liste des types des exceptions pouvant "sortir"** :
  public void methode() throws Type1, Type2, ..., TypeN { ... }

- Le fait de spécifier throws Exception suffit pour n'importe quel type et nombre d'exception. *Quelles conséquences ?*
  Code plus compact, mais un programmeur extérieur ne pourra pas connaître les types d'exception spécifiques sans "rentrer" dans le code.

Déclaration des exceptions

- Une exception lançable est soit
  - traitée direment dans la méthode qui la lance ;
  - sinon il faut obligatoirement spécifier son type (ou un type parent, i.e. Exception) à la suite du mot-clé throws.

  Ainsi, **l'en-tête d'une méthode est complétée par la liste des types des exceptions pouvant "sortir"** :
  public void methode() throws Type1, Type2, ..., TypeN { ... }

- Le fait de spécifier throws Exception suffit pour n'importe quel type et nombre d'exception. *Quelles conséquences ?* Code plus compact, mais un programmeur extérieur ne pourra pas connaître les types d'exception spécifiques sans "rentrer" dans le code.

Lancement des exceptions

- **Une exception** Type qui est une instance de la hiérachie Exception **se lance par** throw new Type(...).

Traitement d'une exception

Un gestionnaire d'exception est un bloc `catch` obligatoirement associé à un bloc `try` dans lequel on va détecter des exceptions.

C'est dans le bloc { ... } du gestionnaire d'exception que l'on traite l'exception en y mettant les instructions adéquates.
On affiche en général un message assez précis (e.g. *Tentative de division par zéro dans la méthode f*).

La dernière instruction d'un bloc catch peut être `System.exit(-1);` pour interrompre l'exécution du programme (l'anomalie détectée est telle qu'on ne peut plus rien faire). Ici le code de retour -1 est transmis à l'environnement.

Traitement d'une exception

Un gestionnaire d'exception est un bloc `catch` obligatoirement associé à un bloc `try` dans lequel on va détecter des exceptions.

C'est dans le bloc { ... } du gestionnaire d'exception que l'on traite l'exception en y mettant les instructions adéquates.
On affiche en général un message assez précis (e.g. *Tentative de division par zéro dans la méthode f*).

La dernière instruction d'un bloc catch peut être `System.exit(-1);` pour interrompre l'exécution du programme (l'anomalie détectée est telle qu'on ne peut plus rien faire). Ici le code de retour -1 est transmis à l'environnement.

Traitement d'une exception

Un gestionnaire d'exception est un bloc `catch` obligatoirement associé à un bloc `try` dans lequel on va détecter des exceptions.

C'est dans le bloc { ... } du gestionnaire d'exception que l'on traite l'exception en y mettant les instructions adéquates.
On affiche en général un message assez précis (e.g. *Tentative de division par zéro dans la méthode f*).

La dernière instruction d'un bloc catch peut être `System.exit(-1);` pour interrompre l'exécution du programme (l'anomalie détectée est telle qu'on ne peut plus rien faire). Ici le code de retour -1 est transmis à l'environnement.

Traitement d'une exception

Un gestionnaire d'exception est un bloc `catch` obligatoirement associé à un bloc `try` dans lequel on va détecter des exceptions.

C'est dans le bloc { ... } du gestionnaire d'exception que l'on traite l'exception en y mettant les instructions adéquates.
On affiche en général un message assez précis (e.g. *Tentative de division par zéro dans la méthode f*).

La dernière instruction d'un bloc catch peut être `System.exit(-1);` pour interrompre l'exécution du programme (l'anomalie détectée est telle qu'on ne peut plus rien faire). Ici le code de retour -1 est transmis à l'environnement.

Exemple avec plusieurs types d'exception

Allez chercher sur spiralconnect le fichier zip contenu dans le dossier Slides/Codes pour la gestion des exceptions : 1er_Exemple_exception.zip :

- Inverser l'ordre des 2 blocs catchs ? Quelles sont les conséquences ?
- A quoi peut bien servir le bloc finally ?
- Si les exceptions ne sont pas traitées dans le programme de test (dans la méthode `main`), quelles modifications dois-je apporter à la méthode `main` ?
- Expliquer le code des constructeurs des 3 classes d'exception.

