Exceptions

Gestion des erreurs et exceptions

Tout programme doit être capable de gérer les erreurs susceptibles de survenir lors de son exécution. Ces erreurs peuvent avoir diverses origines, notamment :

- Erreurs de programmation/Etats internes non prévus : Accès hors bornes, null inattendu, violations de préconditions, etc.
- Entrées non valides fournis par l'utilisateur ;
- Erreurs de communication avec des périphériques ou des réseaux.

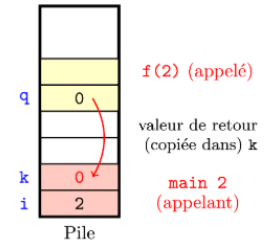
Il s'agit de situations exceptionnelles qui ne sont pas prévues dans le flux normal d'un programme.

Principe des exceptions

Lorsqu'une méthode est appelée, elle peut se terminer de deux façons :

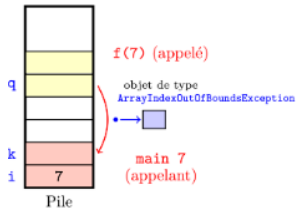
1. Dans un état normal :
   - L'exécution de la méthode se déroule sans problème et se termine correctement ;
   - Le contrôle est alors rendu à la méthode appelante une fois l'exécution terminée.
2. Dans un état d'erreur (exception) :
   - Une exception est levée pendant l'exécution de la méthode, empêchant celle-ci de se terminer normalement ;
   - La méthode se termine **avant sa fin**, et le contrôle est rendu à la méthode appelante avec un signal d'erreur sous la forme d'une exception (un objet de type $Exception$).

Dans les deux cas, la gestion du flux de contrôle revient à la méthode qui appelé celle-ci.



Méthode qui se termine correctement →

Dans le cas de terminaison normale l'appelant reçoit la valeur de retour de la méthode



Méthode qui termine en état d'erreur →

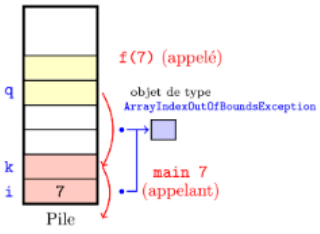
- Lorsqu'une méthode termine en état d'erreur, cela signifie qu'une exception a été levée pendant son exécution ;
- Dans ce cas, l'appelant de la méthode reçoit un objet de type $Exception$, qui contient des informations sur la situation ayant provoqué cette erreur.

Choix de l'appelant en cas d'exception

Lorsqu'une méthode appelée lève une exception, l'appelant a deux choix :

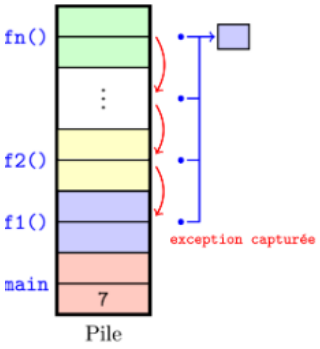
1. Gérer (capturer) l'exception :
   - L'appelant peut choisir de capturer l'exception, gérer l'erreur, et poursuivre son exécution.
2. Ne rien faire (le cas ici) :

- L'appelant peut choisir de ne pas gérer l'exception, ce qui signifie que l'exception est propagée vers l'appelant suivant (s'il y en a un) ;
- Si l'exception atteint la méthode $main$ sans être capturée, le programme **termine en situation d'erreur**.



Capturer une exception

- Une exception peut remonter toute la pile d'exécution (dans l'ordre inverse des appels de méthode) jusqu'à ce qu'elle soit capturée par une méthode ou qu'elle atteigne la méthode $main$.
- Si aucune méthode ne capture l'exception, le programme se termine **en erreur**.



- Si l'exception est capturée par une méthode, celle-là gère et continue son exécution normalement.

