Les Exceptions en JAVA

Exceptions : définition

- Pendant leur exécution les applications peuvent échouer sur plusieurs sortes d'erreurs de différents degrés de sévérité.
- Une exception est un signal qui indique que quelque chose d'anormal s'est produit.
- Les exceptions fournissent un mécanisme de signalisation explicite des erreurs.

Exceptions : définition

- Lancer une exception (*throw*) consiste à signaler le quelque chose d'anormal.
- Capturer une exception (*catch*) consiste à signaler que le quelque chose d'anormal va être géré.
- Une exception se propage en remontant les blocs englobant puis la pile d'appel des méthodes jusqu'à la méthode *main()*.

```
public static void main( String[] args ) {
        meth2(15);
}

static void meth2( int n ) throws RuntimeException {
        int i, j;

        for ( i=0; i<n; i++ ) {
                for ( j=0; j<n; j++ ) {
                        if ( j > 10 ) throw new
RuntimeException ( "Message : attention j > 10" );
                        ...
                }
        }
}
```

Exceptions : exemple

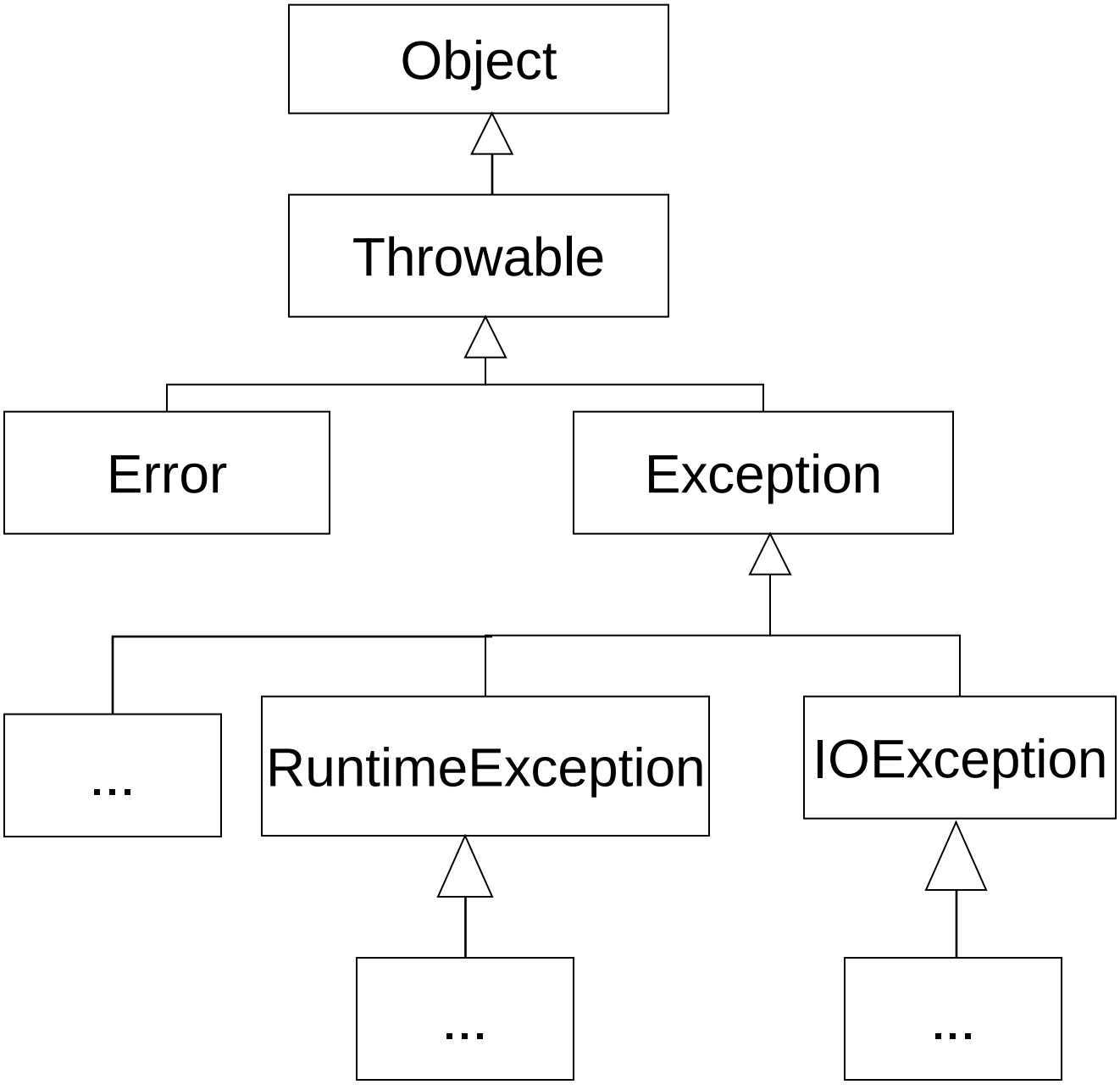
```
class ListeEntiers {

    private int[] tab;

    // constructeur
    public ListeEntiers ( int taille ) throws
RuntimeException {
        if ( taille >= 0 ) this.tab = new int[taille];
        else {
            // cas d'erreur => exception LANCEE
            throw new RuntimeException ( "Taille
impossible (<0)" );
        }
    }
    // accesseur
    public int getTaille () throws RuntimeException {
        int ret = 0;
        if ( this.tab == null ) {
            // cas d'erreur => exception LANCEE
            throw new RuntimeException ( "Taille non
définie" );
        }
        else ret = this.tab.length;
        return ret;
    }
}
```

