Les exceptions

La gestion des cas d'erreur représente un travail important dans la programmation. Les sources d'erreur peuvent être nombreuses dans un programme. Il peut s'agir :

- d'une défaillance physique ou logiciel de l'environnement d'exécution. Par exemple une erreur survient lors de l'accès à un fichier ou à la mémoire.
- d'un état atteint par un objet qui ne correspond pas à un cas prévu. Par exemple si une opération demande à positionner une valeur négative alors que cela n'est normalement pas permis par la spécification du logiciel.
- d'une erreur de programmation. Par exemple, un appel à une méthode est réalisé sur une variable dont la valeur est **null**.
- et bien d'autres cas encore…

La robustesse d'une application est souvent comprise comme sa capacité à continuer à rendre un service acceptable dans un environnement dégradé, c'est-à-dire quand toutes les conditions attendues normalement ne sont pas satisfaites.

En Java, la gestion des erreurs se confond avec la gestion des cas exceptionnels. On utilise alors le mécanisme des exceptions.

Qu'est-ce qu'une exception ?

Une exception est une classe Java qui représente un état particulier et qui hérite directement ou indirectement de la classe Exception. Par convention, le nom de la classe doit permettre de comprendre le type d'exception et doit se terminer par Exception.

Exemple de classes d'exception fournies par l'API standard :

NullPointerException
: Signale qu'une référence **null** est utilisée pour invoquer une méthode ou accéder à un attribut.

NumberFormatException
: Signale qu'il n'est pas possible de convertir une chaîne de caractères en nombre car la chaîne de caractère ne correspond pas à un nombre valide.

IndexOutOfBoundsException
: Signale que l'on tente d'accéder à un indice de tableau en dehors des valeurs permises.

Pour créer sa propre exception, il suffit de créer une classe héritant de la classe java.lang.Exception.

```
package com.cgi.udev.heroes;

public class FinDuMondeException extends Exception {

  public FinDuMondeException() {
  }

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

> La classe Exception fournit plusieurs constructeurs que l'on peut ou non appeler depuis la classe fille.

Une exception étant un objet, elle possède son propre état et peut ainsi stocker des informations utiles sur les raisons de son apparition.

```
package com.cgi.udev.heroes;
import java.time.Instant;

public class FinDuMondeException extends Exception {

  private Instant date;

  public FinDuMondeException() {
    this(Instant.now());
  }

  public FinDuMondeException(Instant instant) {
    super("La fin du monde est survenue le " + instant);
    this.date = instant;
  }
```

```
  public Instant getDate() {
    return date;
  }
}
```

Signaler une exception

Dans les langages de programmation qui ne supportent pas le mécanisme des exceptions, on utilise généralement un code retour ou une valeur booléenne pour savoir si une fonction ou une méthode s'est déroulée correctement. Cette mécanique se révèle assez fastidieuse dans son implémentation car cela signifie qu'un développeur doit tester dans son programme toutes les valeurs retournées par les fonctions ou les méthodes appelées

Les exceptions permettent d'isoler le code responsable du traitement de l'erreur. Cela permet d'améliorer la lisibilité du code source.

Lorsqu'un programme détecte un état exceptionnel, il peut le signaler en *jetant* une exception grâce au mot-clé **throw**.

```
if(isPlanDiaboliqueReussi()) {
  throw new FinDuMondeException();
}
```

> La classe Exception hérite de la classe Throwable. Le mot-clé **throw** peut en fait être utilisé avec n'importe quelle instance qui hérite directement ou indirectement de Throwable.

Jeter une exception signifie que le flot d'exécution normal de la méthode est interrompu jusqu'au point de traitement de l'exception. Si aucun point de traitement n'est trouvé, le programme s'interrompt.

Traiter une exception

Pour traiter une exception, il faut d'abord délimiter un bloc de code avec le mot-clé **try**. Ce bloc de code correspond au flot d'exécution pour lequel on souhaite éventuellement *attraper* une exception qui serait jetée afin d'implémenter un traitement particulier. Le bloc **try** peut être suivi d'un ou plusieurs blocs **catch** pour intercepter une exception d'un type particulier.

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  boolean victoire = heros.combattre(espritDuMal);
  boolean planDejoue = heros.desamorcer(machineInfernale);

  if (!victoire || !planDejoue) {
    throw new FinDuMondeException();
  }

  heros.setPoseVictorieuse();

} catch (FinDuMondeException fdme) {
  // ...
}
```

Dans l'exemple ci-dessus, si la variable *heros* vaut **null** alors le traitement du bloc **try** est interrompu à la ligne 3 par une NullPointerException. Sinon le bloc continue à s'exécuter. La ligne 13 ne sera exécutée que si la condition à la ligne 9 est fausse. Par contre, si cette condition est vraie, le traitement du bloc est interrompu par le lancement d'une *FinDuMondeException* et le traitement reprend dans le bloc **catch** à partir de la ligne 16.

