



JAVA 8 – PROGRAMMATION OBJET

JSE – SEQUENCE 3

LYCEE PASTEUR MONT-ROLAND
Julian.courbez@gmail.com

Table des matières

1.	Introduction.....	3
2.	Mise en œuvre avec Java	4
2.1.	Création d'une classe	4
	Déclaration d'une classe.....	4
	Création des champs	5
	Création de méthodes.....	6
	Les accesseurs	9
	Constructeurs et destructeurs.....	10
	Champs et méthodes statiques.....	10
	Les annotations	12
2.2.	Utilisation d'une classe.....	14
	Création d'une instance	14
	Initialisation d'une instance	15
	Destruction d'une instance	16
2.3.	Héritage	19
	this et super.....	21
	Classes abstraites	23
	Classes finales.....	25
	Conversion de type.....	25
	La classe Object	29
	Equals	32
	getClass.....	34
	toString.....	35
2.4.	Interface	35
	Création d'une interface	36
	Utilisation d'une interface.....	37
	Méthodes par défaut.....	41
2.5.	Classes imbriquées	47
	Classes imbriquées statiques.....	47
	Classes internes	48
	Classes anonymes.....	49
2.6.	Expression lambda.....	54
2.7.	Référence de méthode.....	61
2.8.	Les génériques.....	63
	Classes génériques	64

Définition d'une classe générique	64
Utilisation d'une classe générique	69
Méthodes génériques	71
Les génériques et l'héritage	72
Limitation des génériques	78
2.9. Les packages	80
Création d'un packages	80
Utilisation et importation d'un package.....	81
3. Gestion des exceptions	83
3.1. Les erreurs de syntaxe.....	83
3.2. Les erreurs d'exécution	84
3.3. Les erreurs de logique	84
Les exceptions	85
Récupération d'exceptions.....	85
Exceptions associées à des ressources.....	90
Création et déclenchement d'exceptions	94
4. Les collections.....	95
4.1. La classe ArrayList.....	96
4.2. La classe HashSet.....	100
4.3. La classe LinkedList.....	110
4.4. Streams et pipelines.....	112
5. Exercices	115
Exercice 1.....	115
Exercice 2.....	115
Exercice 3.....	115
Exercice 4.....	116

1. Introduction

Avec Java, la **notion d'objet** est omniprésente et nécessite un minimum d'apprentissage

Dans un langage procédural classique, le fonctionnement d'une application est réglé par une succession d'appels aux différentes procédures et fonctions disponibles dans le code. Ces procédures et fonctions sont chargées de manipuler les données de l'application qui sont représentées par les variables de l'application. Il n'y a aucun lien entre les données et le code qui les manipule.

Dans un langage objet on va au contraire essayer de regrouper le code. Ce regroupement est appelé une classe. Une application développée avec un langage objet est donc constituée de nombreuses classes représentant les différents éléments manipulés par l'application. Les classes vont décrire les caractéristiques de chacun des éléments. C'est ensuite l'assemblage de ces éléments qui va permettre le fonctionnement de l'application.

Les classes sont constituées de champs, attributs ou encore propriétés et de méthodes. Les champs représentent les caractéristiques des objets. Ils sont représentés par des variables et il est donc possible de lire leur contenu ou de leur affecter une valeur directement. Elles sont mises en œuvre par la création de procédures ou de fonctions dans une classe.

Ceci n'est qu'une facette de la programmation orientée objet. Trois autres concepts sont également fondamentaux :

- **L'encapsulation**
- **L'héritage**
- **Le polymorphisme**

L'encapsulation consiste à cacher les éléments qui ne sont pas nécessaires pour l'utilisation d'un objet. Cette technique permet de garantir que l'objet sera correctement utilisé. C'est un principe qui est aussi largement utilisé dans d'autres domaines que l'informatique.

Les éléments d'une classe visibles de l'extérieur de la classe sont appelés l'interface de la classe.

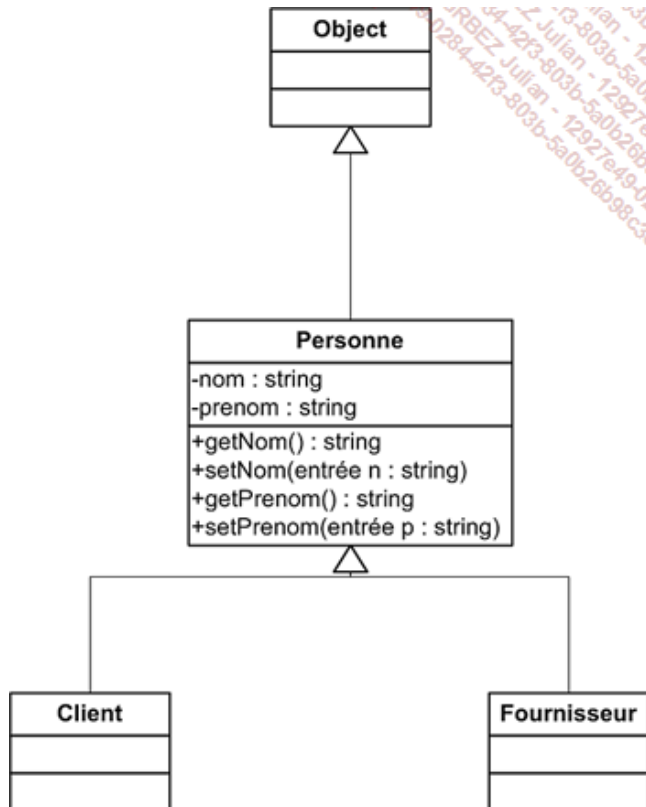
L'héritage permet la création d'une nouvelle classe à partir d'une classe existante. La classe servant de modèle est appelée classe de base. La classe ainsi créée hérite des caractéristiques de sa classe de base. Elle peut aussi être personnalisée en y ajoutant des caractéristiques supplémentaires. Les classes créées à partir d'une classe de base sont appelées classes dérivées.

Le polymorphisme est une autre notion importante de la programmation orientée objet. Par son intermédiaire, il est possible d'utiliser plusieurs classes de manière interchangeable même si le fonctionnement interne de ces classes est différent.

Deux autres concepts sont également associés au polymorphisme. La surcharge et le masquage de méthodes. La surcharge est utilisée pour concevoir dans une classe des méthodes ayant le même nom mais ayant un nombre de paramètres différent ou des types de paramètres différents. Le masquage est utilisé lorsque dans une classe dérivée, on souhaite modifier le fonctionnement d'une méthode dont on a hérité. Le nombre et le type des paramètres restant identiques à ceux définis dans la classe de base.

2. Mise en œuvre avec Java

Pour la suite de la séquence, nous allons travailler sur la classe *Personne* dont la représentation UML (Unified Modeling Language) est disponible ci-dessous. UML est un langage graphique dédié à la représentation des concepts de programmation orientée objet.



2.1. Création d'une classe

La création d'une classe passe par la déclaration de la classe elle-même et de tous les éléments la constituant.

Déclaration d'une classe

La déclaration d'une classe se fait en utilisant le mot clé `class` suivi du nom de la classe puis d'un bloc de code délimité par les caractères `{` et `}`. Dans ce bloc de code, on trouve des déclarations de variables qui seront les champs de la classe et des fonctions qui seront les méthodes de la classe.

Plusieurs mots clés peuvent être ajoutés pour modifier les caractéristiques de la classe. La syntaxe générale de déclaration d'une classe est donc la suivante.

```
[liste de modificateurs] class NomDeLaClasse [extends
NomDeLaClasseDeBase]
    [implements NomDeInterface1, NomDeInterface2, ...]
{
    Code de la classe
}
```

Les signes `[et]` sont utilisés ici pour indiquer le caractère optionnel de l'élément. Ils ne doivent pas être utilisés dans le code de déclaration d'une classe.

Les modificateurs permettent de déterminer la visibilité de la classe et comment vous pouvez l'utiliser. Voici la liste des modificateurs disponibles :

public : indique que la classe peut être utilisée par toutes les autres classes. Sans ce modificateur la classe ne sera utilisable que par les autres classes faisant partie du même package.

abstract : indique que la classe est abstraite et ne peut donc pas être instanciée. Elle ne peut être utilisée que comme classe de base dans une relation d'héritage. En général, dans ce genre de classe seules les déclarations de méthodes sont définies, et il faudra écrire le contenu des méthodes dans les classes dérivées.

final : la classe ne peut pas être utilisée comme classe de base dans une relation d'héritage et peut uniquement être instanciée.

La signification des mots clés **abstract** et **final** étant contradictoire leur utilisation simultanée est bien sûr interdite.

Pour indiquer que votre classe récupère les caractéristiques d'une autre classe par une relation d'héritage, vous devez utiliser le mot clé **extends** suivi du nom de la classe de base. Vous pouvez également implémenter dans votre classe une ou plusieurs interfaces en utilisant le mot **implements** suivi de la liste des interfaces implémentées.

Le début de la déclaration de notre classe **Personne** est donc le suivant :

```
public class Personne
{

}
```

Ce code doit obligatoirement être saisi dans un fichier ayant le même nom que la classe et l'extension **.java**.

Création des champs

Intéressons-nous maintenant au contenu de notre classe. Nous devons créer les différents champs de la classe. Pour cela, il suffit de déclarer des variables à l'intérieur du bloc de code de la classe en indiquant la visibilité de la variable, son type et son nom.

```
[private | protected | public] typeDeLaVariable nomDeLaVariable;
```

La visibilité de la variable répond aux règles suivantes :

La visibilité de la variable répond aux règles suivantes :

private : la variable n'est accessible que dans la classe où elle est déclarée.

protected : la variable est accessible dans la classe où elle est déclarée, dans les autres classes faisant partie du même package et dans les classes héritant de la classe où elle est déclarée.

public : la variable est accessible à partir de n'importe où.

Si aucune information n'est fournie concernant la visibilité, la variable est accessible à partir de la classe où elle est déclarée et des autres classes faisant partie du même package. Lorsque que vous choisissez la visibilité d'une variable vous devez autant que possible respecter le principe d'encapsulation et limiter au maximum la visibilité des variables. L'idéal étant de toujours avoir des variables **private** ou **protected** mais jamais **public**.

La variable doit également avoir un type. Il n'y a pas de limitation concernant le type d'une variable et vous pouvez donc utiliser aussi bien les types de base du langage Java tels que **int**, **float**, **char**, ... que des types d'objets.

Le nom de la variable doit quant à lui simplement respecter les règles de nommage (pas d'utilisation de mot clé du langage).

La classe `Personne` prend donc maintenant la forme suivante :

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;
}
```

Création de méthodes

Les méthodes sont simplement des fonctions définies à l'intérieur d'une classe. Elles sont en général utilisées pour manipuler les champs de la classe. La syntaxe générale de déclaration d'une méthode est décrite ci-dessous.

