TD JAVA n°3: Interfaces, Exceptions

Exercice 1 : Interface

1. Qu'est ce qu'une **Interface** ?

2. On veut pouvoir comparer deux objets d'une même classe.
   Ecrire l'**`interface Comparaison`** qui contiendra les méthodes suivantes :
   - **`plusGrandQue`** qui renverra vrai si l'`Object obj` appelant la méthode est plus grand que l'objet à comparer.
   - **`plusPetitQue`** qui renverra vrai si l'`Object obj` appelant la méthode est plus petit que l'objet à comparer.

3. Ecrire la **`classe Ville`** qui implémente cette interface.
   (Une ville sera caractérisée par son **`nom`** de type **`String`** et son nombre d'habitants(**`nbHabitants`**) de type **`int`**)

4. Instancier la ville **`v1`** (Toulouse ,400000 habitants) et la ville **`v2`**(Limoges,150000 habitants)
   Comment savoir si Toulouse est plus grand que Limoges ?

5. Que se passe-t-il si on compare la ville `v1` à un objet autre qu'une ville ?
   Si un problème surgit, expliquer comment résoudre ce problème …

Exercice 2 : Mise en place d'une Exception métier *dans le projet Cabinet Médical*

1. Qu'est ce qu'une **Exception**? Et comment la traiter ?

2. Pour compléter le projet, nous allons créer une exception métier « contrôlée » que nous appellerons `CabinetMedicalException`. Il est important de donner la possibilité de pouvoir transmettre un message à l'exception levée puisque cette exception pourra être utilisée dans plusieurs cas (NIR incorrect, adresse non mentionnée lors de la création d'un `Patient`)

   En vous aidant du cours, écrire la classe : `CabinetMedicalException.`

3. Dans le TP précédent vous aviez écrit dans la classe `PatientRegle` une méthode statique `verifierNir` qui permettait de contrôler la validité du NIR en renvoyant un booléen, cette méthode était de la forme suivante :

Method Detail

verifierNir

`public static boolean verifierNir(java.lang.String nirATester)`

Teste la validité d'un NIR en recalculant la clé de contrôle à partir de 13 premiers caractères du `nirATester`.

Parameters:
`nirATester` - `String` composé de 15 caractères respectant la norme officielle d'un NIR

Returns:
true si la clé recalculée à partir des 13 premiers caractères du `nirATester` est bien égale à la clé extraite du `nirATester`

Extrait code n°1

```
/** Teste la validité d'un NIR en recalculant la clé de contrôle
 * à partir de 13 premiers caractères du <code>nirATester</code>.
 * @param nirATester <code>String</code> composé de 15 caractères
 * respectant la norme officielle d'un NIR
 * @return true  si la clé recalculée à partir des 13 premiers caractères
 * du <code>nirATester</code> est bien égale à la clé extraite du <code>nirATester</code>
 */

     public static boolean verifierNir (String nirATester)
          {
          if (nirATester == null) return false;
          if (nirATester.length() !=15) return false;
          ////////////////////////////////////////////////////
          // Extraction des 13 premiers caractères et de la clé
          ////////////////////////////////////////////////////
          String nir_13 = nirATester.substring(0,13); //13 car endIndex-1 (javadoc!)
          int cle = Integer.parseInt(nirATester.substring(13,15));

          ////////////////////////////////////////////////////////////
          // Conversion du nir_13 String en long (eh oui… 13 chiffres…)
          // et calcul de la cle dans la variable cleCalculee
          ////////////////////////////////////////////////////////////
          … Voir votre code du TP précédent …

          //////////////////////
          // Validité de la clé
          //////////////////////
          if (cleCalculee == cle) return true ;
          else return false;
          }
```

et vous pouviez alors tester votre code de la manière suivante :

Extrait code n° 2

```
     public static void main(String[] args)
          {
          String unNir = "168072B12345652"; //clé valide
          if (PatientRegle.verifierNir(unNir)==true)
               {
               System.out.println("Cle VALIDE");
               }
          else
               {
               System.out.println("!!!! Cle NON VALIDE !!!!");
               }
          }
```

↳3.1.a Modifier l'*extrait de code n°1* précédent pour que la méthode **`verifierNir`** ne renvoie plus **`boolean`** mais « rien » et ***lève une exception métier*** dans son code et la laisse remonter.
L'exception stockera un message du genre *`"Le NIR proposé est incorrect :"`* et indiquera ensuite la valeur de ce NIR incorrect.