Gérer les exceptions

Les cinq mots-clés

Java gère les exceptions à l'aide de cinq mots-clés : try, $catch$, $throw$, $throws$ et $finally$

- Les instructions du programme qu'on souhaite surveiller sont placées dans un bloc try ;
- Si une exception se produit dans le bloc, elle est lancée ($thrown$) ;
- Le code peut attraper ($catch$) cette expression pour la gérer de manière appropriée.
- Les exceptions générées par le système sont automatiquement lancées par l'environnement d'exécution Java. Pour lancer manuellement une exception, on utilise le mot-clé $throw$;
- Dans certains cas, une exception lancée par une méthode doit être spécifiée via une clause $throws$;
- Enfin, tout code qui doit absolument s'exécuter à la sortie du bloc try est placé dans un bloc $finally$.

Mécanisme $try - catch$

Le bloc $try - catch$ est un mécanisme qui permet de gérer les exceptions qui peuvent se produire pendant l'exécution d'un programme, sans que ce dernier ne se termine brutalement.

- Bloc try : Contient le code qui peut potentiellement lever une exception ;
- Bloc $catch$: Capture et gère l'exception si elle est levée dans le bloc try.

- Si une exception d'un type incompatible avec le type mentionné dans le bloc $catch$ est levée dans le bloc try, la méthode termine en état d'erreur sans exécuter le bloc $catch$. L'exception non capturée sera remontée à l'appelant.
- L'exception non capturée sera remontée à l'appelant.

Afficher la trace de la pile d'exécution

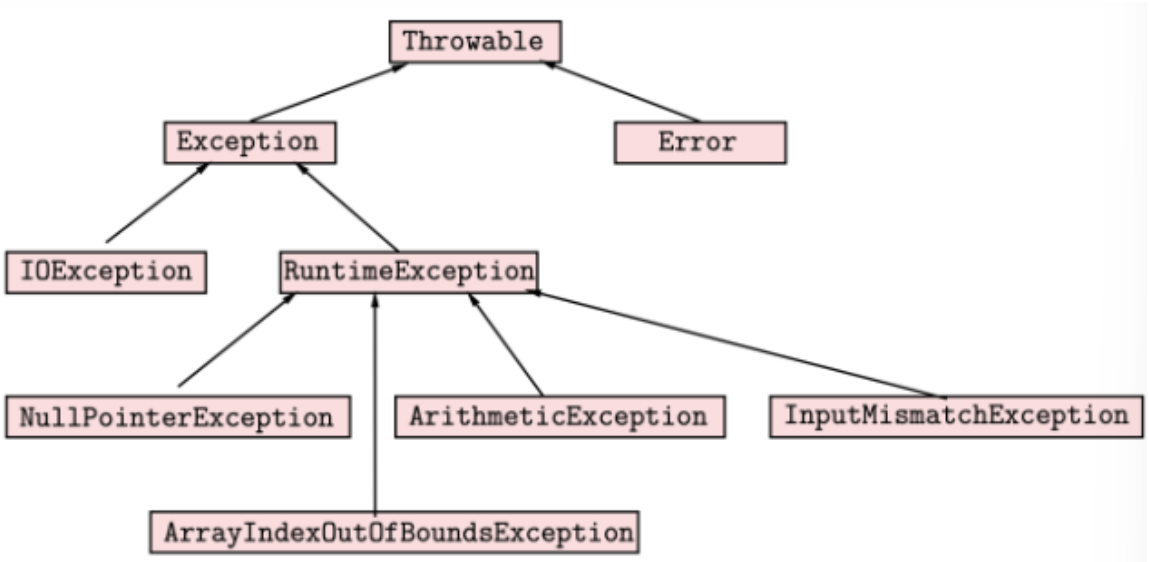
- Lorsqu'une exception est levée, il est souvent utile d'afficher la pile d'exécution (stack trace) pour comprendre précisément **où** et **pourquoi** l'erreur s'est produite ;
- Cela aide à déboguer le programme en fournissant des informations détaillées sur les appels de méthode qui ont conduit à l'exception.