Exemple avec plusieurs types d'exception

Allez chercher sur spiralconnect le fichier zip contenu dans le dossier Slides/Codes pour la gestion des exceptions : 1er_Exemple_exception.zip :

- Inverser l'ordre des 2 blocs catchs ? Quelles sont les conséquences ?
- A quoi peut bien servir le bloc finally ?
- Si les exceptions ne sont pas traitées dans le programme de test (dans la méthode main), quelles modifications dois-je apporter à la méthode main ?
- Expliquer le code des constructeurs des 3 classes d'exception.

Exemple avec plusieurs types d'exception

Allez chercher sur spiralconnect le fichier zip contenu dans le dossier Slides/Codes pour la gestion des exceptions : 1er_Exemple_exception.zip :

- Inverser l'ordre des 2 blocs catchs ? Quelles sont les conséquences ?
- A quoi peut bien servir le bloc finally ?
- Si les exceptions ne sont pas traitées dans le programme de test (dans la méthode main), quelles modifications dois-je apporter à la méthode main ?
- Expliquer le code des constructeurs des 3 classes d'exception.

Exemple avec plusieurs types d'exception

Allez chercher sur spiralconnect le fichier zip contenu dans le dossier Slides/Codes pour la gestion des exceptions : 1er_Exemple_exception.zip :

- Inverser l'ordre des 2 blocs catchs ? Quelles sont les conséquences ?
- A quoi peut bien servir le bloc finally ?
- Si les exceptions ne sont pas traitées dans le programme de test (dans la méthode main), quelles modifications dois-je apporter à la méthode main ?
- Expliquer le code des constructeurs des 3 classes d'exception.

Exemple avec plusieurs types d'exception

Allez chercher sur spiralconnect le fichier zip contenu dans le dossier Slides/Codes pour la gestion des exceptions : 1er_Exemple_exception.zip :

- Inverser l'ordre des 2 blocs catchs ? Quelles sont les conséquences ?
- A quoi peut bien servir le bloc finally ?
- Si les exceptions ne sont pas traitées dans le programme de test (dans la méthode main), quelles modifications dois-je apporter à la méthode main ?
- Expliquer le code des constructeurs des 3 classes d'exception.

Exemple avec plusieurs types d'exception

Questions :

- Dans un bloc try, si plusieurs instructions sont présentes, alors si une exception est déclenchée, est-ce que le programme revient sur l'instruction qui suit ?
- Est-ce que du code peut s'exécuter à la suite d'un bloc try suivi de ses gestionnaires d'exception catch, même en cas de traitement effectif d'une exception ?

Exemple avec plusieurs types d'exception

Questions :

- Dans un bloc try, si plusieurs instructions sont présentes, alors si une exception est déclenchée, est-ce que le programme revient sur l'instruction qui suit ?
- Est-ce que du code peut s'exécuter à la suite d'un bloc try suivi de ses gestionnaires d'exception catch, même en cas de traitement effectif d'une exception ?

Comment transmettre des informations propres à son erreur ?

Pour gérer les informations d'une anomalie, **on peut rajouter les attributs et méthodes nécessaires à sa classe d'exception**. Puis au moment du throw, on spécifie les arguments effectifs du constructeur de sa classe d'exception.

Reprendre l'exemple de spiralconnect et le compléter afin de récupérer les mauvaises coordonnées d'un point 2D positif et d'afficher ces mauvaise coordonnées dans le gestionnaire d'exception associé.

Comment transmettre des informations propres à son erreur ?

Pour gérer les informations d'une anomalie, **on peut rajouter les attributs et méthodes nécessaires à sa classe d'exception**. Puis au moment du throw, on spécifie les arguments effectifs du constructeur de sa classe d'exception.

Reprendre l'exemple de spiralconnect et le compléter afin de récupérer les mauvaises coordonnées d'un point 2D positif et d'afficher ces mauvaise coordonnées dans le gestionnaire d'exception associé.

Poursuite de l'exécution après le traitement d'une exception dans un bloc catch

Si un gestionnaire d'exception ne met pas fin au programme explicitement avec un `System.exit(-1);`, un `return` ou le lancement d'une autre exception, alors l'exécution du programme reprend à la suite du dernier bloc `catch` associé au bloc `try` dans lequel l'exception a été détectée.

