



- La gestion des exceptions en Java -



SOMMAIRE

■ 1	<u>Généralités sur le traitement des erreurs</u>	1
■ 2	<u>Principes de base des exceptions</u>	2
■ 3	<u>La hiérarchie des exceptions</u>	3
■ 4	<u>Les deux types d'erreurs en Java</u>	5
■ 5	<u>Présentation du bloc <i>try/catch</i></u>	11
■ 6	<u>Après quelques exercices de mise en œuvre quelques précisions</u>	17
■ 7	<u>Déclenchement d'une exception</u>	19
■ 8	<u>Exceptions déclenchées par Java</u>	20
■ 9	<u>Exceptions déclenchées par l'application</u>	21
■ 10	<u>Propager une exception : la clause <i>throws</i></u>	22
■ 11	<u>Capter une exception : autres cas</u>	26
■ 12	<u>Résumé</u>	29

1 - Généralités sur le traitement des erreurs

Introduction

Dans tout programme, le **traitement des erreurs** représente une tâche importante, souvent négligée par les développeurs.

- ✓ **Java** dispose d'un mécanisme très efficace pour obliger les développeurs à prendre en compte cet aspect de leur travail, tout en leur facilitant la tâche au maximum.
- ✓ La philosophie de **Java**, en ce qui concerne les erreurs, peut se résumer à deux principes fondamentaux :
 1. La plus grande partie des erreurs qui pourraient survenir doit être détectée par le compilateur, de façon à limiter autant que possible les occasions de les voir se produire pendant l'utilisation des programmes.
 2. Le traitement des erreurs doit être séparé du reste du code, de façon à ce que celui-ci reste lisible. L'examen d'un programme doit faire apparaître, à première vue, ce que celui-ci est censé faire lors de son fonctionnement normal, et non lorsque des erreurs se produisent.

Définition

- Une **exception** est l'apparition, en cours d'exécution d'un programme, d'une erreur qui mène généralement à l'arrêt de ce programme.
- L'**exception est déclenchée** par la machine virtuelle **Java** lorsqu'une situation inattendue survient.
- La création **et la propagation** d'une **exception** représentent le moyen grâce auquel la machine virtuelle **Java** notifie au développeur qu'une activité a échoué.
- Ce peut être tout une **gamme d'anomalies** : une division par zéro, une conversion d'un type vers un autre, une ouverture de fichier dont on n'a pas les droits, une rupture de réseau, un accès à un élément de tableau hors borne,

- 2 - Principes de base des exceptions

- Les **exceptions** ont pour objectif de **séparer le code applicatif** du **code de traitements des erreurs** pouvant survenir.
- La détection des erreurs et la gestion de leurs traitements associés sont donc mises en œuvre grâce à des **objets particuliers** : les **exceptions**.
- **Java** lui-même peut générer des exceptions (il en existe des centaines) pour notifier au développeur d'éventuelles erreurs qu'il aurait oublié de traiter.
- Mais le développeur peut aussi créer **ses propres classes d'exception**, spécifiques à son application et qui contribuent à produire des **programmes robustes**.
- Nous verrons qu'il existe deux stratégies pour traiter une **exception** : soit l'exception est **gérée localement** au code qui la détecte, soit la méthode où l'exception est levée n'est pas candidate à son traitement et **elle transmet cette exception à sa fonction appelante** qui, à son tour, va devoir faire un choix : la traiter ou de nouveau la transmettre à sa fonction appelante, etc.

- 3 - La hiérarchie des exceptions

Extrait de la hiérarchie de *Throwable*

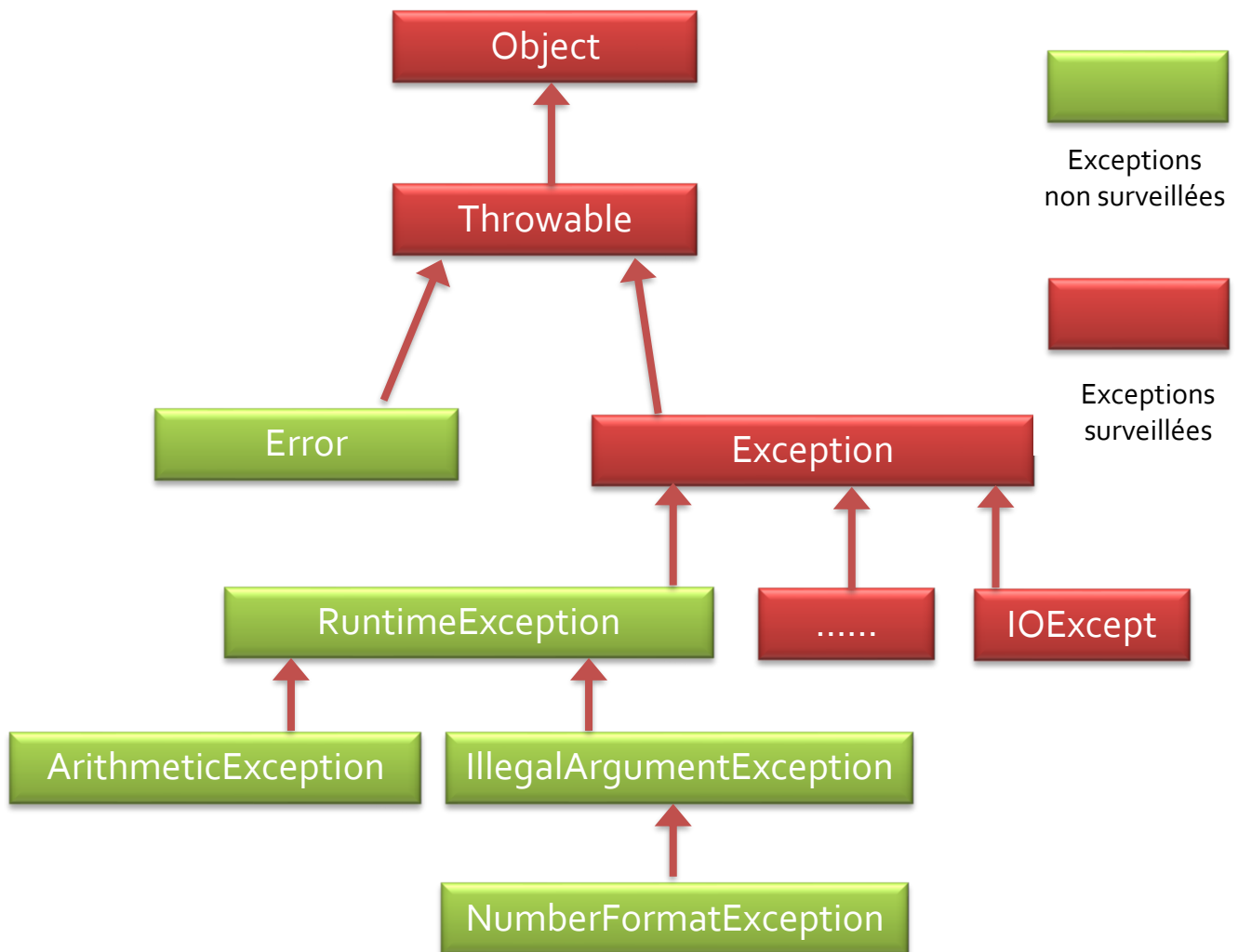


Figure 1 : Extrait de la hiérarchie des exceptions.

- **Toutes les exceptions** dérivent de la classe *Throwable*.