La bloc **catch** permet à la fois d'identifier le type d'exception concerné par le bloc de traitement et à la fois de déclarer une variable qui permet d'avoir accès à l'exception durant l'exécution du bloc **catch**. Un bloc **catch** sera exécuté si une exception du même type ou d'un sous-type que celui déclaré par le bloc est lancée à l'exécution. Attention, si une exception déclenche le traitement d'un bloc **catch**, le flot d'exécution reprend ensuite à la fin des blocs **catch**.

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  boolean victoire = heros.combattre(espritDuMal);
  boolean planDejoue = heros.desamorcer(machineInfernale);

  if (!victoire || !planDejoue) {
    throw new FinDuMondeException();
  }
```

```
  heros.setPoseVictorieuse();

} catch (Exception e) {
  // ...
}
```

Dans le code ci-dessus, le bloc **catch** est associé aux exceptions de type Exception. Comme toutes les exceptions en Java hérite directement ou indirectement de cette classe, ce bloc sera exécuté pour traité la NullPointerException à la ligne 3 ou la *FinDuMondeException* à la ligne 10.

Les blocs **catch** sont pris en compte à l'exécution dans l'ordre de leur déclaration. Déclarer un bloc **catch** pour une exception parente avant un bloc **catch** pour une exception enfant est considéré comme une erreur de compilation.

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  boolean victoire = heros.combattre(espritDuMal);
  boolean planDejoue = heros.desamorcer(machineInfernale);

  if (!victoire || !planDejoue) {
    throw new FinDuMondeException();
  }

  heros.setPoseVictorieuse();

} catch (Exception e) {
  // ...
} catch (FinDuMondeException fdme) {
  // ERREUR DE COMPILATION
}
```

Dans, l'exemple précédent, il faut bien comprendre que Exception est la classe parente de *FinDuMondeException*. Donc si une exception de type *FinDuMondeException* est lancée, alors seul le premier bloc **catch** sera exécuté. Le second est donc simplement du code mort est générera une erreur de compilation. Pour que cela fonctionne, il faut inverser l'ordre des blocs **catch** :

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  boolean victoire = heros.combattre(espritDuMal);
  boolean planDejoue = heros.desamorcer(machineInfernale);

  if (!victoire || !planDejoue) {
    throw new FinDuMondeException();
  }

  heros.setPoseVictorieuse();

} catch (FinDuMondeException fdme) {
  // ...
} catch (Exception e) {
  // ...
}
```

Maintenant, un premier bloc **catch** fournit un traitement particulier pour les exceptions de type *FinDuMondeException* ou de type enfant et un second bloc **catch** fournit un traitement pour les autres exceptions.

Parfois, le code du bloc **catch** est identique pour différents types d'exception. Si ces exceptions ont une classe parente commune, il est possible de déclarer un bloc **catch** simplement pour cette classe parente afin d'éviter la duplication de code. Dans notre exemple, la classe ancêtre commune entre NullPointerException et *FinDuMondeException* est la classe Exception. Donc si nous déclarons un bloc **catch** pour le type Exception, nous fournissons un bloc de traitement pour tous les types d'exception, ce qui n'est pas vraiment le but recherché. Dans cette situation, il est possible de préciser plusieurs types d'exception dans le bloc **catch** en les séparant par **|** :

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  boolean victoire = heros.combattre(espritDuMal);
  boolean planDejoue = heros.desamorcer(machineInfernale);

  if (!victoire || !planDejoue) {
```

```
    throw new FinDuMondeException();
  }

  heros.setPoseVictorieuse();

} catch (NullPointerException | FinDuMondeException ex) {
  // traitement commun aux deux types d'exception...
}
```

> L'exécution d'un bloc **catch** peut très bien être interrompue par une exception. L'exécution d'un bloc **catch** peut même conduire à relancer l'exception qui vient d'être interceptée.