Objets d'exception

Toute exception est un objet instance de la classe Throwable ou d'une sous-classe.



Exceptions contrôlées de type Exception

Il y a une sous-division entre toutes les sous-classes de Exception :

1. La classe RuntimeException et toutes ses sous-classes sont des exceptions que le programmeur PEUT attraper mais ce n'est PAS une obligation.
   Exemples : NullPointerException, IndexOutOfBoundsException, NumberFormatException, ...

2. TOUTES les autres classes et toutes leurs sous-classes DOIVENT être gérées par le programmeur.
   Exemple : IOException, DataFormatException, ClassNotFoundException, NotSerializableException, ...

Exceptions contrôlées : comment les gérer

DONC

SAUF pour la classe RuntimeException (et toutes ses sous-classes),

IL FAUT pour chaque objet de type Exception susceptible d'être lancé au cours de l'exécution, que le programmeur prévoie sa gestion.

Deux solutions :

1. Décider de « laisser passer l'exception » sans l'attraper explicitement => mot-clé throws dans la déclaration de la méthode.

2. Décider d'attraper l'exception : clauses try / catch / finally.

throws ou try / catch / finally

Exceptions contrôlées : throws

Laisser passer l'exception signifie : si une exception est lancée dans le code de la méthode, on ne la capture pas dans la méthode et on la transmet à la méthode appelante.

```
public static void main ( String[] args ) throws
FileNotFoundException {

        lecture("nomFichier.txt");
}

static void lecture ( String nom ) throws
FileNotFoundException {
        FileReader tmp;

        tmp = new FileReader ( nom );
        ...
}
```

Si le constructeur FileReader(…) lance FileNotFoundException, elle n'est capturée ni dans lecture(…), ni dans main(…) et l'exécution s'arrête définitivement.

Exceptions contrôlées : try/catch/finally

Gestion de l'exception FileNotFoundException = capturer l'exception et rattraper le défaut => pas d'arrêt définitif.

```
public static void main( String[] args ) { lecture(); }

static void lecture() {
    FileReader tmp;
    boolean ok = false;
    String nomFich = "monFich.txt";

    while ( !ok ) {    // boucle infinie si fichier introuvable !!
        try {
            tmp = new FileReader ( nomFich );
            ok = true;
        }

        catch ( FileNotFoundException e ) {
            System.out.println ( "Fichier introuvable" );
            ok = false;
        }

        finally { System.out.println("tjrs exécuté"); }
    }
    etc.
}
```

Exceptions : exemple

```
class ListeEntiersTest {

    public static void main( String[] args ) {

        ListeEntiers l1 = null;

        // Test du constructeur et de l'accesseur
        Sop ( "Test du constructeur, cas normal" );
        …
        Sop ( "Test du constructeur, cas d'erreur" );

        try {
            l1 = new ListeEntiers ( -2 );

            // PAS d'ERREUR de construction
            Sop ( "Echec du test constructeur" );
        }

        catch ( RuntimeException e ) {
            // ERREUR de construction
            Sop ( e.getMessage() );
            Sop ( "Test constructeur réussi" );
        }
        // ET CA CONTINUE NORMALEMENT…
    }
}
```