Cheminement des exceptions

Si une exception est déclenchée/lancée au sein d'une méthode :

1. On cherche un gestionnaire d'exception associé à un éventuel bloc `try` :
   1. S'il n'y a pas de bloc `try` ou si aucun gestionnaire d'exception ne peut traiter l'exception déclenchée, l'exception est transmise à la méthode appelante (*après exécution du bloc* `finally`).
   2. Si un gestionnaire d'exception peut traiter l'exception déclenchée, alors il le fait. *Il peut même redéclencher une exception !*
2. Si aucun gestionnaire d'exception ne peut traiter l'exception lancée, ni dans la méthode ni dans une de ses méthodes appelantes, alors une erreur d'exécution se produit et le type d'exception est affiché.

Cheminement des exceptions

Si une exception est déclenchée/lancée au sein d'une méthode :

1. On cherche un gestionnaire d'exception associé à un éventuel bloc `try` :
   1. S'il n'y a pas de bloc `try` ou si aucun gestionnaire d'exception ne peut traiter l'exception déclenchée, l'exception est transmise à la méthode appelante (*après exécution du bloc* `finally`).
   2. Si un gestionnaire d'exception peut traiter l'exception déclenchée, alors il le fait. *Il peut même redéclencher une exception !*
2. Si aucun gestionnaire d'exception ne peut traiter l'exception lancée, ni dans la méthode ni dans une de ses méthodes appelantes, alors une erreur d'exécution se produit et le type d'exception est affiché.

Cheminement des exceptions

Si une exception est déclenchée/lancée au sein d'une méthode :

1. On cherche un gestionnaire d'exception associé à un éventuel bloc `try` :
   1. S'il n'y a pas de bloc `try` ou si aucun gestionnaire d'exception ne peut traiter l'exception déclenchée, l'exception est transmise à la méthode appelante (*après exécution du bloc* `finally`).
   2. Si un gestionnaire d'exception peut traiter l'exception déclenchée, alors il le fait. *Il peut même redéclencher une exception !*
2. Si aucun gestionnaire d'exception ne peut traiter l'exception lancée, ni dans la méthode ni dans une de ses méthodes appelantes, alors une erreur d'exécution se produit et le type d'exception est affiché.

Cheminement des exceptions

Si une exception est déclenchée/lancée au sein d'une méthode :

1. On cherche un gestionnaire d'exception associé à un éventuel bloc `try` :
   1. S'il n'y a pas de bloc `try` ou si aucun gestionnaire d'exception ne peut traiter l'exception déclenchée, l'exception est transmise à la méthode appelante (*après exécution du bloc* `finally`).
   2. Si un gestionnaire d'exception peut traiter l'exception déclenchée, alors il le fait. *Il peut même redéclencher une exception !*
2. Si aucun gestionnaire d'exception ne peut traiter l'exception lancée, ni dans la méthode ni dans une de ses méthodes appelantes, alors une erreur d'exécution se produit et le type d'exception est affiché.

Cheminement des exceptions

Si une exception est déclenchée/lancée au sein d'une méthode :

1. On cherche un gestionnaire d'exception associé à un éventuel bloc `try` :
   1. S'il n'y a pas de bloc `try` ou si aucun gestionnaire d'exception ne peut traiter l'exception déclenchée, l'exception est transmise à la méthode appelante (*après exécution du bloc* `finally`).
   2. Si un gestionnaire d'exception peut traiter l'exception déclenchée, alors il le fait. *Il peut même redéclencher une exception !*
2. Si aucun gestionnaire d'exception ne peut traiter l'exception lancée, ni dans la méthode ni dans une de ses méthodes appelantes, alors une erreur d'exécution se produit et le type d'exception est affiché.

Le bloc finally

Ce bloc est appelé si présent, en particulier dans le cas d'une exception détectée dans son bloc `try` associé et qui n'aurait pas trouvé de gestionnaire d'exception capable de la traiter.

Le bloc `finally` permet donc de garantir que certains traitements seront toujours effectués (*en fait uniquement si on n'est pas passé par le lancement d'une nouvelle exception, un* `System.exit` *ou un* `return` *au sein d'un gestionnaire d'exception*), en particulier la libération de certaines ressources (e.g. on doit rendre la main sur une ressource accédée en exclusion mutuelle, par exemple une connexion sur une base de données).

Le bloc finally

Ce bloc est appelé si présent, en particulier dans le cas d'une exception détectée dans son bloc `try` associé et qui n'aurait pas trouvé de gestionnaire d'exception capable de la traiter.

