Contenu du cours

1. Introduction à la POO
2. Une **classe** : définition de nouveaux objets
3. **Instanciation** et utilisation d'objets
4. Création des objets : les **constructeurs**
5. **Références**, visibilité des variables
6. **Encapsulation** et masquage des données
7. **Statique**, ou d'instance ?
8. **Héritage**
9. **Polymorphisme**
10. **Classes abstraites** et **interfaces**
11. **Introduction aux types génériques**
12. **Exceptions** en java
13. *Compléments syntaxiques*

Les Exceptions en Java

Exceptions en Java : vue d'ensemble

- Mécanisme Java pour gérer les erreurs à l'exécution
- Ce sont les méthodes (*callee* = code appelé) qui déclenchent les exceptions signalant un problème
  - Plutôt que de retourner un code d'erreur (langage C)
  - Une méthode lance (ou lève) une exception
- Le code appelant (= *caller*)
  - Essaye (try) d'appeler la méthode susceptible de lancer une exception
  - Peut rattraper (catch) l'exception éventuellement soulevée
- Attraper une exception empêche le programme de « planter »
- L'exécution se poursuit après le catch

Exceptions : un exemple schématique

callee

```
void toto() throws Exception {
    …
    if (…) // problème détecté
        throw new Exception(…);
    …
}
```

caller

```
void callerToto() {
    …
    try {
        toto();
        …
    }
    catch (Exception e) {
            System.out.println("Erreur");
        …
    }
    …
}
```

En cas de problème, toto() lancera (throw) une exception, son exécution s'arrêtera

Si pas de problème, l'exécution continuera normalement

Le caller ***essaye*** (try) d'appeler toto()

Si pas de problème, l'exécution se poursuit après toto()

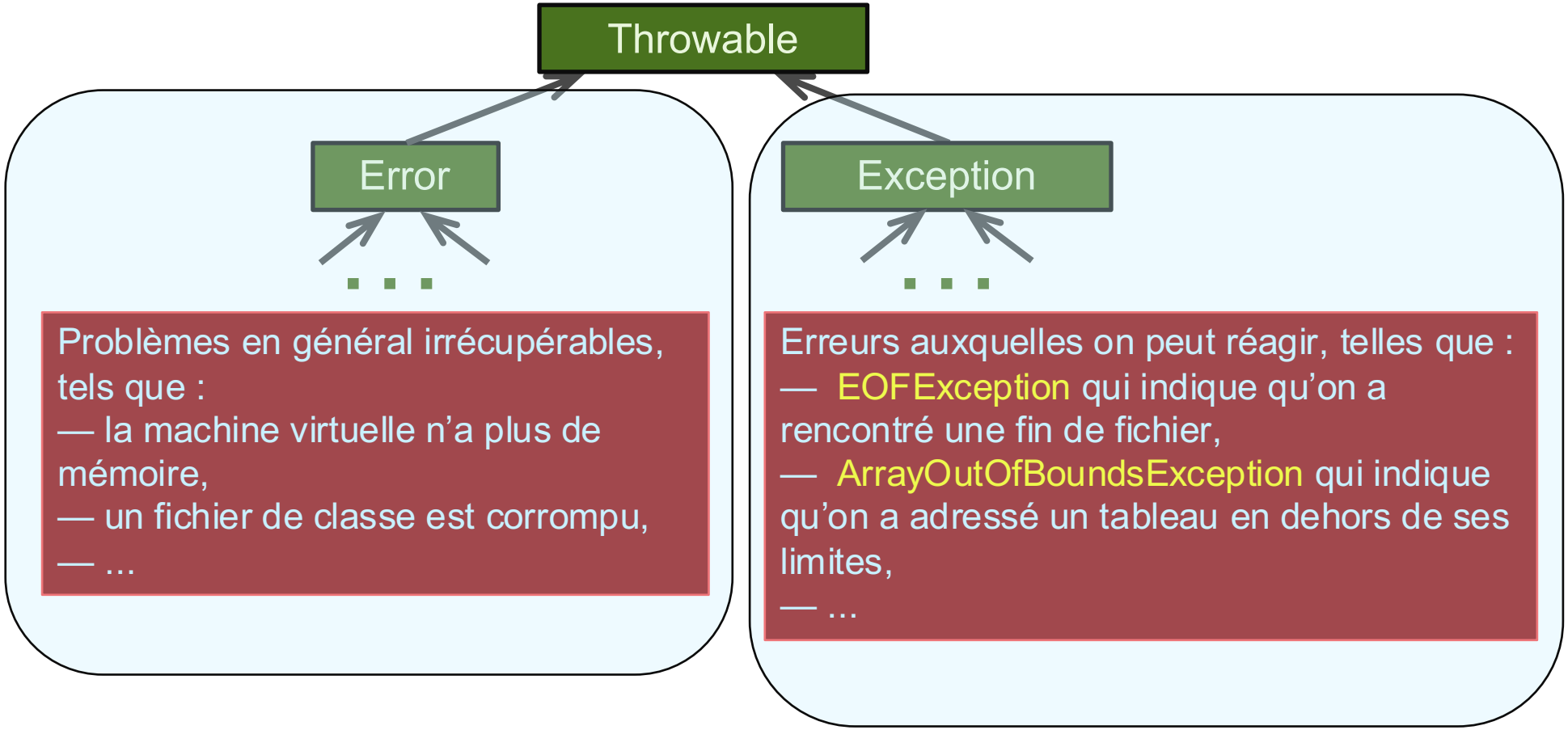
Si problème, le caller ***rattrape*** (catch) l'exception (et signale le problème)