(to throw : jeter, lancer)

- La classe *Error* représente des erreurs graves conduisant à l'arrêt du programme et encapsule des erreurs liées à la machine virtuelle.
- La classe *Exception* représente les erreurs que le développeur peut gérer lui-même sans provoquer l'arrêt du programme.
- Ce qu'il faut bien comprendre c'est qu'il existe dans **Java** des centaines de classes d'exception, chacune dédiée à un cas d'erreur particulier.

- Il est **capital** d'avoir une fenêtre ouverte sur la **doc de Java** des exceptions lorsqu'on les étudie ou lorsque l'on développe.

Classe Error

La classe Error modélise des erreurs d'exécution d'une application que l'on ne gère pas. Citons par exemple : la saturation de la mémoire (OutOfMemoryError) ou le dépassement de capacité de la pile d'appel (StackOverflowError). La génération d'une telle erreur signifie que l'application est vraiment très malade. Sa fin de vie est en général assez proche.

Classe RuntimeException

La classe RuntimeException modélise des erreurs d'exécution d'une application que l'on ne gère pas non plus, mais qui ont un statut différent. Elles signifient qu'une opération non prévisible a eu lieu. Par exemple l'appel d'une méthode au travers d'un pointeur nul, qui va générer la bien connue NullPointerException. Autre exemple : la division par zéro (ArithmeticException) ou la lecture d'un tableau au-delà de sa limite (ArrayIndexOutOfBoundsException). Ce genre de choses n'est pas censé arriver dans une application normalement constituée.

Classe Exception

À la différence de la classe Error et la classe RuntimeException, la classe Exception modélise les erreurs d'exécution que l'on doit prévoir. Parmi elles : l'impossibilité d'ouvrir un fichier ou de se connecter à une ressource réseau. Ces erreurs sont prévisibles, et le développeur doit proposer un comportement si elles interviennent.

La classe Exception n'ajoute aucune méthode à la classe Throwable.

Exception (Java SE 21 & JDK 21 [a...]

← → ↺ 🔒 https://cr.openjdk.org/~jlaskey/templates/docs/api/java.base/java/lang/Exception.html 📄 🔍 ⭐ 🏠 ⌵

This specification is not final and is subject to change. Use is subject to license terms.

OVERVIEW MODULE PACKAGE **CLASS** USE TREE PREVIEW NEW DEPRECATED INDEX HELP

Java SE 21 & JDK 21
DRAFT 21-internal-adhoc.jlaskey open

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH 🔍 Search

Module java.base

Package java.lang

Class Exception

java.lang.Object
 java.lang.Throwable
 java.lang.Exception

All Implemented Interfaces:
Serializable

Direct Known Subclasses:
AbsentInformationException, AgentInitializationException, AgentLoadException, AlreadyBoundException, AttachNotSupportedException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CardException, CertificateException, ClassNotLoadedException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionControl.ExecutionControlException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSEException, IllegalClassFormatException, IllegalConnectorArgumentsException, IncompatibleThreadStateException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMididataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvalidTypeException, InvocationException, IOException, JMXException, JShellException, KeySelectorException, LambdaConversionException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NotBoundException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SQLException, StringConcatException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, VMStartException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

public class **Exception**
extends Throwable

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

The class Exception and any subclasses that are not also subclasses of RuntimeException are *checked exceptions*. Checked exceptions need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

See Java Language Specification:
11.2 Compile-Time Checking of Exceptions⁴⁹

Since:
1.0

See Also:

AFPA/ Novembre 2024

La gestion des exceptions en Java

5/40

- 4 - Les deux types d'erreurs en Java

Classification

En **Java**, on peut classer les erreurs qui surviennent en cours d'exécution d'un programme en **deux catégories** :

1. Les erreurs **surveillées** (accès à un fichier, ...) .
2. Les erreurs **non surveillées** (que l'on appelle les « *runtime* ») pour lesquelles il n'est pas nécessaire de prévoir systématiquement un traitement spécifique (division par zéro, accès à un élément de tableau, ...).

- **Java** oblige le programmeur à traiter les **erreurs surveillées**.
- Les erreurs **non surveillées** n'obligent pas un traitement à priori pour les gérer.

Par défaut, lorsqu'une erreur survient, la **JVM** crée une **exception caractéristique** de l'événement et la propage dans les méthodes de votre programme.

Cette exception est un **objet particulier** (nous sommes en **Java** ...), qui est une instance issue de la **hiérarchie présentée ci-dessus**, qui est envoyée par la **JVM** à destination d'une méthode qui s'est **déclarée candidate pour la gérer**.

Le comportement par défaut, consécutif à la génération de cette exception, consiste à produire un affichage décrivant le diagnostic de l'erreur.

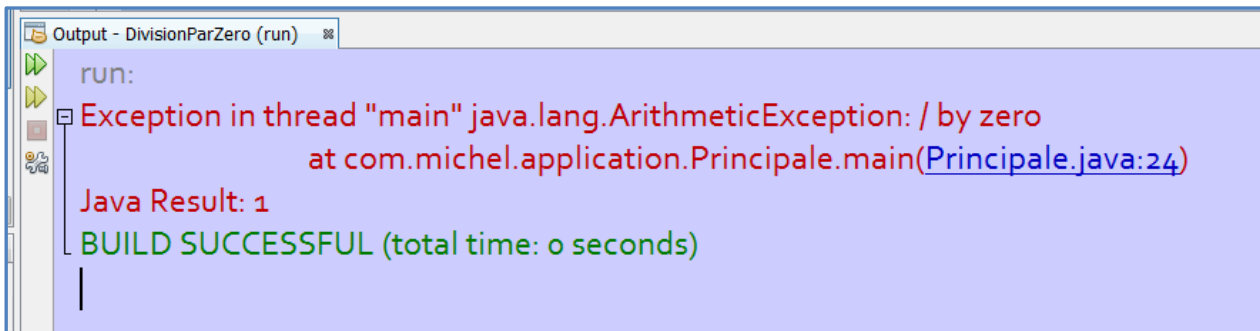
Mettons en œuvre un premier exemple d'une **erreur non surveillée**.

Erreurs non surveillées

Codons en **Java** une **division par zéro** volontaire :

```
public class Principale {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
  
        int numerateur = 10 ;  
        int denominateur = 0 ;  
        int resultat ;  
  
        resultat = numerateur/denominateur ;  
        System.out.println("Le résultat de la division est : " + resultat );  
    }  
}
```

- Exécutons le code précédent et constatons ses effets :



The screenshot shows an IDE output window titled "Output - DivisionParZero (run)". The output text is as follows:

```
run:  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at com.michel.application.Principale.main(Principale.java:24)  
Java Result: 1  
BUILD SUCCESSFUL (total time: 0 seconds)
```

- La machine virtuelle Java (**JVM**) propose dans la fenêtre de sortie le **diagnostic** suivant :
- Tout d'abord, **le fait que ce diagnostic soit affiché** indique que l'erreur a été traitée. Un message est produit dans la console et la suite de l'exécution est interrompue. Il s'agit d'un traitement exceptionnel.
- Analysons les informations affichées :

Exception in thread "main" java.lang.ArithmeticException

Ici, **Java** nous renseigne qu'une exception de classe :

java.lang.ArithmeticException a été émise et déclenchée.

- Vient ensuite l'information :

/by zero :

- **Java** produit un message indiquant **la cause de l'émission de l'exception** : « une division par zéro ».
- Ensuite, est affichée la trace de la pile d'exécution, au moment de l'exception :

at com.michel.application.Principale.main (Principale.java:24)

- Ici, nous n'avons qu'une seule méthode impliquée : *main*.



- Enfin, **Java** indique l'endroit, dans le code source, à l'origine de la levée d'exception avec le nom du fichier **Principale.java** et la ligne dans ce fichier (24).