```
[modificateurs] typeDeRetour nomDeLaMethode ([listeDesParamètres])
                                   [throws listeException]
{
}
}
```

Si aucun de ces mots clés n'est utilisé alors la visibilité sera limitée au package dans lequel la classe est définie.

static : indique que la méthode est une méthode de classe.

abstract : indique que la méthode est abstraite et qu'elle ne contient pas de code. La classe où elle est définie doit elle aussi être abstraite.

final : indique que la méthode ne peut pas être substituée dans une sous-classe.

native : indique que le code de la méthode se trouve dans un fichier externe écrit dans un autre langage.

synchronized : indique que la méthode ne peut être exécutée que par un seul thread à la fois.

Le type de retour peut être n'importe quel type de données, type de base du langage ou type objet. Si la méthode n'a pas d'information à renvoyer vous devez utiliser le mot clé **void** en remplacement du type de retour.

La liste des paramètres est identique à une liste de déclaration de variables. Il faut spécifier le type du paramètre et le nom du paramètre. Si plusieurs paramètres sont attendus il faut séparer leur déclaration par une virgule. Même si aucun paramètre n'est attendu, les parenthèses sont tout de même obligatoires.

Le mot clé **throws** indique la liste des exceptions que cette méthode peut déclencher lors de son exécution.

Ajoutons deux méthodes à notre classe `Personne`.

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;

    public long calculAge()
    {
        return date_nais.until(LocalDate.now(), ChronoUnit.YEARS);
    }
}
```

```

public void affichage()
{
    System.out.println("nom : " + nom);
    System.out.println("prénom : " + prenom);
    System.out.println("âge : " + calculAge());
}
}

```

Dans certains langages de programmation, il n'est pas possible d'avoir plusieurs fonctions portant le même nom. Le langage Java comme beaucoup de langages objet permet de contourner le problème en créant des fonctions surchargées. Une fonction surchargée porte le même nom qu'une autre fonction de la classe mais présente une signature différente. Les informations suivantes sont prises en compte pour déterminer la signature d'une fonction :

- **Le nom de la fonction**
- **Le nombre de paramètres attendus par la fonction**
- **Le type des paramètres.**

Pour pouvoir créer une fonction surchargée il faut qu'au moins un de ses éléments change par rapport à une fonction déjà existante. Le nom de la fonction devant rester le même pour pouvoir réellement parler de surcharge, nous ne pouvons donc agir que sur le nombre de paramètres ou leur type. Nous pouvons par exemple ajouter la fonction suivante à la classe `Personne` :

```

public void affichage(boolean francais)
{
    if (francais)
    {
        System.out.println("nom : " + nom);
        System.out.println("prénom : " + prenom);
        System.out.println("âge : " + calculAge());
    }
    else
    {
        System.out.println("name : " + nom);
        System.out.println("first name : " + prenom);
        System.out.println("age : " + calculAge());
    }
}
}

```

Elle possède effectivement une signature différente de la première fonction `affichage` que nous avons créée puisqu'elle attend un paramètre de type `boolean`. Si maintenant nous ajoutons la fonction suivante le compilateur refuse la compilation du code.

```

public void affichage(boolean majuscule)
{
    if (majuscule)
    {
        System.out.println("nom : " + nom.toUpperCase());
        System.out.println("prénom : " + prenom.toUpperCase());
        System.out.println("âge : " + calculAge());
    }
    else
    {
        System.out.println("nom : " + nom.toLowerCase());
        System.out.println("prénom : " + prenom.toLowerCase());
        System.out.println("âge : " + calculAge());
    }
}
}

```



```
    }  
}
```

Il détermine en fait que deux fonctions ont rigoureusement la même signature, même nom, même nombre de paramètres, même type de paramètre. Cet exemple nous montre également que le nom des paramètres n'est pas pris en compte pour déterminer la signature d'une fonction.

Une fonction peut être conçue pour accepter un nombre variable de paramètres. La première solution consiste à utiliser comme paramètre un tableau et à vérifier dans le code de la fonction la taille du tableau pour obtenir les paramètres. Cette solution nécessite cependant la création d'un tableau au moment de l'appel de la fonction. Elle est donc moins évidente que l'utilisation d'une liste de paramètres. Pour simplifier l'appel de ce type de fonction, vous pouvez utiliser la déclaration suivante pour indiquer qu'une fonction attend un nombre quelconque de paramètres.

```
public void affichage(String...couleurs)  
{  
    {  
        if (couleurs==null)  
        {  
            System.out.println("pas de couleur");  
            return;  
        }  
        switch (couleurs.length)  
        {  
            case 1:  
                System.out.println("une couleur");  
                break;  
            case 2:  
                System.out.println("deux couleurs");  
                break;  
            case 3:  
                System.out.println("trois couleurs");  
                break;  
            default :  
                System.out.println("plus de trois couleurs");  
        }  
    }  
}
```

À l'intérieur de la méthode, le paramètre couleurs est considéré comme un tableau (de chaînes de caractères dans notre cas).

Par contre, lors de l'appel de la fonction, vous utilisez une liste de chaînes de caractères séparées par des virgules. Les syntaxes suivantes sont tout à fait valides pour l'appel de cette fonction.

```
p.affichage("rouge");  
p.affichage("vert", "bleu", "rouge");  
p.affichage();
```

Il y a juste une petite anomalie au moment de l'exécution car le dernier appel de la fonction affichage n'exécute pas la fonction que nous venons de concevoir mais la première version qui n'attend aucun paramètre. En effet le compilateur ne peut pas deviner qu'il s'agit de la version qui attend un nombre variable de paramètres que nous souhaitons exécuter sans lui passer de paramètre. Pour lui montrer notre intention nous devons donc appeler la fonction avec la syntaxe suivante.

```
p.affichage(null);
```

Lors de l'appel d'une fonction les paramètres sont passés par valeur aussi bien pour les types de base du langage (**int**, **float**, **boolean**...) que pour les types objets. Cependant si un objet est passé comme paramètre à une fonction, le code de la fonction a accès aux champs de l'objet et peut donc modifier leurs valeurs. Par contre si le code de la fonction modifie la référence vers l'objet, celle-ci sera rétablie après le retour de la fonction.

Il faut noter que dans les méthodes de la classe nous pouvons utiliser les champs de la classe même s'ils sont déclarés **private**, car nous sommes à l'intérieur de la classe.

Les accesseurs

La déclaration des attributs avec une visibilité privée est une bonne pratique pour respecter le principe d'encapsulation. Toutefois cette solution est limitative puisque seul le code de la classe où ils sont déclarés peut y accéder. Pour pallier ce problème vous devez mettre en place des accesseurs. Ce sont des fonctions ordinaires qui ont simplement pour but de rendre visibles les champs à partir de l'extérieur de la classe. Par convention, les fonctions chargées d'affecter une valeur à un champ sont nommées **set** suivi du nom du champ, les fonctions chargées de fournir la valeur d'un champ sont nommées **get** suivi du nom du champ. Si le champ est de type **boolean**, le préfixe **get** est remplacé par le préfixe **is**. Si un champ doit être en lecture seule, l'accesseur set ne doit pas être disponible, si un champ doit être en écriture seule alors c'est la fonction **get** qui doit être omise.

Avec cette technique vous pouvez contrôler l'utilisation qui sera faite des champs d'une classe. Nous pouvons donc modifier la classe *Personne* en y ajoutant quelques règles de gestion :

- Le nom doit être en majuscules.
- Le prénom doit être en minuscules.

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_naiss;

    public String getNom()
    {
        return nom;
    }

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

    public String getPrenom()
    {
        return prenom;
    }

    public void setPrenom(String p)
    {
        prenom = p.toLowerCase();
    }
}
```

Les champs de la classe sont maintenant accessibles depuis l'extérieur grâce à ces méthodes.

```
p.setNom("dupont");
p.setPrenom("albert");
```

```
System.out.println(p.getNom());  
System.out.println(p.getPrenom());
```

Constructeurs et destructeurs

Les constructeurs sont des méthodes particulières d'une classe par différents aspects. Le constructeur est une méthode portant toujours le même nom que la classe elle-même. Il ne retourne aucun type, même pas **void**. Il n'est jamais appelé explicitement dans le code mais de manière implicite, à la création d'une instance de classe. Comme pour une méthode classique, un constructeur peut attendre des paramètres. Le constructeur d'une classe qui n'attend pas de paramètres est désigné comme le constructeur par défaut de la classe. Le rôle du constructeur est principalement l'initialisation des champs d'une instance de classe. Comme les autres méthodes d'une classe, les constructeurs peuvent également être surchargés. La création d'un constructeur par défaut n'est pas obligatoire car le compilateur en fournit un automatiquement. Ce constructeur effectue simplement un appel au constructeur de la super classe qui doit bien sûr exister. Par contre si votre classe contient un constructeur surchargé, elle doit également posséder un constructeur par défaut.

Une bonne habitude à prendre consiste à toujours créer un constructeur par défaut pour chacune de vos classes. Ajoutons à la classe *Personne* des constructeurs.

```
public Personne()  
{  
    nom="";  
    prenom="";  
    date_nais=null;  
}  
public Personne(String n,String p,LocalDate d)  
{  
    nom=n;  
    prenom=p;  
    date_nais=d;  
}
```

Les destructeurs sont d'autres méthodes particulières d'une classe. Comme les constructeurs ils sont appelés implicitement mais uniquement lors de la destruction d'une instance de classe. La signature du destructeur est imposée. Cette méthode doit être **protected**, elle ne retourne aucune valeur, se nomme obligatoirement **finalize**, ne prend aucun paramètre et elle est susceptible de déclencher une exception de type **Throwable**.

Du fait de cette signature imposée, il ne peut y avoir qu'un seul destructeur pour une classe, donc pas de surcharge possible pour les destructeurs. La déclaration d'un destructeur est donc la suivante :

```
protected void finalize() throws Throwable  
{  
  
}
```

Le code présent dans le destructeur doit permettre la libération des ressources utilisées par la classe. On peut par exemple y trouver du code fermant un fichier ouvert par la classe ou la fermeture d'une connexion vers un serveur de base de données.

Champs et méthodes statiques

Les membres statiques sont des champs ou méthodes qui sont accessibles par la classe elle-même ou par toutes les instances de la classe.

Ils sont très utiles lorsque vous avez à gérer dans une classe des informations qui ne sont pas spécifiques à une instance de la classe mais à la classe elle-même. Par opposition aux membres d'instance pour lesquels il existe un exemplaire par instance de la classe, les membres statiques existent en un exemplaire unique.

La modification de la valeur d'un membre d'instance ne modifie la valeur que pour cette instance de classe alors que la modification de la valeur d'un membre statique modifie la valeur pour toutes les instances de la classe. Les membres statiques sont assimilables à des variables globales dans une application. Ils sont utilisables dans le code en y faisant référence par le nom de la classe ou grâce à une instance de la classe. Cette deuxième solution n'est pas conseillée car elle ne met pas évidence le fait que l'on travaille avec un membre statique. Le compilateur déclenche un avertissement s'il détecte cette situation.

Les méthodes statiques suivent les mêmes règles et peuvent servir à la création de bibliothèques de fonctions. L'exemple classique est la classe Math dans laquelle de nombreuses fonctions statiques sont définies. Les méthodes statiques possèdent cependant une limitation car elles ne peuvent utiliser que des variables locales ou d'autres membres statiques de la classe. Elles ne peuvent pas utiliser des membres d'instance d'une classe car il se peut que la méthode soit utilisée sans qu'il existe une instance de la classe.

Le compilateur détectera cette situation en indiquant :

```
non-static variable cannot be referenced from a static context.
```

Les membres statiques doivent être déclarés avec le mot clé **static**. Vous pouvez également, comme pour n'importe quel autre membre d'une classe, spécifier une visibilité. Par contre une variable locale à une fonction ne peut pas être statique.

Pour illustrer l'utilisation des membres statiques nous allons ajouter à la classe Personne un champ numéro. La valeur de ce champ sera renseignée automatiquement à la création de chaque instance de la classe et sera unique pour chaque instance. Les constructeurs de notre classe sont tout à fait adaptés pour réaliser ce travail. Par contre, il nous faut mémoriser combien d'instances ont été créées pour pouvoir affecter un numéro unique à chaque instance. Une variable statique privée va se charger de cette opération. Voici ci-dessous le code correspondant.

```
public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;
    // champ privé représentant le numéro de la Personne
    private int numero;
    // champ statique privé représentant le compteur de Personnes
    private static int nbInstances;

    public String getNom()
    {
        return nom;
    }

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

    public String getPrenom()
```

```

    {
        return prenom;
    }

    public void setPrenom(String p)
    {
        prenom = p.toLowerCase();
    }
    // méthode d'instance permettant d'obtenir le numéro d'une Personne
    public int getNumero()
    {
        return numero;
    }
    // methode statique permettant d'obtenir le nombre d'instances créées
    public static int getNbInstances()
    {
        return nbInstances;
    }
    public Personne()
    {
        nom="";
        prenom="";
        date_nais=null;
        // création d'une nouvelle Personne donc incrémentation du
compteur
        nbInstances++;
        // affectation à la nouvelle Personne de son numéro
        numero=nbInstances;
    }
}

```

Les annotations

Les annotations sont utilisées pour ajouter à un élément des informations supplémentaires. Ces informations n'ont aucun effet sur le code mais peuvent être utilisées par le compilateur, la machine virtuelle qui se chargera de l'exécution de l'application ou certains outils de développement. Elles peuvent s'appliquer à une classe, un champ, une méthode. Elle doit être spécifiée avant l'élément auquel elle se rapporte. Une annotation est précédée du symbole **@** et suivie du nom de l'annotation. Le compilateur reconnaît trois types d'annotations qui vont permettre la modification de son comportement au moment de la compilation du code.

@Deprecated est utilisé pour indiquer qu'une méthode ne doit plus être utilisée. C'est par exemple le cas lorsque vous avez décidé de faire évoluer une méthode et que vous souhaitez que la version précédente ne soit plus utilisée. Cette annotation ne change pas le résultat de la compilation mais ajoute des informations supplémentaires au code compilé. Si une autre classe essaie d'utiliser cette méthode un avertissement est déclenché lors de la compilation du code de cette classe. Si nous ajoutons cette annotation à la méthode affichage de la classe Personne le code qui utilise la classe Personne ne doit plus appeler cette méthode sous peine d'avoir un avertissement au moment de la compilation.