Le caller peut exécuter du code complémentaire

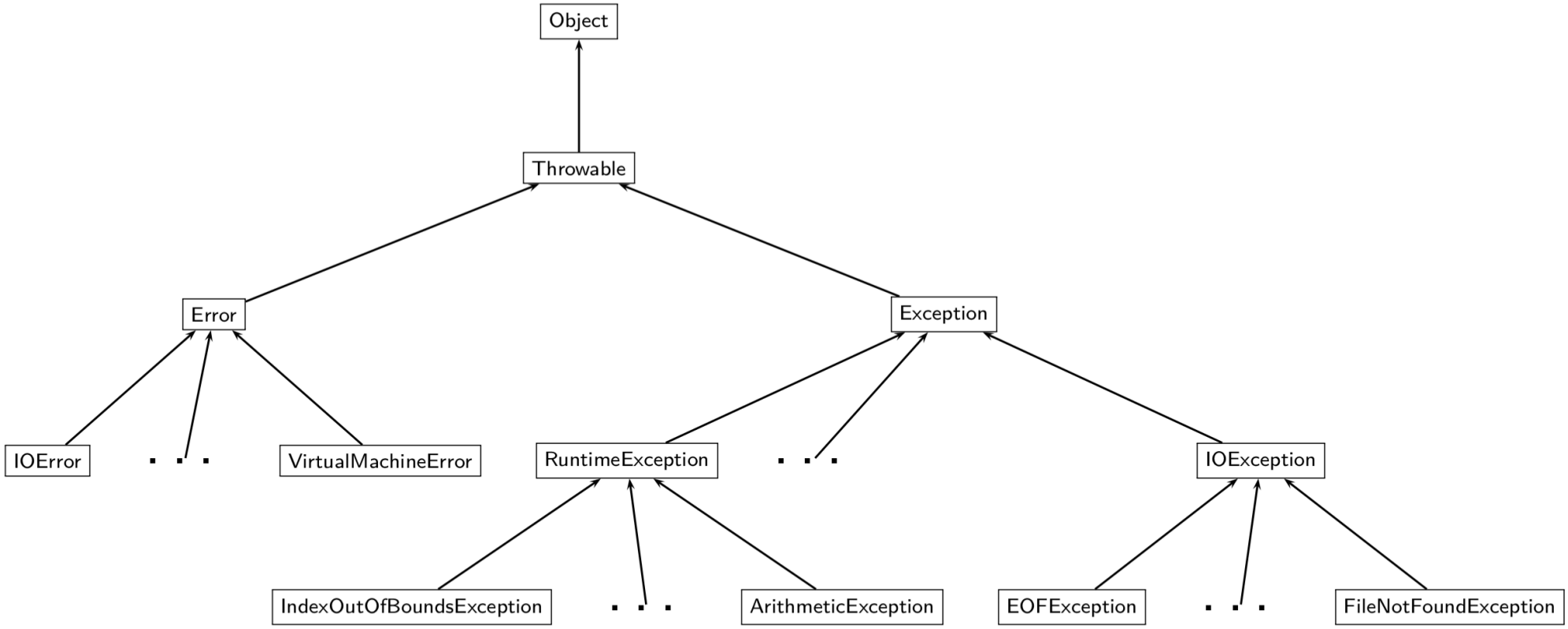
L'exécution continue dans tous les cas (pas de plantage) après le catch

