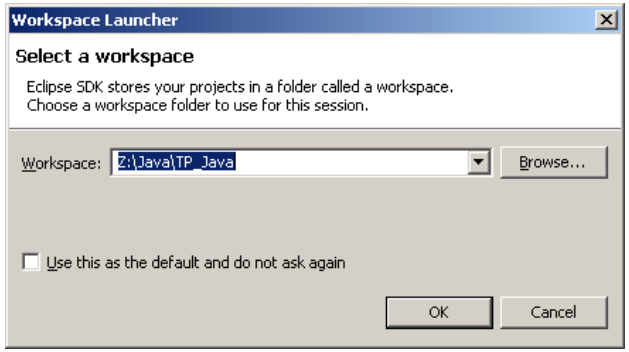
TP JAVA n°2: Héritage et Polymorphisme

Ouvrez Eclipse en cliquant sur l'icône correspondante.
Vérifiez que le **Workspace Launcher** contient bien le chemin que vous aviez tapé au TP précédent lors de la création du workspace à savoir : `Z:\Java\TP_Java`



Le workspace TP_Java s'ouvre avec les projets créés au TP précédent .
Redépliez dans la vue `Package Explorer` ce qui concerne le `cabinetMedical.`

Exercice 1 : Optimisation du code dans la méthode `toString`

Cet exercice permet de mettre en pratique le transparent n°35 ***Concaténer des chaînes de caractères*** du cours ***Classe et Objet***

Si vous avez utilisé l'opérateur **+** pour la concaténation de la chaîne dans la méthode `toString`, placer votre curseur sur la ligne de commande correspondant à cette concaténation,
et utiliser le raccourci-clavier **`Ctrl+1`** :
Eclipse vous propose une liste d'actions possibles relatives à la concaténation :

Use 'StringBuilder' for string concatenation
Use 'MessageFormat' for string concatenation

Le premier permet d'utiliser un objet **`StringBuilder`** pour générer la chaîne, alors que le second se base sur la classe **`MessageFormat`** qui propose des options de formatage plus évoluées .

Choisir ***Use 'StringBuilder' for string concatenation*** pour optimiser les performances de votre application.

Dans un premier temps, il est bien sûr plus commode d'écrire la méthode `toString` avec des `+`. Mais dans un soucis d'optimisation, et notamment lors de la remise des sources de votre application, vous veillerez bien à transformer vos méthodes `toString` afin qu'elles ne manipulent plus que des `Stringbuilder` et non des `+.`

Exercice 2 :

a. Transformer la classe **`Personne`** en classe abstraite mère vue en TD.
Ne pas oublier de passer les attributs en **`protected`** !

b. Ajouter dans le paquetage **`com.iut.cabinet.metier`** la classe fille **`Patient`** vue en TD.

➔ ***Pour gagner du temps***, après avoir écrit la déclaration des deux attributs, vous pouvez ***générer automatiquement les getteurs et setteurs*** comme nous l'avions évoqué lors du TP précédent.
Sélectionnez :
`Source → Generate Getter and Setter`

➔ N'oubliez pas de rajouter dans la classe `Patient`, toutes les méthodes nécessaires afin de bien respecter les règles d'écritures d'une classe métier (voir énoncé du TD n°1, notamment : `equals, hashCode` et `toString`)

c. Utilisation d'une annotation : **`@Override`**
`@Override` est une annotation qui demande au compilateur de vérifier que l'on redéfinit bien une méthode.
Rajouter cette annotation devant la déclaration de la méthode `equals`
Passer la souris sur **`@Override`** et lire le message.
… Pour l'instant rien ne change…
Pour illustrer l'intérêt de l'annotation, faire une faute d'orthographe en écrivant par exemple `equal` à la place d'`equals.` Sauver, et là le compilateur détecte alors une erreur…

Exercice 3 :

Tester votre code dans une application que vous nommerez **`EssaiCabMed_v2.java`** que vous placerez dans le package
`com.iut.cabinet.essai`

Cette application devra instancier deux patients qui auront les caractéristiques données dans le ***tableau n°1 de l'annexe 1***.

