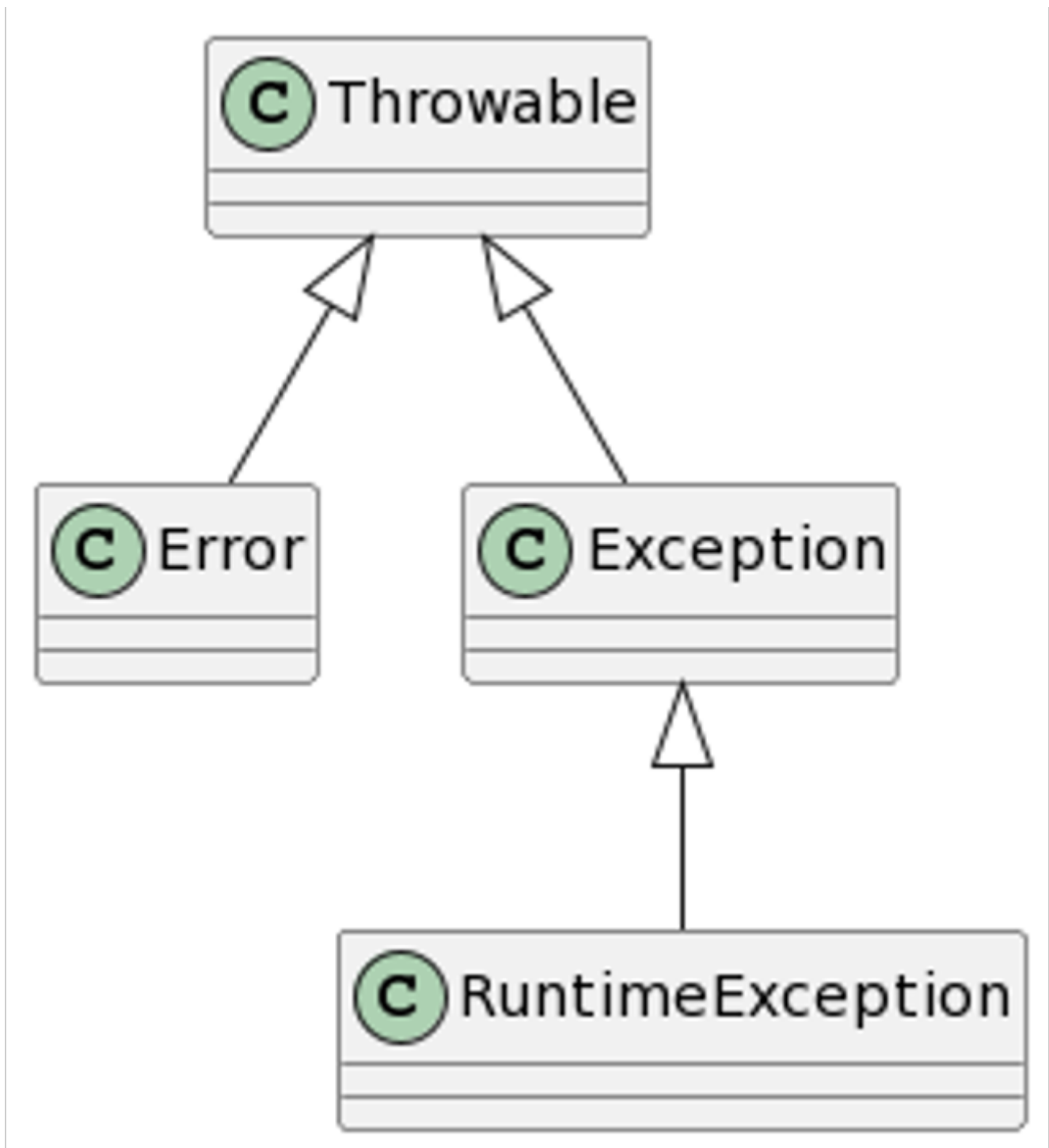


Exceptions



On va surtout utiliser **Exception** et **Runtime Exception**, mais pas directement (on va plus tot utiliser des exceptions dérivées de ceux-là, tel que **NullPointerException** ou **IllegalArgumentException**).