La méthode $printStackTrace()$:

- $printStackTrace()$ est une méthode de la classe $Throwable$ en Java ;
- Elle permet d'afficher la trace d'exécution d'une exception à partir du moment où elle a été levée jusqu'à l'endroit où elle est capturée ;
- Cette méthode est souvent utilisée pour déboguer le code, car elle donne des informations détaillées sur l'origine de l'erreur.

Nature et hiérarchie des exceptions

- En Java, une exception est un **objet**, instance de la classe $Exception$;
- La classe $Exception$ hérite de la classe $Throwable$, qui est la superclasse de toutes les exceptions et erreurs en Java.



La classe $Error$

- La classe $Error$ et ses sous-classes représentent des situations anormales qui peuvent se produire au sein de la JVM (Java Virtual Machine) ;
- Les Errors correspondent à des problèmes graves qui ne sont pas sensés être récupérés par les applications.

Remarque :

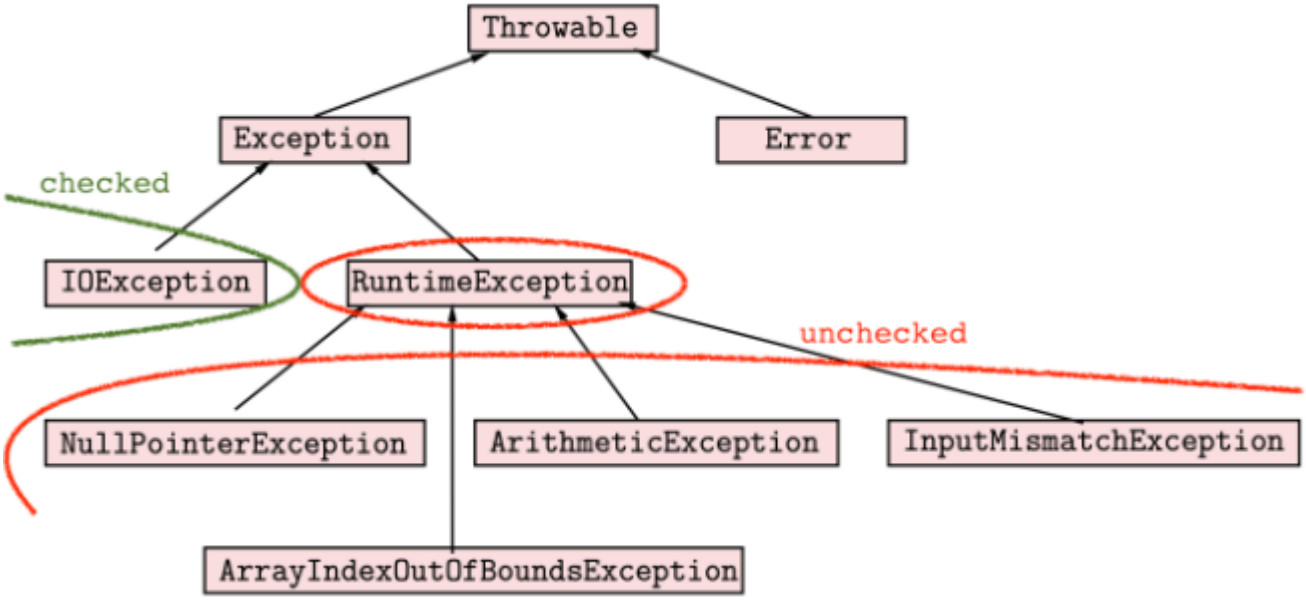
- La classe $Error$ représente des erreurs graves qui indiquent que la machine virtuelle Java (JVM) ne peut pas continuer son exécution correctement ;
- Il est généralement impossible de récupérer après de telles erreurs ;
- Contrairement aux exceptions, les Errors ne doivent pas être rattrapées par l'application via un bloc $try - catch$.