Propagation d'une exception

Si une exception n'est pas interceptée par un bloc **catch**, alors elle remonte la pile d'appel, jusqu'à ce qu'un bloc **catch** prenne cette exception en charge. Si l'exception remonte tout en haut de la pile d'appel du thread, alors le thread s'interrompt. S'il s'agit du thread principal, alors l'application s'arrête en erreur.

Le mécanisme de propagation permet de séparer la partie de l'application qui génère l'exception de la partie qui traite cette exception.

Si nous reprenons notre exemple précédent, nous pouvons grandement l'améliorer. En effet, les méthodes *combattre* et *desamorcer* devraient s'interrompre par une exception plutôt que de retourner un booléen. L'exception jetée porte une information plus riche qu'un simple booléen car elle dispose d'un type et d'un état interne.

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  heros.combattre(espritDuMal);
  heros.desamorcer(machineInfernale);
  heros.setPoseVictorieuse();

} catch (FinDuMondeException ex) {
  // ...
}
```

Le code devient beaucoup plus lisible. On comprend que le bloc **try** peut être interrompu par une exception de type *FinDuMondeException* et le code du bloc n'est plus contaminé par des variables et des instructions **if** spécifiquement utilisées pour la gestion des erreurs.

La langage Java impose que les méthodes signalent les types d'exception qu'elles peuvent jeter. Ainsi, le code ci-dessus ne compilera que si au moins une des instructions du bloc **try** peut générer une *FinDuMondeException*. Cela permet au compilateur de détecter d'éventuel code mort. La déclaration des exceptions jetées par une méthode fait donc partie de sa signature et utilise le mot-clé **throws**.

```
package com.cgi.udev.heroes;

public class Heros {

  public void combattre(Vilain vilain) throws FinDuMondeException {
    // ...
  }

  public void desamorcer(Piege piege) throws FinDuMondeException {
    // ...
  }

  public void setPoseVictorieuse() {
    // ...
  }
}
```

Grâce aux exceptions, il est maintenant possible d'interrompre une méthode. Il est même possible d'interrompre un constructeur. Cela aura pour effet de stopper la construction de l'objet et ainsi d'empêcher d'avoir une instance dans un état invalide.

```
package com.cgi.udev.heroes;

public class Heros {

  public Heros(String classePerso) throws ClasseDePersoInvalideException {
    if (classePerso == null || "".equals(classePerso) {
      throw new ClasseDePersoInvalideException();
    }
  }
```

La déclaration des exceptions dans la signature d'une méthode permet à la fois de documenter dans le code lui-même le comportement de la méthode tout en contrôlant à la compilation que les cas d'exception sont gérés par le code.

```
public Marchandise acheter(long montant, Currency devise)
  throws CreditInsuffisantException, DeviseRefuseeException,
         MarchandiseNonDisponibleException {
  // ...
}
```

Dans l'exemple ci-dessus, même sans avoir accès au code source, la signature suffit à renseigner sur les cas d'erreur que l'on va pouvoir rencontrer lorsqu'on appelle la méthode *acheter*.

Exceptions et polymorphisme

Comme la déclaration des exceptions jetées par une méthode fait partie de sa signature, certaines règles doivent être respectées pour la redéfinition de méthode afin que le polymorphisme fonctionne correctement.

Selon le principe de substitution de Liskov, dans la redéfinition d'une méthode, les préconditions ne peuvent pas être renforcées par la sous-classe et les postconditions ne peuvent pas être affaiblies par la sous-classe. Rapporté au mécanisme des exceptions, cela signifie qu'une méthode redéfinie ne peut pas lancer des exceptions supplémentaires. Par contre, elle peut lancer des exceptions plus spécifiques. Le langage Java ne permet pas de distinguer les exceptions qui signalent une violation des préconditions ou des postconditions. C'est donc aux développeurs de s'assurer que les postconditions ne sont pas affaiblies dans la sous-classe.

Ainsi, si la classe *SuperHeros* hérite de la classe *Heros*, elle peut redéfinir les méthodes en ne déclarant pas d'exception.

