Exceptions

Maria-Virginia Aponte, François Barthélémy

NFA031

1 Introduction : Quand les programmes plantent

Imaginez la situation suivante : vous travaillez depuis une heure sur un document important, vous saisissez des données dans un programme, et soudain... tout s'arrête brutalement. Un message d'erreur incompréhensible s'affiche, le programme se ferme, et vous perdez tout votre travail. Cette expérience frustrante est malheureusement courante avec des programmes mal conçus.

En Java, de nombreuses situations peuvent provoquer un arrêt brutal du programme. Considérons ce programme simple qui demande à l'utilisateur de saisir des nombres pour les stocker dans un tableau :

```
package nfa031;
public class ProblemeSimple {
   public static void main(String[] args) {
      int[] nombres = new int[3];

      System.out.println("Saisissez 3 nombres :");
      for (int i = 0; i < 3; i++) {
         System.out.print("Nombre " + (i+1) + " : ");
         nombres[i] = Terminal.lireInt();
      }

      System.out.print("Quel nombre voulez-vous doubler ? (1-3) : ");
      int position = Terminal.lireInt() - 1; //Conversion 1-3 vers 0-2
      nombres[position] = nombres[position] * 2;

      System.out.println("Résultat : " + nombres[position]);
   }
}
```

Ce programme semble simple et fonctionne parfaitement quand tout se passe bien :

```
Saisissez 3 nombres :
Nombre 1 : 10
Nombre 2 : 20
Nombre 3 : 30
Quel nombre voulez-vous doubler ? (1-3) : 2
Résultat : 40
```

Mais que se passe-t-il si l'utilisateur commet une erreur de saisie ? Plusieurs situations problématiques peuvent survenir :

Problème 1 : Saisie non numérique

```
Saisissez 3 nombres :
Nombre 1 : abc
```

```
Exception in thread "main" TerminalException
    at Terminal.lireInt(Terminal.java:25)
    at ProblemeSimple.main(ProblemeSimple.java:8)
```

Problème 2 : Position invalide

```
Saisissez 3 nombres :
Nombre 1 : 10
Nombre 2 : 20
Nombre 3 : 30
Quel nombre voulez-vous doubler ? (1-3) : 5
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at ProblemeSimple.main(ProblemeSimple.java:13)
```

Dans les deux cas, le programme s'arrête brutalement sans aucune possibilité de récupération. L'utilisateur perd toutes les données saisies et doit recommencer depuis le début. C'est exactement le type de situation qu'un programme professionnel doit éviter à tout prix.

1.1 L'impact sur l'expérience utilisateur

Un programme qui plante à la moindre erreur de saisie présente plusieurs problèmes majeurs :
— **Perte de données** : tout le travail effectué avant l'erreur est perdu
— **Frustration de l'utilisateur** : personne n'aime voir un programme se fermer brutalement
— **Manque de professionnalisme** : un logiciel qui plante donne une mauvaise image
— **Perte de productivité** : l'utilisateur doit recommencer son travail

1.2 Vers une solution élégante

Pour créer des programmes robustes et professionnels, nous avons besoin d'un mécanisme qui permette de :
— **Détecter** les erreurs avant qu'elles ne fassent planter le programme
— **Informer** l'utilisateur du problème de manière compréhensible
— **Récupérer** de l'erreur en donnant une seconde chance à l'utilisateur
— **Continuer** l'exécution normale du programme après résolution du problème

Java propose un mécanisme puissant pour atteindre ces objectifs : les **exceptions**. Les exceptions permettent de transformer un arrêt brutal en une gestion élégante et contrôlée des situations problématiques.

2 Le mécanisme des exceptions

2.1 Concept de base

Une exception en Java est un événement qui interrompt le cours normal d'exécution d'un programme lorsqu'une erreur ou une situation exceptionnelle se produit. Plutôt que de laisser le programme planter, le mécanisme d'exception permet de "capturer" cette situation et de la traiter de manière appropriée.

Le principe fondamental est simple : quand une méthode rencontre une situation qu'elle ne peut pas gérer normalement, elle "lance" une exception. Cette exception remonte dans la pile d'appels jusqu'à ce qu'elle soit "attrapée" par du code capable de la traiter.