Par défaut, le comportement d'une exception est de print l'erreur dans la console, cependant on ne va pas tellement l'utiliser en cours car on va utiliser un système de logging plus élaboré.

Try - catch

```
try {  
    // On essaye d'exécuter un certain code ici...  
} catch (NullPointerException e) {  
    // Le code ici s'exécutera si il y a une NullPointerException et l'erreur sera mise dans la variable e.  
} catch (IllegalArgumentException | IndexOutOfBoundsException e) {  
    // Le code ici s'exécutera dans le cas d'un IllegalArgumentException ou d'une IndexOutOfBoundsException et  
    sera mise dans la variable e.  
}  
  
// Si aucun problème n'est arrivé ou qu'elle a été bien catchée, le code ici s'exécutera...
```

Les try-catch permettent d'exécuter son propre code dans le cas d'une erreur.

Réexécution de fonction

Si on veut réexécuter une fonction il vaut mieux éviter de rappeler simplement la fonction récursivement dans le `catch` car si une erreur persiste, cela augmente grandement le stack des fonctions appelée ce qui peut mener à faire crash le programme.

Il vaut mieux utiliser un do-while

```
// On initialise les variables en dehors de la boucle  
boolean locked = false;  
int entier = 0;  
do {  
    System.out.print("Entrer un entier : ");  
    try {  
        entier = lireEntier();  
        // Si le code a fonctionné, on remet le locked à false pour sortir de la boucle  
        // On doit le remettre à false car si elle a raté la première fois, le locked aura été mis à true par le catch  
        locked = false;  
    } catch {  
        // Si un soucis survient, on met la variable locked à true, pour que la boucle réexécute le code  
        // Il n'y a ainsi aucune récursion, donc pas de risque de stackoverflow  
        locked = true;  
    }  
} while (locked);
```

Throw - throws

```
public class PersonalException extends Exception {  
    // Ici on met le code de l'exception, par exemple on peut y mettre des messages d'erreurs, des fonctions  
    spéciales, etc.  
}  
  
// La méthode suivante retourne une exception  
public static void method() {  
    // Si quelque chose ne fonctionne pas on peut retourner notre exception custom  
    throw new PersonalException();  
}
```

On peut créer nos propre exceptions pour des cas particulier de nos programmes en étendant la classe Exception puis en utilisant `throw new` pour l'appeller.

Finally

```
try {  
    // On essaye d'exécuter un certain code  
} catch (NullPointerException e) {  
    // On exécute le code ici si le code dans le try ne fonctionne pas  
}  
finally {  
    // Quoi qu'il arrive, le code ici sera exécuté, même si le catch retourne une exception.  
}  
// Si une erreur arrive dans catch ou finally, le code ici ne sera pas exécuté
```

Lorsque l'on veut qu'un code s'exécute quoi qu'il arrive, on peut utiliser le bloc `finally`, ainsi même si l'un des `catch` retourne une exception, on exécutera quand même le bloc finally.

Tester les exceptions

On peut également tester des exceptions avec `assertThrows`

```
assertThrows(RuntimeException.class, ()=>maMethode());
```

On passe directement la méthode à `assertThrows`. La syntaxe étrange s'appelle une *lambda*, c'est une fonction anonyme temporaire qui exécute la méthode à tester. Cette *lambda* est nécessaire, sinon on passe le résultat de `maMethode` à la place de passer la méthode elle-même. La fonction `assertThrows` va ensuite tester pour voir si une exception est émise par la méthode, et si oui, elle va tester que l'exception est bien de la classe `RuntimeException`.

Revision #2

Created 18 September 2023 16:39:45 by SnowCode

Updated 18 September 2023 16:54:57 by SnowCode