Le bloc `finally` permet donc de garantir que certains traitements seront toujours effectués (*en fait uniquement si on n'est pas passé par le lancement d'une nouvelle exception, un* `System.exit` *ou un* `return` *au sein d'un gestionnaire d'exception*), en particulier la libération de certaines ressources (e.g. on doit rendre la main sur une ressource accédée en exclusion mutuelle, par exemple une connexion sur une base de données).

Les exceptions implicites ou hors contrôle

Jusqu'à présent, nous avons vu uniquement les *exceptions explicites* ou *sous contrôle* : on doit soit les attraper et les traiter au sein de la méthode, soit on doit déclarer que cette méthode peut laisser sortir ce type d'exception et de même pour les méthodes appelantes.

Java les distingue des exceptions qui peuvent avoir lieu presque partout : les *exceptions implicites* ou *hors contrôle* : afin que la programmation ne soit pas fastidieuse, les types de ces exceptions ne doivent pas être déclarés explicitement dans le throws, ils le sont déjà implicitement.

Les exceptions implicites ou hors contrôle

Jusqu'à présent, nous avons vu uniquement les *exceptions explicites* ou *sous contrôle* : on doit soit les attraper et les traiter au sein de la méthode, soit on doit déclarer que cette méthode peut laisser sortir ce type d'exception et de même pour les méthodes appelantes.

Java les distingue des exceptions qui peuvent avoir lieu presque partout : les *exceptions implicites* ou *hors contrôle* : afin que la programmation ne soit pas fastidieuse, les types de ces exceptions ne doivent pas être déclarés explicitement dans le throws, ils le sont déjà implicitement.

Les exceptions implicites ou hors contrôle

Jusqu'à présent, nous avons vu uniquement les *exceptions explicites* ou *sous contrôle* : on doit soit les attraper et les traiter au sein de la méthode, soit on doit déclarer que cette méthode peut laisser sortir ce type d'exception et de même pour les méthodes appelantes.

Java les distingue des exceptions qui peuvent avoir lieu presque partout : les *exceptions implicites* ou *hors contrôle* : afin que la programmation ne soit pas fastidieuse, les types de ces exceptions ne doivent pas être déclarés explicitement dans le throws, ils le sont déjà implicitement.

Les exceptions implicites ou hors contrôle

Jusqu'à présent, nous avons vu uniquement les *exceptions explicites* ou *sous contrôle* : on doit soit les attraper et les traiter au sein de la méthode, soit on doit déclarer que cette méthode peut laisser sortir ce type d'exception et de même pour les méthodes appelantes.

Java les distingue des exceptions qui peuvent avoir lieu presque partout : les *exceptions implicites* ou *hors contrôle* : afin que la programmation ne soit pas fastidieuse, les types de ces exceptions ne doivent pas être déclarés explicitement dans le throws, ils le sont déjà implicitement.

Les exceptions implicites ou hors contrôle

Les *exceptions implicites* ou *hors contrôle* sont dérivées de la classe Exception (on les distingue des Error telles que le manque de mémoire qui entrainent un arrêt brutal du programme).

Quelques exemples d'exceptions implicites :

- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- ArithmeticException (e.g. division par zéro)
- NullPointerException (utilisation d'une référence null)
- etc. (`http://www.tutorialspoint.com/java/java_exceptions.htm`)

Les exceptions implicites ou hors contrôle

Les *exceptions implicites* ou *hors contrôle* sont dérivées de la classe Exception (on les distingue des Error telles que le manque de mémoire qui entrainent un arrêt brutal du programme).

Quelques exemples d'exceptions implicites :

- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- ArithmeticException (e.g. division par zéro)
- NullPointerException (utilisation d'une référence null)
- etc. (`http://www.tutorialspoint.com/java/java_exceptions.htm`)

Les exceptions implicites ou hors contrôle

Les *exceptions implicites* ou *hors contrôle* sont dérivées de la classe Exception (on les distingue des Error telles que le manque de mémoire qui entrainent un arrêt brutal du programme).

Quelques exemples d'exceptions implicites :

- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- ArithmeticException (e.g. division par zéro)
- NullPointerException (utilisation d'une référence null)
- etc. (`http://www.tutorialspoint.com/java/java_exceptions.htm`)

Les exceptions implicites ou hors contrôle

Les *exceptions implicites* ou *hors contrôle* sont dérivées de la classe Exception (on les distingue des Error telles que le manque de mémoire qui entrainent un arrêt brutal du programme).