```

public class Personne
{
    private String nom;
    private String prenom;
    private LocalDate date_nais;
    private int numero;
    private static int nbInstances;
}

```

```

...

@Deprecated
public void affichage()
{
    System.out.println("nom : " + nom);
    System.out.println("prénom : " + prenom);
    System.out.println("âge : " + calculAge());
}

...
}

```

La compilation d'une classe contenant un appel à la méthode affiche de la classe Personne déclenche un avertissement sur la ligne contenant l'appel à cette méthode.

```

javac -Xlint:deprecation Principale.java
Principale.java:16: warning: [deprecation] affichage() in
Personne has been deprecated
        p.affichage();
           ^
1 warning

```

Pour avoir un message détaillé sur les avertissements il faut utiliser l'option `-Xlint:deprecation` lors de l'appel du compilateur.

@Override est utilisé pour indiquer qu'une méthode se substitue à une méthode héritée. Cette annotation n'est pas obligatoire mais elle oblige le compilateur à vérifier que la substitution est correctement réalisée (signature identique des méthodes dans la classe de base et dans la classe actuelle). Si ce n'est pas le cas, une erreur de compilation est déclenchée.

L'exemple suivant substitue la méthode calculAge dans une classe héritant de la classe Personne.

```

public class Client extends Personne
{
    @Override
    public long calculAge()
    {
        ...
    }
}

```

Cette classe est compilée sans problème, car la méthode calculAge possède bien la même signature que dans la classe Personne. Par contre, si nous essayons de compiler le code suivant :

```

public class Client extends Personne
{
    @Override
    public long calculAge(int unite)
    {
        ...
    }
}

```

Nous obtenons la réponse suivante du compilateur.

```

Client.java:6: method does not override or implement a method

```

```

from a supertype
@Override
^
1 error

```

Il a effectivement raison (il faut bien avouer que c'est pratiquement toujours le cas !) nous lui avons annoncé notre intention de substituer la méthode calculAge et, en fait, nous avons effectué une surcharge, car il n'existe pas dans la classe Personne de méthode avec cette signature. Si nous enlevons le mot clé @Override, le code se compile mais il s'agit dans ce cas d'une surcharge.

@SuppressWarnings("...,...") indique au compilateur de ne pas générer certaines catégories d'avertissements. Si par exemple vous voulez discrètement utiliser dans une fonction une méthode marquée comme **@Deprecated** vous devez utiliser l'annotation suivante devant la fonction où se trouve l'appel de cette méthode.

```

@SuppressWarnings("deprecation")
public static void main(String[] args)
{
    Personne p;
    p=new Personne();
    p.affichage();
}

```

Ce code se compile correctement sans aucun avertissement, jusqu'au jour où la méthode concernée aura disparu complètement de la classe correspondante. Car il faut avoir à l'esprit que le but de l'annotation **@Deprecated** est de déconseiller l'utilisation d'une méthode, avec certainement l'intention de la faire disparaître dans une version future de la classe.

2.2. Utilisation d'une classe

L'utilisation d'une classe dans une application passe par trois étapes :

- **La déclaration d'une variable permettant l'accès à l'objet ;**
- **La création de l'objet ;**
- **L'initialisation d'une instance.**

Création d'une instance

Les variables objet sont des variables de type référence. Elles diffèrent des variables classiques par le fait que la variable ne contient pas directement les données mais une référence sur l'emplacement dans la mémoire où se trouvent les informations. Comme pour toutes les variables elles doivent être déclarées avant leur utilisation. La déclaration se fait de manière identique à celle d'une variable classique (**int** ou autre).

```
Personne p;
```

Après cette étape la variable existe mais ne référence pas d'emplacement valide. Elle contient la valeur **null**.

La deuxième étape consiste réellement à créer l'instance de la classe. Le mot clé **new** est utilisé à cet effet. Il attend comme paramètre le nom de la classe dont il est chargé de créer une instance.

L'opérateur **new** fait une demande pour obtenir la mémoire nécessaire au stockage de l'instance de la classe, puis initialise la variable avec cette adresse mémoire. Le constructeur de la classe est ensuite appelé pour initialiser la nouvelle instance créée.

```
p=new Personne();
```

Les deux opérations peuvent être combinées en une seule ligne.

```
Personne p=new Personne();
```

Le constructeur par défaut est appelé dans ce cas. Pour utiliser un autre constructeur vous devez spécifier une liste de paramètres et en fonction du nombre et du type des paramètres l'opérateur **new** appelle le constructeur correspondant.

```
Personne pe=new Personne("Dupont","albert",  
    LocalDate.of(1956,12,13));
```

Initialisation d'une instance

Plusieurs possibilités sont disponibles pour initialiser les champs d'une instance de classe. La première consiste à initialiser les variables constituant les champs de la classe au moment de leur déclaration.

```
public class Personne  
{  
    private String nom="nouveauNom";  
    private String prenom="nouveauPrenom";  
    private LocalDate date_nais=LocalDate.of(1963,11,29);  
    ...  
    ...  
}
```

Cette solution, bien que très simple, est relativement limitée puisqu'il n'est pas possible d'utiliser des structures de contrôle telles qu'une boucle à l'extérieur d'un bloc de code. La solution qui nous vient immédiatement à l'esprit est de placer le code d'initialisation à l'intérieur d'un constructeur. C'est effectivement une très bonne idée et c'est même l'objectif principal du constructeur que d'initialiser les variables d'instance. Par contre, un problème se pose avec les champs statiques, car pour eux il n'est bien sûr pas nécessaire d'avoir une instance de classe pour pouvoir les utiliser. Donc si nous plaçons le code chargé de les initialiser dans un constructeur, rien ne nous garantit que ce constructeur aura été appelé au moins une fois avant l'utilisation des champs statiques.

Pour pallier ce problème, Java propose les blocs d'initialisation statiques. Ce sont simplement des blocs de code précédés du mot clé **static** et délimités par les caractères { et }. Ils peuvent apparaître n'importe où dans le code de la classe et ils sont exécutés au chargement de la classe par la machine virtuelle dans l'ordre dans lequel ils apparaissent dans le code.

Nous pouvons par exemple utiliser le code suivant pour initialiser un champ statique avec une valeur aléatoire supérieure ou égale à 1000.

```
public class Personne  
{  
    private String nom="nouveauNom";  
    private String prenom="nouveauPrenom";  
    private LocalDate date_nais=LocalDate.of(1963,11,29);  
    private int numéro=0;  
    private static int numInstance;  
  
    static  
    {  
        while(numInstance<1000)  
        {  
            numInstance=(int) (10000*Math.random());  
        }  
    }  
}
```



```

    ...
    ...
}

```

Le même résultat peut être obtenu en créant une fonction privée statique et en l'appelant pour initialiser la variable.

```

public class Personne
{
    private String nom="nouveauNom";
    private String prenom="nouveauPrenom";
    private LocalDate date_nais=LocalDate.of(1963,11,29);
    private int numero=0;
    private static int numInstance=initCompteur();

    private static int initCompteur()
    {
        int cpt=0;
        while(cpt<1000)
        {
            cpt=(int) (10000*Math.random());
        }
        return cpt;
    }
    ...
    ...
}

```

Cette solution présente l'avantage de pouvoir utiliser la fonction dans une autre partie du code de la classe.

Le même principe peut être appliqué pour l'initialisation des champs d'instance. Le bloc de code chargé de l'initialisation ne doit pas, dans ce cas, être précédé du mot clé **static**. Ce bloc de code est implicitement recopié au début de chaque constructeur de la classe pendant la compilation.

L'initialisation d'un champ d'instance par un appel de fonction est également possible. Une petite restriction doit cependant être appliquée sur la fonction utilisée car elle ne doit pas pouvoir être substituée dans une sous-classe. Il faut pour cela la déclarer avec le mot clé **final**.

Destruction d'une instance

Java prend totalement à sa charge la gestion de la mémoire et nous évite cette tâche fastidieuse.

Il détermine de lui-même qu'un objet n'est plus utilisé dans l'application et le supprime alors de la mémoire. Ce mécanisme est appelé **Garbage Collector** (ramasse-miettes). Il considère qu'un objet peut être supprimé lorsque l'application n'a plus aucun moyen d'y accéder. Cette situation se produit, par exemple à la sortie d'une fonction, lorsqu'une variable locale est utilisée pour référencer l'objet. Elle peut également être provoquée par l'affectation de la valeur **null** à une variable. Pour qu'un objet soit réellement effacé de la mémoire, il faut que tous les moyens d'y accéder depuis l'application aient disparu. Il ne faut pas oublier que si un objet est stocké dans une collection ou un tableau, la collection ou tableau, conserve une référence vers l'objet.

Regardons un peu plus dans le détail le fonctionnement du **Garbage Collector**. Il existe plusieurs algorithmes pour mettre en œuvre le mécanisme de gestion de la mémoire. Ces mécanismes sont implémentés par les concepteurs de la machine virtuelle Java.

Mark and Sweep

Avec ce mécanisme le Garbage Collector travaille en deux étapes. Il commence par une exploration de la mémoire depuis la racine de l'application, la méthode main, et parcourt ainsi tous les objets accessibles depuis cette racine. Chaque objet accessible est marqué pendant cette exploration (Mark). Il effectue ensuite un deuxième passage au cours duquel il supprime tous les objets qui ne sont pas marqués, donc inaccessibles, et retire les marques des objets restant qu'il a placées lors du premier passage. Cette solution rudimentaire présente quelques inconvénients :

- Pendant la première étape l'exécution de l'application est interrompue.
- Sa durée est proportionnelle à la quantité de mémoire utilisée par l'application.
- Après plusieurs passages la mémoire risque d'être fragmentée.

Stop and Copy

Cette solution découpe l'espace mémoire disponible pour l'application en deux parties identiques. Dès que le Garbage Collector entre en action, il effectue, comme pour la solution précédente, une exploration de la mémoire depuis la racine de l'application. Dès qu'il trouve un objet accessible, il le recopie vers la deuxième zone mémoire et modifie bien sûr les variables pour qu'elles référencent ce nouvel emplacement. À la fin de l'exploration tous les objets accessibles ont été copiés dans la deuxième zone mémoire. Il est alors possible d'effacer complètement le contenu de la première zone. Le même mécanisme peut alors être répété ultérieurement avec la zone mémoire qui vient d'être libérée. Cette solution présente l'avantage d'éliminer la fragmentation de la mémoire puisque les objets sont copiés les uns à la suite des autres. Par contre, un inconvénient important réside dans le déplacement fréquent des objets ayant une longue durée de vie dans l'application. Une solution intermédiaire consiste à répartir les objets en mémoire en fonction de leur espérance de vie ou de leur âge. La moitié de la mémoire disponible est donc parfois découpée en trois zones :

- Une zone pour les objets ayant une très longue durée de vie et qui n'ont pratiquement aucun risque de disparaître pendant le fonctionnement de l'application.
- Une zone pour les objets créés récemment.
- Une zone pour les objets les plus anciens.

Lorsque le Garbage Collector intervient, il traite tout d'abord la zone réservée aux objets récents. Si après ce premier passage l'application dispose de suffisamment de mémoire, le Garbage Collector arrête son traitement ; à noter que pendant ce premier passage il peut transférer des objets dans la zone réservée aux objets anciens, s'ils existent depuis suffisamment longtemps. Si la mémoire disponible n'est pas suffisante, il refait le traitement sur la zone réservée aux objets les plus anciens.

L'efficacité de cette solution repose sur un constat cruel : les objets Java n'ont pas une grande espérance de vie. Il y a donc plus de chance d'en trouver à éliminer parmi ceux qui ont été créés le plus récemment.

Parmi ces deux solutions il est difficile de dire laquelle est utilisée par une machine virtuelle Java puisque l'implémentation du Garbage Collector est laissée libre au concepteur de la machine virtuelle.

Avant d'éliminer une instance de la mémoire, le Garbage Collector appelle le destructeur de cette instance.

Le dernier point à éclaircir concernant le Garbage Collector concerne son déclenchement. C'est en fait la machine virtuelle Java qui surveille les ressources mémoire disponibles et provoque l'entrée en action du Garbage Collector lorsque celles-ci ont atteint un seuil limite (environ 85 %). Il est

cependant possible de forcer l'activation du Garbage Collector en appelant la méthode gc() de la classe System. Cette utilisation doit être exceptionnelle, car une utilisation trop fréquente affecte les performances de l'application. Elle peut être utilisée juste avant que l'application utilise une quantité de mémoire importante, comme par exemple, la création d'un tableau volumineux.

Le code ci-dessous permet de mettre en évidence l'action du Garbage Collector.