2.2 Les trois temps des exceptions

Le mécanisme d'exception se déroule en trois phases distinctes :

1. **Création de l'exception** : un objet exception est créé pour décrire le problème

2. **Lancement de l'exception** : l'exception est "lancée" pour interrompre l'exécution normale
3. **Capture de l'exception** : l'exception est "attrapée" et traitée par du code approprié

2.3 Terminologie

Java utilise une terminologie spécifique pour décrire ces opérations :
— **Lancer une exception** (en anglais : *throw*) : déclencher une exception quand un problème est détecté
— **Attraper une exception** (en anglais : *catch*) : capturer et traiter une exception

2.4 Structure try/catch de base

La structure fondamentale pour gérer les exceptions en Java utilise les mots-clés `try` et `catch` :

```
try {
   // Code susceptible de lever une exception
   instruction1;
   instruction2;
   // ...
} catch (TypeException e) {
   // Code pour traiter l'exception
   // Ce code s'exécute si TypeException est lancée
}
```

2.5 Fonctionnement détaillé

Voici comment cette structure fonctionne :
Exécution normale :
— Les instructions dans le bloc `try` s'exécutent séquentiellement
— Si aucune exception n'est lancée, le bloc `catch` est ignoré
— L'exécution continue après la structure `try/catch`
Quand une exception se produit :
— L'instruction qui lance l'exception s'arrête immédiatement
— Les instructions suivantes dans le bloc `try` ne sont pas exécutées
— Le contrôle passe directement au bloc `catch` correspondant
— Après traitement de l'exception, l'exécution continue après la structure `try/catch`

2.6 Premier exemple pratique

Reprenons notre problème de saisie et appliquons le mécanisme d'exception pour gérer la saisie incorrecte :

```
public class PremierEssai {
   public static void main(String[] args) {
      try {
         System.out.print("Entrez un nombre : ");
         int nombre = Terminal.lireInt();
         System.out.println("Vous avez saisi : " + nombre);
         System.out.println("Le double est : " + (nombre * 2));
      } catch (TerminalException e) {
         System.out.println("Erreur : vous devez saisir un "
                        + "nombre entier !");
         System.out.println("Le programme se termine.");
```

```
        }
    }
}
```

Testons ce programme avec différentes saisies :
Saisie correcte :

```
Entrez un nombre : 42
Vous avez saisi : 42
Le double est : 84
```

Saisie incorrecte :

```
Entrez un nombre : abc
Erreur : vous devez saisir un nombre entier !
Le programme se termine.
```

Le programme ne plante plus ! Au lieu d'un arrêt brutal avec un message technique incompréhensible, nous obtenons un message d'erreur clair et le programme se termine proprement.

2.7 Amélioration : plusieurs types d'exceptions

Un même bloc `try` peut gérer plusieurs types d'exceptions différentes en utilisant plusieurs blocs `catch` :

```
try {
    // Code susceptible de lever différentes exceptions
    int[] tableau = new int[taille]; // NegativeArraySizeException
    int nombre = Terminal.lireInt(); // TerminalException
    tableau[indice] = nombre; // ArrayIndexOutOfBoundsException
} catch (TerminalException e) {
    System.out.println("Erreur de saisie : nombre attendu");
} catch (NegativeArraySizeException e) {
    System.out.println("Erreur : la taille doit être positive");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Erreur : indice invalide");
}
```

Cette approche nous donne un contrôle précis sur le traitement de chaque type d'erreur, permettant d'afficher des messages spécifiques et adaptés à chaque situation.

3 Gestion des exceptions prédéfinies

Java fournit de nombreuses exceptions prédéfinies qui correspondent aux erreurs les plus courantes en programmation. Apprendre à les gérer correctement est essentiel pour créer des programmes robustes.