Afficher les caractéristiques de ces deux patients comme le montre la copie d'écran ci-contre.

```
-----------------------------------------
--------     TEST des PATIENTS     -------
-----------------------------------------

 --- Premier Patient ---
Numéro: 1
Nom : DUPONT
Prenom : Julie
DateNaissance : 21/05/1960
isMale : false
Telephone :  0555434355
Portable : 0606060606
Email : julie.dupont@tralala.fr
Adresse :
         numéro: 15
         rue: avenue Jean Jaurès
         voie: null
         batiment: null
         codePostal: 87000
         ville: Limoges
         pays: France
Ascendant : null
NIR: LEDOC Paul
Medecin Traitant: 260058700112367

 --- Second Patient ---
Numéro: 2
Nom : DUPONT
Prenom : Toto
DateNaissance : 25/12/1991
isMale : true
Telephone :  0555430000
Portable : 0605040302
Email : toto.dupont@etu.unilim.fr
Adresse :
         numéro: 185
         rue: avenue Albert Thomas
         voie: null
         batiment: Résidence La Borie
         codePostal: 87065
         ville: Limoges
         pays: France
Ascendant : Numéro: 1
Nom : DUPONT
Prenom : Julie
DateNaissance : 21/05/1960
isMale : false
Telephone :  0555434355
Portable : 0606060606
Email : julie.dupont@tralala.fr
Adresse :
         numéro: 15
         rue: avenue Jean Jaurès
         voie: null
         batiment: null
         codePostal: 87000
         ville: Limoges
         pays: France
Ascendant : null
NIR: LEDOC Paul
Medecin Traitant: 260058700112367
NIR: LEDOC Paul
Medecin Traitant: 260058700112367
```

Exercice 4 :

↳ Revenir dans la classe `Personne` et modifier la méthode `toString()` en rajoutant un `super.toString()` comme dans le transparent n°15 du cours. Recompiler la classe `Personne` et le fichier précédent afin de re-exécuter le fichier `EssaiCabMed_v2` …Que constatez-vous ?

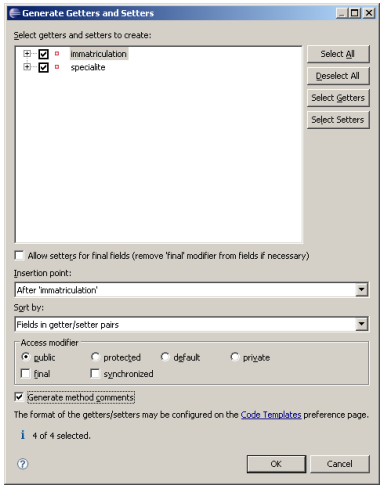
Exercice 5 :

Ajouter la classe fille **`Professionnel`** (description dans le TD n°2) dans le package **`com.iut.cabinet.metier`**

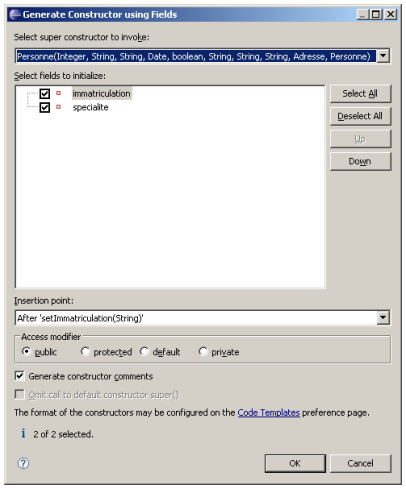
→ ***Pour gagner du temps***, après avoir écrit la déclaration des deux attributs,

```
private String immatriculation;
private String specialite;
```

vous pouvez ***générer automatiquement les getteurs et setteurs*** comme nous l'avions évoqué lors du TP précédent.
Sélectionnez :
`Source → Generate Getter and Setter`



→ ***Remarque :*** **vous pouvez également *générer automatiquement le constructeur avec tous les arguments (de `Personne` et `Professionnel`)***



Sélectionnez :
`Source → Generate constructor using Fields`

Dans la zone : *`Select super constructor to invoke`*, sélectionner le constructeur de `Personne` avec le plus d'arguments.

Vérifiez que les champs `immatriculation` et `specialite` soient bien cochés.
Si vous le souhaitez, cochez `Generate constructor comments`

Validez avec **`OK`**.

