Licence Pro DevOps

Rappels Java :

généricité++,
Classes internes, locales et anonymes
Exceptions

Généricité

- Permet de définir une classe en fonction d'un type qu'on ne connaît pas (qu'on ne connaîtra qu'à l'instanciation)
- Exemple classique : conteneurs (ex : listes, ensembles)

```
public class monConteneur<T> {
    private T premierElt;
    private T deuxiemeElt;
    public monConteneur(T first, T second) {
            premierElt = first;
            deuxiemeElt = second;
    }
}
```

Puis :

```
MonConteneur<Integer> = new MonConteneur<Integer>();
MonConteneur<String> = new MonConteneur<String>();
```

Généricité : un peu plus loin

- Permet de placer des contraintes sur le type générique :
  - Obliger le type à implémenter telles interfaces
  - Obliger le type à hériter de telle classe
- Exemple classique : on veut un conteneur d'objets qui peuvent être comparés les uns avec les autres

```
public class monConteneur<T extends Comparable<T>> {
    private T premierElt;
    private T deuxiemeElt;
    public monConteneur(T first, T second) {
            premierElt = first;
            deuxiemeElt = second;
            premierElt.compareTo(deuxiemeElt);
    }
}
```

- Remarque : mot-clé **extends** même pour une interface

Classes **internes**, locales, anonymes

- Java permet de définir une classe à l'intérieur d'une autre classe. On l'appelle **classe interne**
- La classe interne a accès aux attributs et méthodes de la classe englobante

```
public class A {

    public class B {
        int x;
        void m() {
            y = 1337;
        }
    }

    public B b;
    public int y;

    public A() {
        b = new B();
        b.m();
    }
}
```

Classes internes, **locales**, anonymes

- Une classe peut être définie dans n'importe quel bloc
- Sa portée est alors limitée au bloc
- Elle a accès aux attributs de la classe et aux variables et paramètres de la méthode

```
public class A {
    public int m(int t) {
        class Paire {
            public int x;
            public int y;

            public int foo() {
                return (x+y)*t;
            }
        }
        Paire p = new Paire();
        p.x = 8;
        p.y = 9;
        return p.foo();
    }

    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.m(10));
    }
}
```

Classes internes, locales, **anonymes**

- Permet de definir une classe interne locale sans nom
- Valable seulement pour définir une sous-classe d'une certaine classe ou une classe implémentant une interface

```
public interface Affichable {
        public void affiche();
}
```

```
public class A {

    public void foo(Affichable a) {
        a.affiche();
    }

    public static void main(String[] args) {

        A a = new A();
        a.foo(new Affichable() {
            public void affiche() {
                System.out.println("Hello world !");
            }
        });
    }
}
```

Les exceptions en Java

But :

- Assurer la robustesse du code
- Séparer le code de la gestion des erreurs
  - pour une meilleure organisation
  - et donc une meilleure maintenance

Les exceptions en Java

- Quand est-ce qu'une exception est "levée" ?
  - erreur d'exécution "native" :
    - division par 0
    - appel de méthode d'un objet null
    - accès à une case d'un tableau hors bornes
    - …
  - ou si vous (ou un développer d'une API utilisée) le décidez :
    - lecture d'un fichier qui n'existe pas
    - requête à une @ ip qui n'existe pas
    - ...

Les exceptions en Java

- Exemple "classique" :

```
13  public class A {
14
15      B b;
16
17      public A() {
18
19      }
20
21      public void maMethode() {
22          System.out.println(b.getAttribut());
23      }
24
25      public static void main(String[] args) {
26          A a = new A();
27          a.maMethode();
28      }
```

Les exceptions en Java

- Exemple "classique" :

```
13  public class A {
14
15      B b;
16
17      public A() {
18
19      }
20
21      public void maMethode() {
22          System.out.println(b.getAttribut());
23      }
24
25      public static void main(String[] args) {
26          A a = new A();
27          a.maMethode();
28      }
```

Exception in thread "main"
java.lang.NullPointerException
at monPg.A.maMethode(A.java:22)
at monPg.A.main(A.java:27)

→ **l'exécution s'arrête**
car l'exception n'a pas été capturée

Les exceptions en Java

```java
public class A {

    B b;

    public A() {

    }

    public void maMethode() {
        System.out.println(b.getAttribut());
    }

    public static void main(String[] args) {
        try{
            A a = new A();
            a.maMethode();
        } catch (Exception e) {
            System.out.println("On a eu un soucis : "+e.getMessage());
        }

        System.out.println("Mais on peut continuer !");
    }
```

Les exceptions en Java

```java
public class A {

    public void maMethode() throws Exception  {
        uneAutreMethode();
    }

    private void uneAutreMethode() throws Exception  {
        throw new Exception("ceci est un soucis");
    }

    public static void main(String[] args) {
        try{
            A a = new A();
            a.maMethode();
        } catch (Exception e) {
            System.out.println("On a eu un soucis : "+e.getMessage());
        }

        System.out.println("Mais on peut continuer !");
    }
```

Les exceptions en Java

```
public void maMethode() throws Exception  {
    uneAutreMethode();
}

private void uneAutreMethode() throws Exception  {
    throw new MonException1("ceci est un soucis");
}

public static void main(String[] args) {

    try{
        A a = new A();
        a.maMethode();
        System.out.println("Pas exécuté si exception");

    } catch (MonException1 e) {
        System.out.println(e.getMessage());
    } catch (UneDeuxiemeException e) {
        System.out.println(e.getMessage());
    } catch (Exception e) {
        System.out.println(e.getMessage());
    } finally {
        System.out.println("sera toujours exécuté");
    }

    System.out.println("ça repart de là");(si l'exception a été attrapée)
}
```

Les exceptions en Java

Le bloc **finally** est toujours exécuté*, même si :

- Aucune exception n’est levée
- Une exception attrapée dans un catch est levée
- Une exception non prévue est levée

Bonne pratique : y placer les fermetures de fichiers/flux/connexions

* n’est pas exécuté si la JVM crashe ou si une interruption stoppe le bloc try...