Quelques autres sous-classes de $Error$

- $StackOverflow$: Cette erreur se produit lorsque la pile d'appels (stack) est saturée. Cela se produit souvent dans les cas récursions infinies ou profondes, où une méthode s'appelle elle-même de manière répétée sans condition d'arrêt appropriée.

- $InternalError$: Cette erreur est levée lorsqu'une défaillance interne inattendu se produit dans la JVM. Cela peut signaler un bug dans l'implémentation de la JVM ou dans le code natif utilisé par la JVM. Ces erreurs sont rares.

Exceptions « checked » et « unchecked »



Exception checked

Les exceptions checked sont des exceptions qui doivent être déclarées ou gérées explicitement dans le code. Elles décrivent de la classe $Exception$, mais ne sont pas des sous-classes de $RuntimeException$:
Erreurs qui **ne sont pas des erreurs de programmation** et qui dépendent de **conditions externes** (entrées/sorties, fichiers, réseaux, etc.).
Ces exceptions sont vérifiées à la compilation. Si elles ne sont pas correctement gérées (avec un bloc $try - catch$ ou avec $throws$ dans la déclaration de méthode), le programme ne compile pas.

Liste des exceptions checked en Java

- Liste non exhaustive.

Exception	Signification
ClassNotFoundException	Classe introuvable.
CloneNotSupportedException	Tentative de cloner un objet qui n'implémente pas l'interface Cloneable.
IllegalAccessException	Accès à une classe refusé.
InstantiationException	Tentative de créer un objet d'une classe abstraite ou d'une interface.
InterruptedException	Un thread a été interrompu par un autre thread.
NoSuchFieldException	Un champ demandé n'existe pas.
NoSuchMethodException	Une méthode demandée n'existe pas.
ReflectiveOperationException	Superclasse des exceptions liées à la réflexion.
IOException	Erreur d'entrée/sortie, par exemple lors de la lecture ou de l'écriture de fichiers.
FileNotFoundException	Fichier introuvable.
EOFException	Fin de fichier atteinte de manière inattendue.

Le mot-clé $throws$

- Le mot-clé $throws$ est utilisé dans la signature d'une méthode pour indiquer que cette méthode peut potentiellement lancer des exceptions ;
  Il est utilisé pour propager des exceptions à la méthode appelante au lieu de les gérer localement avec un bloc $try - catch$.
- Si une méthode déclare une exception dans $throws$, elle ne gère pas l'exception directement ;
- L'appelant de cette méthode doit gérer l'exception ou déclarer qu'il la lance à son tour.

Exemple de gestion d'une exception checked : La méthode $Clone$

- $CloneNotSupportedException$ est une exception $checked$, donc elle doit être gérée.

Première solution :

- On ajoute un bloc $try - catch$ pour la capturer et la gérer.

Deuxième solution :

- Si on ne souhaite pas gérer l'exception, il est dans ce cas obligatoire d'ajouter dans la signature de $main\ throws\ CloneNotSupportedException$;
- Cela indique que $main$ ne gère pas cette exception non plus et qu'elle la lance à son tour.

Si on ne le fait pas, on aura l'erreur :

$unreported\ exception\ java.lang.CloneNotSupportedException;$
$must\ be\ caught\ or\ declared\ to\ be\ thrown.$