3.1 Les exceptions les plus courantes

Voici les exceptions que vous rencontrerez le plus fréquemment :
— **ArrayIndexOutOfBoundsException** : accès à un indice invalide dans un tableau
— **NegativeArraySizeException** : tentative de création d'un tableau avec une taille négative
— **TerminalException** : erreur de saisie avec les méthodes de Terminal
— **ArithmeticException** : erreur arithmétique (division par zéro avec des entiers)
— **NullPointerException** : tentative d'utilisation d'une référence null

3.2 Approche progressive

Reprenons notre exemple initial et améliorons-le étape par étape. Commençons par gérer une seule exception : **Étape 1 : Gestion basique de TerminalException**

```
public class Etape1 {
public static void main(String[] args) {
int[] nombres = new int[3];
   try {
      System.out.println("Saisissez 3 nombres :");
      for (int i = 0; i < 3; i++) {
         System.out.print("Nombre " + (i+1) + " : ");
         nombres[i] = Terminal.lireInt();
      }

      System.out.print("Quel nombre voulez-vous doubler ? (1-3) : ");
      int position = Terminal.lireInt() - 1;
      nombres[position] = nombres[position] * 2;

      System.out.println("Résultat : " + nombres[position]);
   } catch (TerminalException e) {
      System.out.println("Erreur de saisie, un nombre était attendu.");
      System.out.println("Le programme se termine.");
   }
}
}
```

Test de cette première version :

```
Saisissez 3 nombres :
Nombre 1 : 10
Nombre 2 : abc
Erreur de saisie : un nombre était attendu.
Le programme se termine.
```

Le programme ne plante plus, mais le comportement n'est pas optimal : toute erreur de saisie fait perdre le travail déjà effectué. **Étape 2 : Gestion de toutes les exceptions possibles**

```
public class Etape2 {
public static void main(String[] args) {
int[] nombres = new int[3];
   try {
      System.out.println("Saisissez 3 nombres :");
      for (int i = 0; i < 3; i++) {
         System.out.print("Nombre " + (i+1) + " : ");
         nombres[i] = Terminal.lireInt();
      }

      System.out.print("Quel nombre voulez-vous doubler ? (1-3) : ");
      int position = Terminal.lireInt() - 1;
      nombres[position] = nombres[position] * 2;

      System.out.println("Résultat : " + nombres[position]);

   } catch (TerminalException e) {
      System.out.println("Erreur, saisissez un nombre entier.");
   } catch (ArrayIndexOutOfBoundsException e) {
```

```
      System.out.println("Erreur, le numéro doit être entre 1 et 3.");
   }
}
}
```

Cette version gère les deux types d'erreurs les plus probables avec des messages spécifiques, mais présente encore le même problème : une erreur fait tout perdre.

3.3 Granularité du traitement

Pour éviter de perdre tout le travail en cas d'erreur ponctuelle, il faut diviser le code en plusieurs blocs `try/catch` indépendants. Cela permet de traiter chaque risque d'erreur de manière ciblée.

```
public class EtapeAmelioree {
public static void main(String[] args) {
int[] nombres = new int[3];
   // Saisie des nombres avec gestion d'erreur individuelle
   System.out.println("Saisissez 3 nombres :");
   for (int i = 0; i < 3; i++) {
      boolean saisieReussie = false;
      while (!saisieReussie) {
         try {
            System.out.print("Nombre " + (i+1) + " : ");
            nombres[i] = Terminal.lireInt();
            saisieReussie = true;
         } catch (TerminalException e) {
            System.out.println("Erreur : veuillez saisir "
                        + "un nombre entier.");
         }
      }
   }

   // Saisie de la position avec gestion d'erreur
   boolean positionValide = false;
   while (!positionValide) {
      try {
         System.out.print("Quel nombre voulez-vous doubler (1-3) ? ");
         int position = Terminal.lireInt() - 1;
         nombres[position] = nombres[position] * 2;
         System.out.println("Résultat : " + nombres[position]);
         positionValide = true;

      } catch (TerminalException e) {
         System.out.println("Erreur, entrez un nombre entier.");
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Erreur, le numéro doit être entre 1 et 3.");
      }
   }
}
}
```

Maintenant le programme offre une vraie seconde chance après chaque erreur :

```
Saisissez 3 nombres :
Nombre 1 : 10
Nombre 2 : abc
```

```
Erreur : veuillez saisir un nombre entier.
Nombre 2 : 20
Nombre 3 : 30
Quel nombre voulez-vous doubler ? (1-3) : 5
Erreur : le numéro doit être entre 1 et 3.
Quel nombre voulez-vous doubler ? (1-3) : 2
Résultat : 40
```

4 Techniques de gestion robuste

Le code de la section précédente fonctionne, mais il devient rapidement verbeux et répétitif. Cette section présente des techniques pour créer une gestion d'erreur plus élégante et réutilisable.