↳ 3.1.b Modifier également la javadoc pour faire apparaître la « remontée » de l'exception.

↳3.2 Modifier l'*extrait de code n°2* afin que ***l'exception soit traitée*** dans la méthode **`main`**.

4. ↳ 4.1 Quelle(s) méthode(s) de la classe `Patient` pourrai(en)t être susceptible(s) d'appeler `verifierNir` et de propager une `CabinetMedicalException`?

↳ 4.2 Comment faire ***pour faciliter la maintenance de votre code*** et faire intervenir l'appel à **`verifierNir`** dans une seule méthode de la classe `Patient.`
Quelle nouvelle règle d'écriture des classes métier en déduisez-vous ?
Mettez à jour le document reprenant les règles d'écriture des classes métier du projet **CabinetMedical**

↳ 4.3 D'après les deux questions précédentes, effectuer les modifications nécessaires dans la classe métier `Patient` pour tester la validité du NIR et lever des `CabinetMedicalException` .
Volontairement, on ne traitera pas d'exception(s) dans la classe métier `Patient`, mais on la(les) laissera se propager…

Correction TD JAVA n°3: Interfaces, Exceptions, Polymorphisme

Exercice 1 : Interface

↳ 1. Une **Interface** est une **classe purement abstraite** où toutes les méthodes sont abstraites et publiques.

↳ 2. **Remarque :** j'appelle volontairement cette interface **`Comparaison`** car attention, il existe déjà une interface **`Comparable`** dans l'API sous le package **`java.lang`**

```
public interface Comparaison{
// renvoie vrai si this est plus grand que o
public boolean plusGrandQue (Object obj);   //
    public abstract implicite

// renvoie vrai si this est plus petit que o
public boolean plusPetitQue (Object obj);
}
```

Rq : le nom de la variable `obj` est obligatoire (≠ avec specification (.h) du C/C++)

Remarque : Cette interface pourra être utilisée par d'autres classes héritant de `Object`, c'est-à-dire toutes les classes…

↳ 3.

```
public class Ville implements Comparaison{
      //Attributs
      private String nom;
      private int nbHabitants;

      //Constructeur demandé…et constructeur par défaut !
      public Ville()
            {}

      public Ville(String nom, int nbHabitants)
            {
            this.nom = nom;
            this.nbHabitants = nbHabitants;
            }

      //Getteurs et Setteurs
      public String getNom() {
            return nom;
      }

      public void setNom(String nom) {
            this.nom = nom;
      }

      public int getNbHabitants() {
            return nbHabitants;
      }

      public void setNbHabitants(int nbHabitants) {
            this.nbHabitants = nbHabitants;
      }
```

```
	//////////////////////////////////////////
	// Autres méthodes
	// Implémentation des méthodes de l'interface
	//////////////////////////////////////////
	public boolean plusGrandQue(Object objet) {
	// Attention il faut garder la même signature que dans l'interface
	// public boolean plusGrandQue(Ville autreVille) n'est pas possible

	Ville autreVille = (Ville) objet;	// Cast : transformer l'objet
	 // en  variable de type Ville
		if (nbHabitants >autreVille.nbHabitants) return true;
		else return false;
	// on peut utiliser directement nbHabitants car même s'il est privé
	// dans la classe toutes les variables de type Ville partage
	// la même classe (private => à l'intérieur de la classe)

		}
		public boolean plusPetitQue(Object objet) {
		Ville autreVille = (Ville) objet;	// Cast : objet en Ville
		if (nbHabitants <autreVille.nbHabitants) return true;
		else return false;
		}
}
```

Remarque :
On aurait pu écrire : `if (objet instanceof Ville)`
Avant l'instruction de cast : `Ville autreVille = (Ville) objet;`

… mais ce problème sera traité à la question 5 … => if ou exception ?