Contraintes sur les exceptions checked

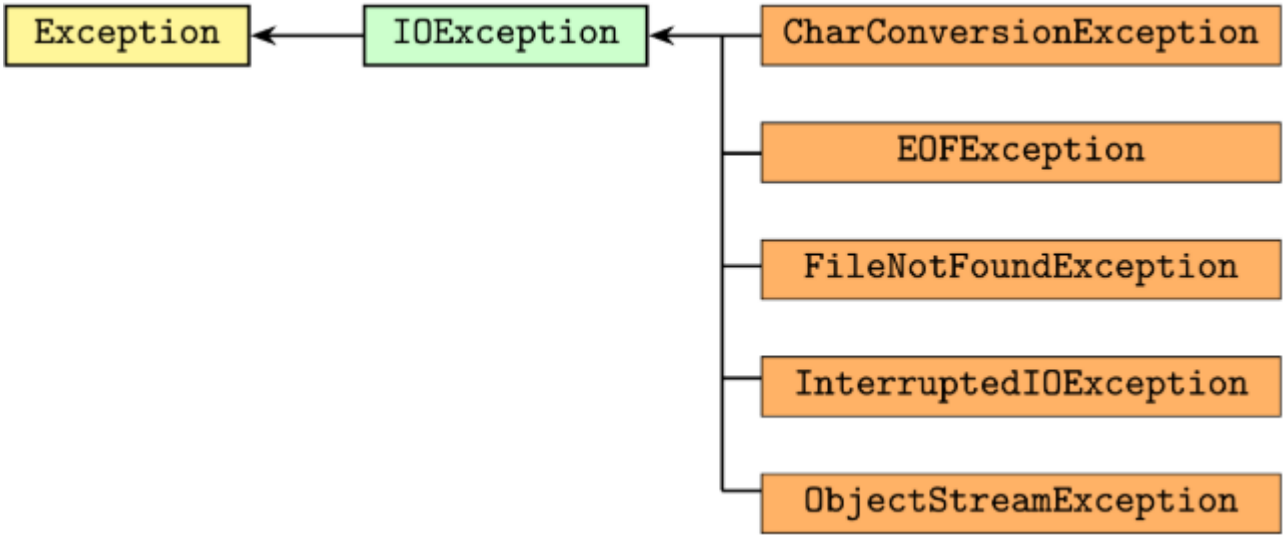
Toute sa méthode qui peut lever une exception checked doit déclarer cette possibilité dans sa signature avec $throws$.

Remarque : Pas de $throws$ nécessaire pour les exceptions unchecked.

- Les exceptions unchecked (comme $NullPointerException$ ou $ArrayIndexOutOfBoundsException$) ne doivent pas être déclarées avec $throws$, car toute instruction peut en lever une à tout moment.

Un autre exemple avec $IOException$

- $FileNotFoundException$ est une sous-classe de $IOException$, donc la déclaration $throws\ IOException$ couvre également cette exception.



Gestion d'une exception checked

Si une méthode capture une exception, elle n'a plus besoin de la déclarer dans sa signature avec $throws$.

Exception unchecked

- Les exceptions unchecked sont des exceptions qui dérivent de $RunTimeException$ et n'ont pas besoin d'être déclarées ou capturées explicitement. Elles sont généralement liées à des erreurs de programmation.

Caractéristiques :

- Mauvais fonctionnement du programme (bugs) : ce sont des erreurs qui ne devrait pas se produire si le code est bien écrit et testé ;

- Ces exceptions surviennent souvent à cause d'une mauvaise utilisation (par exemple, accès à un indice invalide dans le tableau) ;
- Elles ne sont pas vérifiées à la compilation, ce qui signifie que le programme compilera même si ces exceptions ne sont pas gérées.

Exception	Signification
ArithmeticException	Erreur arithmétique, comme une division entière par zéro.
ArrayIndexOutOfBoundsException	Indice de tableau hors limites.
ArrayStoreException	Affectation à un élément de tableau d'un type incompatible.
ClassCastException	Cast invalide.
EnumConstantNotPresentException	Tentative d'utiliser une valeur d'énumération non définie.
IllegalArgumentException	Argument illégal utilisé pour invoquer une méthode.
IllegalMonitorStateException	Opération de moniteur illégale, comme l'attente sur un thread non verrouillé.
IllegalStateException	L'environnement ou l'application est dans un état incorrect.
IllegalThreadStateException	Opération demandée incompatible avec l'état actuel du thread.
IndexOutOfBoundsException	Un type d'indice hors limites.
NegativeArraySizeException	Création d'un tableau avec une taille négative.
NullPointerException	Utilisation invalide d'une référence nulle.
NumberFormatException	Conversion invalide d'une chaîne de caractères en un format numérique.
SecurityException	Tentative de violer la sécurité.
StringIndexOutOfBoundsException	Tentative d'accès en dehors des limites d'une chaîne de caractères.
TypeNotPresentException	Type non trouvé.
UnsupportedOperationException	Une opération non supportée a été rencontrée.