4.1 Factorisation des boucles de retry

La répétition du pattern "boucle + try/catch" peut être factorisée dans des méthodes spécialisées. Créons une méthode pour la saisie sécurisée d'entiers :

```
public class SaisieSecurisee {
/**
 * Lit un entier au clavier avec gestion d'erreur automatique
 */
public static int lireEntierSecurise(String message) {
   int resultat = 0;
   boolean saisieReussie = false;

   while (!saisieReussie) {
      try {
         System.out.print(message);
         resultat = Terminal.lireInt();
         saisieReussie = true;
      } catch (TerminalException e) {
         System.out.println("Erreur, entrez un nombre entier.");
      }
   }

   return resultat;
}

public static void main(String[] args) {
   int[] nombres = new int[3];

   // Saisie simplifiée grâce àla méthode factiorisée
   System.out.println("Saisissez 3 nombres :");
   for (int i = 0; i < 3; i++) {
      nombres[i] = lireEntierSecurise("Nombre " + (i+1) + " : ");
   }

   // Gestion de la position avec validation de l'intervalle
   boolean positionValide = false;
   while (!positionValide) {
      try {
         int numPosition = lireEntierSecurise("Quel nombre voulez-vous "
                                    + "doubler (1-3) ? ");
```

```
            int position = numPosition - 1;
            nombres[position] = nombres[position] * 2;
            System.out.println("Résultat : " + nombres[position]);
            positionValide = true;

        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Erreur, le numéro doit être entre 1 et 3.");
        }
    }
}
}
```

4.2 Validation avec gestion d'erreur intégrée

Pour aller plus loin, nous pouvons créer des méthodes qui combinent saisie et validation :

```
/**

Lit un entier dans un intervalle donné avec gestion d'erreur
*/
public static int lireEntierDansIntervalle(String message, int min,
                                    int max) {
int resultat = 0;
boolean saisieValide = false;
while (!saisieValide) {
try {
System.out.print(message);
resultat = Terminal.lireInt();
    if (resultat < min || resultat > max) {
       System.out.println("Erreur : le nombre doit être entre " +
                          min + " et " + max + ".");
    } else {
       saisieValide = true;
    }

 } catch (TerminalException e) {
    System.out.println("Erreur : veuillez saisir un nombre entier.");
 }
}
return resultat;
}
```

4.3 Programme final optimisé

En utilisant ces méthodes utilitaires, le programme principal devient beaucoup plus lisible :

```
public class ProgrammeOptimise {
public static int lireEntierSecurise(String message) {
   int resultat = 0;
   boolean saisieReussie = false;

   while (!saisieReussie) {
      try {
         System.out.print(message);
```

```
            resultat = Terminal.lireInt();
            saisieReussie = true;
        } catch (TerminalException e) {
            System.out.println("Erreur, entrez un nombre entier.");
        }
    }

    return resultat;
}

public static int lireEntierDansIntervalle(String message, int min,
                                    int max) {
    int resultat = 0;
    boolean saisieValide = false;

    while (!saisieValide) {
        try {
            System.out.print(message);
            resultat = Terminal.lireInt();

            if (resultat < min || resultat > max) {
                System.out.println("Erreur : le nombre doit être entre "
                            + min + " et " + max + ".");
            } else {
                saisieValide = true;
            }

        } catch (TerminalException e) {
            System.out.println("Erreur, entrez un nombre entier.");
        }
    }

    return resultat;
}

public static void main(String[] args) {
    int[] nombres = new int[3];

    // Saisie des nombres
    System.out.println("Saisissez 3 nombres :");
    for (int i = 0; i < 3; i++) {
        nombres[i] = lireEntierSecurise("Nombre " + (i+1) + " : ");
    }

    // Saisie de la position avec validation automatique
    int numPosition = lireEntierDansIntervalle("Quel nombre voulez-vous"
                                    + "doubler (1-3)?", 1, 3);
    int position = numPosition - 1;

    nombres[position] = nombres[position] * 2;
    System.out.println("Résultat : " + nombres[position]);
}
}
```

Le programme principal est maintenant très lisible et robuste. Toute la complexité de la gestion d'erreur est encapsulée dans les méthodes utilitaires réutilisables.