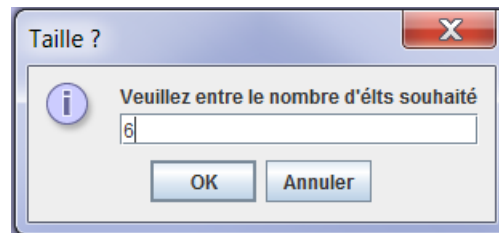
- A noter que cette référence au code source fait l'objet d'un lien hypertexte.
- On clique dessus et on est positionné dans le code source sur la ligne en question. Pratique et efficace.

Mettons en œuvre un deuxième exemple d'une erreur non surveillée.

- Le code suivant est un extrait de l'exemple (mis en œuvre précédemment dans le support dédié aux collections) de la création d'un tableau d'entiers dont la taille est demandée à l'utilisateur **dynamiquement**, à travers une boîte de dialogue.
- A noter que la variable **taille**, affectée par le retour de la fermeture de la boîte de dialogue est de type **String**.
- Puisque que l'on se sert de cette information pour dimensionner le tableau, il faut d'abord convertir cette **String** en un entier (**tailleTableau**). On le fait en instanciant un **Integer** et, par la propriété d'**unboxing** de **Java**, on affecte l'entier **tailleTableau**.

```
{ ...  
// Variables nécessaires à la manipulation du tableau  
int tailleTableau = 0;  
String taille;  
int[ ] tableau;  
  
taille = showInputDialog(null, "Veuillez entrer le nombre d'élts souhaité", "Taille ?",  
                        INFORMATION_MESSAGE);  
  
tailleTableau = new Integer(taille);  
  
tableau = new int[tailleTableau];  
  
for (int i = 0; i < tableau.length; i++) {  
    tableau[i] = (int) (Math.random() * 50 + 1);  
}  
  
afficher(tableau);  
}  
  
public static void afficher(int[] tableau) {  
    String resultat = "";  
    int taille = tableau.length;  
    for (int i = 0; i < tableau.length; i++) {  
        resultat += "tableau[" + i + "] = " + tableau[i] + "\n";  
        System.out.println("tableau[" + i + "] = " + tableau[i]);  
    }  
    showMessageDialog(null, resultat, "Tableau de " + taille + " entiers",  
                     INFORMATION_MESSAGE);  
}  
}
```

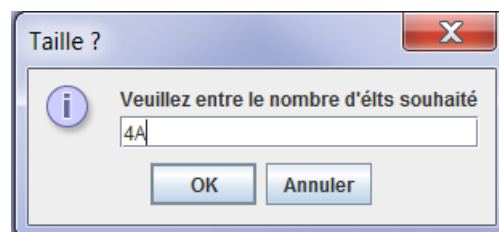
- Exécutons le programme précédent :



- On a bien saisi une chaîne de caractères («6») dont la conversion vers un entier équivalent ne posera pas de problème. Le tableau peut être créé, rempli et affiché.
- La preuve :



- Exécutons de nouveau le programme :



- Validons la boîte, après avoir saisi la chaîne «4A».
- **Java** affiche le diagnostic suivant dans la console :

```
Output - ConversionStringEntier (run) #2
run:
Exception in thread "main" java.lang.NumberFormatException: For input string: "4A"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.<init>(Integer.java:677)
    at conversionstringentier.ConversionStringEntier.main(ConversionStringEntier.java:35)
Java Result: 1
BUILD SUCCESSFUL (total time: 5 seconds)
```

- Le résultat est implacable : une **exception** a été générée.
- Examinons, comme précédemment, les informations fournies :

- Une **exception** de classe `java.lang.NumberFormatException` a été émise.

- La raison: For input string : «4A».

... « Pour une chaîne de caractères saisie « 4A ».

- S'ensuit derrière l'état de la pile, c'est-à-dire les **appels successifs** aux méthodes des différentes classes impliquées.
- La dernière information est capitale, c'est l'endroit à l'origine de l'exception :

```
at conversionstringentier.ConversionStringEntier.main(CConversionStringEntier.java :35)
```

- On constate qu'une **nouvelle instance d'exception** a été produite par **Java** suite à notre erreur de saisie. C'est une exception de type *NumberFormatException*.
- On remarquera également qu'à travers ces exemples, **Java** ne nous a rien imposé concernant la mise en place préventive d'un dispositif de traitement de ces incidents.
- C'est tout simplement parce que ces deux classes d'exception étudiées (*ArithmeticException* et *NumberFormatException*) descendent toutes deux de la branche :

RuntimeException

- A partir de ce niveau hiérarchique de **l'API** des exceptions, toutes les classes d'exception héritées **ne sont pas surveillées**.
- C'est-à-dire que **Java** n'impose pas au développeur une gestion préventive pour traiter ces éventuels dysfonctionnements.



- « N'impose pas » ne veut pas dire « **interdit** ».
- En d'autres termes, il est possible de mettre en place un **gestionnaire d'exception** pour ces catégories d'exceptions non surveillées.

- Nous allons le faire pour les deux exemples précédents après avoir vu la structure syntaxique **Java** de mise en œuvre de ce gestionnaire.

- 5 - Présentation du bloc *try/catch*

- Le traitement classique des **exceptions**, qu'elles soient **surveillées** ou **non surveillées**, consiste à :
 - Repérer le bloc de code susceptible et potentiellement **générateur d'une erreur** ou **exception** et ...
 - Entourer ce bloc par la structure suivante, qui est le **canevas classique** et **universel**, quelle que soit la classe d'exception générée par **Java**.
- Cette structure est la suivante. Elle se nomme le bloc «**try/catch**». Elle est incontournable en **Java** et représente **le fondement des gestionnaires d'exception**.

```
try {  
    // bloc d'instructions du try  
    ....  
}  
catch ( ClassException ce) {  
    // bloc d'instructions du catch  
    // Gestion de l'exception  
    ....  
}  
....  
// suite du programme ....  
.....
```

- Traduisons **littéralement** cette construction programmatique :

« J'essaie d'exécuter les instructions du bloc **try**, dont je sais qu'une (ou plusieurs) d'entre elles peut (peuvent) être à l'origine de l'émission d'une (ou plusieurs) exception(s), déclenchée(s) par une anomalie. Si au moins une exception est déclenchée pendant l'exécution du bloc **try**, alors je l'**intercepte** par le biais du bloc **catch** (attrape) qui lui correspond. »

- A noter que le bloc **catch** spécifie une **catégorie d'exception précise** : toute exception de type **ClassException** (ou une de ses descendantes, par héritage) levée dans le bloc **try** est ici interceptée.
- A nous, développeurs, de placer du code pertinent pour remédier au dysfonctionnement constaté.
- Si, dans le bloc **try**, plusieurs catégories d'exceptions peuvent potentiellement être levées, alors, on peut placer autant de blocs **catch** correspondant à la nature de chacune des catégories d'exception : en d'autres termes, on peut spécialiser un traitement particulier à chaque catégorie d'exception.

- Quand une **exception** est émise (ou levée) dans le bloc **try**, la **JVM** parcourt tous les blocs **catch** à la recherche de celui spécifique et dédié pour cette exception particulière.
- On a, chronologiquement, les étapes suivantes dans la survenue d'une exception :



1. **Emission** par **Java** de l'exception survenue dans le bloc **try**.
2. **Interception** de cette exception dans le bloc **catch** qui lui est consacré.
3. **Exécution** du code présent dans le bloc **catch**.
4. La suite des instructions du programme se poursuit après le bloc **try/catch**.
5. Les instructions qui sont présentes dans le bloc **try** derrière celle qui a provoqué l'exception **sont purement abandonnées**.