4.

```
	public static void main(String[] args) {
		Ville v1 = new Ville ("Toulouse",400000);
		Ville v2 = new Ville ("Limoges",150000);

		System.out.print ("Toulouse plus grand que Limoges ? : ");
		if (v1.plusGrandQue(v2)) System.out.println("VRAI");
		else System.out.println("FAUX");
	}
```


5.
```
System.out.print ("Ville plus grand que Rectangle ? : ");
if (v1.plusGrandQue(new Rectangle())) System.out.println("VRAI");
else System.out.println("FAUX");
```

```
Ville plus grand que Rectangle ? :
Exception in thread "main" java.lang.ClassCastException
        at Ville.plusGrandQue(Ville.java:17)
        at TestComparable.<init>(TestComparable.java:35)
        at TestComparable.main(TestComparable.java:6)
```

Pour résoudre le problème utiliser **instance of** avant de réaliser le cast :
Plusieurs solutions :

1. **retourner faux** si l'objet n'est pas de classe ville (bof….⇒ ce qui aurait été fait avant de connaître les exceptions…)… Par exemple : `Object objet = new Rectangle();`
```
if (objet instanceof Ville)
	return nbHabitants > ((Ville)objet).nbHabitants;
else
	return false ; => n'interrompt pas le programme
```

2. ***Traiter*** l'exception **`ClassCastException`** (si on sait que c'est cette exception qui va se déclencher)… Pour être sûr de tout récupérer, on peut aussi faire une clause **`catch : ( Exception e)`**
   2.1 on traite avec un **bloc try…catch** …. soit **directement** dans la méthode **`plusGrandQue`**
   2.2 soit **dans le programme test** appelant cette méthode : dans ce cas-là, on « **laisse remonter** » l'exception ClassCastException …

3. ou bien mieux : **lancer** une exception personnelle que l'on **devra attraper plus tard** dans le programme, ce qui nous amène aux exceptions ….

Pour information pour les étudiants, je pense que cela n'est pas la peine de leur faire écrire car il y a l'exercie 2 qui traite vraiment des exceptions, le dire peut être juste oralement …

Pour la mise en place de l'exception personnelle, il faut modifier :
- le code de la méthode `plusGrandQue` dans la classe Ville
- la déclaration de la méthode `plusGrandQue` dans l'interface Comparaison si on propage l'interface
- le code de la classe test appelant `plusGrandQue` qui doit try..catch.. la nouvelle exception lancée
- créer une classe `MonException`..

```
public boolean plusGrandQue(Object objet) throws MonException
	{
	try
	{
	Ville autreVille = (Ville) objet;
	if (nbHabitants >autreVille.nbHabitants) return true;
    else return false;
	}
	catch (ClassCastException e)
	{
		throw new MonException(" Le paramètre passé n'est pas une Ville");
		  // => on devra attraper l'exception plus tard …
		  // ne pas oublier de préciser throws dans l'entête
		 }
	}
```

Remarque : si je lance une Exception Personnelle, je serais obligée de la déclarer : Cest une ***exception contrôlée par la compilateur …***

```
public boolean plusGrandQue(Object objet) throws MonException
```

… mais si je lance une : `throw new IllegalArgumentException();`
`IllegalArgumentException` n'est pas une exception contrôlée par le compilateur… donc Je ne serais pas obligé de la déclarer dans l'entête de la méthode et donc d'ecrire:

```
public boolean plusGrandQue(Object objet) throws IllegalArgumentException
```

je laisse tout simplement : `public boolean plusGrandQue(Object objet)`
alors quel intérêt de faire un try… catch et de lancer `new IllegalArgumentException();`
alors que `ClassCastException` remonte déjà toute seule exception non contrôlée car hérite de RuntimeException (voir doc) … aucun à mon avis…

Ce qui doit alors changer d'autre dans le projet…

→ Dans l'interface Comparaison, il faut aussi lancer l'exception dans la déclaration …

```
public interface Comparaison {
        // renvoie vrai si this est plus grand que o
        public boolean plusGrandQue (Object o) throws MonException;
        // renvoie vrai si this est plus petit que o
        public boolean plusPetitQue (Object o);
}
```

→ Il faut créer la classe MonException …

```
public class MonException extends Exception{

    public MonException(){
        super();
    }

    public MonException(String msg){
        super(msg);
    }

}
```

→ … et dans le fichier test appelant plus grand que, il faut attraper l'exception …