5 Créer ses propres exceptions

Jusqu'à présent, nous avons utilisé des exceptions prédéfinies par Java. Mais il est parfois nécessaire de créer ses propres exceptions pour gérer des erreurs spécifiques à notre application – on parle de domaine métier.

5.1 Quand créer une exception personnalisée

Une exception personnalisée est justifiée dans les situations suivantes :

— **Erreurs métier spécifiques** : validation de règles propres à votre application
— **Séparation des responsabilités** : détecter l'erreur dans une méthode, la traiter dans une autre
— **Clarté du code** : donner un nom explicite à un type d'erreur particulier

Il est important de noter qu'une exception ne se justifie que si la détection et le traitement de l'erreur se font dans des méthodes différentes. Si vous pouvez traiter l'erreur localement avec un simple `if`, c'est souvent préférable.

5.2 Syntaxe de création

Pour créer une exception personnalisée, il faut définir une classe qui hérite de `RuntimeException` :

```
package nfa031;
public class MonException extends RuntimeException {
  // Constructeur avec message (recommandé)
  public MonException(String message) {
    super(message);
  }
}
```

Il est fortement recommandé de définir un constructeur qui accepte un message d'erreur. Cela permet de donner des informations contextuelles précises sur l'erreur, ce qui facilite grandement le débogage et l'affichage d'erreurs informatives pour l'utilisateur.

La définition d'exception consiste à écrire une classe. Comme toute classe, celle-ci doit être écrite dans un fichier qui porte le même nom suivi de l'extension `.java`.

La clause `extends RuntimeException` ainsi que l'instruction `super` sont des éléments liés à l'héritage entre classes qui ne seront pas expliqués ici. Vous pouvez simplement faire un copier-coller de la classe donnée ici et changer le nom de la classe et du constructeur en fonction de vos besoins chaque fois que vous voulez créer un nouveau type d'exceptions.

5.3 Lancement d'une exception

Pour lancer une exception, on utilise l'instruction `throw` suivie de la création d'un objet exception avec `new` :

```
throw new MonException("Description précise du problème");
```

L'utilisation d'un message permet de donner des informations précises sur le contexte de l'erreur, ce qui facilite le débogage et améliore l'expérience utilisateur.

5.4 Exemple concret : gestion d'un compte bancaire

Considérons un exemple : la gestion d'un compte bancaire simple.

```
// Définition des exceptions métier
public class SoldeInsuffisantException extends RuntimeException {
  public SoldeInsuffisantException(String message) {
    super(message);
  }
}
public class MontantInvalideException extends RuntimeException {
  public MontantInvalideException(String message) {
    super(message);
  }
}
public class CompteBancaire {
 private double solde;
 private String titulaire;
 public CompteBancaire(String titulaire, double soldeInitial) {
  this.titulaire = titulaire;
  this.solde = soldeInitial;
 }

 /**
 * Effectue un retrait sur le compte
 */
 public void retirer(double montant) {
  if (montant <= 0) {
    throw new MontantInvalideException(""+montant);
  }
  if (montant > solde) {
    throw new SoldeInsuffisantException("solde : " + solde +
                                "montant : " + montant);
  }
  solde = solde - montant;
  System.out.println("Retrait de " + montant + " effectué.");
 }

 /**
 * Effectue un dépôt sur le compte
 */
 public void deposer(double montant) {
  if (montant <= 0) {
     throw new MontantInvalideException();
  }

  solde = solde + montant;
  System.out.println("Dépôt de " + montant + " effectué.");
 }

 public double getSolde() {
  return solde;
 }

 public String getTitulaire() {
```

```
    return titulaire;
  }
}
```