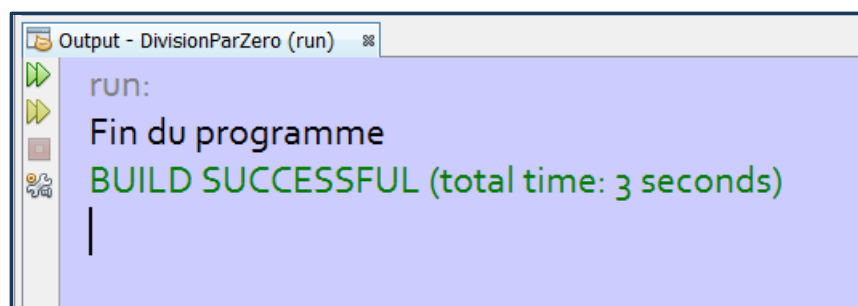
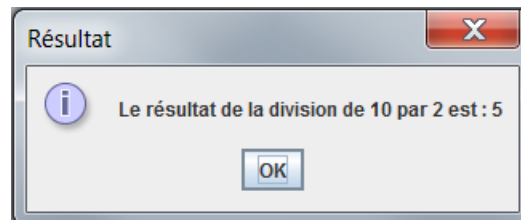
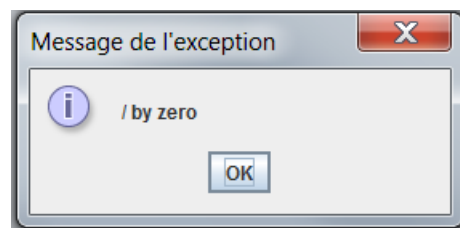
- Reprenons nos deux exemples et remédions au dysfonctionnement :

Premier exemple : la division par zéro

```
public class Principale {  
  
    public static void main(String[] args) {  
  
        int numerateur = 10;  
        int denominateur = 0;  
        int resultat = 0;  
  
        try {  
            resultat = numerator/denominateur ;  
            System.out.println("Suite du bloc try"); // Instruction jamais exécutée  
        }  
        catch (ArithmeticException ae ) {  
            showMessageDialog(null,ae.getMessage(), "Message de l'exception",  
                INFORMATION_MESSAGE);  
            denominateur = 2 ;  
            resultat = numerateur/denominateur ;  
        }  
  
        showMessageDialog(null, "Le résultat de la division de " + numerateur + " par " +  
            denominateur +" est : " + resultat ,"Résultat", INFORMATION_MESSAGE );  
  
        System.out.println("Fin du programme");  
    }  
}
```

- Cette fois, nous avons **encadré** la division dans un bloc **try**.
- On a prévu le traitement particulier suivant : si une **ArithmeticException** (lors de l'exécution du bloc **try**) survient, alors nous la capturons dans le bloc **catch** correspondant. Et nous la traitons.
- On décide de produire un affichage avec une boite de dialogue qui renseigne l'utilisateur sur la nature de l'incident :
- Toute exception dispose par héritage de la méthode **getMessage()** qui renvoie une **String** concernant la cause de l'exception.
- D'une manière totalement arbitraire, on affecte la valeur 2 à la variable **denominateur** et l'on recommence la division.
- Puis, on sort du bloc **catch** et l'on poursuit en séquence l'exécution du programme.

- Examinons les boîtes de dialogue affichées durant l'exécution du programme ci-dessus.



- Il est important de noter que l'instruction d'affichage présente dans le bloc *try* :

```
System.out.println("Suite du bloc try");
```

... se situant derrière celle qui a provoqué l'exception n'est pas exécutée : c'est ce que nous avons dit, **la suite du bloc est abandonnée**.

- En revanche, on constate que derrière le traitement d'une exception, on poursuit l'exécution du programme. (Affichage de « Fin du programme »).
- Apportons une correction maintenant pour l'exemple N°2 :

Deuxième exemple : la conversion d'un entier sécurisée.

```
public class ConversionStringEntier {

    public static void main(String[] args) {

        // Variables nécessaires à la manipulation du tableau
        int tailleTableau = 0;
        String taille;
        int[] tableau;

        boolean encore = true;

        while (encore == true) {

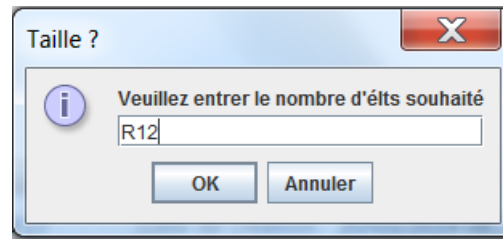
            taille = showInputDialog(null, "Veuillez entrer le nombre d'élts souhaité", "Taille
                ?", INFORMATION_MESSAGE);

            try {
                tailleTableau = new Integer(taille);
                encore = false;
            }
            catch (NumberFormatException nfe) {
                showMessageDialog(null, "Veuillez entrer un entier SVP ", "Entier",
                    WARNING_MESSAGE);
            }
        }
        tableau = new int[tailleTableau];

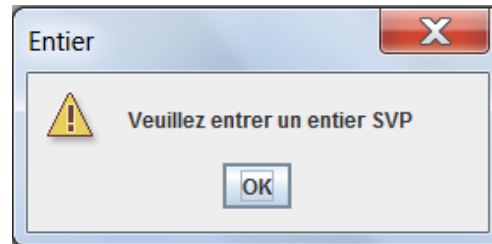
        for (int i = 0; i < tableau.length; i++) {
            tableau[i] = (int) (Math.random() * 50 + 1);
        }
        afficher(tableau);
    }

    public static void afficher(int[] tableau) {
        String resultat = "";
        int taille = tableau.length;
        for (int i = 0; i < tableau.length; i++) {
            resultat += "tableau[" + i + "] = " + tableau[i] + "\n";
            System.out.println("tableau[" + i + "] = " + tableau[i]);
        }
        showMessageDialog(null, resultat, "Tableau de " + taille + " entiers",
            INFORMATION_MESSAGE);
    }
}
```

- Saisissons un entier erroné :



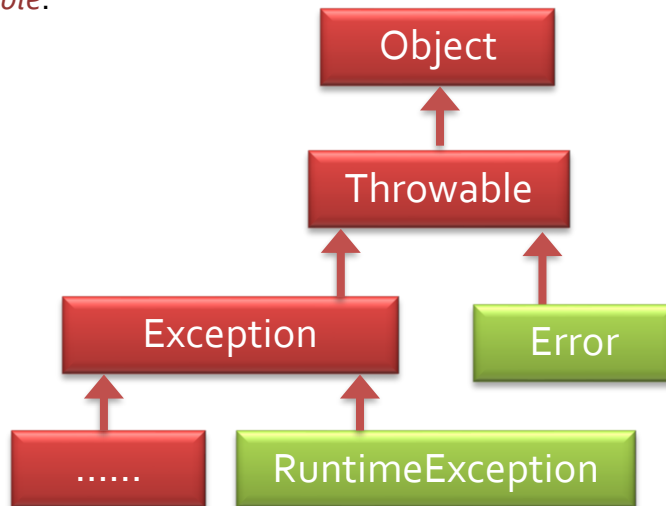
- Le gestionnaire d'exceptions prévu dans le bloc **catch** s'exécute : il nous invite à entrer un entier par la suite.



- Après l'exécution du bloc **catch**, on poursuit en séquence : ici, cela consiste à boucler tant qu'un entier n'est pas correctement saisi.