```
import java.util.GregorianCalendar;
public class Personne
{
    private String nom="nouveauNom";
    private String prenom="nouveauPrenom";
    private LocalDate date_nais=LocalDate.of(1963,11,29);
    private int numero=0;
    private static int numInstance;
    public String getNom()
    {
        return nom;
    }

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

    public String getPrenom()
    {
        return prenom;
    }

    public void setPrenom(String p)
    {
        prenom = p.toLowerCase();
    }

    @Override
    protected void finalize() throws Throwable
    {
        System.out.print("\u2020");
        super.finalize();
    }
    public int getNumero()
    {
        return numero;
    }

    public static int getNbInstances()
    {
        return numInstance;
    }

    public Personne()
    {
        nom="";
        prenom="";
        date_nais=null;
        numInstance++;
        numero=numInstance;
    }
    public Personne(String n,String p,LocalDate d)
    {

```

```

        nom=n;
        prenom=p;
        date_nais=d;
        numInstance++;
        numero=numInstance;
    }
}

/*****
import java.time.LocalDate;

public class GestionMemoire
{
    public static void main(String[] args) throws
    InterruptedException
    {
        double total;
        double reste;
        double pourcentage;

        for (int j=0;j<1000;j++)
        {
            creationTableau();
            total=Runtime.getRuntime().totalMemory();
            reste=Runtime.getRuntime().freeMemory();
            pourcentage=100-(reste/total)*100;
            System.out.println("création du " + j + "ième
tableau mémoire pleine à : " + pourcentage + "%" );
            // une petite pause pour pouvoir lire les messages
            Thread.sleep(1000);
        }
    }

    public static void creationTableau()
    {
        // création d'un tableau de 1000 Personnes dans une variable locale
        // à la fin de cette fonction les éléments du tableau ne sont
        // plus accessibles et peuvent être supprimés de la mémoire
        Personne[] tablo;
        tablo=new Personne[1000];
        for (int i=0;i<1000;i++)
        {
            tablo[i]=new
Personne("Dupont","albert",LocalDate.of(1956,12,13));
        }
    }
}

```

2.3. Héritage

L'héritage est une puissante fonctionnalité d'un langage orienté objet mais peut parfois être utilisée mal à propos.

Deux types de relations peuvent être envisagés entre deux classes. Nous pouvons avoir la relation « est une sorte de » et la relation « concerne un ». La relation d'héritage doit être utilisée lorsque la relation « est une sorte de » peut être appliquée entre deux classes. Prenons un exemple avec trois classes : Personne, Client, Commande.

Essayons la relation « est une sorte de » pour chacune des classes.

- Une commande est une sorte de client.
- Une commande est une sorte de personne.
- Un client est une sorte de commande.
- Un client est une sorte de personne.
- Une personne est une sorte de client.
- Une personne est une sorte de commande.

Parmi toutes ces tentatives, il n'y en a qu'une seule qui nous semble logique : un client est une sorte de personne. Nous pouvons donc envisager une relation d'héritage entre ces deux classes.

La mise en œuvre est très simple au niveau du code puisque dans la déclaration de la classe il suffit juste de spécifier le mot clé **extends** suivi du nom de la classe dont on souhaite hériter. Java n'acceptant pas l'héritage multiple, vous ne pouvez spécifier qu'un seul nom de classe de base. À l'intérieur de cette nouvelle classe vous pouvez :

- Utiliser les champs hérités de la classe de base (à condition bien sûr que leur visibilité le permette).
- Ajouter des nouveaux champs.
- Masquer un champ hérité en le déclarant avec le même nom que celui utilisé dans la classe de base. Cette technique doit être utilisée avec modération.
- Utiliser une méthode héritée sous réserve que sa visibilité le permette.
- Substituer une méthode héritée en la déclarant à l'identique (même signature).
- Surcharger une méthode héritée en la créant avec une signature différente.
- Ajouter une nouvelle méthode.
- Ajouter un ou plusieurs constructeurs.

Voici par exemple la création de la classe Client qui hérite de la classe Personne et à laquelle est ajouté le champ type et les accesseurs correspondants.

```
public class Client extends Personne
{
    // détermination du type de client
    // P -> particulier
    // E -> entreprise
    // A -> administration
    private char type;
    public char getType()
    {
        return type;
    }
    public void setType(char t)
    {
        type = t;
    }
}
```

La classe peut ensuite être utilisée et propose toutes les fonctionnalités définies dans la classe Client plus celles héritées de la classe Personne.

```
Client c;
c=new Client();
c.setNom("COURS");
c.setPrenom("Java");
c.setDate_nais(LocalDate.of(1991,05,15));
c.setType('E');
```

```
c.affichage();
```

this et super

Il est légitime de vouloir ensuite modifier le fonctionnement de certaines méthodes héritées pour les adapter à la classe Client. Par exemple la méthode affichage peut être substituée pour tenir compte du nouveau champ disponible dans la classe.

```
public void affichage()
{
    System.out.println("nom : " + getNom());
    System.out.println("prenom : " + getPrenom());
    System.out.println("age : " + calculAge());
    switch (type)
    {
        case 'P':
            System.out.println("type de client : Particulier");
            break;
        case 'E':
            System.out.println("type de client : Entreprise");
            break;
        case 'A':
            System.out.println("type de client : Administration");
            break;
        default:
            System.out.println("type de client : Inconnu");
            break;
    }
}
```

Ce code fonctionne très bien mais ne respecte pas un des principes de la programmation objet qui veut que l'on réutilise au maximum ce qui existe déjà. Dans notre cas nous avons déjà une portion de code chargée de l'affichage du nom, du prénom, et de l'âge d'une personne. Pourquoi ne pas la réutiliser dans la méthode affichage de la classe Client puisque l'on en hérite ?

Notre méthode devient donc :

```
public void affichage()
{
    affichage();
    switch (type)
    {
        case 'P':
            System.out.println("type de client : Particulier");
            break;
        case 'E':
            System.out.println("type de client : Entreprise");
            break;
        case 'A':
            System.out.println("type de client : Administration");
            break;
        default:
            System.out.println("type de client : Inconnu");
            break;
    }
}
```

Essayons de l'utiliser :

```
Client c;
```

```
c=new Client();
c.setNom("JAVA");
c.setPrenom("");
c.setDate_nais(LocalDate.of(1991,05,15));
c.setType('E');
c.affichage();
```

Hélas le résultat n'est pas à la hauteur de nos espérances !

```
.Exception in thread 'main' java.lang.StackOverflowError
```

Que s'est-il passé lors de l'exécution ?

Lors de l'appel de la méthode `affichage` la première ligne de code a consisté à appeler la méthode `affichage` de la classe de base. En fait la machine virtuelle Java recherche la première méthode `affichage` qu'elle trouve. Pour cela, elle commence la recherche dans la classe à partir de laquelle est créé l'objet, dans notre cas la classe `Client`. Elle appelle donc ainsi en boucle la méthode `affichage` de la classe `Client` d'où l'erreur de débordement de pile que nous obtenons.

Pour éviter ce genre de problème, nous devons lui préciser que la méthode `affichage` à appeler se trouve dans la classe de base. Pour cela, nous devons utiliser le mot clé `super` pour qualifier la méthode `affichage` appelée.

```
public void affichage()
{
    super.affichage();
    switch (type)
    {
        case 'P':
            System.out.println("type de client : Particulier");
            break;
        case 'E':
            System.out.println("type de client : Entreprise");
            break;
        case 'A':
            System.out.println("type de client : Administration");
            break;
        default:
            System.out.println("type de client : Inconnu");
            break;
    }
}
```

Le même mot clé peut être utilisé pour appeler le constructeur de la classe de base.

L'appel au constructeur de la classe de base doit être la première ligne d'un constructeur.

Nous pouvons donc créer un constructeur pour la classe `Client` qui réutilise le constructeur de la classe `Personne`.

Après la création d'un constructeur surchargé, le constructeur par défaut généré automatiquement par le compilateur n'est plus disponible. C'est donc une bonne habitude de toujours créer un constructeur par défaut. Celui-ci devra simplement faire un appel au constructeur par défaut de la classe de base. Nous devons donc ajouter le code suivant à la classe `Client`.

```
public Client()
{
    // appel au constructeur par défaut de la super classe
    super();
}

public Client(String nom, String prenom, LocalDate date_nais, char
type)
{
    // appel au constructeur surchargé de la super classe
    super(nom, prenom, date_nais);
    // initialisation du type de client
    type=type;
}
```

Les informations ont bien été prises en compte sauf le type de client qui n'a pas été initialisé. Regardons de plus près le code du constructeur. Nous découvrons qu'un paramètre du constructeur porte le même nom qu'un champ de la classe.

Lorsque nous écrivons la ligne `type=type`, le compilateur considère que nous souhaitons affecter au paramètre `type` la valeur contenue dans le paramètre `type`. Rien d'illégal, mais ce n'est absolument pas ce que nous souhaitons faire. Nous devons indiquer que l'affectation doit se faire au champ de la classe. Pour cela nous devons préfixer son nom avec le mot clé `this`.

Le constructeur devient donc :

```
public Client(String nom, String prenom, LocalDate date_nais,
char type)
{
    // appel au constructeur surchargé de la super classe
    super(nom, prenom, date_nais);
    this.type=type;
}
```

Classes abstraites

Les classes abstraites sont des classes qui peuvent uniquement être utilisées comme classe de base dans une relation d'héritage. Il est impossible de créer une instance d'une classe abstraite. Elles

servent essentiellement de modèle pour la création de classe devant toutes avoir un minimum de caractéristiques identiques. Elles peuvent contenir des champs, des propriétés et des méthodes comme une classe ordinaire. Pour qu'une classe soit une classe abstraite vous devez utiliser le mot clé **abstract** lors de sa déclaration.

Cette technique facilite l'évolution de l'application car si une nouvelle fonctionnalité doit être disponible dans les classes dérivées, il suffit d'ajouter cette fonctionnalité dans la classe de base. Il est également possible de ne pas fournir d'implémentation pour certaines méthodes d'une classe abstraite et ainsi laisser à l'utilisateur de la classe le soin de créer l'implémentation dans la classe dérivée. Ces méthodes doivent également être déclarées avec le mot clé `abstract`. Il n'y a dans ce cas pas de bloc de code correspondant à cette méthode et sa déclaration doit se terminer avec un point-virgule.

Une classe abstraite n'a pas obligatoirement de méthodes abstraites, par contre une classe contenant une méthode abstraite doit obligatoirement être elle aussi abstraite. La classe qui héritera de la classe abstraite devra implémenter les méthodes déclarées abstraites dans sa classe de base ou être elle aussi abstraite. Si elle implémente une méthode abstraite, elle ne peut pas réduire la visibilité déclarée dans la classe abstraite.

Voici ci-dessous un exemple de classe abstraite :

```
public abstract class EtreVivant
{
    private double taille;
    private double poids;
    public double getTaille()
    {
        return taille;
    }
    public void setTaille(double taille)
    {
        this.taille = taille;
    }
    public double getPoids()
    {
        return poids;
    }
    public void setPoids(double poids)
    {
        this.poids = poids;
    }
    // cette méthode devra être implémentée dans les classes dérivées
```

```
protected abstract void seDeplacer();
}
```

Classes finales

Les classes sont des classes ordinaires qui peuvent être instanciées mais ne sont pas utilisables comme classe de base dans une relation d'héritage. Elles doivent être déclarées avec le mot clé **final**. C'est le cas de plusieurs classes du langage Java, comme par exemple la classe **String** dont voici la déclaration extraite de la documentation Java.