Exceptions contrôlées : try/catch/finally

La syntaxe du triplet *try/catch/finally* est la suivante :

```
try { // OBLIGATOIRE pour capturer
        // Normalement ce code s'exécute sans
        // problème. Mais il peut parfois lancer une
        // exception.
        // Try = permettre la capture si exception lancée.
}

catch ( SomeException e ) { // OBLIGATOIRE si try
        // CAPTURE un objet exception « e » de type
        // SomeException ou une sous-classe.
        // ICI, il y a du code qui permet de traiter
        // l'exception. Exemple : recommencer le try{}.
        // Ensuite l'exécution se poursuit normalement,
        // l'application ne plante PAS !!
}

finally { // PAS OBLIGATOIRE
        // Ce code est toujours exécuté lorsqu'on quitte
        // la clause try :
        // 1) Normalement (pas d'exception lancée)
        // 2) Avec une exception gérée ou pas par catch
        // finally est peu utilisé en pratique.
}
```

throw (lancer) n’est pas catch (capturer) ni throws (laisser passer)

Une exception lancée par l'API

Lorsqu'une méthode de l'API Java est susceptible de lancer (throw) une exception (d'un type TrucException), c'est CLAIREMENT indiqué dans sa javaDoc.

Exemple :
dans la javaDoc méthode parseInt de la classe Integer :

```
public static int parseInt ( String s )
                throws NumberFormatException
```

Ce qui signifie :
Cette méthode lance (throw) NumberFormatException ET laisse passer cette exception (throws) SANS la capturer à l'intérieur de son code.

Une exception lancée par l'API

Supposons que le programmeur utilise parseInt de la classe Integer dans une de ses méthodes.

```
public int stringToInt ( String st ) {
        int ret;

        ret = Integer.parseInt ( st );
        return ret;
}
```

Que doit faire le programmeur au niveau de l'utilisation de parseInt de la classe Integer ?

Consulter le javaDoc de NumberFormatException.

La gestion de NumberFormatException est-elle OBLIGATOIRE ? NumberFormatException hérite de RuntimeException DONC sa gestion n'est PAS obligatoire.

Une exception lancée par l'API

Supposons que le programmeur veuille quand même gérer l'exception, il peut :

- soit la laisser passer (throws) au niveau de sa méthode stringToInt(…)

```
public int stringToInt ( String st ) throws
NumberFormatException { … }
```

- soit la capturer dans la méthode stringToInt(…)

```
public int stringToInt ( String st ) {
        int ret = 0;

        try {
                ret = Integer.parseInt ( st );
        }

        catch ( NumberFormatException e ) {
                System.out.println ( e.getMessage() );
        }

        return ret;
}
```

Une exception lancée par le programmeur

Le programmeur peut décider de lancer (throw) lui-même une exception.
Intérêt : signaler à l'utilisateur de sa méthode qu'il y a un problème au niveau de son exploitation.

Exemple : dans la classe Cercle du programmeur, le constructeur prend en paramètre int unRayon. Si ce rayon a une valeur négative ou nulle, le Cercle construit n'a aucun sens…

```
// Constructeur de la classe Cercle
// Pas de throws dans la signature… Pourquoi ?
public Cercle ( int unRayon ) {

        // Lancer une exception qui signale le problème
        // Cette exception est de type RuntimeException
        if ( unRayon <= 0 ) {
                throw new RuntimeException ( "Un cercle ne
peut pas avoir de rayon négatif !!" );
        }
        etc.
}
```

Une exception créée par le programmeur

Le programmeur peut créer ses propres exceptions personnalisées. Il crée alors une classe qui doit hériter de Exception.

```
// La classe MonException définit une exception
// personnalisée : elle doit hériter de Exception
class MonException extends Exception {

        public MonException ( String message ) {
                // c'est le message d'explication de l'erreur
                super ( message );
        }
}
```

L'héritage de la classe Exception (qui hérite elle-même de Throwable) permet l'accès aux méthodes : getMessage(), printStackTrace(), etc.

ET DONC une exception de type MonException DOIT être gérée :

- soit en la laissant passer (throws)
- soit en la capturant (try/catch)