Exceptions unchecked définies dans $java.lang$

- Liste non-exhaustive.

Exemple d'exception unchecked

$NullPointerException$ (NPE) :

- Un $NullPointerException$ est levée lorsqu'un programme essaie d'utiliser (par exemple, lire un champ ou invoquer une méthode) une référence qui a la valeur $null$;
- **Depuis Java 14 :** Les messages d'erreur de $NullPointerException$ sont améliorés pour indiquer quelle variable contient la référence $null$ que le programme a essayé d'utiliser (et pas seulement la ligne où l'exception a été levée).

Remarque :

- Avant Java 14, seule la ligne qui a levé l'exception était affiché. Mais cela ne suffisait pas à identifier précisément quelle référence est null (a ? $a.b$? $a.b.c$?) ;
- Maintenant, l'exception indique exactement quelle partie de l'expression est $null$, facilitant ainsi le débogage.

Définir des exceptions personnalisées en Java

Il est possible de créer ses propres classes d'exceptions pour gérer des situations spécifiques à notre programme. Cela permet de :

- Fournir des messages d'erreur plus précis et adaptés au contexte de notre application ;
- Gérer des erreurs qui ne sont pas couvertes par les exceptions standard fournies par Java.

Héritage :

- Une exception personnalisée peut hériter de la classe $Exception$ (pour créer une exception **checked**) ;
- Elle peut aussi hériter de $RuntimeException$ (pour créer une exception **unchecked**).

Lever une exception

- Une conception peut être levée automatiquement par une opération « primitive » comme :
  - Accès à un tableau ;
  - Accès à un pointeur $null$;
  - Opération sur un fichier.
- Une méthode peut également lever une exception lorsqu'une erreur est rencontrée.

Le mot-clé $throw$

- Il est possible de lever une exception manuellement (qu'elle soit **checked** ou unchecked) lorsqu'on détecte une situation d'erreur spécifique) ;
- Le mot-clé $throw$ permet de lancer une exception manuellement dans le programme ;

- Utilisé lorsqu'une condition anormale ou une erreur est rencontrée et qu'on souhaite signaler cette erreur pour qu'elle soit gérée ailleurs.

$throw\ new\ ExceptionType("Message\ d'erreur");$

Gestion d'une sous-classe d'exception avec $getClass()$

- Le bloc $catch$ utilise la méthode $getClass()$ pour vérifier si l'exception levée est une $FileNotFoundException$;
- Si c'est le cas, un message d'erreur est affiché ;
- Si ce n'est pas le cas une $FileNotFoundException$, l'exception est relancée pour être gérée ailleurs dans le programme.

Permet une meilleure gestion des erreurs, en distinguant les erreurs de type ($FileNotFoundException$) des autres erreurs d'entrées/sorties ($IOException$).

Bloc $try - catch$:

- **Détails :**
  Il est possible de capturer plusieurs exceptions dans un même bloc $try - catch$.
  Explication :
  - Le premier bloc $catch$ qui est compatible avec le type de l'exception levée est celui qui est exécuté ;
  - Si le fichier n'est pas trouvé, le bloc $catch$ pour $FileNotFoundException$ sera exécuté.
  - Si le fichier est trouvé mais on rencontre une autre erreur du type $IOException$, le bloc $catch$ pour $IOException$ sera exécuté.
- **Ordre des blocs :**
  L'ordre des blocs $catch$ doit être respecté :
  - Un bloc $catch$ qui capture une exception plus générale (comme $IOException$) doit apparaître après les blocs $catch$ plus spécifiques (comme $FileNotFoundException$).