```
public final class String
extends Object
implements Serializable, Comparable, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

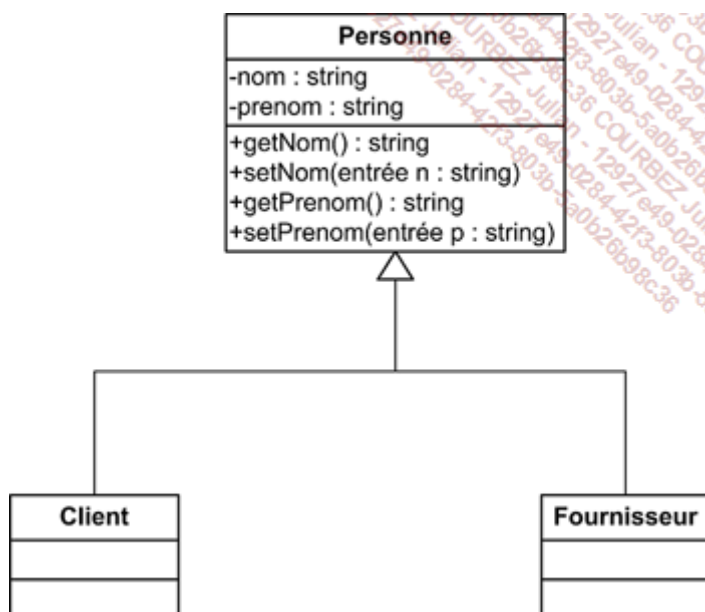
Une méthode peut également être déclarée avec le mot clé `final` pour interdire sa redéfinition dans une sous-classe.

La signification des mots clés `abstract` et `final` étant contradictoire, l'utilisation simultanée de ces deux mots clés est bien sûr interdite. Le compilateur détecte cette situation et génère une erreur.

Conversion de type

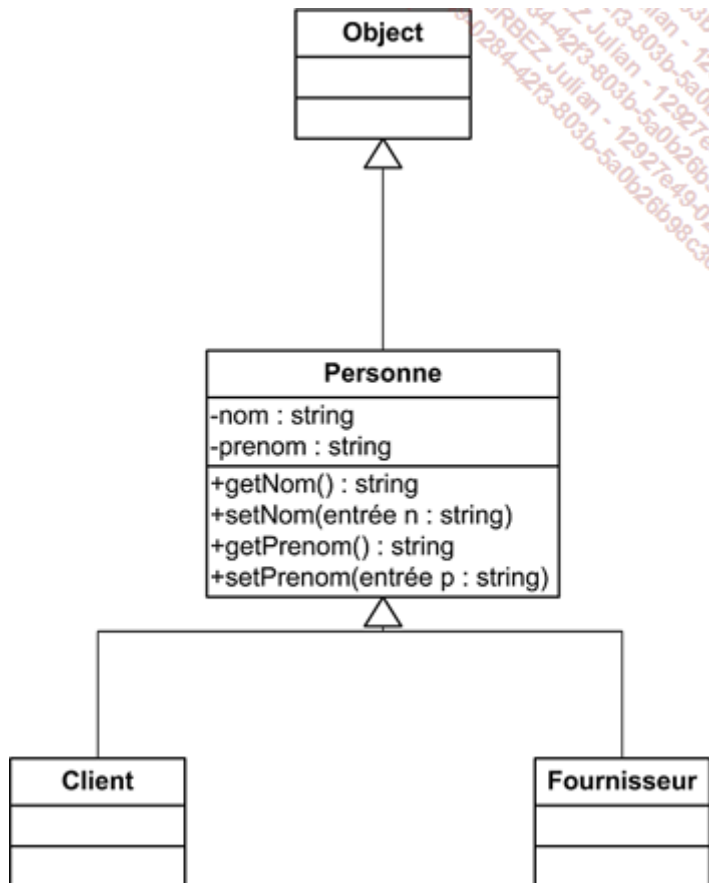
Pour bien comprendre les opérations de conversion de type il faut toujours garder présent à l'esprit que une fois créé en mémoire un objet ne changera jamais de type jusqu'à la fin de sa vie. Ainsi si vous créez une instance de la classe `Personne`, cette instance restera toujours une instance de la classe `Personne`. Avec ce préambule vous devez vous poser des questions sur la signification réelle du terme "conversion". En fait la conversion n'intervient pas sur l'objet lui-même mais sur la façon de le manipuler. Elle va agir sur le type de la variable utilisée pour accéder à l'objet.

La relation d'héritage entre classes est l'élément fondamental permettant l'utilisation des conversions de type. Nous avons vu qu'une classe pouvait récupérer les caractéristiques d'une autre classe par cet intermédiaire. C'est-à-dire qu'elle récupère automatiquement les caractéristiques de sa classe de base. Nous pouvons par exemple avoir la hiérarchie de classe suivante.



La classe `Client` est une évolution de la classe `Personne` au même titre que la classe `Fournisseur` est une évolution de la classe `Personne`. Toutes les caractéristiques disponibles avec une instance de la classe `Personne` seront également disponibles avec une instance de la classe `Client` ou une instance de la classe `Fournisseur`.

En fait notre schéma n'est pas complet, il devrait plutôt avoir la forme suivante :



Il ne faut jamais oublier qu'une classe qui n'a pas de classe de base explicite hérite implicitement de la classe `Object`. Nous pouvons donc dire que n'importe quelle instance de classe aura au minimum les caractéristiques d'une instance de la classe `Object`.

En partant du code suivant :

```

Object o;

Personne p;

Client c;

Fournisseur f;
  
```

Avec une variable du type `X` nous pouvons bien sûr référencer une instance de classe de type `X` mais aussi une instance de n'importe quelle sous-classe de la classe `X`. Ainsi une variable de type **`Object`** peut être utilisée pour référencer une instance de n'importe quelle classe.

```
f=new Fournisseur();
```

<code>o=f;</code>	<input checked="" type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>p=f;</code>	<input checked="" type="checkbox"/> fonctionne	<input type="checkbox"/> ne fonctionne pas
<code>c=f;</code>	<input type="checkbox"/> fonctionne	<input checked="" type="checkbox"/> ne fonctionne pas

c=new Client();		
o=c;	■ fonctionne	□ ne fonctionne pas
p=c;	■ fonctionne	□ ne fonctionne pas
f=c;	□ fonctionne	■ ne fonctionne pas
p=new Personne();		
o=p;	■ fonctionne	□ ne fonctionne pas
c=p;	□ fonctionne	■ ne fonctionne pas
f=p;	□ fonctionne	■ ne fonctionne pas
o=new Object();		
p=o;	□ fonctionne	■ ne fonctionne pas
f=o;	□ fonctionne	■ ne fonctionne pas
c=o;	□ fonctionne	■ ne fonctionne pas

Lors de l'appel de la fonction **essai** le passage du paramètre est équivalent à une affectation à une variable de type **Object**. Nous pouvons donc appeler cette fonction en lui fournissant une instance de n'importe quelle classe.

```
...

o=new Object();
p=new Personne();
c=new Client();
f=new Fournisseur();

essai(o);      ■ fonctionne      □ ne fonctionne pas
essai(p);      ■ fonctionne      □ ne fonctionne pas
essai(f);      ■ fonctionne      □ ne fonctionne pas
essai(c);      ■ fonctionne      □ ne fonctionne pas
...

public void essai(Object obj)
{
    Object o;
    Personne p;
    Client c;
    Fournisseur f;

    o=obj;      ■ fonctionne      □ ne fonctionne pas
    p=obj;      □ fonctionne      ■ ne fonctionne pas
    c=obj;      □ fonctionne      ■ ne fonctionne pas
}
```

f=obj;	□ fonctionne	■ ne fonctionne pas
}		

À l'intérieur de la fonction `essai` le problème est inversé. Le paramètre **obj** référence une instance de classe mais le compilateur ne peut pas savoir de quelle classe il s'agit. C'est pour cette raison qu'il n'accepte que l'affectation **o=obj**; qui est sans risque puisque les deux variables sont de même type. Il est toutefois possible de contourner ce problème en effectuant une opération de transtypage. Cette opération est réalisée simplement en faisant précéder la variable du nom de la classe vers laquelle nous voulons réaliser le transtypage. Le nom de la classe doit être placé entre les caractères (et).

```
public static void essai(Object obj)
{
    Object o;
    Personne p;
    Client c;
    Fournisseur f;

    o=obj;
    p=(Personne)obj;
    c=(Client)obj;
    f=(Fournisseur)obj;
}
```

Nous n'avons plus d'erreurs au moment de la compilation. Par contre au moment de l'exécution nous obtenons une exception de type **java.lang.ClassCastException**. Cette exception est déclenchée par la machine virtuelle lorsqu'elle découvre que nous lui avons menti en essayant de lui faire croire que la variable **obj** référence une instance d'une certaine classe alors qu'il n'en est rien. Nous devons donc être plus prudents avec nos opérations de transtypage et vérifier ce que référence réellement la variable **obj**. L'opérateur **instanceof** permet de faire cette vérification. La fonction `essai` doit donc s'écrire de la façon suivante :

```
public static void essai(Object obj)
{
    Object o;
    Personne p;
    Client c;
    Fournisseur f;

    o=obj;
    if (obj instanceof Personne)
    {
```

```

        p=(Personne)obj;
    }
    if (obj instanceof Client)
    {
        c=(Client)obj;
    }
    if (obj instanceof Fournisseur)
    {
        f=(Fournisseur)obj;
    }
}

```

Cet opérateur doit pratiquement toujours être utilisé pour vérifier la faisabilité d'une opération de transtypage.

La classe Object

La classe `Object` est directement ou indirectement présente dans la hiérarchie de toute classe d'une application. Les méthodes définies dans la classe `Object` sont donc disponibles pour n'importe quelle classe. L'inconvénient de ces méthodes est qu'elles ne font pas grand-chose d'utile ou leur fonctionnement n'est pas adapté aux différentes classes de l'application. Elles ont donc souvent besoin d'être substituées dans les classes de l'application pour pouvoir être utilisées efficacement. Il est donc important de bien connaître leur utilité pour adapter leur fonctionnement.

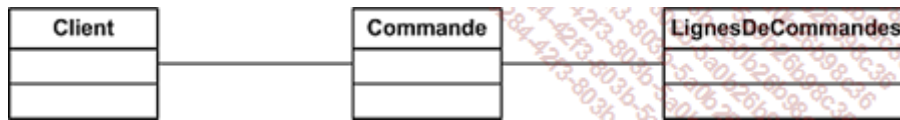
hashCode

Cette méthode permet d'obtenir l'adresse mémoire où est stockée une instance de classe. En elle-même elle n'est pas très utile, par contre elle est utilisée en interne par d'autres méthodes et principalement par la méthode **equals**.

clone

La méthode **clone** peut revendiquer le titre de "photocopieur" d'objets. Par son intermédiaire nous pouvons obtenir une copie conforme d'un objet présent en mémoire. Pour que cette méthode soit disponible, il est indispensable que la classe à partir de laquelle l'objet à copier a été créé, implémente l'interface **Cloneable**. Cette interface exige la création d'une méthode **clone** dans la classe. Cette méthode est souvent très simple puisqu'il suffit qu'elle appelle la méthode **clone** de la classe **Object**. Cette version par défaut se contente d'effectuer une copie de la zone mémoire correspondant à l'instance de classe à dupliquer. Si l'instance à dupliquer contient des références vers d'autres objets, ces objets ne seront pas dupliqués et seront partagés par l'original et la copie. Si ce fonctionnement n'est pas adapté à l'application, il faut concevoir la méthode **clone** pour qu'elle effectue également une copie des objets référencés.