Exceptions : objets de Java

- **Une exception est un objet à instancier**
- **Type : java.lang.Throwable (ou une sous-classe)**



Hiérachie des Exceptions Java



Personnaliser les exceptions

- **Exception = objet ==> *attributs, méthodes, constructeurs***
- **Throwable et ses sous-classes possèdent**
  - **Attribut String pour message décrivant l'anomalie**
  - **Méthode getMessage() pour récupérer ce message**
  - **Constructeur avec paramètre String pour initialiser le message**
- **Exemple (si fileName est une chaîne contenant un nom de fichier)**

```
throw new FileNotFoundException("Le fichier "+fileName+" n'existe pas");
```

- **Créer sa propre classe d'exceptions ?**
  - **Il suffit d'hériter d'une classe d'exception existante**
  - **Utile pour nommer (getClass().getName()) l'objet exception**

Levée d'une exception

- **Anomalie détectée au cours d'une méthode : on « lance » (ou « lève ») une exception**
  - **Instancier l'exception (avec new)**
  - **Lancer l'exception (avec throw)**
  - **On dit de la méthode (*callee*) qu'elle lève l'exception**
- **Exemple**

```
public static long factorielle(int n) {
    if (n < 0)
      throw new IllegalArgumentException("Factorielle d'un nombre négatif indéfinie") ;
    long fact = 1;
    for (int i=n; i > 1; i--)
        fact *= i ;
     return fact ;
}
```

On n'ira pas plus loin dans le code de la méthode si n<0

Propagation d'une exception

- **Aussitôt qu'une exception est levée (avec throw)**
  - **L'interpréteur java interrompt l'exécution normale**
  - **Exception propagée de bloc de code appelant en bloc de code appelant (de *caller* en *caller*)**
  - **Jusqu'à être rattrapée par un bloc catch qui la traite**
- **Si aucun bloc catch rencontré**
  - **L'exception remonte jusqu'au main()**
  - **Si non traitée dans le main()** printStackTrace() *dans* Throwable
    - **Affichage du message de l'exception**
    - **Affichage de la trace de la pile des appels**
    - **Interruption (plantage) du programme**
- **Exemple**

```
Exception in thread "main" java.lang.IllegalArgumentException: Factorielle d'un nombre négatif indéfinie
        at ExFactorielle.factorielle(ExFactorielle.java:4)
        at ExFactorielle.main(ExFactorielle.java:12)
```

Exceptions contrôlées ? Ou non-contrôlées ?

- Exceptions **non-contrôlées**
  - Pour des problèmes a priori imprévisibles
  - (Sous) classes Error et RunTimeException
  - Le langage n'impose pas d'en prendre le contrôle
  - Aboutissent en général à un plantage du programme
- Exceptions **contrôlées**
  - Erreurs prévisibles
  - Le langage **impose** d'en prendre le contrôle
  - (Sous) classes Exception (sauf RunTimeException)
  - Pour une méthode (***callee***) ***levant*** une exception contrôlée
    - Déclaration throws obligatoire dans sa signature

```
void readFile(String fileName) throws FileNotFoundException {
    …
    if ( … ) throw new FileNotFoundException(…);
    …
}
```

Invocation (caller) de méthode (callee) levant une exception

- **J'appelle une méthode levant une exception ? Je dois la traiter**
  - **Soit je l'arrête (catch (et/ou finally))**
  - **Soit je la propage (throws)**

Comment savoir si une méthode que j'appelle pourrait lever une exception ?

Indiqué dans son API (signature inclut throws) + erreur compilation si pas traitée.

```
error: unreported exception FileNotFoundException; must be caught or declared to be thrown
```

- **Propager une exception**
  - **Le caller ajoute throws dans sa propre signature**
  - **L'exception ne fait que transiter (transmise à son appelant)**
- **Arrêter une exception**
  - **Insérer l'invocation dans blocs try/catch/finally**
  - **Elle ne sera pas propagée plus loin**

Rattrapage d'une exception : *catch me if you can!*

Le bloc *try* englobe du code susceptible de lever une exception

- **Instruction** try/catch/finally

```
try {
    … // code levant une exception
}
catch (TypeDException e) {
    … // code réagissant à l'exception
}
finally {
    … // code de « nettoyage »
}
```

Un bloc *catch* est conçu pour réagir à un type d'exception spécifique. Le type (au sens de classe java) de l'exception traitable est précisé entre parenthèses. Ce bloc attrape toutes les exceptions de ce type ou d'un type dérivé levées dans le bloc try associé.