Quelques exemples d'exceptions implicites :

- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- ArithmeticException (e.g. division par zéro)
- NullPointerException (utilisation d'une référence null)
- etc. (`http://www.tutorialspoint.com/java/java_exceptions.htm`)

Les exceptions implicites ou hors contrôle

Les *exceptions implicites* ou *hors contrôle* sont dérivées de la classe Exception (on les distingue des Error telles que le manque de mémoire qui entrainent un arrêt brutal du programme).

Quelques exemples d'exceptions implicites :

- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- ArithmeticException (e.g. division par zéro)
- NullPointerException (utilisation d'une référence null)
- etc. (http://www.tutorialspoint.com/java/java_exceptions.htm)

Les exceptions implicites ou hors contrôle

Les *exceptions implicites* ou *hors contrôle* sont dérivées de la classe Exception (on les distingue des Error telles que le manque de mémoire qui entrainent un arrêt brutal du programme).

Quelques exemples d'exceptions implicites :

- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- ArithmeticException (e.g. division par zéro)
- NullPointerException (utilisation d'une référence null)
- etc. (`http://www.tutorialspoint.com/java/java_exceptions.htm`)

Sa classe d'exception doit dériver d'Exception

```
class MonException1 extends Exception /* pas public car dans le fichier d'une autre classe */
{
    public MonException1(String s)
    {
        super(s) ; // on utilise le constructeur avec une String de Exception
    }
}

class MonException2 extends Exception
{
    public MonException2()
    {
        super() ; // on utilise cette fois le constructeur sans argument de Exception
    }
}

class MonException3 extends Exception
{
    public MonException3()
    {
        super("Erreur") ;
    }
}
```

3 méthodes à connaître

- public String getMessage() : retourne une chaîne contenant le message décrivant l'anomalie ;
- public String toString() : retourne une chaîne contenant la concatenation du nom de la classe d'exception et du message fourni par getMessage() ;
- public void printStackTrace() : affiche le message de toString() suivi par suite des derniers appels de méthodes trouvés sur la pile d'exécution.

Vous l'aurez compris, appeler la méthode printStackTrace() fournira le message le plus intéressant.

3 méthodes à connaître

- public String getMessage() : retourne une chaîne contenant le message décrivant l'anomalie ;
- public String toString() : retourne une chaîne contenant la concatenation du nom de la classe d'exception et du message fourni par getMessage() ;
- public void printStackTrace() : affiche le message de toString() suivi par suite des derniers appels de méthodes trouvés sur la pile d'exécution.

Vous l'aurez compris, appeler la méthode printStackTrace() fournira le message le plus intéressant.

3 méthodes à connaître

- public String getMessage() : retourne une chaîne contenant le message décrivant l'anomalie ;
- public String toString() : retourne une chaîne contenant la concatenation du nom de la classe d'exception et du message fourni par getMessage() ;
- public void printStackTrace() : affiche le message de toString() suivi par suite des derniers appels de méthodes trouvés sur la pile d'exécution.

Vous l'aurez compris, appeler la méthode printStackTrace() fournira le message le plus intéressant.

3 méthodes à connaître

- public String getMessage() : retourne une chaîne contenant le message décrivant l'anomalie ;
- public String toString() : retourne une chaîne contenant la concatenation du nom de la classe d'exception et du message fourni par getMessage() ;
- public void printStackTrace() : affiche le message de toString() suivi par suite des derniers appels de méthodes trouvés sur la pile d'exécution.

Vous l'aurez compris, appeler la méthode printStackTrace() fournira le message le plus intéressant.

3 méthodes à connaître

- public String getMessage() : retourne une chaîne contenant le message décrivant l'anomalie ;
- public String toString() : retourne une chaîne contenant la concatenation du nom de la classe d'exception et du message fourni par getMessage() ;
- public void printStackTrace() : affiche le message de toString() suivi par suite des derniers appels de méthodes trouvés sur la pile d'exécution.

Vous l'aurez compris, appeler la méthode printStackTrace() fournira le message le plus intéressant.

3 méthodes à connaître

- public String getMessage() : retourne une chaîne contenant le message décrivant l'anomalie ;
- public String toString() : retourne une chaîne contenant la concatenation du nom de la classe d'exception et du message fourni par getMessage() ;
- public void printStackTrace() : affiche le message de toString() suivi par suite des derniers appels de méthodes trouvés sur la pile d'exécution.