→ N'oubliez pas de rajouter dans la classe `Professionnel`, le constructeur par défaut, ainsi que tout autre constructeur que vous jugerez utile et toutes les méthodes nécessaires afin de bien respecter les règles d'écritures d'une classe métier (voir énoncé du TD n°1)

↳ ***Tester le code de la même manière*** que précédemment, en instanciant des professionnels ayant les caractéristiques données dans le ***tableau n°2 de l'annexe 1***.

↳ Afficher toutes les caractéristiques de ces professionnels.

Exercice 6 :

Ajouter dans le package **`com.iut.cabinet.essai`**, la classe `ListePersonneCabinetTab` vue en TD (aidez-vous éventuellement des transparents du cours concernant le **`GroupeTD`** et la **`ListeDeFormes`**)

Exercice 7 : Ecrire l'application **`EssaiCabMed_v22.java`** dans le package **`com.iut.cabinet.essai`**

Cette application va permettre de manipuler un objet de type `ListePersonneCabinetTab` : instancier cet objet, ajouter des `Personne` (`Patient` et `Professionnel`) dont vous choisirez les caractéristiques.
Tester vos méthodes de la classe `ListePersonneCabinetTab` en affichant :

- toutes les personnes présentes dans cet objet
- uniquement les patients stockés dans cet objet
- uniquement les professionnels stockés dans cet objet

Exercice 8 : **méthode statique `verifierNir`**

↳ Ajouter dans le paquetage **`com.iut.cabinet.metier`** une classe **`PatientRegle.`**
Dans cette classe, on demande d'écrire la **méthode statique `verifierNir` (voir cours n°2 pour explication `static`)** qui prend en paramètre d'entrée un `String` correspondant à un **NIR** à tester et qui renvoie un booléen à VRAI si le nir est correct (c.a.d si la clé est conforme à la clé attendue pour les 13 premiers chiffres du nir) . On demande ici de contrôler uniquement la validité de la clé.
La calcul de la clé est donné au point [4] de l'Annexe 2.

```
public static boolean verifierNir(String nirATester)
```

Tester votre méthode avec les numéros de sécurité sociales « valides » suivants :

- `260058700112367`
- `191128708545628` *… et votre propre nir ! ! !*
- `297112A10102401`
- `168072B12345652`

Tester également avec des numéros non « valides » (en falsifiant les numéros précédents par exemple)
Et des numéros incomplets (qui ne comportent pas 15 caractères)

Attention : Pour la prochaine séance les classes :

- `Patient`
- `Professionnel`

doivent absolument être implémentées, vérifier les règles d'écriture de classe,
… être testées et fonctionner ! ! ! !

- Egalement la classe `PatientRegle` doit implémenter une méthode `static verifierNir` qui fonctionne et cette méthode doit être utilisée à bon escient dans le programme

N'oubliez pas les commentaires javadoc ! ! !

Annexe1 :

Jeu d'essai pour les patients :

idPerson ne	nom	prenom	dateNaissance	sexe	telephone	portable	email	adresse	unAscendant	nir	medecinTraitant
1	DUPONT	Julie	21/05/1960	F	0555434355	0606060606	julie.dupont@tralala.fr	15 avenue Jean Jaurès 87000 Limoges France	NON	260058700112367	LEDOC Paul
2	DUPONT	Toto	25/12/1991	M	0555430000	0605040302	toto.dupont@etu.unilim.fr	185 avenue Albert Thomas Résidence La Borie 87065 Limoges France	DUPONT Julie	260058700112367	LEDOC Paul

Même nir car même ascendant…

Jeu d'essai pour les professionnels :

idPerson ne	nom	prenom	dateNaissance	sexe	telephone	portable	email	adresse	unAscendant	immmatriculation	spécialité
3	**LEDOC**	Paul	10/07/1976	M	0555434343	0612345678	paul.ledoc@lesmedecins.fr	3 rue de Limoges 87170 Isle	NON	871255358	Médecine générale
4	**CHILDREN**	Rose	16/02/1970	F	0555434343	0687654321	rose.children@lesmedecins.fr	10 avenue de la gare 87000 Limoges	NON	312444555	Pédiatrie