```
package com.cgi.udev.heroes;

public class SuperHeros extends Heros {

  @Override
  public void combattre(Vilain vilain) {
    // ...
  }

  @Override
```

```
  public void desamorcer(Piege piege) {
    // ...
  }
}
```

Cette nouvelle classe peut aussi changer les types d'exception déclarés par les méthodes redéfinies à condition que ces types soient des classes filles des exceptions d'origine.

```
package com.cgi.udev.heroes;

public class SuperHeros extends Heros {

  @Override
  public void desamorcer(Piege piege) throws PlanMachiaveliqueException {
    // ...
  }

}
```

Le code précédent ne compile que si l'exception *PlanMachiaveliqueException* hérite directement ou indirectement de *FinDuMondeException*.

```
package com.cgi.udev.heroes;

public class PlanMachiaveliqueException extends FinDuMondeException {
  // ...
}
```

> Même si cela est maladroit, il est possible de conserver la déclaration des exceptions dans la signature même si la méthode ne jette pas ces types d'exception. Le compilateur ne vérifie pas si une méthode jette effectivement tous les types d'exception déclarés par sa signature.

Le bloc finally

À la suite des blocs **catch** il est possible de déclarer un bloc **finally**. Un bloc **finally** est exécuté systématiquement, que le bloc **try** se soit terminé normalement ou par une exception.

> Si un bloc **try** se termine par une exception et qu'il n'existe pas de bloc **catch** approprié, alors le bloc **finally** est exécuté et ensuite l'exception est propagée.

```
try {
  if (heros == null) {
    throw new NullPointerException("Le heros ne peut pas être nul !");
  }

  heros.combattre(espritDuMal);
  heros.desamorcer(machineInfernale);
  heros.setPoseVictorieuse();

} catch (FinDuMondeException fdme) {
  // ...
} finally {
  // Ce bloc sera systématiquement exécuté
  jouerGeneriqueDeFin();
}
```

> Un bloc **finally** est exécuté même si bloc **try** exécute une instruction **return**. Dans ce cas, le bloc **finally** est d'abord exécuté puis ensuite l'instruction **return**.

Le bloc **finally** est le plus souvent utilisé pour gérer les ressources autre que la mémoire. Si le programme ouvre une connexion, un fichier..., le traitement est effectué dans le bloc **try** puis le bloc **finally** se charge de libérer la ressource.

```
java.io.FileReader reader = new java.io.FileReader(filename);
try {
  int nbCharRead = 0;
  char[] buffer = new char[1024];
  StringBuilder builder = new StringBuilder();
  // L'appel à reader.read peut lancer une java.io.IOException
  while ((nbCharRead = reader.read(buffer)) >= 0) {
    builder.append(buffer, 0, nbCharRead);
  }
  // le retour explicite n'empêche pas l'exécution du block finally.
  return builder.toString();
} finally {
  // Ce block est obligatoirement exécuté après le block try.
  // Ainsi le flux de lecture sur le fichier est fermé
  // avant le retour de la méthode.
```

```
  reader.close();
}
```

Le try-with-resources

La gestion des ressources peut également être réalisée par la syntaxe du try-with-resources.

```
try (java.io.FileReader reader = new java.io.FileReader(filename)) {
  int nbCharRead = 0;
  char[] buffer = new char[1024];
  StringBuilder builder = new StringBuilder();
  while ((nbCharRead = reader.read(buffer)) >= 0) {
    builder.append(buffer, 0, nbCharRead);
  }
  return builder.toString();
}
```

Après le mot-clé **try**, on déclare entre parenthèse une ou plusieurs initialisations de variable. Ces variables doivent être d'un type qui implémente l'interface AutoCloseable ou Closeable. Ces interfaces ne déclarent qu'une seule méthode : **close**. Le compilateur ajoute automatiquement un bloc **finally** à la suite du bloc **try** pour appeler la méthode **close** sur chacune des variables qui ne valent pas **null**.

Ainsi pour ce code :