5.5 Utilisation avec gestion d'exceptions

Voici comment utiliser cette classe avec une gestion appropriée des exceptions :

```
package nfa031;
public class GestionCompte {

   public static double lireDouble(String message) {
      double resultat = 0;
      boolean saisieReussie = false;

      while (!saisieReussie) {
         try {
            System.out.print(message);
            resultat = Terminal.lireDouble();
            saisieReussie = true;
         } catch (TerminalException e) {
            System.out.println("Erreur : veuillez saisir un nombre.");
         }
      }

      return resultat;
   }

   public static void main(String[] args) {
      CompteBancaire compte = new CompteBancaire("Alice Dupont",
                                    1000.0);

      System.out.println("=== Gestion de compte bancaire ===");
      System.out.println("Titulaire : " + compte.getTitulaire());
      System.out.println("Solde initial : " + compte.getSolde());

      boolean continuer = true;
      while (continuer) {
         System.out.println("\n1. Retirer 2. Déposer 3. "
                     + "Consulter 4. Quitter");
         int choix = (int) lireDouble("Votre choix : ");

         switch (choix) {
            case 1:
               boolean retraitReussi = false;
               while (!retraitReussi) {
                  try {
                     double montant = lireDouble("Montant àretirer ");
                     compte.retirer(montant);
                     retraitReussi = true;

                  } catch (SoldeInsuffisantException e) {
                     System.out.println("Erreur, solde insuffisant"
                                   + " Solde actuel : " +
```

```
                                compte.getSolde());
                    } catch (MontantInvalideException e) {
                        System.out.println("Erreur : le montant "
                                + "doit être positif.");
                    }
                }
                break;

            case 2:
                boolean depotReussi = false;
                while (!depotReussi) {
                    try {
                        double montant = lireDouble("Montant àdéposer ");
                        compte.deposer(montant);
                        depotReussi = true;

                    } catch (MontantInvalideException e) {
                        System.out.println("Erreur : le montant doit "
                                + "être positif.");
                    }
                }
                break;

            case 3:
                System.out.println("Solde actuel : " +
                                compte.getSolde());
                break;

            case 4:
                continuer = false;
                System.out.println("Au revoir !");
                break;

            default:
                System.out.println("Choix invalide.");
        }
    }
}
```

Dans cet exemple, les exceptions métier (`SoldeInsuffisantException`, `MontantInvalideException`) permettent de séparer clairement la logique métier (dans la classe `CompteBancaire`) de la gestion de l'interface utilisateur (dans la méthode `main`).

5.6 Utilisation du message d'exception dans le catch

Lorsqu'une exception contient un message (défini lors du lancement avec `throw new Exception("message")`), ce message peut être récupéré dans le bloc `catch` pour afficher des informations plus précises à l'utilisateur ou pour le débogage.

La méthode `getMessage()` de l'objet exception permet d'accéder au message :

```
try {
    // Code susceptible de lever une exception avec message
    compte.retirer(montant);
```

```
} catch (SoldeInsuffisantException e) {
   // Récupération et utilisation du message
   String messageErreur = e.getMessage();
   System.out.println("Erreur détectée : " + messageErreur);

   // Ou plus directement :
   System.out.println("Détail : " + e.getMessage());
}
```

Cette approche est particulièrement utile pour les exceptions métier qui peuvent contenir des informations contextuelles spécifiques :

```
try {
   compte.retirer(1500.0);

} catch (SoldeInsuffisantException e) {
   // Affichage du message détaillé défini dans CompteBancaire
   System.out.println("Opération impossible : " + e.getMessage());
   // Affichera : "Opération impossible : Solde insuffisant.
   // Demandé : 1500.0, disponible : 1000.0"

} catch (MontantInvalideException e) {
   System.out.println("Saisie incorrecte : " + e.getMessage());
}
```

L'utilisation des messages d'exception permet de centraliser la logique de description des erreurs dans les classes métier, tout en gardant la flexibilité d'affichage dans l'interface utilisateur.

6 Propagation des exceptions

Une caractéristique fondamentale des exceptions est leur capacité à "remonter" automatiquement dans la pile des appels de méthodes jusqu'à être capturées par un bloc `catch` approprié.

6.1 Principe de la propagation

Quand une méthode appelle une autre méthode qui lance une exception, et qu'elle ne capture pas cette exception, c'est comme si elle la lançait elle-même. L'exception continue à remonter jusqu'à trouver un gestionnaire approprié.