  Ce code **ne compilera pas** !
  - $FileNotFoundException$ est une sous-classe de $IOException$;

  Si le bloc $catch$ pour $IOException$ est placé avant celui pour $FileNotFoundException$, l'exception sera capturée par le premier bloc et le second bloc ne sera jamais atteint.
  L'ordre est correct : $FileNotFoundException$ (plus spécifique) est capturée avant $IOException$ (plus générale).
  Résumé :
  - Il est important de respecter l'ordre des bloc catch pour que les exceptions spécifiques soient capturées avant les exceptions générales.

La clause $finally$

- La clause $finally$ peut être ajoutée à un bloc $try - catch$;
- Elle contient du code qui sera exécuté dans tous les cas après le bloc $try - catch$, quelle que soit la manière dont on sort du bloc.

Façons possibles de sortir d'un bloc $try - catch$:

- Par l'exécution correcte du bloc try ;
- Par l'exécution d'un des blocs $catch$;
- Par une exception non capturée, levée par le bloc try ou un bloc $catch$;
- Par un $return$ exécuté dans le bloc try ou $catch$.

Scénarios possibles avec le programme $FinallyTest$

- Cas 1 : Pas d'exception levée ($arg: 1$)

  $java\ FinallyTest\ 1$
  $> Valeur\ de\ i : 100$
  $> Suite\ du\ main..$
- Cas 2 : Levée de $CharacterCodingException$ ($arg: 2$)

  $java\ FinallyTest\ 2$
  $> Exception\ CharacterCoding\ capturée$
  $> Valeur\ de\ i : 0$
  $> Suite\ du\ main..$
- Cas 3 : Levée de $FileNotFound\ Exception$ ($arg: 3$)

  $java\ FinallyTest\ 3$
  $> Exception\ FileNotFound\ capturée + RuntimeException\ lancée$
  $> Valeur\ de\ i : 0$
  $Exception\ in\ thread\ "main"\ java.lang.RuntimeException$
- Cas 4 : Levée de $EOFException$ avec retour ($arg: 4$)

  $java\ FinallyTest\ 4$
  $> Exception\ EOF\ capturée + return$
  $> Valeur\ de\ i : 0$
- Cas 5 : Levée de $IOException$ ($arg: 5$)

  $java\ FinallyTest\ 3$
  $> Valeur\ de\ i : 0$
  $Exception\ in\ thread\ "main"\ java.lang.IOException$

Travailler avec des ressources

- Lorsqu'on travaille avec des ressources (comme un fichier ou une connexion), il est essentiel de s'assurer que ces ressources sont fermées, quelle que soit la manière dont on sort du bloc try (exécution correcte ou exception) ;
- Par exemple, un $Scanner$ ou une connexion à une base de données doivent être fermés pour éviter les fuites de ressources ;
- Les instructions suivantes sont typiques :

```
/* ouvrir la ressource */
try{  /* travailler avec la ressource */
}
finally{ /* fermer la ressource */
}
```

Le bloc $try - with - resources$

- Depuis Java 7, un raccourci permet de gérer automatiquement la fermeture des ressources avec le bloc $try - with - resources$;
- Cette syntaxe permet de simplifier la gestion des ressources tout en garantissant leur fermeture, quelle que soit la façon dont on sort du bloc try.

Remarques

- **Interface $AutoCloseable$:**

- Pour qu'un objet puisse être utilisé dans un bloc $try - with - reources$, il doit implémenter l'interface $AutoCloseable$, qui fournit la méthode $close()$.
- Exemple avec $Scanner$ → [FinallyTest.java, class Scanner]

Le bloc $catch$ est exécuté **après la fermeture de la ressource**.

- **Depuis Java 9 :**
  - Depuis Java 9, la ressource utilisée dans un bloc $try - with - resources$ n'a pas besoin d'être créée dans le bloc try ;
  - Elle peut être variable existante, à condition qu'elle soit effectively final.