Pour illustrer ce mécanisme, nous allons travailler avec la classe `Client` à laquelle nous allons associer une classe `Commande` elle-même associée à une classe `LignesDeCommandes`.



Si nous effectuons une copie d'une commande nous conservons la référence vers le client par contre les lignes de commande doivent être dupliqués. Nous devons donc concevoir la méthode `clone` de la classe `Commande` pour qu'elle duplique également l'instance de la classe `LignesDeCommandes` référencée. Voici ci-dessous un extrait du code de ces classes.

```

public class Commande implements Cloneable
{
    Client leClient;
    LignesDeCommande lesLignes;

    public Commande()
    {
        super();
        lesLignes=new LignesDeCommande();
    }

    public Object clone() throws CloneNotSupportedException
    {
        Commande cmd;
        // création d'une copie de la commande
        cmd=(Commande) super.clone();
        // duplication des lignes de la commande
        cmd.lesLignes=(LignesDeCommande) lesLignes.clone();
        return cmd;
    }

    public Client getLeClient()
    {
        return leClient;
    }

    public void setLeClient(Client leClient)
    {
        this.leClient = leClient;
    }
}

```

```

    public LignesDeCommande getLesLignes()
    {
        return lesLignes;
    }

    public void setLesLignes(LignesDeCommande lesLignes)
    {
        this.lesLignes = lesLignes;
    }
}

/*****/
public class LignesDeCommande implements Cloneable
{
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

```

Nous pouvons maintenant créer des instances de nos classes et vérifier le bon fonctionnement.

```

Client c;
c=new Client("JAVA","",new GregorianCalendar(1991,04,15),'E');
Commande cmd1,cmd2;
// création et initialisation d'une commande
cmd1=new Commande();
cmd1.setLeClient(c);
System.out.println("hashCode de la commande : " +cmd1.hashCode());
System.out.println("hashCode du Client : "
+cmd1.getLeClient().hashCode());
System.out.println("hashCode des lignes : "
+cmd1.getLesLignes().hashCode());
cmd2=(Commande)cmd1.clone();
System.out.println("hashCode de la copie : " +cmd2.hashCode());

```



```
System.out.println("hashCode du Client de la copie: "
+cmd2.getLeClient().hashCode());
System.out.println("hashCode des lignes de la copie:"
+cmd2.getLesLignes().hashCode());
```

Nous obtenons le résultat suivant à l'exécution de ce code :

```
hashCode de la commande : 6413875
hashCode du Client : 21174459
hashCode des lignes : 827574
hashCode de la copie : 17510567
hashCode du Client de la copie: 21174459
hashCode des lignes de la copie:27744459
```

Nous avons bien deux commandes distinctes qui référencent le même client.

Equals

La méthode `equals` est utilisée pour comparer deux instances de classe. Le code suivant permet de vérifier l'égalité de deux clients.

```
Client c1,c2;
c1=new Client("JAVA","",LocalDate.of(1991,04,15),'E');
c2=new Client("JAVA","",LocalDate.of(1991,04,15),'E');
if (c1.equals(c2))
{
    System.out.println("les deux clients sont identiques");
}
else
{
    System.out.println("les deux clients sont différents");
}
```

Malgré les apparences, l'exécution de ce code nous affiche le résultat suivant :

```
les deux clients sont différents
```

L'implémentation par défaut de cette méthode effectue une comparaison des références pour déterminer s'il y a égalité des deux objets. Ce sont en fait les **hashCode** des deux objets qui sont comparés. Si nous voulons avoir un critère de comparaison différent, nous devons substituer la méthode **equals** dans la classe `Client` en utilisant nos propres critères de comparaison.

```
public boolean equals(Object obj)
{
    Client c;
```

```

// vérification si obj est null ou référence une instance
// d'une autre classe
if (obj ==null || obj.getClass()!=this.getClass())
{
    return false;
}
else

{
    c=(Client)obj;
    // vérification des critères d'égalité sur
    // - le nom
    // - le prénom
    // - la date de naissance
    // - le type de client
    if (c.getNom().equals(getNom()) &
        c.getPrenom().equals(getPrenom()) &
        c.getDate_nais().equals(getDate_nais()) &
        c.getType()== getType() )
    {
        return true;
    }
    else
    {
        return false;
    }
}
}

```

Pour conserver une cohérence entre la méthode **equals** et la méthode **hashCode**, il est important de redéfinir cette dernière pour qu'elle calcule le **hashcode** à partir de la valeur des champs utilisés dans les critères de comparaison. La valeur renvoyée par cette fonction a peu d'importance, seule compte réellement la garantie d'obtenir toujours le même résultat pour toute instance de classe ayant les mêmes valeurs de champs.

Voici un exemple possible d'implémentation de la méthode **hashCode** :

```
public int hashCode()
```

```

{

    return this.getNom().hashCode()+

           this.getPrenom().hashCode() +

           this.getDate_nais().hashCode()+

           (int)this.getType();

}

```

getClass

Cette méthode fournit une instance de la classe `Class` contenant les caractéristiques de la classe à partir de laquelle l'objet a été créé. Nous pouvons par exemple obtenir le nom de la classe, les méthodes disponibles, les champs, etc. Par mesure de sécurité, cette méthode ne peut pas être substituée.

Voici une fonction affichant quelques informations sur la classe de l'objet qui lui est passé en paramètre.

```

public static void infoClasse(Object o)
{
    Class c;
    c=o.getClass();
    System.out.println("nom de la classe : " + c.getName());
    System.out.println("elle est dans le package : " +
c.getPackage().getName());
    System.out.println("elle hérite de la classe : " +
c.getSuperclass().getName());
    System.out.println("elle possède les champs : ");
    for (int i=0;i<c.getFields().length;i++)
    {
        System.out.print("\t" + c.getFields()[i].getName());
        System.out.println(" de type : " +
c.getFields()[i].getType().getName());
    }
    System.out.println("elle possède les méthodes : ");
    for (int i=0;i<c.getMethods().length;i++)
    {
        System.out.print("\t" + c.getMethods()[i].getName());
        System.out.print(" qui attend comme paramètre (");

```

```

        for (int
j=0;j<c.getMethods()[i].getParameterTypes().length;j++)
        {

            System.out.print(c.getMethods()[i].getParameterTypes()[j]+ " ");

        }

        System.out.println("");

    }

}

```

toString

À l'inverse de la précédente, cette méthode devrait pratiquement toujours être surchargée. Elle permet d'obtenir la représentation d'un objet sous forme d'une chaîne de caractères. Par défaut l'implémentation de cette méthode dans la classe `Object` retourne le nom de la classe suivi du **hashCode** de l'instance. Une représentation plus parlante est bien sûr conseillée. Elle doit être construite à partir des valeurs contenues dans les champs de l'objet. Une version possible de la méthode **toString** pour la classe `Personne` pourrait par exemple être la suivante.

```

public String toString()
{
    String chaine;

    chaine="nom : " + getNom()+ "\r\n";

    chaine=chaine + "prénom : " + getPrenom();

    return chaine;
}

```

L'appel de la méthode `toString` est parfois implicite lorsque l'on passe un objet comme paramètre à une fonction. Les deux syntaxes suivantes sont donc équivalentes.

```

Client c;

c=new Client("JAVA","",LocalDate.of(1991,04,15),'E');

System.out.println(c);

```

Ou

```

Client c;

c=new Client("JAVA","",LocalDate.of(1981,05,15),'E');

System.out.println(c.toString());

```

2.4. Interface

Si plusieurs classes doivent implémenter la même méthode, il est plus pratique d'utiliser les interfaces. Comme les classes, les interfaces permettent de définir un ensemble de constantes et méthodes. Généralement, une interface ne contient que des signatures de méthodes, elles sont dans ce cas similaires aux méthodes abstraites définies dans une classe abstraite.

L'interface constitue un contrat que la classe signe. En déclarant que la classe implémente une interface, elle s'engage à fournir tout ce qui est défini dans l'interface. Il faut être prudent si vous utilisez les interfaces et ne jamais modifier une interface qui est déjà utilisée sinon vous courez le risque de devoir reprendre le code de toutes les classes qui implémentent cette interface.

À partir de la version 8 du langage Java, cette contrainte peut être contournée en créant des méthodes par défaut dans l'interface.

Création d'une interface

Pour pouvoir utiliser une interface, il faut la définir au préalable. La déclaration est semblable à la déclaration d'une classe mais on utilise le mot clé **interface** en lieu et place du mot clé **class**.

Vous pouvez éventuellement utiliser le mot clé **extends** pour introduire une relation d'héritage dans l'interface. Contrairement aux classes les interfaces autorisent l'héritage multiple. Dans ce cas, les noms des interfaces héritées sont séparés par des virgules après le mot clé **extends**. Il faut être raisonnable avec cette possibilité car les classes qui implémenteront cette interface devront fournir toutes les méthodes définies dans la hiérarchie de l'interface.

Créons donc notre première interface. Celle-ci va imposer la présence d'une fonction **compare** attendant comme paramètre un objet. Lors de la définition d'une interface, il est recommandé de fournir, sous forme de commentaires, une description du travail que devra accomplir chaque méthode et les résultats qu'elle devra fournir.

```
// cette interface devra être implémentée par les classes
// pour lesquelles un classement des instances est envisagé
public interface Classable
{
    // cette méthode pourra être appelée pour comparer l'instance
    // courante avec celle reçue en paramètre
    // la méthode retourne un entier dont la valeur dépend
    // des règles suivantes
    // 1 si l'instance courante est supérieure à celle reçue
    // en paramètre
    // 0 si les deux instances sont égales
    // -1 si l'instance courante est inférieure à celle reçue
    // en paramètre
    // -99 si la comparaison est impossible

    int compare(Object o);

    public static final int INFÉRIEUR=-1;
    public static final int EGAL=0;
    public static final int SUPÉRIEUR=1;
```

```

        public static final int ERREUR=-99;

    }

```

Utilisation d'une interface

Mais quels critères allons-nous utiliser pour dire qu'un objet est supérieur à un autre ?

Dans la description de notre interface ce n'est pas notre souci ! Nous laissons le soin à l'utilisateur qui va définir une classe implémentant l'interface de définir quels sont les critères de comparaison. Par exemple dans notre classe `Personne`, nous pourrions implémenter l'interface `Classable` de la manière suivante en choisissant de comparer deux instances de la classe `Personne` sur le nom :

```

public class Personne
implements Classable
{

    public int compare(Object o)
    {
        Personne p;
        if (o instanceof Personne)
        {
            p=(Personne)o;
        }
        else
        {
            return Classable.ERREUR;
        }
        if (getNom().compareTo(p.getNom())<0)
        {
            return Classable.INFERIEUR;
        }
        if (getNom().compareTo(p.getNom())>0)
        {
            return Classable.SUPERIEUR;
        }

        return Classable.EGAL;
    }
}

```

```
...  
...  
}
```

Deux modifications sont visibles dans la classe :

- Le fait qu'elle implémente l'interface **Classable**.
- L'implémentation réelle de la fonction **compare**.

Dans cette fonction la comparaison se fera sur le nom des clients. Très bien, mais à quoi sert-il ?

Il arrive fréquemment que l'on ait besoin de trier des éléments dans une application. Deux solutions :

- Créer une fonction de tri spécifique pour chaque type d'élément que l'on veut trier.
- Créer une routine de tri générique et faire en sorte que les éléments que l'on utilise soient triables par cette routine.

Les interfaces vont nous aider à mettre en œuvre cette deuxième solution. Pour pouvoir trier des éléments, et quelle que soit la méthode utilisée pour le tri, nous aurons besoin de comparer deux éléments. Pour être certain que notre routine de tri fonctionnera sans problème, il faut s'assurer que les éléments qu'elle devra trier auront la possibilité d'être comparés les uns aux autres.