Le bloc finally contient du code qui sera toujours exécuté, que l'exception se soit produite ou non. Son but est de contenir du « code de nettoyage », par exemple qui referme des fichiers ouverts.

- **Pour chaque** try**, soit un** catch**, soit un** finally**, soit les deux**
- **Plusieurs** catch **autorisés pour un même** try **(typiquement un par type d'exception)**

Exemple complet : (1) exception personnalisée

- Classe d'exception personnalisée pour signaler des indices de tableau non autorisés

```
public class BadIndexesException extends Exception {
        BadIndexesException() {
            super();
        }

        BadIndexesException(String message) {
            super(message);
        }
}
```

Exemple complet : (2) callee levant exception

- Extraction d'un sous-tableau dans une chaîne

```
public static String subArrayString(int[] tab, int from, int to)
                                        throws BadIndexesException {
    if (from < 0 || from >= tab.length || to < 0 || to >= tab.length)
        throw new BadIndexesException("Les indices "+from+" et "+to+
           " doivent être dans l'intervalle [0, "+(tab.length-1)+"]");

    if (from > to)
        throw new BadIndexesException("L'indice de début "+from+
                    " doit être plus petit que l'indice de fin "+to);

    // Si ce point est atteint, alors on n'a pas levé d'exception.
    // Les indices sont corrects, on peut commencer le traitement
    // sans risque.
    String sas = "[";
    for (int i=from; i <= to; i++) {
        sas += tab[i];
        if (i < to)
            sas += ", ";
    }
    return sas+"]";
}
```

Exemple complet : (3) caller traitant exception

- **Cas 1 : déléguer au code appelant**

```
public static void printSubArrayString1(int[] tab, int from, int to)
                                                    throws BadIndexesException {
    // version 1 : l'exception, si elle se produit, ne fait que transiter
    // sans être traitée
    System.out.println("de "+from+" à "+to+" : "+subArrayString(tab, from, to));
}
```

- **Cas 2 : arrêter l'exception**

```
public static void printSubArrayString2(int[] tab, int from, int to) {
    // version 2 : l'exception, si elle se produit, est saisie (ou arrêtée)
    try {
        System.out.println("de "+from+" à "+to+" : "+subArrayString(tab, from, to));
    }
    catch (BadIndexesException e) {
        System.out.println(e.getMessage());
    }
}
```

Cas 1 : l'exception peut continuer à se propager, au risque de planter le programme si personne ne l'arrête

Cas 2 : l'exception est attrapée, le programme peut continuer sans planter

Exemple complet : (4) un main() qui plante ou pas

```
public static void main(String[] args) throws BadIndexesException {
    // Selon les exécutions (au hasard), ce programme peut s'exécuter
    // avec ou sans erreur (ou exception). Si une exception se produit
    // le programme peut soit planter (cas 1, exception non rattrapée),
    // soit arriver quand même à son terme (cas 2, exception rattrapée)

    int[] tab = new int[20];

    for (int i=0; i < tab.length; i++)
        tab[i] = i; // on remplit par les valeurs de 0 à 19

    // On génère les indices au hasard (classe Hasard perso)
    int from = Hasard.entier(20)-3;
    int to = Hasard.entier(20)+3;

    // Cas 1 ou 2 tiré au hasard :
    if (Hasard.entier(2) == 0) {
        System.out.println("Cas 1 : appel non protégé par un try");
        printSubArrayString1(tab, from, to);
    }
    else {
        System.out.println("Cas 2 : appel protégé par un try");
        printSubArrayString2(tab, from, to);
    }

    // En cas d'exception, ce point ne peut être atteint que si celle-ci
    // a été rattrapée (cas 2). Sinon (cas 1), le programme a planté avant
    // d'atteindre ce point
    System.out.println("FIN NORMALE DU PROGRAMME");
}
```