```
try (java.io.FileReader reader = new java.io.FileReader(filename)) {
  // ...
}
```

Le compilateur générera le bytecode correspondant à :

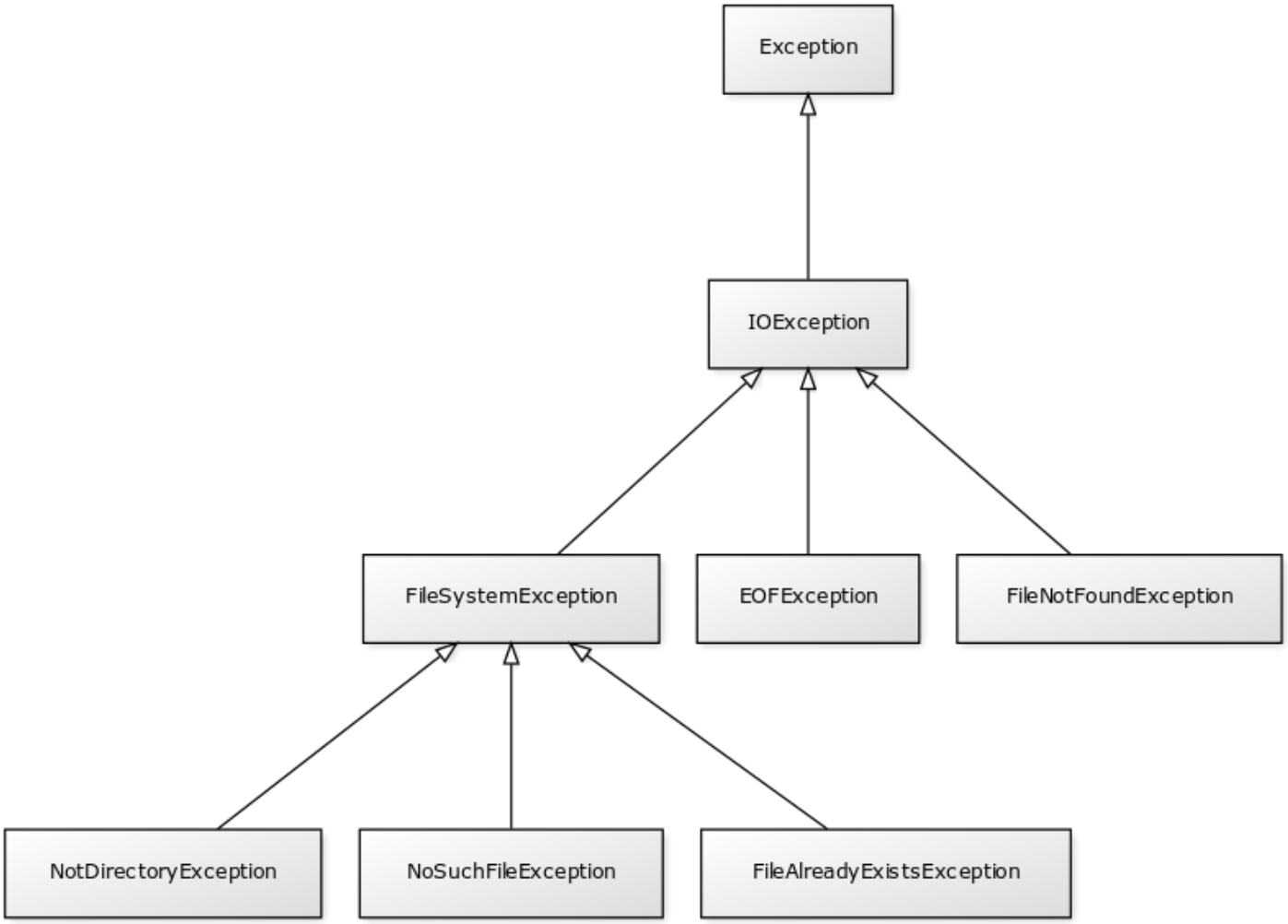
```
{
  java.io.FileReader reader = new java.io.FileReader(filename)
  try {
    // ...
  } finally {
    if (reader != null) {
      reader.close();
    }
```

```
    }
}
```

La syntaxe try-with-resources est à la fois simple à lire et évite d'oublier de libérer des ressources puisque le compilateur se charge d'introduire le code pour nous.

Hiérarchie applicative d'exception

Comme les exceptions sont des objets, il est possible de créer une hiérarchie d'exception par héritage. C'est par exemple le cas pour les exceptions d'entrée/sortie en Java.



Un extrait de la hiérarchie de java.io.IOException

La hiérarchie d'exception permet de grouper des erreurs en concevant des types d'exception de plus en plus généraux. Une application pourra donc traiter à sa convenance des exceptions générales comme IOException mais pourra, au besoin, fournir une bloc **catch** pour traiter des

exceptions plus spécifiques.