Nous ne pouvons garantir cela que si tous nos éléments implémentent l'interface **Classable**. Nous allons donc l'exiger dans la déclaration de notre routine de tri.

```
public static void tri(Classable[] tablo)  
{  
}
```

Définie ainsi, notre fonction sera capable de trier toutes sortes de tableaux pourvu que leurs éléments implémentent l'interface **Classable**. Elle retournera le tableau trié. Nous pouvons donc écrire le code suivant et utiliser la méthode **compare** sans risque.

```
public static Classable[] tri(Classable[] tablo)  
{  
    int i,j;  
    Classable c;  
    for (i=0;i< tablo.length;i++)  
    {  
        for( j = i + 1; j<tablo.length;j++)  
        {  
            if (tablo[j].compare(tablo[i])==Classable.INFERIEUR)  
            {  
                c = tablo[j];  
                tablo[j] = tablo[i];  
            }  
        }  
    }  
}
```

```

        tablo[i] = c;
    }
    else if (tablo[j].compare(tablo[i])==Classable.ERREUR)
    {
        return null;
    }
}
}
return tablo;
}

```

Puis pour tester notre procédure, créons quelques clients et essayons de les trier, puis d'afficher leurs noms.

```

Personne[] tab;
tab=new Personne[5];
tab[0] = new Personne("toto2", "prenom2",new
GregorianCalendar(1922,2,15));
tab[1] = new Personne("toto1", "prenom1 ",new
GregorianCalendar(1911,1,15));
tab[2] = new Personne("toto5", "prenom5 ",new
GregorianCalendar(1955,05,15));
tab[3] = new Personne("toto3", "prenom3 ",new
GregorianCalendar(1933,03,15));
tab[4] = new Personne("toto4", "prenom4 ",new
GregorianCalendar(1944,04,15));
Personne[] tabTrie;
tabTrie=(Personne[])tri(tab);
for (int i=0;i<tabTrie.length;i++)
{
    System.out.println(tabTrie[i]);
}

```

Nous obtenons le résultat suivant :

```

Mr toto1 prenom1   né le 01/01/0001 00:00:00 code Client : 1
Mr toto2 prenom2   né le 01/01/0001 00:00:00 code Client : 2
Mr toto3 prenom3   né le 01/01/0001 00:00:00 code Client : 3
Mr toto4 prenom4   né le 01/01/0001 00:00:00 code Client : 4

```

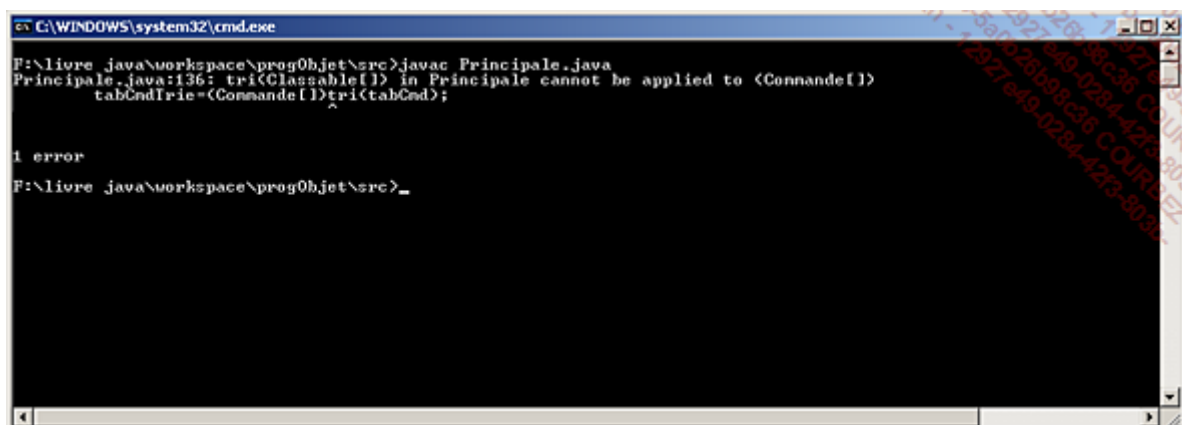

Mr toto5 prenom5 né le 01/01/0001 00:00:00 code Client : 5

Nous avons bien la liste de nos clients triée par ordre alphabétique sur le nom.

Essayons d'utiliser notre procédure de tri avec un tableau d'objets qui n'implémentent pas l'interface **Classable**.

```
Commande[] tabCmd;  
  
    tabCmd=new Commande[5];  
  
    tabCmd[0] = new Commande();  
    tabCmd[1] = new Commande();  
    tabCmd[2] = new Commande();  
    tabCmd[3] = new Commande();  
    tabCmd[4] = new Commande();  
  
    Commande[] tabCmdTrie;  
    tabCmdTrie=(Commande[])tri(tabCmd);  
  
    for (int i=0;i<tabCmdTrie.length;i++)  
    {  
        System.out.println(tabCmdTrie[i]);  
    }
```

À la compilation les choses se compliquent.



Cette erreur intervient au moment de l'appel de la procédure de tri. Les éléments du tableau que nous avons passé comme paramètre n'implémentent pas l'interface **Classable** et nous ne sommes pas certains qu'ils contiennent une fonction **compare**. À noter que, même s'il existe une fonction **compare** correcte dans la classe `Commande`, il faut obligatoirement spécifier que cette classe implémente l'interface **Classable**, pour que le code puisse fonctionner.

Méthodes par défaut

Maintenant que notre code fonctionne correctement, nous pouvons envisager l'optimisation de l'algorithme de tri utilisé. Pour cette amélioration, nous avons besoin que les objets que nous souhaitons trier fournissent deux méthodes supplémentaires. Nous pouvons donc ajouter à l'interface **Classable** la signature de ces deux méthodes.

```
// cette interface devra être implémentée par les classes
// pour lesquelles un classement des instances est envisagé

public interface Classable
{
    // cette méthode pourra être appelée pour comparer l'instance
    // courante avec celle reçue en paramètre
    // la méthode retourne un entier dont la valeur dépend des règles
    // suivantes
    // 1 si l'instance courante est supérieure à celle reçue en
    // paramètre
    // 0 si les deux instances sont égales
    // -1 si l'instance courante est inférieure à celle reçue en
    // paramètre
    // -99 si la comparaison est impossible

    int compare(Object o);

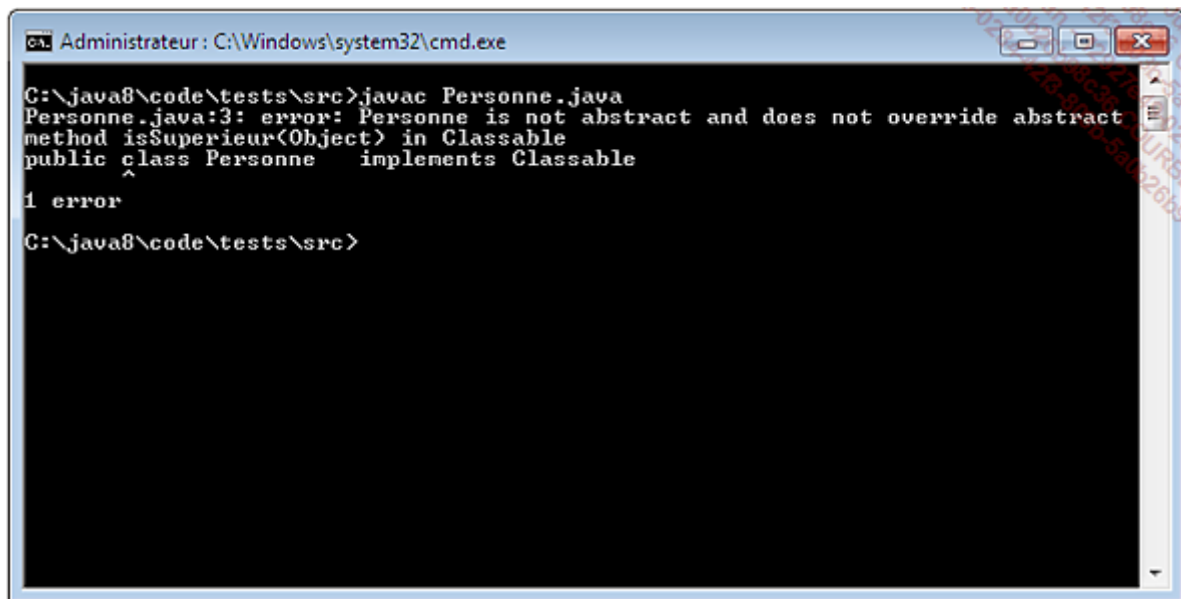
    boolean isInferieur(Object o);

    boolean isSuperieur(Object o);

    public static final int INFERIEUR=-1;
    public static final int EGAL=0;
    public static final int SUPERIEUR=1;
    public static final int ERREUR=-99;

}
```

L'ajout de ces deux méthodes provoque maintenant un problème lors de la compilation de la classe `Personne`, et plus généralement celle de toutes les classes qui ont implémenté l'ancienne version de l'interface.



```
Administrateur : C:\Windows\system32\cmd.exe

C:\java8\code\tests\src>javac Personne.java
Personne.java:3: error: Personne is not abstract and does not override abstract
method isSuperieur(Object) in Classable
public class Personne implements Classable
      ^
1 error

C:\java8\code\tests\src>
```

Nous pouvons définir les deux méthodes ajoutées à l'interface avec le mot clé **default**. Il faut également fournir une implémentation par défaut pour ces deux méthodes. Celles-ci doivent donc comporter un bloc de code délimité par des accolades.

Mais que peut-on placer dans ce bloc de code puisque nous ne savons pas sur quelle classe l'interface va être appliquée ?

Peu importe, ce bloc de code est important par sa présence et non par son contenu. Il permet simplement aux classes ayant implémenté l'ancienne version de l'interface d'être compatibles avec la nouvelle version.

```
// cette interface devra être implémentée par les classes
// pour lesquelles un classement des instances est envisagé

public interface Classable
{
    // cette méthode pourra être appelée pour comparer l'instance
    // courante avec celle reçue en paramètre
    // la méthode retourne un entier dont la valeur dépend des règles
    // suivantes
    // 1 si l'instance courante est supérieure à celle reçue en
    // paramètre
    // 0 si les deux instances sont égales
    // -1 si l'instance courante est inférieure à celle reçue en
    // paramètre
    // -99 si la comparaison est impossible

    int compare(Object o);

    default boolean isInferieur(Object o)
```

```

    {

        return false;

    }

    default boolean isSuperieur(Object o)

    {

        return false;

    }


    public static final int INFERIEUR=-1;
    public static final int EGAL=0;
    public static final int SUPERIEUR=1;
    public static final int ERREUR=-99;

}

```

Si certaines classes n'implémentent pas toutes les méthodes de l'interface, elles héritent des méthodes définies par défaut dans l'interface. Elles ont bien sûr la possibilité de fournir leur propre implémentation de ces méthodes. Avec cette nouvelle définition de l'interface, la classe `Personne` n'a pas besoin d'être modifiée. Nous pouvons cependant créer d'autres classes qui implémentent complètement l'interface en fournissant toutes les méthodes de l'interface.

```

public class Voiture implements Classable

{

    private String immatriculation;
    private String marque;
    private String modele;
    private int puissance;

    public Voiture()

    {

        super();

    }


    public Voiture(String immatriculation,String marque,String
modele,int puissance)

    {

        this.immatriculation=immatriculation;
        this.marque=marque;