6.2 Exemple de chaîne d'appels

Voici un exemple qui illustre clairement ce mécanisme :

```
package nfa031;
import java.util.Scanner;

public class ProblemeCalculException extends RuntimeException {
  public ProblemeCalculException(String msg){
     super(msg);
  }
}

public class PropagationException {

   public static void methodeA(int x) {
```

```
   System.out.println("Début de methodeA");
   methodeB(x);
   System.out.println("Fin de methodeA");
}

public static void methodeB(int x) {
   System.out.println("Début de methodeB");
   methodeC(x);
   System.out.println("Fin de methodeB");
}

public static void methodeC(int x) {
   System.out.println("Début de methodeC");

   if (x == 0) {
      throw new ProblemeCalculException("message");
   }

   System.out.println("Calcul effectué avec x = " + x);
   System.out.println("Fin de methodeC");
}

public static double lireDouble(String message) {
   Scanner scanner = new Scanner(System.in);
   double resultat = 0;
   boolean saisieReussie = false;

   while (!saisieReussie) {
      try {
         System.out.print(message);
         resultat = scanner.nextDouble();
         saisieReussie = true;
      } catch (Exception e) {
         System.out.println("Erreur : veuillez saisir un nombre.");
         scanner.nextLine(); // Consommer la ligne incorrecte
      }
   }

   return resultat;
}

public static void main(String[] args) {
   System.out.println("=== Début du programme ===");

   int valeur = (int) lireDouble("Entrez une valeur : ");

   try {
      methodeA(valeur);
      System.out.println("Traitement terminé avec succès");

   } catch (ProblemeCalculException e) {
      System.out.println("Erreur capturée : problème de " +
                  "calcul détecté");
   }
```

```
        System.out.println("=== Fin du programme ===");
    }
}
```

6.3 Traces d'exécution

Exécution normale (valeur = 5) :

```
=== Début du programme ===
Entrez une valeur : 5
Début de methodeA
Début de methodeB
Début de methodeC
Calcul effectué avec x = 5
Fin de methodeC
Fin de methodeB
Fin de methodeA
Traitement terminé avec succès
=== Fin du programme ===
```

Exécution avec exception (valeur = 0) :

```
=== Début du programme ===
Entrez une valeur : 0
Début de methodeA
Début de methodeB
Début de methodeC
Erreur capturée : problème de calcul détecté
=== Fin du programme ===
```

6.4 Analyse du mécanisme

Dans le second cas, on observe que :

1. L'exception est lancée dans `methodeC`
2. `methodeC` se termine brutalement (pas d'affichage "Fin de methodeC")
3. `methodeB` se termine également sans affichage de fin
4. `methodeA` se termine également sans affichage de fin
5. L'exception remonte jusqu'au `try/catch` dans `main`
6. Le programme continue normalement après le traitement de l'exception

6.5 Choix du niveau de capture

La propagation des exceptions offre une grande flexibilité dans le choix du niveau où traiter l'erreur. Dans notre exemple, nous aurions pu placer le `try/catch` dans `methodeA`, `methodeB`, ou `main`. Le choix dépend de la responsabilité de chaque niveau :

— **Niveau bas** (`methodeC`) : détection et lancement de l'exception
— **Niveau intermédiaire** (`methodeA`, `methodeB`) : propagation transparente
— **Niveau haut** (`main`) : gestion globale et interface utilisateur

Cette séparation des responsabilités rend le code plus modulaire et maintenable.

7 Bonnes pratiques et recommandations

7.1 Exceptions vs validation simple

La première question à se poser avant de créer ou utiliser une exception est : "Ai-je vraiment besoin d'une exception ?" Les exceptions ne doivent pas être utilisées comme un mécanisme de contrôle de flux normal.

Utilisez les exceptions quand :

— La détection de l'erreur et son traitement se font dans des méthodes différentes
— L'erreur est vraiment exceptionnelle (peu fréquente)
— Vous devez interrompre plusieurs niveaux d'appels de méthodes
— L'erreur nécessite un traitement spécial (logging, notification, etc.)