```
try {

  // ... opérations sur des fichiers

} catch (NoSuchFileException nsfe) {

  // ...

} catch (IOException ioe) {

  // ...

}
```

Exception cause

Il est souvent utile d'encapsuler une exception dans une autre exception. Par exemple, imaginons une méthode qui souhaite réaliser une opération distante sur un serveur. Se le serveur distant n'est pas joignable, le programme devra intercepter une IOException. Mais cela n'a peut-être pas beaucoup de sens pour le reste du programme, la méthode peut décider de jeter à la place une exception définit par l'application comme une *OperationNonDisponibleException*.

```
package com.cgi.udev;

public class OperationNonDisponibleException extends Exception {

  public OperationNonDisponibleException(Exception cause) {
    super(cause);
  }
}
```

Cette exception n'a pas de lien d'héritage avec une IOException. Par contre, elle expose un constructeur qui accepte en paramètre une exception. Cela permet d'indiquer que l'exception a été causée par une autre exception.

```
try {
```

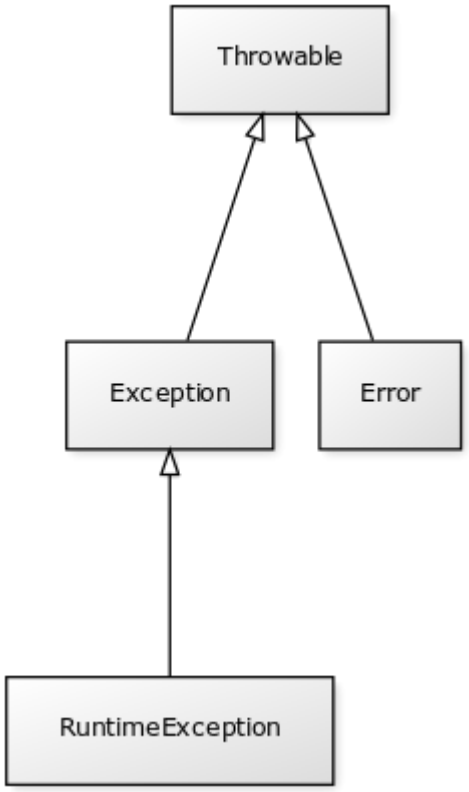
```
 // ... opérations d'entrée / sortie vers le serveur

} catch (IOException ioe) {
  throw new OperationNonDisponibleException(ioe);
}
```

La classe Exception fournit la méthode getCause (qu'elle hérite de Throwable) pour connaître l'exception qui est la cause du problème.

Les erreurs et les exceptions runtime

En regardant plus en détail la hiérarchie à la base des exceptions, on découvre le modèle d'héritage suivant :



La classe Throwable est la classe indiquant qu'il est possible d'utiliser ce type avec le mot clé **throw**. De plus la classe Throwable fournit des méthodes utilitaires. Par exemple, la méthode printStackTrace permet d'afficher sur la sortie d'erreur standard la pile d'appel de l'application.

```
try {
  double d = 1/0; // produit une ArithmeticException
} catch (ArithmeticException e) {
  // Afficher la pile d'appel sur la sortie d'erreur standard
  e.printStackTrace();
}
```

La classe Error hérite de Throwable comme Exception. Error est la classe de base pour représenter les erreurs sérieuses que l'application ne devrait pas intercepter. Lorsqu'une erreur survient cela signifie souvent que l'environnement d'exécution est dans un état instable. Par exemple, la classe OutOfMemoryError hérite indirectement de cette classe. Cette erreur signale que la JVM ne dispose plus d'assez de mémoire (généralement pour allouer de l'espace pour les nouveaux objets).

La classe RuntimeException représente des problèmes d'exécution qui proviennent la plupart du temps de bug dans l'application. Parmi les classes filles de cette classe, on trouve :

ArithmeticException
: signale une opération arithmetique invalide comme une division par zéro.

NullPointerException
: signale que l'on tente d'accéder à une méthode ou un attribut à travers une référence **null**.

ClassCastException
: signale qu'un transtypage invalide a été réalisé.

Généralement, les exceptions qui héritent de RuntimeException ne sont pas interceptées ni traitées par l'application. Au mieux, elles sont interceptées au plus haut de la pile d'appel pour signaler une erreur à l'utilisateur ou dans les fichiers de log.

Les classes Error, RuntimeException et toutes les classes qui en héritent sont appelées des *unchecked exceptions*. Cela signifie que le compilateur n'exige pas que ces exceptions apparaissent dans la signature des méthodes. En effet, elles représentent des problèmes internes graves de la JVM ou des bugs. Donc virtuellement toutes les méthodes en Java sont susceptibles de lancer de telles exceptions.

Si nous reprenons notre exemple des véhicules, les méthodes pour accélérer et décélerer devraient contrôler que le paramètre passé est bien un nombre positif. Si ce n'est pas le cas, elle peut jeter une IllegalArgumentException qui est une exception runtime fournie par l'API standard et qui sert à signaler qu'un paramètre est invalide. Cette exception ne doit pas être obligatoirement déclarée dans la signature de la méthode.