- 6 - Après quelques exercices de mise en œuvre : quelques précisions.

- Les exceptions sont donc des objets **Java** issus de classes héritant de la classe *Throwable*.



- Le nom de cette classe indique qu'une exception est « **jetable** » : toute exception, après son instantiation, est « jetée » ou « lancée » via l'empilement des appels de méthodes à la recherche d'un gestionnaire dédié pour la traiter.
- De la classe *Throwable* dérivent deux autres classes correspondant à deux branches : *Error* pour les erreurs graves (que l'on ne traite généralement pas) et *Exception*. La classe *RuntimeException* hérite d'*Exception* et est à la base des erreurs **non surveillées**.
- Tout objet de type *Throwable* gère un message d'erreur traduisant les caractéristiques de l'exception. Elle propose également un certain nombre de méthodes dédiées à la *stack trace* de l'exception que l'on approfondira.
- L'interprétation de cette *stack trace* est très importante et aide à résoudre les erreurs inévitables présentes dans le code.
- Comme nous l'avons précisé, cette *stack trace* représente la pile des appels du programme à l'instant où l'exception est émise.
- On retrouve, comme l'expérience des exemples ci-dessus nous l'a montré, les noms de classes impliquées, ainsi que leur package d'appartenance.
- Contrairement aux exceptions issues de la branche *Error*, toute exception issue de *Exception* et toutes ses descendantes modélise les erreurs d'exécution qu'il est raisonnable de traiter.

- Toute classe d'exception à partir d'*Exception* sont **surveillées** et imposent un traitement particulier.
- En revanche, à partir de la classe *RuntimeException*, les exceptions **ne sont pas surveillées**.
- Parmi elles, citons : déclenchement d'une méthode sur un référence nulle (*NullPointerException*), division par zéro (*ArithmeticException*), connexion réseau défectueuse, conversion de type impossible, ...)
- Ces erreurs sont **prévisibles**, et le développeur doit proposer un comportement si elle intervient.

- 7 - Déclenchement d'une exception

- Une **exception** qui se produit pendant l'exécution d'un programme peut avoir deux **origines possibles** :
 - L'exécution nominale du programme ne se passe comme prévu, et c'est la machine virtuelle de **Java** qui génère une exception (parmi toutes celles de l'**API**.)
 - Dans son code, le développeur a prévu - lors d'une situation particulière - d'émettre lui-même une exception **adaptée à l'anomalie** et caractérisant cet événement. Il notifie, à partir de la méthode où se produit l'incident, à la méthode appelante qu'un événement d'erreur a été constaté en émettant une exception que quelque chose ne se déroule pas comme prévu.
- Lorsqu'une erreur est rencontrée, **Java** crée immédiatement un objet, instance d'une classe particulière, sous-classe de la classe **Throwable**.
- Cet objet est créé normalement à l'aide de l'opérateur **new**. Puis, il est lancé grâce au mot-clé **throw**. Ensuite, l'interpréteur part à la recherche d'une portion de code capable de recevoir cet objet et d'effectuer le traitement approprié.
- S'il s'agit d'une erreur surveillée par le compilateur, celui-ci a obligé le programmeur à fournir ce code. Dans le cas contraire (erreur non surveillée), le traitement est fourni par l'interpréteur lui-même.
- Cette opération est appelée « lancement (en anglais **throw**) d'une exception », par analogie avec le lancement d'un objet qui doit être attrapé par le code prévu pour le traiter.
- Pour trouver le code capable de traiter l'objet, l'interpréteur se base sur **le type de l'objet**, c'est-à-dire sur la classe dont il est une instance. S'il existe un tel code, l'objet lui est transmis, et l'exécution se poursuit à cet endroit.
- Il faut noter que le **premier bloc de code capable de traiter** l'objet événement, reçoit celui-ci et le gère.
- Du fait du **polymorphisme**, il peut s'agir d'un bloc de code traitant une classe parente de celle de l'objet en question.
- Dans le cas de notre programme précédent, une instance de la classe **ArithmeticException** est donc lancée.

- 8 - Exceptions déclenchées par Java

- Nous avons déjà expérimenté quelques exemples de déclenchement d'exceptions émises par Java :

`ArithmeticException` (division par zéro)

`NumberFormatException` (conversion d'un entier impossible).

- Il en existe beaucoup d'autres parmi lesquelles, les très classiques :

`NullPointerException` (tentative d'envoi de message sur une référence nulle)

`IndexOutOfBoundsException` (accès illégal à un élément de tableau)

- 9 - Exceptions déclenchées par l'application

- Le développeur peut lui-même décider de prévoir dans son code **le déclenchement d'exceptions**.

Quelles exceptions peut-il émettre ?

- Naturellement : toute exception issue de la hiérarchie *Throwable* de l'API **Java**.
- Mais aussi une exception d'une classe dont il **est l'auteur** et qui représente une anomalie spécifique à son application.
- Voyons cela sur l'exemple suivant où il lève une exception de l'API de **Java**.

```
....  
if ( magasin.getStock() < 10) {  
    throw (new Exception("Seuil du stock critique")) ;  
}  
....
```

- Comme on va bientôt le constater, le développeur peut donc lui-même créer **ses propres classes d'exceptions**, totalement adaptées à son code.
- On a pu vérifier que le mot-clé pour émettre une exception, lors d'un constat particulier est **throw**. C'est l'un des mots-clés de **Java**.
- Il doit être accompagné d'une instance issue de l'arborescence des exceptions : ici

```
throw (new Exception (« Seuil du stock critique »)) ;
```

- Ici dans l'exemple, on instancie la classe *Exception* en passant une chaîne à son constructeur, et l'on « jette » (**throw**) cette exception par l'objet qu'on lui passe en paramètre.

- 10 - Propager une exception : la clause *throws*

- Nous avons vu que le mot-clé **Java** pour émettre une exception est *throw*.
- Ce mot-clé ne doit pas être confondu avec le mot-clé *throws*.
- En **Java**, on indique au code appelant qu'une méthode est susceptible de lever une exception grâce au mot clé *throws*.
- Une méthode signale cet éventuel envoi d'exception **lors de son fonctionnement** grâce à la fourniture de mot-clé *throws* dans sa signature.
- Reprenons l'exemple de la conversion d'une chaîne de caractères en un entier qui s'appuie sur la construction d'un *Integer* à partir de cette chaîne. Examinons la documentation de **Java** qui précise la configuration du constructeur de *Integer* :

`Integer public Integer(String s) throws NumberFormatException`

Constructs a newly allocated Integer object that represents the int value indicated by the String parameter. The string is converted to an int value in exactly the manner used by the parseInt method for radix 10.

Parameters:

s - the String to be converted to an Integer.

Throws:

NumberFormatException - if the String does not contain a parsable integer.

- Il est précisé que l'instanciation d'un *Integer* peut lever une *NumberFormatException* si la chaîne fournie n'est pas convertible en un *Integer*. Cette notification est réalisée en précisant dans l'entête du constructeur la clause *throws* suivie de la classe d'exception potentiellement émise.

Autre cas : la méthode *read* de la classe *InputStreamReader*

```
public int read(char[] cbuf, int offset, int length) throws IOException
```

Reads characters into a portion of an array.