Vous l'aurez compris, appeler la méthode printStackTrace() fournira le message le plus intéressant.

3 méthodes à connaître

- public String getMessage() : retourne une chaîne contenant le message décrivant l'anomalie ;
- public String toString() : retourne une chaîne contenant la concatenation du nom de la classe d'exception et du message fourni par getMessage() ;
- public void printStackTrace() : affiche le message de toString() suivi par suite des derniers appels de méthodes trouvés sur la pile d'exécution.

Vous l'aurez compris, appeler la méthode printStackTrace() fournira le message le plus intéressant.

Allez chercher sur spiralconnect le fichier zip contenu dans le dossier Slides/Codes pour la gestion des exceptions : 2nd_Exemple_exception.zip :

- Rajoutez-y les méthodes crediter et debiter avec la gestion des anomalies.
- Testez le fonctionnement de ces méthodes ainsi que la gestion des anomalies associées.

Allez chercher sur spiralconnect le fichier zip contenu dans le dossier Slides/Codes pour la gestion des exceptions : 2nd_Exemple_exception.zip :

- Rajoutez-y les méthodes crediter et debiter avec la gestion des anomalies.
- Testez le fonctionnement de ces méthodes ainsi que la gestion des anomalies associées.

Plan

Les tests unitaires de non-regression

Lorsqu'un développeur modifie le code d'un programme, il a besoin de s'assurer que ses dernières modifications ne vont pas remettre en question les fonctionnalités qui fonctionnaient jusqu'alors.

C'est là qu'interviennent les tests unitaires de non-régression.

Les tests unitaires de non-regression

Lorsqu'un développeur modifie le code d'un programme, il a besoin de s'assurer que ses dernières modifications ne vont pas remettre en question les fonctionnalités qui fonctionnaient jusqu'alors.

C'est là qu'interviennent les tests unitaires de non-régression.

Les tests unitaires de non-regression

JUnit est un framework opensource pour le développment et l'exécution des tests unitaires automatisables :

- Il fournit des *assertions* qui testent les résultats attendus.
- Il fournit des *applications pour exécuter les tests et afficher les résultats*.
- Toutes les méthodes d'une classe doivent être testées qu'elles soient `public` ou non.

Les tests unitaires de non-regression

JUnit est un framework opensource pour le développment et l'exécution des tests unitaires automatisables :

- Il fournit des *assertions* qui testent les résultats attendus.
- Il fournit des *applications pour exécuter les tests et afficher les résultats*.
- Toutes les méthodes d'une classe doivent être testées qu'elles soient public ou non.

Les tests unitaires de non-regression

JUnit est un framework opensource pour le développment et l'exécution des tests unitaires automatisables :

- Il fournit des *assertions* qui testent les résultats attendus.
- Il fournit des *applications pour exécuter les tests et afficher les résultats*.
- Toutes les méthodes d'une classe doivent être testées qu'elles soient `public` ou non.

JUnit sur NetBeans

Vollet de gauche : Clic droit sur un package ou sur un fichier, puis choisir Tools et Create Tests : Pour l'emplacement choisir Test Packages et pour le framework choisir JUnit. Enfin, décocher tout dans la partie Generated Code sauf le Default Method Bodies, et cliquer sur OK.

- Une classe de test est générée par classe MaClasse dans votre package avec pour nom : MaClasseTest ;
- Pour chaque méthode de votre classe à tester, une méthode de nom testNomMethodeDansMaClasse a été créée au sein de la classe MaClasseTest ;
- Vous n'avez qu'à réécrire le corps de chaque méthode de test et à exécuter les tests d'une classe en cliquant sur la classe en question (clic droit) et en choisissant Test File.

JUnit sur NetBeans

Vollet de gauche : Clic droit sur un package ou sur un fichier, puis choisir Tools et Create Tests : Pour l'emplacement choisir Test Packages et pour le framework choisir JUnit. Enfin, décocher tout dans la partie Generated Code sauf le Default Method Bodies, et cliquer sur OK.

- Une classe de test est générée par classe MaClasse dans votre package avec pour nom : MaClasseTest ;
- Pour chaque méthode de votre classe à tester, une méthode de nom testNomMethodeDansMaClasse a été créée au sein de la classe MaClasseTest ;
- Vous n'avez qu'à réécrire le corps de chaque méthode de test et à exécuter les tests d'une classe en cliquant sur la classe en question (clic droit) et en choisissant Test File.