```
package com.cgi.udev.conduite;
```

```
public class Vehicule {

  private final String marque;
  protected float vitesse;

  public Vehicule(String marque) {
    this.marque = marque;
  }

  public void accelerer(float deltaVitesse) {
    if (deltaVitesse < 0) {
      throw new IllegalArgumentException("deltaVitesse doit être positif");
    }
    this.vitesse += deltaVitesse;
  }

  public void decelerer(float deltaVitesse) {
    if (deltaVitesse < 0) {
      throw new IllegalArgumentException("deltaVitesse doit être positif");
    }
    this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f);
  }

  // ...

}
```

> Il est tout de même intéressant de signaler les exceptions runtime qui sont engendrées par des violations de préconditions ou de postconditions. Cela permet de documenter explicitement ces préconditions et ces postconditions.

```
/**
 * Accélère le véhicule
 *
 * @param deltaVitesse la vitesse à ajouter à la vitesse courante.
 * @throws IllegalArgumentException si deltaVitesse est un nombre négatif.
 */
public void accelerer(float deltaVitesse) throws IllegalArgumentException {
  if (deltaVitesse < 0) {
```

```
    throw new IllegalArgumentException("deltaVitesse doit être positif");
  }
  this.vitesse += deltaVitesse;
}
```

Par opposition, toutes les autres exceptions sont appelées des *checked exception*. Une méthode qui est susceptible de laisser se propager une *checked exception* doit le signaler dans sa signature à l'aide du mot-clé **throws**.

Choix entre checked et unchecked

En tant que développeurs, lorsque nous créons de nouvelles classes pour représenter des exceptions, nous avons le choix entre hériter de la classe Exception ou de la classe RuntimeException. C'est-à-dire entre créer une *checked* ou une *unchecked* exception. La frontière entre les deux familles a évolué au cours des versions de Java.

> Il ne faut jamais créer un classe qui hérite de Error. Les classes qui en héritent sont faites pour signaler un problème dans la JVM.

On considère généralement qu'il est préférable de créer une *unchecked exception* lorsque l'exception représente une erreur technique, un événement qui ne relève pas du domaine de l'application mais qui est plutôt lié à son contexte d'exécution. Généralement il s'agit d'exceptions que l'application ne pourra pas traiter correctement à part signaler un problème aux utilisateurs ou aux administrateurs. Par exemple, si votre application se connecte à un service distant, vous pouvez avoir besoin de créer une exception *RemoteServiceUnavailableException* pour signaler que le service ne répond pas. Ce type d'exception est probablement une *unchecked exception* et devrait hériter de RuntimeException.

Par contre, les exceptions qui peuvent avoir une valeur pour le domaine applicatif devraient être des *checked exception*. Généralement, elles traduisent des états particuliers identifiés par les analystes du domaine.

Par exemple, si vous développez une application bancaire pour réaliser des transactions, certaines transactions peuvent échouer lorsqu'un compte bancaire n'est pas suffisamment approvisionné. Pour représenter cet état, on peut créer une classe *SoldeInsuffisantException*. Il est probable que cette exception devrait être une *checked exception* afin que le compilateur puisse vérifier qu'elle est correctement traitée.

Created 2025-02-10 12:23:34 UTC by Nicolas
Updated 2025-02-10 12:27:44 UTC by Nicolas