Annexe 2 : Numéro de sécurité sociale (source : wikipédia)

Le **numéro de Sécurité sociale** est un numéro formé de **13 chiffres**, plus une clé de 2 chiffres, servant en France à l'identification des individus par la Sécurité sociale, et créé sous l'Occupation allemande, à des fins de mobilisation militaire clandestine, par René Carmille.
Son nom administratif a été *Numéro national d'identité* (N.N.I.) et est actuellement *Numéro d'inscription au répertoire* (NIR), par référence au "Répertoire national d'identification des personnes physiques" (RNIPP) géré par l'INSEE.

Dans les années 1970, quand l'informatique se généralisa, **une clé à deux chiffres** fut ajoutée ***pour vérification par l'ordinateur : c'est le complément à 97 du reste de la division du numéro à 13 chiffres par 97 (pour plus de détails voir paragraphe [4])***

Le numéro de sécurité sociale est inscrit sur la Carte vitale.

La signification des chiffres est la suivante :

Cas	Positions	Signification	Valeurs possibles
Tous	1	sexe : 1 pour les hommes, 2 pour les femmes	1 ou 2
	2 et 3	deux derniers chiffres de l'année de naissance (ce qui donne l'année à un siècle près)	de 00 à 99
	4 et 5	mois de naissance	de 01 à 12, ou 20 [1]
A	6 et 7	département de naissance métropolitain (2A ou 2B pour la Corse) [2]	de 01 à 95
	8, 9 et 10	numéro d'ordre de la commune de naissance dans le département [2] [3]	de 001 à 989, ou 990 [1]
B	6, 7 à 8	département de naissance en outre-mer [2]	de 970 à 989
	9 et 10	numéro d'ordre de la commune de naissance dans le département [2] [3]	de 01 à 89, ou 90 [1]
C	6 et 7	naissance hors de France [2]	99
	8, 9 et 10	identifiant du pays de naissance [2]	de 001 à 989, ou 990 [1]
Tous	11, 12 et 13	numéro d'ordre de l'acte de naissance dans le mois et la commune (ou le pays) [3]	de 001 à 999
	14 et 15	clé de contrôle modulo 97 [4]	de 01 à 97

[1] Des codes spécifiques existent pour les personnes inscrites à partir d'un acte d'état civil incomplet (code mois supérieur à 20 si le mois de naissance est inconnu, code commune 990 si la commune (ou le pays) de naissance est inconnue). Ces cas sont extrêmement rares avec les formalités de déclaration de naissance actuelles.

[2] Pour les départements d'outre-mer, on retient le numéro de département à trois chiffres, et le numéro de commune sur deux chiffres. Les personnes nées à l'étranger ont un code département égal à 99 et un code commune remplacé par le code du pays de naissance à trois chiffres.
[3] Dans le cas où le nombre de naissances dépasse 999 un mois donné, un code extension commune est créé dans le même département (ou collectivité d'outre-mer) : il ne correspond à une commune donnée que pour un mois et une année donnée : certaines communes ont donc maintenant plusieurs codes d'extension attribués de façon permanente.

[4] Calcul de la clé de contrôle : diviser par 97 le nombre formé par les 13 premiers chiffres, prendre le reste de cette division, puis le complément à 97 (c'est-à-dire la différence entre 97 et le reste de la division). La clé de contrôle est égale à ce complément. Pour faire le calcul avec la calculatrice scientifique de l'ordinateur, procéder ainsi : taper le NIR (les 13 premiers chiffres), puis sur le MOD 97 ; on obtient ainsi le reste de la division que l'on retranche à 97 pour avoir les deux derniers chiffres qui forment la clé. Pour la Corse, les lettres A et B sont remplacées par des zéros, et on soustrait du nombre à 13 chiffres ainsi obtenu 1 000 000 pour A et 2 000 000 pour B.

Unicité

Le numéro de sécurité sociale n'est pas suffisant pour identifier de façon unique et certaine un individu, du fait de l'existence de doublons :

- pour un assuré et ses ayant-droits éventuels, jusqu'à ce que chacun des ayant-droits se voit attribuer et communiquer son numéro propre,
- pour les personnes nées hors métropole.
- pour les personnes de même sexe, nées à 100 ans d'intervalle, dans la même commune, dans le même numéro d'ordre.