JUnit sur NetBeans

Vollet de gauche : Clic droit sur un package ou sur un fichier, puis choisir Tools et Create Tests : Pour l'emplacement choisir Test Packages et pour le framework choisir JUnit. Enfin, décocher tout dans la partie Generated Code sauf le Default Method Bodies, et cliquer sur OK.

- Une classe de test est générée par classe MaClasse dans votre package avec pour nom : MaClasseTest ;
- Pour chaque méthode de votre classe à tester, une méthode de nom testNomMethodeDansMaClasse a été créée au sein de la classe MaClasseTest ;
- Vous n'avez qu'à réécrire le corps de chaque méthode de test et à exécuter les tests d'une classe en cliquant sur la classe en question (clic droit) et en choisissant Test File.

JUnit sur NetBeans

Vollet de gauche : Clic droit sur un package ou sur un fichier, puis choisir Tools et Create Tests : Pour l'emplacement choisir Test Packages et pour le framework choisir JUnit. Enfin, décocher tout dans la partie Generated Code sauf le Default Method Bodies, et cliquer sur OK.

- Une classe de test est générée par classe MaClasse dans votre package avec pour nom : MaClasseTest ;
- Pour chaque méthode de votre classe à tester, une méthode de nom testNomMethodeDansMaClasse a été créée au sein de la classe MaClasseTest ;
- Vous n'avez qu'à réécrire le corps de chaque méthode de test et à exécuter les tests d'une classe en cliquant sur la classe en question (clic droit) et en choisissant Test File.

JUnit sur NetBeans

Vollet de gauche : Clic droit sur un package ou sur un fichier, puis choisir Tools et Create Tests : Pour l'emplacement choisir Test Packages et pour le framework choisir JUnit. Enfin, décocher tout dans la partie Generated Code sauf le Default Method Bodies, et cliquer sur OK.

- Une classe de test est générée par classe MaClasse dans votre package avec pour nom : MaClasseTest ;
- Pour chaque méthode de votre classe à tester, une méthode de nom testNomMethodeDansMaClasse a été créée au sein de la classe MaClasseTest ;
- Vous n'avez qu'à réécrire le corps de chaque méthode de test et à exécuter les tests d'une classe en cliquant sur la classe en question (clic droit) et en choisissant Test File.

JUnit sur NetBeans

Vollet de gauche : Clic droit sur un package ou sur un fichier, puis choisir Tools et Create Tests : Pour l'emplacement choisir Test Packages et pour le framework choisir JUnit. Enfin, décocher tout dans la partie Generated Code sauf le Default Method Bodies, et cliquer sur OK.

- Une classe de test est générée par classe MaClasse dans votre package avec pour nom : MaClasseTest ;
- Pour chaque méthode de votre classe à tester, une méthode de nom testNomMethodeDansMaClasse a été créée au sein de la classe MaClasseTest ;
- Vous n'avez qu'à réécrire le corps de chaque méthode de test et à exécuter les tests d'une classe en cliquant sur la classe en question (clic droit) et en choisissant Test File.

Ce qui est important pour utiliser JUnit

```
import junit.framework.*                    // Pour JUnit 3
                                            // Pour JUnit 4:
                                            // import org.junit.Test;
                                            // import static org.junit.Assert.*;
public class CompteTest extends TestCase {// A partir de JUnit 4 ça n'est plus obligatoire
                                            // d'hériter de TestCase

    public CompteTest(String testName) {  // Constructeur qui initialise la partie propre
        super(testName);                    // à TestCase (uniquement si on hérite de TestCase)
    }                                       // car sinon un contructeur sans argument suffit

    // Méthode qui va tester la méthode affiche de la classe Compte
    public void testAffiche() { // A partir de JUnit 4 plus besoin d'utiliser le préfixe test
                                // car une annotation @Test est indiquée au-dessus de la méthode
        System.out.print("affiche... ");

        // C'est ici que vous écrivez les instructions de test

        // méthodes fournies par JUnit:
        //assertEquals(expectedResult, result);
        //assertTrue( true );
        //assertFalse( false );

        System.out.println("Done. ");
    }
}
```

- Reprenez votre classe Compte avec la gestion des exceptions et testez-la avec JUnit. Un exemple de test avec JUnit est disponible sur spiralconnect : *Exemple_TestJUnit4.zip*.