Specified by:

read in class Reader

Parameters:

cbuf - Destination buffer

offset - Offset at which to start storing characters

length - Maximum number of characters to read

Returns:

The number of characters read, or -1 if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

- La méthode *read* de la classe *InputStreamReader* spécifie que sa méthode peut lever une *IOException*.
- Cette information, que l'on trouve dans l'entête des méthodes ou des constructeurs des classes est **capitale** : elle notifie au développeur qu'en utilisant ces méthodes (ou constructeurs), il s'expose à recevoir potentiellement une exception.
- Cette exception, précisée derrière la clause *throws* est soit non surveillée (elle hérite de *RuntimeException*), soit surveillée : elle hérite de *Exception*.
- C'est précisément le cas ici : *IOException* est une classe **d'exception surveillée**.
- Ceci implique qu'en utilisant cette méthode *read*, le développeur est contraint d'une **manière obligatoire** de prévoir un traitement approprié pour gérer cette exception.
- La réponse à ce traitement est la mise en œuvre du bloc *try/catch*.
- En revanche, s'il décide de ne pas traiter cette exception et donc de ne pas fournir le bloc *try/catch* alors, à son tour, il va signaler dans sa méthode qui fait appel à la méthode *read* qu'il propage à sa méthode appelante cette exception (*throws*).

- Il le fera en ajoutant dans l'entête de sa méthode la clause **throws** suivie de l'exception qu'il ne souhaite pas gérer et qui peut être émise par **read**.

Exemple simple :

- La plupart des méthodes des classes qui gèrent les fichiers émettent potentiellement des exceptions.
- La classe **FileReader**, qui permet de lire des fichiers, déclare par exemple dans l'entête de son constructeur l'exception suivante :

```
public FileReader(String fileName) throws FileNotFoundException
```

- Si l'on tente de construire une instance de **FileReader** en fournissant un nom de fichier qui n'existe pas, notre objet ne sera pas construit, et une exception du type **FileNotFoundException** sera générée.
- On notera la différence entre la génération manuelle d'une exception qui utilise le mot-clé **throw** (sans **s**), et la déclaration au niveau de la signature d'une méthode, qui utilise le mot-clé **throws** (avec un **s**).
- Dans un programme **Java**, toute exception jetée explicitement doit être soit propagée vers la méthode appelante, soit traitée localement.

Exemple de traitement de l'exception *FileNotFoundException*

1. Je traite **localement** l'exception liée au constructeur de *FileReader* évoqué ci-dessus.

```
public void lireFichier ( String nomFichier ) {  
    try {  
        FileReader fichier = new FileReader(nomFichier);  
    }  
    catch ( FileNotFoundException fnfe ) {  
        System.out.println( « Le fichier » + nomFichier + « n'existe pas ! » ) ;  
    }  
    ...  
}
```

2. Je ne **suis pas intéressé** pour traiter l'exception : je propage l'exception à la méthode appelante :

```
public void lireFichier ( String nomFichier ) throws FileNotFoundException {  
    FileReader fichier = new FileReader(nomFichier) ;  
}
```

- Dans ce cas, on constate que l'exception *FileNotFoundException* potentiellement émise par le constructeur de *FileReader* est propagée à la méthode appelante.
- Celle-ci pourra, à son tour, la traiter ou la propager. Il est possible de spécifier derrière la clause **throws** autant d'exceptions que l'on souhaite. On le fait en spécifiant les noms des classes d'exception séparés par une virgule.
- Ceci signifie que dans une méthode plusieurs catégories d'exception peuvent être émises et que l'on peut propager vers la méthode appelante ces exceptions.

- 11 Capturer une exception : autres cas

Lorsque l'on décide de capturer une **exception**, on émet le souhait implicite de ne pas confier à la méthode appelante le soin de la gérer (on ne la propage pas) : en d'autres termes, on **décide autoritairement de prendre la main** sur les actions à entreprendre dans une telle situation.

Il est alors d'une importance capitale pour le développeur de **notifier** à l'utilisateur, aux équipes de développement et à l'application elle-même les circonstances et le contexte de cette exception qui représente, rappelons-le, **un incident plus ou moins grave**.

Chaque exception étant d'un type particulier, dotée d'un message, porteuse de l'ensemble des méthodes impliquées, du fichier initial et de la ligne où elle s'est produite, il est possible de notifier à l'utilisateur à l'aide d'une page dédiée ou d'une boîte de dialogue affichée avec toutes ces informations, **l'environnement complet de la nature de l'incident**.

De même, il est tout à fait recommandé **d'historiser** dans les fichiers de logs ou dans une base de données tout le contexte d'apparition de l'exception.

Précisons maintenant comment le bloc `try/catch` peut être étendu.

Pour rappel, le bloc **`try`** contient l'ensemble des instructions liées fonctionnellement entre elles et représente le code potentiellement à l'origine de l'émission d'une exception particulière.

Lorsqu'une exception est déclenchée au sein de ce bloc, **Java** parcourt les blocs **`catch`** jusqu'à en trouver un qui correspond au type de l'exception émise. Lorsque ce bloc **`catch`** est trouvé, **Java** exécute les instructions se trouvant dans son bloc associé.

On peut donc affirmer, consécutivement à la lecture précédente, que plusieurs blocs **`catch`** **`}`** peuvent être fournis l'un derrière l'autre pour un même et unique bloc **`try`**. Il n'y a pas de limite théorique.

Le premier bloc **`catch`** trouvé correspondant au type de l'exception générée est alors exécuté. Attention : si un bloc **`catch`** est fourni avec un type d'exception plus haut dans la hiérarchie des exceptions par rapport au type d'exception réellement déclenchée alors ce bloc est exécuté. On peut dire que, par lien de parenté ou héritage, un bloc **`catch`** avec un type d'exception parent peut être vu comme un collecteur de toute exception descendante de ce parent.

Si un bloc plus précis est situé derrière ce bloc « englobant » il ne sera jamais atteint.

Cette situation ne devrait jamais se produire, **Java** interdisant à la compilation ce type d'organisation des blocs **catch**.

Si aucun bloc **catch** ne convient parmi tous ceux fournis, alors **Java** propage à la méthode appelante. A-elle de prendre les mesures adéquates.

Il est possible également de prévoir derrière le dernier bloc **catch** la clause **finally**.

Cette clause représente un bloc incontournable par lequel passe **Java** et exécute les instructions s'y trouvant quelle que soit la situation : qu'il y ait eu levée d'exception ou non, on passera inévitablement par ce bloc, cela permet de **libérer des ressources** qui auraient été ouvertes préalablement :

```
....  
try {  
    ....  
}  
catch ( ClassException ce ) {  
    ...  
}  
finally {  
    // Passage systématique dans ce bloc finally  
}  
...
```

Reprenons les étapes chronologiques susceptibles de se passer dans l'exécution d'un bloc **try/catch**.

- Exécution du bloc **try** et de ses instructions.
- Si aucune exception n'est déclenchée dans le bloc **try** alors le programme se poursuit après le dernier bloc **catch** (situation nominale).
- Si la clause **finally** est prévue alors son code est exécuté. Pour rappel : le contenu d'un bloc **finally** est systématiquement exécuté lors de la fin de l'exécution d'un bloc **catch**.
- Si une exception est générée dans le bloc **try**, et qu'il existe un bloc **catch** correspondant à son type, alors ce bloc est exécuté.

- Si une exception est générée dans le bloc *try* et qu'il n'existe aucun bloc *catch* correspond à son type, alors, si le bloc *finally* existe, il est exécuté, et l'exception est propagée au code appelant.
- Si une exception est générée, qu'aucune clause *catch* ne lui correspond et qu'il n'y a pas de bloc *finally*, alors l'exception est propagée par la clause *throws* déclarée dans la méthode.

RESUME

- Une **exception** est un événement lié à une **situation anormale** qui modifie ou interrompt le flux nominal d'exécution d'un programme.
- **Java** fournit un gestionnaire d'exceptions pour remédier à une telle situation.