Ce dernier point mérite d'être détaillé. L'année de naissance de l'individu est identifiée par ses deux derniers chiffres, soit à 100 ans près. Il y a par conséquent un risque de doublon entre individus nés avec 100 ans d'intervalle. Ce risque peu paraître très faible entre individus en vie, mais il ira croissant avec l'augmentation de l'espérance de vie. Il a été décuplé par l'informatisation du RNIPP en 1972, qui a entrainé la conservation illimitée des numéros de sécurité sociale y compris lorsque la personne est décédée.


```
package com.iut.cabinet.essai;

import com.iut.cabinet.metier.Personne;
import com.iut.cabinet.metier.Patient;
import com.iut.cabinet.metier.Professionnel;

public class ListePersonneCabinetTab {

     //attributs
     Personne[] liste = new Personne[30];
     int nbPersonnes = 0;

     // méthode permettant d'ajouter une personne maPers
     // dans le tableau
     public void ajouterPersonne (Personne maPers)
          {
          if (nbPersonnes < liste.length)
               liste[nbPersonnes++] = maPers;
          }

// méthode permettant d'afficher toutes les personnes du tableau
     public void afficheListe()
     {

          for (int i=0; i<nbPersonnes;i++)
               {
               System.out.println(liste[i]);
               System.out.println("------------------------------");
               }
       }


// méthode permettant d'afficher les patients présents ds le tableau
     public void affichePatient()
      {
          for (int i=0; i<nbPersonnes;i++)
          {
          if (liste[i] instanceof Patient)
               { System.out.println(liste[i]);
               System.out.println("------------------------------");
               }
          }
       }

//    méthode permettant d'afficher les professionnels présents
//    ds le tableau
     public void afficheProfessionnel()
          {
          for (int i=0; i<nbPersonnes;i++)
          {
          if (liste[i] instanceof Professionnel)
               { System.out.println(liste[i]);
```

```
                System.out.println("------------------------------");
                }
            }
        }
}//fin classe




public class EssaiCabMed_v22 {

     public static void main(String args[])
     {
    // ici on se contente d'appeler le constructeur.
          new EssaiCabMed_v22 ();
     }


     EssaiCabMed_v22 ()
     {
          // Instanciation de la Liste de Personne
          ListePersonneCabinetTab maListe = new
ListePersonneCabinetTab();


……

// Instanciation des patients
Patient patient1 = new  Patient(1,"DUPONT","Julie",
          DateUtil.toDate("21/05/1960",DateUtil.FRENCH_DEFAUT),
          false,"0555434355","0606060606","julie.dupont@tralala.fr",
new Adresse("15","avenue Jean Jaurès",null,null,"87000","Limoges",
"France"),
          null,
          "260058700112367","MARTIN Paul");

Patient  patient2 = new Patient(2,"DUPONT","Toto",
          DateUtil.toDate("25/12/1991",DateUtil.FRENCH_DEFAUT),
          true,"0555430000","0605040302","toto.dupont@etu.unilim.fr",
new Adresse("185","avenue Albert Thomas",null,"Résidence La Borie",
               "87065","Limoges","France"),
         patient1,
        "260058700112367","MARTIN Paul");

// Ajout des patients dans la liste...
if (patient1!= null) maListe.ajouterPersonne(patient1);
if (patient2!= null)maListe.ajouterPersonne(patient2);

// Instanciation des professionnels
Professionnel pro1=new Professionnel(3,"LEDOC","Paul",
     DateUtil.toDate("10/07/1976",DateUtil.FRENCH_DEFAUT),
     true,"0555434343","0612345678","paul.ledoc@lesmedecins.fr",
 new Adresse("3","rue de Limoges",null,null,"87170","Isle","France"),
```



```
 null,
 "871255358","generaliste");

Professionnel pro2=new Professionnel(4,"CHILDREN","Rose",
     DateUtil.toDate("16/02/1970",DateUtil.FRENCH_DEFAUT),
     true,"0555434343","0687654321","rose.children@lesmedecins.fr",
new Adresse("3","avenue de la gare",
            null,null,"87000","Limoges","France"),
            null, "312444555","pediatrie");

// Ajout des professionels dans la liste...
if (pro1!= null) maListe.ajouterPersonne(pro1);
if (pro2!= null)maListe.ajouterPersonne(pro2);

// Affichage des caractéristiques de TOUTES LES PERSONNES
          System.out.println(" \n ---- TOUTE LA LISTE----");
          maListe.afficheListe();

          System.out.println(" \n ---- LES PATIENTS----");
          maListe.affichePatient();

          System.out.println(" \n ---- LES PROFESSIONNELS----");
          maListe.afficheProfessionnel();

          System.out.println();
     }
}
```