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

        System.out.print ("Ville plus
grand que Rectangle ? : ");
        try {
            if (v1.plusGrandQue(new Rectangle())) System.out.println("VRAI");
            else System.out.println("FAUX");
        } catch (MonException e) {
            e.printStackTrace();
        }

    }
```

java.lang
Class ClassCastException

java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.lang.RuntimeException
                    |
                    +--**java.lang.ClassCastException**

All Implemented Interfaces:
Serializable

public class **ClassCastException**
extends RuntimeException

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a `ClassCastException`:

```
Object x = new Integer(0);
System.out.println((String)x);
```


Exercice 2 : Mise en place d'une* Exception métier *dans le projet Cabinet Médical

↳ 1. ***Extrait du cours …***
Une ***exception*** est un signal : qui **indique** que quelque chose d'exceptionnel s'est produit
et qui **interrompt** le flot d'exécution normal du programme
Une exception peut être « capturée » par l'instruction **catch**
⇒ Le ***blocs : try…catch…*** permet la séparation du bloc d'instructions de la gestion des erreurs pouvant survenir dans ce

	Bloc optionnel finally
try { …… **// lignes de code à protéger** …… **// susceptibles** …… **// de provoquer une exception** **}** **catch** (UneException e1) **{** …… **// traitement à effectuer si** …… **// l'exception UneException** …… **// est générée** **}** …… **/ / suite du programme** ……	**try** **{** …… …… **}** **catch** (Exception e) **{** …… …… **}** **finally** **{** …… **// exécution garantie** …… **// quoi qu'il arrive que** **}**

↳ 2. Pour compléter le projet du cabinet médical, nous allons créer une exception métier « contrôlée » que nous appelerons `CabinetMedicalException`.

Remarque : Nous allons créer ici une ***exception métier « générique ».*** Nous allons créer une seule classe d'exception que nous ***personnaliserons ensuite avec un message suivant le cas où l'exception pourra être levée.*** En effet différentes situations se prêtent à l'utilisation d'une exception….

- cas d'un NIR incorrect => lèvera `CabinetMedicalException` avec un message approprié
- cas de saisie d'un patient qui n'a pas d'adresse => exception levée `CabinetMedicalException`
- et plus tard si un Rendez-Vous est donné sur une plage horaire déjà réservée

Pas la peine de déclarer autant de classe `Exception` que de « cas à risque »…
Une seule classe métier `CabinetMedicalException` suffira… Les cas d 'utilisation seront différenciés grâce au message passé en paramètre au constructeur. Il est donc indispensable de créer un constructeur de cette forme avec un argument de type String

`Remarque :` *rien de difficile ici, ce code est donné dans un transparent du cours…*

```
package com.iut.cabinet.metier;

public class CabinetMedicalException extends Exception{

    public CabinetMedicalException(){
        super();
    }

    public CabinetMedicalException(String msg){
        super(msg);
    }
}
```

Dans le cas d'une ***exception métier***, il faut obliger la prise en compte de l'exception (`catch` obligatoire) pour pouvoir corriger l'erreur soulevée ⇒ `CabinetMedicalException` hérite de **`Exception`**…L'exception métier doit absolument être traitée : souvent propagée (jusqu 'au contrôleur de use case) et finalement attrapée (en principe dans le Controleur de Use Case qui les traite, on verra plus tard….)
Rappel : `Runtime` pour les exceptions « techniques »

3. Lever une exception contrôlée et propager l'exception dans `verifierNir` .

3 modifications à effectuer : `void, throw et throws`…
→ Pour lever une exception dans le code, on utilise le mot clé **`throw`** :

```
      if (cleCalculee !=cle)
      {
throw new CabinetMedicalException("Le NIR proposé est incorrect : "+ nirATester);
       }
```

→ *« Déclarer » ou « Traiter »* : l'énoncé demande de laisser remonter (jusqu'au contrôleur de Use Cas) donc on déclare en utilisant le mot clé : **`throws`** . Le `boolean` devient `void`…

```
public static void verifierNir(String nirATester) throws CabinetMedicalException