La gestion des exceptions **doit être découplée du déroulement normal** d'un programme **Java**.

La hiérarchie d'exceptions

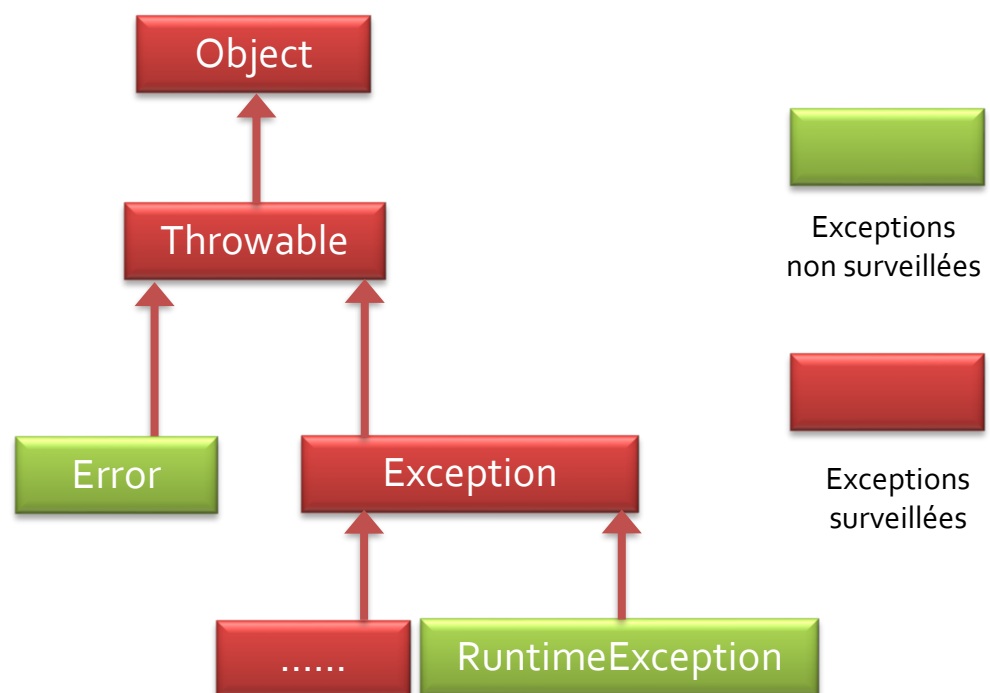


Figure 2. Extrait de la hiérarchie des exceptions

Comme le montre la figure 2, toutes les exceptions et erreurs de l'API **Java** héritent de la classe **Throwable**, qui hérite elle-même de la classe **Object**.

Exceptions surveillées, non surveillées et les *Errors*

Les *exceptions* et les *erreurs* en *Java* se répartissent en *trois catégories* : les exceptions *surveillées*, les exceptions *non surveillées* et les *erreurs (Error)*.

■ Les exceptions *surveillées*

- Les exceptions surveillées sont vérifiées par le compilateur de *Java*.
- Les méthodes qui peuvent lever une exception surveillée doivent l'indiquer dans leur entête en précisant la clause *throws*.
- Toutes les exceptions surveillées doivent explicitement être contrôlées avec un bloc *catch*.
- L'exception remonte toute la pile d'appel de méthodes jusqu'à ce qu'un gestionnaire d'exception soit trouvé.
- Les exceptions surveillées comprennent toutes les exceptions du type *Exception* et ses sous-classes, sauf la classe *RuntimeException* et ses sous-classes.

Exemple de méthode qui peut lever une exception surveillée :

```
...  
// Méthode levant une IOException  
void readFile(String filename) throws IOException {  
    ...  
}  
...
```

■ Les exceptions *non surveillées*

- Le compilateur *Java* ne vérifie pas les exceptions non surveillées.
- Les exceptions non surveillées se produisent pendant l'exécution d'un programme *Java* et sont dues à des erreurs de logique du développeur (indice d'un tableau hors limites, division par zéro, méthode sur une référence *null*, ...) ou par exemple atteinte des limites des ressources du système.
- Les exceptions non surveillées n'ont pas l'obligation d'être gérées par un *catch*.
- Les méthodes qui peuvent lever une exception non surveillée ne doivent pas (mais peuvent) l'indiquer dans leur entête.
- Les exceptions non surveillées sont les exceptions du type *RuntimeException* et tous les sous-types de *RuntimeException*.

■ Les *Errors*

- Les erreurs (*Errors*) sont généralement liées à des événements graves et représentent souvent une *situation irréversible*.
- Les erreurs *ne sont pas surveillées* au moment de la compilation et n'ont pas la vocation à (mais peuvent) être capturées (*catch*) et gérées.

Les exceptions surveillées, non surveillées et les *Errors* classiques

Il existe une très grande diversité d'exceptions surveillées, non surveillées, et d'erreurs non vérifiées (*Errors*) qui font partie de la plate-forme **Java** standard.

Certaines sont plus rencontrées que d'autres. Faisons un bref résumé de celles-ci.

▪ Exceptions *surveillées* communes

<i>ClassNotFoundException</i>	Levée lorsqu'une classe ne peut pas être chargée car sa définition ne peut être trouvée.
<i>IOException</i>	Levée lorsqu'une opération d'entrée/sortie s'interrompt ou échoue. Deux sous-classes communes d' <i>IOException</i> : <i>EOFException</i> et <i>FileNotFoundException</i> .
<i>FileNotFoundException</i>	Levée lorsqu'une ouverture se fait sur un fichier non trouvé.
<i>SQLException</i>	Levée pour une erreur liée à la base de données.
<i>InterruptedException</i>	Levée lorsqu'un thread est interrompu.
<i>NoSuchMethodException</i>	Levée lorsqu'une méthode appelée n'est pas trouvée.
<i>CloneNotSupportedException</i>	Levée lorsque la méthode <i>clone()</i> est appelée sur un objet non clonable.

■ Exceptions **non surveillées** communes

<i>ArithmeticException</i>	Levée lorsqu'une erreur arithmétique survient.
<i>ArrayIndexOutOfBoundsException</i>	Levée lorsqu'un indice de tableau hors-plage est utilisé.
<i>ClassCastException</i>	Levée lors d'un casting d'un objet vers une sous-classe dont il n'est pas une instance.
<i>IllegalArgumentException</i>	Levée pour indiquer qu'un argument invalide a été transmis à une méthode.
<i>IndexOutOfBoundsException</i>	Levée lorsqu'un indice hors-plage est utilisé.
<i>NullPointerException</i>	Levée lorsque le code référence un objet <i>null</i> alors qu'un non <i>null</i> est attendu.
<i>NumberFormatException</i>	Levée pour indiquer qu'une conversion d'une chaîne vers un type numérique a échoué.

■ Les **Errors** communes

<i>AssertionError</i>	Levée lorsqu'une assertion a échoué.
<i>ExceptionInInitializerError</i>	Levée lorsqu'une erreur dans un initialiseur static survient.
<i>VirtualMachineError</i>	Levée pour indiquer une erreur de la JVM.
<i>OutOfMemoryError</i>	Levée pour indiquer un manque de mémoire lors de l'allocation d'un objet.
<i>NoClassDefFoundError</i>	Levée lorsque la JVM ne peut trouver la définition d'une classe.
<i>StackOverflowError</i>	Levée lorsqu'un dépassement de la pile est atteint.

Rappels des principes des gestionnaires d'exceptions

- En **Java**, le code de gestion d'une exception **doit être séparé** du code qui génère cette exception.
- On dit que le code qui génère l'exception « **lève** » une exception, alors que l'on dit que le code qui gère l'exception « **attrape** » ou « **capture** » cette exception.