Exercice 8: ***A propos de la méthode*** **`verifierNumSecu`** ***`Patient`***

Pourquoi mettre cette méthode dans une classe « à part » ?

La clé du numéro de sécurité sociale répond à des règles qui s'appuient sur la législation.
Dans une entreprise, le développeur qui s'occupera de la législation ne sera peut être pas le même que celui qui s'occupera des classes métier.
Si le protocole de calcul de la clé venait à changer (à se complexifier par exemple, car aujourd'hui il est relativement simple…), le développeur chargé de la législation interviendrait au niveau de la classe `PatientRegle` et en aucun cas au niveau de la classe métier `Patient`.

Différentes étapes de cette méthode

4 étapes :

Première Etape : Extraire :

- les 13 premiers caractères dans un `String`
- et la cle dans un `int`

Deuxième Etape : Conversion du `String` en `long` pour le numéro à 13 chiffres

- Pour la conversion utiliser les **long**

car les entiers de type `int` sont codés sur **4 octets** [4 -2 147 483 648 à 2 147 483 647]
=> 10 chiffres : c'est trop court !!!… on a besoin de 13 chiffres…
mais
les entiers de type `long` sont des entiers codés sur **8 octets**
=> [-9223372036854775808 à 9223372036854775807] => ça passe ! ! !

- Au préalable test pour savoir si on a une lettre ou pas : Corse 2A ou 2B
  - ⇨ A ou B position 7 donc indice=6 à remplacer par '0'
  - ⇨ si A on soustraira 1000000
  - ⇨ si B on soustraira 2000000

Troisième Etape : Calcul de la clé ...

Ne pas donner la formule au tableau, laisser les interpréter et formuler tout seul le point [4] de l'annexe : ils doivent être capables de se « dépatouiller » avec un cahier des charges…
diviser par 97 le nombre formé par les 13 premiers chiffres,
prendre le reste de cette division,
puis le complément à 97 (c-a-d la différence entre 97 et le reste de la division).

Quatrième Etape : Vérification des clés : extraite et calculée


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

public class PatientRegle {

public static boolean verifierNir (String nirATester)
      {
      ////////////////////////////////////////////////////
      // Premiere Etape :
      // Extraire les 13 premiers caractères et le cle
      ////////////////////////////////////////////////////
      if (nirATester == null) return false;
      if (nirATester.length() !=15) return false;
      String nir_13 = nirATester.substring(0,13); //13 car endIndex- 1  voir doc
      int cle = Integer.parseInt(nirATester.substring(13,15));

      ////////////////////////////////////////////////////
      // Deuxième Etape :
      // Conversion du String en int...
      ////////////////////////////////////////////////////
      // au préalable test pour savoir si on a une lettre ou pas
      // Corse 2A ou 2B => A ou B position 7 donc indice=6

      long nir ;//déclaration avant dans le case, sinon "Duplicate local variable..."
      // ATTENTION, il faut bien des long
// car les int sont codés sur 4 octets [4 -2 147 483 648 à 2 147 483 647]
//    => 10 chiffres : c'est trop court !!!… on a besoin de 13 chiffres…
// long sont des entiers codés sur 8 octets [-9223372036854775808 à 9223372036854775807]
// => ça passe ! ! !