Préférez la validation simple quand :

— L'erreur peut être détectée et traitée localement
— La situation est prévisible et courante
— Un simple `if/else` suffit pour gérer le cas

Exemple de mauvaise utilisation d'exception :

```
// Àéviter : exception utilisée comme contrôle de flux
public static void methodeInappropriee() {
   try {
      int age = Terminal.lireInt();
      if (age < 0) {
         throw new AgeInvalideException();
      }
      System.out.println("Âge valide : " + age);
   } catch (AgeInvalideException e) {
      System.out.println("Âge invalide");
   }
}
```

Meilleure approche :

```
// Préférable : validation simple et directe
public static void methodePreferable() {
   int age = Terminal.lireInt();
   if (age < 0) {
      System.out.println("Âge invalide");
   } else {
      System.out.println("Âge valide : " + age);
   }
}
```

7.2 Hiérarchie et types d'exceptions

En Java, il existe deux grandes catégories d'exceptions :

— **Checked exceptions** : doivent être déclarées ou capturées (héritent d'`Exception`)
— **Unchecked exceptions** : peuvent être ignorées (héritent de `RuntimeException`)

Dans ce cours, nous utilisons `RuntimeException` car elle est plus simple à manipuler pour l'apprentissage. Dans un contexte professionnel, le choix dépend de la politique de gestion d'erreur de l'équipe.

7.3 Performance et lisibilité

Les exceptions ont un coût en performance lors de leur lancement (création de la pile d'appels). Évitez donc :

— D'utiliser les exceptions dans des boucles intensives
— De lancer des exceptions pour des cas fréquents
— De créer des exceptions "pour rien" qui ne seront jamais capturées

Pour la lisibilité :

— Donnez des noms explicites à vos exceptions (`CompteClotureException` plutôt que `ErreurException`)
— Utilisez des messages informatifs et contextuels
— Documentez les exceptions que peuvent lancer vos méthodes
— Évitez les blocs `catch` vides qui masquent les erreurs

7.4 Patterns courants

Pattern de retry avec limite :

```
public static void operationAvecRetry(int maxTentatives) {
   int tentatives = 0;
   boolean succes = false;

   while (!succes && tentatives < maxTentatives) {
      try {
         // Opération risquée
         operationRisquee();
         succes = true;
      } catch (OperationException e) {
         tentatives++;
         if (tentatives >= maxTentatives) {
            System.out.println("Échec après " + maxTentatives +
                               " tentatives");
            throw e; // Relancer l'exception
         }
         System.out.println("Tentative " + tentatives +
                            " échouée, retry...");
      }
   }
}
```

8 Conclusion

Les exceptions constituent un mécanisme fondamental pour créer des programmes Java robustes et professionnels. Elles permettent de transformer des arrêts brutaux en gestion élégante des situations exceptionnelles, améliorant considérablement l'expérience utilisateur et la maintenabilité du code.

Les concepts clés à retenir sont :

— Les exceptions interrompent le flux normal d'exécution pour gérer les erreurs
— La structure `try/catch` permet de capturer et traiter les exceptions
— La granularité du traitement détermine la robustesse du programme
— Les exceptions personnalisées séparent la détection du traitement d'erreur
— La propagation permet une gestion flexible selon les niveaux de responsabilité

L'approche progressive présentée dans ce chapitre - du traitement basique vers la gestion robuste, puis vers les exceptions personnalisées - reflète l'évolution naturelle d'un programmeur qui apprend à créer des applications de qualité professionnelle.

Les bonnes pratiques en matière d'exceptions s'acquièrent avec l'expérience, mais les principes fondamentaux restent constants : privilégier la clarté, éviter l'usage abusif, et toujours garder à

l'esprit l'expérience de l'utilisateur final. Un programme qui gère intelligemment ses erreurs est un programme qui inspire confiance et facilite le travail de ses utilisateurs.

Ces compétences en gestion d'exception vous serviront tout au long de votre parcours en programmation Java, que ce soit pour des projets académiques, des applications d'entreprise, ou des systèmes critiques où la robustesse est primordiale.

Ce chapitre a été rédigé avec l'assistance d'une intelligence artificielle, Claude 4 Sonnet de la société Anthropic