Exemple :

```
// Déclaration d'une levée d'exception par methodA()
public void methodA() throws IOException {
    ...
    throw new IOException();
    ...
}

// Capture d'une exception
public void methodB() {
    ...
    /* L'appel à la methodA() doit être réalisé dans un bloc try/catch
    ** puisque l'exception levée par celle-ci est surveillée;
    */
    try {
        methodA();
    } catch (IOException ioe) {
        System.err.println(ioe.getMessage());
        ioe.printStackTrace();
    }
}
```

Le mot-clé **throw**

- Pour déclencher une exception, il faut utiliser le mot-clé **throw**. N'importe quelle exception surveillée, non surveillée ou **Error** peut être levée.

```
if ( n==1)
    throw new EOFException();
```

Rappel des règles de la structure *try/catch/finally*

- Les exceptions sont gérées en **Java** par un bloc *try, catch, finally*.
- La **JVM** recherche le code qui s'est déclaré candidat pour gérer l'exception, explorant d'abord dans les éventuels blocs *catch*, et en remontant dans la pile pour propager l'exception si besoin.
- Si l'exception n'est pas gérée dans le code, le programme s'arrête et une trace de la pile est imprimée (comportement par défaut).
- L'instruction *try-catch* comprend un bloc *try* et un ou plusieurs blocs *catch*.
- Le bloc *try* contient le code susceptible de lever des exceptions.
- Toute exception surveillée qui peut être émise doit avoir un bloc *catch* permettant de gérer l'exception.
- Si aucune exception n'est levée, le bloc *try* se termine normalement.

```
try {  
    method();  
}  
catch (EOFException eofe) {  
    eofe.printStackTrace();  
}  
catch (IOException ioe) {  
    ioe.printStackTrace();  
}  
finally {  
    // Nettoyage, mise à jour ,...}
```

- Il ne peut y avoir de code entre le bloc *try* et l'un des blocs *catch* ou le bloc *finally*
- Le(s) bloc (s) *catch* contient du code pour gérer les exceptions levées et donc capturées y compris l'impression des informations sur l'exception dans un fichier, ou permettre à l'utilisateur la possibilité de fournir des informations correctes ...
- A noter que les blocs *catch* ne doivent jamais être vides car cette gestion d'exception « silencieuse » et masquée rend les erreurs difficiles à déboguer.
- Une convention commune consiste à nommer le paramètre de la clause *catch* (le nom de l'exception) avec l'ensemble de lettres représentant chacun des mots du nom de la classe d'exception. Par exemple :

```
...
catch (ArrayIndexOutOfBoundsException aioobe) {
    aioobe.printStackTrace();
}
...
```

- L'ordre des blocs *catch* dans une structure *try/catch* définit la préséance pour la capture d'exceptions : il faut toujours commencer par prévoir le type d'exception spécifique et finir avec un bloc *catch* capturant le type d'exception le plus général.
- Une exception levée dans un bloc *try* est dirigée vers la première clause *catch* contenant l'argument du même type ou du type d'une classe supérieure de cette exception : le bloc *catch* avec le paramètre *Exception* doit toujours être le dernier de la liste ordonnée de ces *catch*.

L'instruction *try-catch-finally*

- L'instruction *try-catch-finally* comprend un bloc *try*, un ou plusieurs blocs *catch*, et un bloc *finally*.
- Le bloc *finally* est typiquement utilisé pour le nettoyage et la libération des ressources :

```
public void testMethod() {
    FileWriter fileWriter = null;
    try {
        fileWriter = new FileWriter("\\data.txt");
        fileWriter.write("Information...");
    } catch (IOException ex) {
        ex.printStackTrace();
    } finally {
        try {
            fileWriter.close();
        } catch (Exception e) {
            e.printStackTrace();
        } // catch
    } // finally
} // testMethod()
```

- Ce bloc *finally* est optionnel et est utilisé uniquement en cas de besoin.
- Lorsqu'il est utilisé, il est exécuté en dernier dans un *try-catch-finally*. Il est systématiquement exécuté, qu'une exception ait été émise (et capturée) ou pas.
- Si le bloc *finally* lève une exception, elle doit être gérée.

L'instruction *try-with-resources*

- L'instruction *try-with-resources* est utilisée pour déclarer des ressources qui doivent être fermées impérativement quand elles ne sont plus nécessaires.
- Ces ressources sont déclarées dans le bloc *try* :

```
...  
  
static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}  
...  
// le code ci-dessus remplace avantageusement :  
static String readFirstLineFromFileWithFinallyBlock(String path)  
    throws IOException {  
    BufferedReader br = new BufferedReader(new FileReader(path));  
    try {  
        return br.readLine();  
    } finally {  
        if (br != null) br.close();  
    }  
}  
....
```

- Toute ressource qui implémente l'interface *AutoClosable* peut être utilisée avec l'instruction *try-with-resources*.

La clause *multi-catch*

- La clause *multi-catch* est utilisée pour autoriser plusieurs types d'exception dans une clause *catch* :

```
boolean isTest = false;  
public void testMethod() {  
    try {  
        if (isTest) {  
            throw new IOException();  
        } else {  
            throw new SQLException();  
        }  
    } catch (IOException | SQLException e) {  
        e.printStackTrace();  
    }  
}
```


Définition de ses propres classes d'exception

- Les exceptions définies par les développeurs sont créées lorsqu' aucune des exceptions existantes de l'API Java ne convient.
- En général, les exceptions Java doivent être réutilisées à chaque fois que possible.
- Pour définir une exception surveillée, votre classe d'exception doit étendre la classe *Exception*, directement ou indirectement.
- Pour définir une exception non surveillée, votre classe d'exception doit étendre la classe *RuntimeException*, directement ou indirectement.
- Pour définir une erreur non surveillée, la nouvelle classe d'erreur doit étendre la classe *Error*.
- Les exceptions définies par l'utilisateur doivent avoir au moins deux constructeurs :
 - Un constructeur sans argument.
 - Un constructeur qui admet une *String* qui représente le message à afficher :

```
public class ReportException extends Exception {  
    public ReportException () {}  
    public ReportException (String message) {  
        super ( message );  
        ...  
    }  
}
```

Afficher/imprimer les informations sur les exceptions

- Les méthodes de la classe *Throwable* qui fournissent des informations sur exceptions levées sont *getMessage()*, *toString()*, et *printStackTrace()*.
- Dans la pratique, au moins une de ces méthodes doit être appelée dans la clause *catch* qui gère l'exception.
- Les développeurs peuvent aussi écrire du code pour obtenir des informations supplémentaires utiles lorsqu'une exception se produit (par exemple, le nom du fichier qui n'a pas été trouvé).

La méthode *getMessage()*

- La méthode *getMessage()* retourne une chaîne de message détaillée sur l'exception:

```
try {  
    new FileReader("file.js");  
} catch (FileNotFoundException fnfe) {  
    System.err.println(fnfe.getMessage());  
}
```

La méthode *toString()*

- La méthode *toString()* retourne une chaîne détaillée sur l'exception, y compris son nom de classe :

```
try {  
    new FileReader("file.js");  
} catch (FileNotFoundException fnfe) {  
    System.err.println(fnfe.toString());  
}
```

La méthode *printStackTrace()*

- La méthode *printStackTrace()* retourne une chaîne détaillée sur l'exception, y compris son nom de classe et l'état de la pile au moment de l'erreur :

```
try {  
    new FileReader("file.js");  
} catch (FileNotFoundException fnfe) {  
    fnfe.printStackTrace();  
}
```