      switch  (nir_13.charAt(6))    // Attention il faut à tout prix les () autour
                                    // de la variable à tester...
            {
            case 'a' :  nir_13=nir_13.replace('a','0');
                        nir = Long.parseLong(nir_13);
                        nir = nir - 1000000;
                        break;
            case 'A' :  nir_13=nir_13.replace('A','0');
                        nir = Long.parseLong(nir_13);
                        nir = nir - 1000000;
                        break;
            case 'b' :  nir_13=nir_13.replace('b','0');
                        nir = Long.parseLong(nir_13);
                        nir = nir - 2000000;
                        break;

            case 'B' :  nir_13=nir_13.replace('B','0');
                        nir = Long.parseLong(nir_13);
                        nir = nir - 2000000;
                        break;

            default  : nir = Long.parseLong(nir_13);// on avait déjà un chiffre
                        break; // juste la conversion String  int...
            }

      ////////////////////////////////////////////////////
      // Troisième Etape : Calcul de la clé ...
      ////////////////////////////////////////////////////
      long reste = nir%97; // reste de la division par 97
      long cleCalculee = 97-reste;       // puis le complément à 97

      ////////////////////////////////////////////////////
      //  Quatrième Etape : Vérification des clés ...
      ////////////////////////////////////////////////////
       if (cleCalculee==cle) return true ;
       else return false;
```

}

Rappel : méthode statique, pas besoin d'instancier la classe pour qu'on puisse appeler la méthode puisque cette méthode est partagée entre toutes les instances de la classe…

→ Utiliser la **méthode `verifierNir`** à bon escient dans le classe Patient… ce qui veut dire…
appel de `verifierNir` à chaque fois que le NIR est susceptible d'être modifié c-a-d:

- dans les **constructeurs** et
- dans le **setteur**…

```
public void setNir(String nir) {
if (PatientRegle.verifierNir(nir)==true)
                this.nir = nir;
}
```

Pour ce TD/TP on peut se contenter d'appel à verfierNIR dans le setteur et les constructeurs…

… Mais pour aller un peu plus loin… Soyons plus malin (cette démarche sera détaillée au TD/TP suivant)

Si on utilise un appel à **`setNir`** dans les constructeurs au lieu d'une simple mise à jour du **`nir`** de type **`this.nir = …`**
…
et il est alors nécessaire d'appeler `verifierNir` uniquement dans le setteur…
puisque le(s) constructeur(s) appelle le setteur qui appellera la métode **`verifierNir`** …

Donc il faut procéder dans toutes les classes métiers aux changements suivants :
Changer tous les constructeurs pour qu'ils appellent des **`set`** (et non effectuent de simlples m-a-j avec `this`)

… et dorénavant prendre l'habitude de procéder comme cela…
… en vue d'éventuelles autres vérifications de paramètres…

Method Summary

char	**charAt**(int index) Returns the char value at the specified index.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
String	**replace**(char oldChar, char newChar) Returns a new string resulting from replacing all occurrenc string with newChar.

java.lang
Class String

java.lang.Object
  └ **java.lang.String**

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

substring

```
public String substring(int beginIndex,
                        int endIndex)
```

Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.

Examples:

```
"hamburger".substring(4, 8) returns "urge"
"smiles".substring(1, 5) returns "mile"
```

Parameters:
`beginIndex` - the beginning index, inclusive.
`endIndex` - the ending index, exclusive.
Returns:
the specified substring.
Throws:
`IndexOutOfBoundsException` - if the `beginIndex` is negative, or `endIndex` is larger than the length of this `String` object, or `beginIndex` is larger than `endIndex`.

Remarque : Dans la conception objet il y a une différence entre une classe statique et un singleton :
singleton : une seule instance de la classe dans le programme
classe statique : plusieurs instances manipulant les mêmes paramêtres
Donc dans le concept, un singleton ne peut se dériver qu'en singleton, une classe statique, elle, peut se dériver en classe 'classique'. Un singleton peut néanmoins être un paramêtre d'instance dans d'autres classes.
La classe static n'a pas besoin d'être instanciée pour qu'on puisse appeller ses méthodes et accéder à ses attributs. Le constructeur n'a pas besion d'être appellé.
Le singleton (comme toute classe non statique) doit obligatoirement être instancié.

Si ça peut éclaircir les choses:
final class **interdit la création de *classe* dérivée**
alors que "singleton" **interdit la création de plus d'une *instance* de la classe.**