/** Teste la validité d'un NIR en recalculant la clé de contrôle
 * à partir de 13 premiers caractères du <code>nirATester</code>.
 * @param nirATester <code>String</code> composé de 15 caractères
 * respectant la norme officielle d'un NIR
 * @return true  si la clé recalculée à partir des 13 premiers caractères
 * du <code>nirATester</code> est bien égale à la clé extraite du <code>nirATester</code>
* @exception  CabinetMedicalException  si la clé recalculée à partir des 13 premiers
* caractères du <code>nirATester</code> n'est pas égale à la clé extraite du
* <code>nirATester</code>
          */
           //        void

      public static boolean verifierNir (String nirATester) throws
                                                    CabinetMedicalException
            {
            if (nirATester == null) return false
                                                    throw new
                                                    CabinetMedicalException("Aucun NIR
                                                    passé en paramètre (objet null)");
            if (nirATester.length() !=15) return false ;   throw new
            ////////////////////////////////////////////   CabinetMedicalException("Le
            // Extraction des 13 premiers caractères et de la  NIR proposé est incorrect :
            ////////////////////////////////////////////   il faut 15 caractères "+
            String nir_13 = nirATester.substring(0,13); //  nirATester);
            int cle = Integer.parseInt(nirATester.substring(13,15));

            ////////////////////////////////////////////////////////////
            // Conversion du nir_13 String en long (eh oui… 13 chiffres…)
            // et calcul de la cle dans la variable cleCalculee
            ////////////////////////////////////////////////////////////
            … Voir votre code du TP précédent …

            ///////////////////////
            // Validité de la clé
            ///////////////////////
            if (cleCalculee != cle)
                  return true ;     throw new CabinetMedicalException("Le NIR proposé
            else return false;      est incorrect : "+ nirATester);
            }
```


le beau programme…

```
/** Teste la validité d'un NIR en recalculant la clé de contrôle
 * à partir de 13 premiers caractères du <code>nirATester</code>.
 * @param nirATester <code>String</code> composé de 15 caractères
 * respectant la norme officielle d'un NIR
* @exception  CabinetMedicalException  si la clé recalculée à partir des 13 premiers
* caractères du <code>nirATester</code> n'est pas égale à la clé extraite du
* <code>nirATester</code>

      public static void verifierNir (String nirATester) throws
                                                    CabinetMedicalException
            {
            if (nirATester == null) throw new CabinetMedicalException("Aucun NIR
      passé en paramètre (objet null)");

      if (nirATester.length() !=15) throw new CabinetMedicalException("Le NIR
      proposé est incorrect : il faut 15 caractères "+ nirATester);
            ////////////////////////////////////////////////////////////
            // Extraction des 13 premiers caractères et de la clé
            ////////////////////////////////////////////////////////////
            String nir_13 = nirATester.substring(0,13); //13 car endIndex-1 (javadoc!)
            int cle = Integer.parseInt(nirATester.substring(13,15));

            ////////////////////////////////////////////////////////////
            // Conversion du nir_13 String en long (eh oui… 13 chiffres…)
            // et calcul de la cle dans la variable cleCalculee
            ////////////////////////////////////////////////////////////
            … Voir votre code du TP précédent …

            ///////////////////////
            // Validité de la clé
            ///////////////////////
            if (cleCalculee != cle)
throw new CabinetMedicalException("Le NIR proposé est incorrect : "+ nirATester);
            }
```

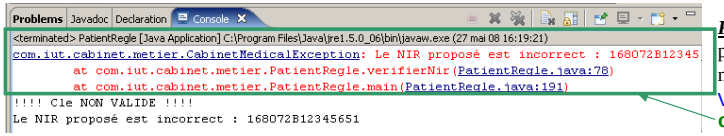
↳3.2 Modifier l'extrait de code n°2 afin que ***l'exception soit traitée*** dans la méthode **`main`**.

```
public static void main(String[] args) {
            String unNir="168072B12345652";

             try
                {
                 PatientRegle.verifierNir(unNir);
                 System.out.println("Cle VALIDE");
                }
                catch (CabinetMedicalException cme)
                      {
                      System.out.println("!!!! Cle NON VALIDE !!!!");

                      //facultatif, seulement si on veut récupérer le message
                      // stocké dans l'exception et l'afficher
                      String message=cme.getMessage();
                      System.out.println(message);
                      }
            }
```

➢ S'il n'y a pas de bloc `try…catch…`, le programme ne compile pas ! ! ! !
puisque `CabinetMedicalException` doit absolument être attrapée. C'est une exception « normale » qui hérite de « `Exception` » (voir doc) et qu'il est indispensable d'attraper…
Rappel : le `try..catch` est facultatif pour seulement les « `Error` » et « `RuntimeError` » ! ! !



Remarque : dans le `catch`, on pourrait aussi rajouter avant le message `"!!!! Cle NON VALIDE !!!! "`, la commande : **`cme.printStackTrace();`**

`printStackTrace` met le message stocké dans l'exception.

4. ↳ 4.1 Quelle(s) méthode(s) de la classe `Patient` pourrai(en)t être susceptible(s) d'appeler `verifierNir` et de propager une `CabinetMedicalException`?

La question revient à se demander : quand doit-on appeler la méthode **`verifierNir`** et continuer à propager l'exception puisqu'on souhaite que cette exception se propage jusqu'au contrôleur de Use Case.

Dans un premier temps, on pense à ***toutes les méthodes*** de la classe **`Patient`** qui ***manipulent et modifient un nir*** et sont susceptibles d'enregistrer son état dans un `Patient` c'est-à-dire :

- le *setteur du nir* : **`setNir`** où un contrôle du ***nir*** s'impose ! ! !
- et les *constructeurs…*

↳ 4.2 Comment faire ***pour faciliter la maintenance de votre code*** et faire intervenir l'appel à **`verifierNir`** dans une seule méthode de la classe `Patient.`
Quelle nouvelle règle d'écriture des classes métier en déduisez-vous ?
Mettez à jour le document reprenant les règles d'écriture des classes métier du projet Cabinet Médical

Afin de faciliter la maintenance du code, nous ferons intervenir la méthode **`verifierNir`** uniquement dans le ***setteur***.
Dans les constructeurs pour mettre à jour la valeur des attributs :

- il suffit alors de faire appel au setteur : `setNir(nir);`
- plutôt que d'écrire une affectation du type : `this.nir=nir;`.

Cela devient d'ailleurs une de nos **nouvelles règles d'écriture** des classes métier

Nouvelle règle d'écriture des classes métier

Les constructeurs doivent utiliser des appels aux *setteurs* plutôt que des affectations directes du type `this.unAttribut=uneNouvelleValeur`

Mettez à jour le document reprenant les règles d'écriture des classes métier du projet Cabinet Médical

↳ 4.3 D'après les deux questions précédentes, effectuer les modifications nécessaires dans la classe métier `Patient` pour tester la validité du NIR et lever des `CabinetMedicalException` .

Volontairement, on ne traitera pas d'exception(s) dans la classe métier `Patient`, mais on la(les) laissera se propager…
On laissera remonter les exceptions jusqu'au contrôleur de Use Case (que l'on a pas encore écrit…)

→ ***Modification du setteur :***

```
////////////
// nir
public void setNir(String nir) throws CabinetMedicalException
  {
  PatientRegle.verifierNir(nir);
  this.nir = nir;
  }
```

… si les constructeurs utilisent des setteurs, la modification de `setNir` entraîne des erreurs de compilations dans les constructeurs puisque l'exception `CabinetMedicalException` se propage dans les constructeurs.
Pour supprimer ces erreurs de compilation, il faut alors :

- soit ***traiter*** l'exception `CabinetMedicalException`
- soit la ***déclarer*** (c-a-d ***la propager***)

Notre choix de conception est de ***propager l'exception jusqu'au contrôleur de use case*** (cf énoncé).
Donc on doit aussi utiliser des **`throws`** dans les constructeurs.

→ ***Modification des constructeurs :… on laisse l'exception se propageait, on ne la traite pas***
=> **`throws CabinetMedicalException`**

```
public Patient (Integer idPersonne,String nom,String prenom,
                Date dateNaissance, boolean isMale,String telephone,
                String portable, String email, Adresse adresse,
                Personne unAscendant,
                String nir, String medecinTraitant)
                                throws CabinetMedicalException

public Patient (String nom,String prenom,Date dateNaissance,boolean
isMale, Adresse adresse, String nir)
                        throws CabinetMedicalException

public Patient (String nom,String prenom,Date dateNaissance,boolean
isMale, Adresse adresse,Personne unAscendant,String nir)
                        throws CabinetMedicalException
```