```

```
        this.modele=modele;
        this.puissance=puissance;
    }

    public String getImmatriculation()
    {
        return immatriculation;
    }

    public void setImmatriculation(String immatriculation)
    {
        this.immatriculation = immatriculation;
    }
    public String getMarque()
    {
        return marque;
    }
    public void setMarque(String marque)
    {
        this.marque = marque;
    }
    public String getModele()
    {
        return modele;
    }
}

    public void setModele(String modele)
    {
        this.modele = modele;
    }
    public int getPuissance()
    {
        return puissance;
    }
}
```

```
public void setPuissance(int puissance)
{
    this.puissance = puissance;
}
```

```
@Override
```

```
public int compare(Object o)
{
    Voiture v;
    if (o instanceof Voiture)
    {
        v=(Voiture)o;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (getPuissance()<v.getPuissance())
    {
        return Classable.INFERIEUR;
    }
    if (getPuissance()>v.getPuissance())
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
}
```

```
@Override
```

```
public boolean isInferieur(Object o)
{
    Voiture v;
    if (o instanceof Voiture)
    {
```

```

        v=(Voiture)o;
    }
    else
    {
        return false;
    }
    if (getPuissance()<v.getPuissance())
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```

@Override
public boolean isSuperieur(Object o)
{
    Voiture v;
    if (o instanceof Voiture)
    {
        v=(Voiture)o;
    }
    else
    {
        return false;
    }
    if (getPuissance()>v.getPuissance())
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

```

    }

}

}

```

2.5. Classes imbriquées

La majorité des classes que vous utiliserez dans une application seront définies dans leur propre fichier source. Cependant Java fournit la possibilité de déclarer une classe à l'intérieur d'une autre classe, voire à l'intérieur d'une méthode. Cette technique permet de définir une classe uniquement dans le contexte où elle est réellement utile. On désigne également les classes imbriquées par le terme classes assistantes. Suivant l'emplacement de leur déclaration elles ont accès, soit aux autres membres de la classe dans laquelle elles sont déclarées (y compris les membres privés) soit uniquement aux variables locales des méthodes.

Classes imbriquées statiques

Comme n'importe quel élément déclaré dans une classe (champ ou méthode) une classe imbriquée peut être déclarée avec le mot clé **static**. Elle est dans ce cas soumise aux mêmes règles imposées par ce mot clé que les autres éléments d'une classe.

- Elle ne peut utiliser que les champs et méthodes statiques de sa classe container.
- Elle peut être utilisée (instanciée) sans qu'il existe une instance de sa classe container.
- Elle peut utiliser les membres d'instance de sa classe container uniquement par l'intermédiaire d'une instance de celle-ci.

Voici ci-dessous un exemple simple de classe imbriquée statique.

```

public class Externe
{
    static class Interne
    {
        public double calculTTC(double prix)
        {
            return prix*taux;
        }
    }

    static double taux=1.196;
}

```

Elle peut être utilisée de façon tout à fait semblable à n'importe quelle autre classe. La seule contrainte réside dans le nom utilisé pour y faire référence dans le code qui doit être précédé du nom de la classe container.

```

Externe.Interne ci;

ci=new Externe.Interne();

System.out.println(ci.calculTTC(100));

```


Classes internes

Les classes imbriquées sont également connues sous la désignation de classes internes. Elles sont soumises aux mêmes règles que n'importe quel élément déclaré dans une classe.

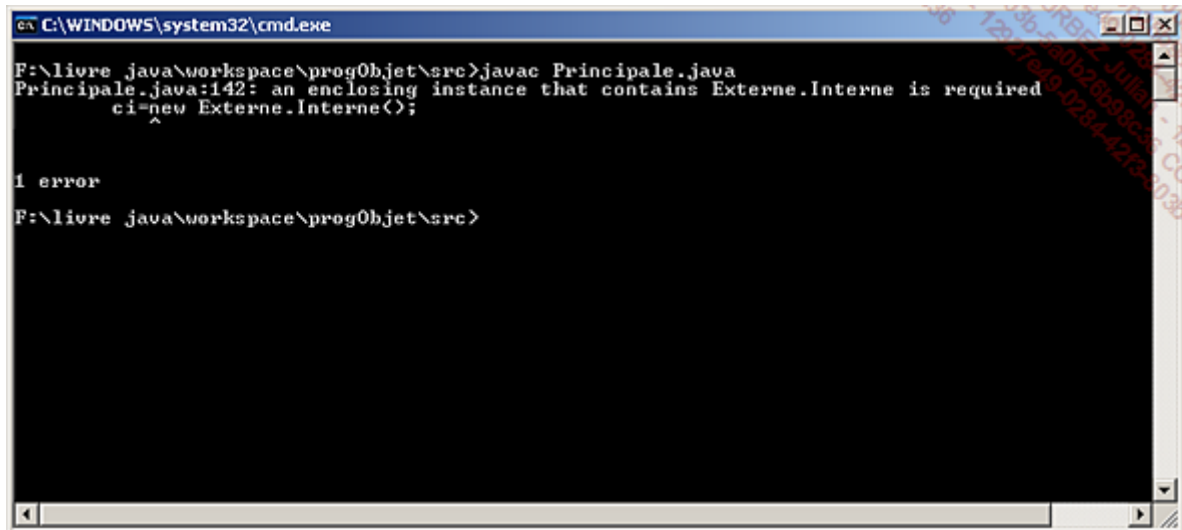
- Elles peuvent avoir accès à tous les membres de la classe dans laquelle elles sont déclarées y compris les membres privés.
- Elles ne peuvent être utilisées que si une instance de la classe container est disponible.

La déclaration est très simple et tout à fait similaire à la précédente hormis le mot clé `static` qui disparaît.

```
public class Externe
{
    class Interne
    {
        public double calculTTC(double prix)
        {
            return prix*taux;
        }
    }

    double taux=1.196;
}
```

Par contre, si nous tentons de l'utiliser avec le même code que la version **static** de la classe, nous rencontrons quelques soucis au moment de la compilation.



Effectivement nous devons obligatoirement avoir une instance de la classe container pour que la classe interne soit disponible. Nous devons donc au préalable instancier la classe puis lui demander de nous fournir une instance de la classe interne. La bonne syntaxe est donc :

```
Externe e;

e=new Externe();
```

```

Externe.Interne ci;

ci=e.new Interne();

System.out.println(ci.calculTTC(100));

```

Classes anonymes

Une classe anonyme est une classe interne pour laquelle aucun nom n'est fourni. Les classes anonymes sont tout à fait adaptées pour réaliser une opération nécessitant un objet mais qui ne justifie pas la création d'une classe normale. C'est le cas par exemple si la classe est très simple ou si elle n'est utilisée que dans une seule méthode. Ce mécanisme est très utilisé pour la gestion des actions de l'utilisateur dans une application en mode graphique.

Pour illustrer l'utilisation des classes internes anonymes, nous allons reprendre la fonction de tri de tableau créée précédemment et la rendre encore plus universelle. Dans sa version actuelle, elle exige que les éléments du tableau implémentent l'interface **Classable** pour qu'elle puisse les comparer deux à deux. Nous modifions légèrement la fonction pour que, lors de l'appel, on puisse lui fournir en plus du tableau à trier, la fonction qu'elle devra utiliser pour effectuer les comparaisons. Une solution pour transmettre une fonction à une autre fonction est de lui fournir une instance de classe qui contient la fonction à transmettre. Par mesure de sécurité, nous pouvons exiger que cette instance soit créée à partir d'une classe implémentant une interface pour garantir l'existence de la fonction. Voici le code de cette nouvelle interface :

```

// cette interface devra être implémentée par les classes
// pour lesquelles une comparaison des instances est envisagée
public interface Compareur
{
    // cette méthode pourra être appelée pour comparer les deux
    // objets reçus en paramètre
    // la méthode retourne un entier dont la valeur dépend
    // des règles suivantes
    // 1 si l'instance o1 est supérieure à o2
    // 0 si les deux instances sont égales
    // -1 si l'instance o1 est inférieure à o2
    // -99 si la comparaison est impossible
    int compare(Object o1, Object o2);

    public static final int INFERIEUR=-1;
    public static final int EGAL=0;
    public static final int SUPERIEUR=1;
    public static final int ERREUR=-99;
}

```

Une telle interface qui ne contient que la définition d'une seule et unique méthode est appelée interface fonctionnelle.

Nous pouvons maintenant revoir légèrement notre fonction de tri en prenant en compte nos améliorations.

```
public static Object[] tri(Object[] tablo, Comparateur trieur)
{
    int i, j;
    Object c;
    Object[] tabloTri;
    tabloTri = Arrays.copyOf(tablo, tablo.length);
    for (i = 0; i < tabloTri.length; i++)
    {
        for (j = i + 1; j < tabloTri.length; j++)
        {
            // utilise la fonction compare de l'objet reçu en paramètre
            // pour comparer le contenu de deux cases du tableau
            if
(trieur.compare(tabloTri[j], tabloTri[i]) == Comparateur.INFERIEUR)
            {
                c = tabloTri[j];
                tabloTri[j] = tabloTri[i];
                tabloTri[i] = c;
            }
            else if
(trieur.compare(tabloTri[j], tabloTri[i]) == Comparateur.ERREUR)
            {
                return null;
            }
        }
    }
    return tabloTri;
}
```

Pour utiliser cette nouvelle fonction de tri nous devons maintenant lui fournir deux paramètres :

- Le tableau à trier.
- Une instance de classe ayant implémenté l'interface `comparateur`.

Pour fournir cette instance de classe nous avons plusieurs solutions :

- Créer une instance d'une classe "normale" qui implémente l'interface.
- Créer une instance classe interne qui implémente l'interface.
- Créer une instance d'une classe interne anonyme qui implémente l'interface.

C'est cette dernière solution que nous allons utiliser. La syntaxe de création d'une instance d'une classe interne anonyme est au premier abord assez bizarre.

```
new "nom de la classe de base ou de l'interface implémentée" ()
{
    // déclaration des champs

    int i;

    float j;

    // déclaration des méthodes

    public int method1(..., ...)
    {
        } // accolade de fermeture du bloc de code de la méthode

    } // accolade de fermeture du bloc de code de la classe
```

Comme pour créer une instance de classe normale, nous utilisons l'opérateur `new` suivi du nom de la classe. En fait nous n'indiquons pas directement le nom de la classe dont nous souhaitons obtenir une instance, mais le nom de la classe de base ou de l'interface implémentée par cette classe (la classe n'a pas de nom puisqu'elle est anonyme !). Nous fournissons ensuite un bloc de code délimité par des accolades correspondant au contenu de la classe, comme pour la définition d'une classe normale. À noter que si la classe anonyme implémente une interface, ce bloc de code doit obligatoirement fournir toutes les méthodes exigées par l'interface.

Voici la mise en application de ces principes pour l'appel de notre fonction de tri avec comme critère de tri le nom de la personne.

```
Personne[] tab;

tab=new Personne[5];

tab[0] = new Personne("toto2", "prenom2",new
GregorianCalendar(1922,2,15));

tab[1] = new Personne("toto1", "prenom1 ",new
GregorianCalendar(1911,1,15));

tab[2] = new Personne("toto5", "prenom5 ",new
GregorianCalendar(1955,05,15));

tab[3] = new Personne("toto3", "prenom3 ",new
GregorianCalendar(1933,03,15));

tab[4] = new Personne("toto4", "prenom4 ",new
```

```
GregorianCalendar(1944,04,15));
```

```
tabTrie=(Personne[])tri(tab,
    // création d'une instance de classe implémentant
    // l'interface Comparateur
    new Comparateur()
    // voici le code de la classe
    {
        // comme l'exige l'interface voici la méthode compare
        public int compare(Object o1, Object o2)
        {
            Personne p1,p2;
            if (o1 instanceof Personne & o2 instanceof Personne)
            {
                p1=(Personne)o1;
                p2=(Personne)o2;
            }
            else
            {
                return Classable.ERREUR;
            }
            if (p1.getNom().compareTo(p2.getNom())<0)
            {
                return Classable.INFERIEUR;
            }
            if (p1.getNom().compareTo(p2.getNom())>0)
            {
                return Classable.SUPERIEUR;
            }

            return Classable.EGAL;

        } // accolade de fermeture de la méthode compare

    } // accolade de fermeture de la classe
```

```

        ); // fin de l'appel de la fonction de tri

// affichage du tableau trié

for (int i=0;i<tabTrie.length;i++)
{
    System.out.println(tabTrie[i]);
}

```

Si nous voulons effectuer un autre tri avec un critère différent, nous devons simplement appeler la fonction **tri** avec une nouvelle instance d'une classe interne anonyme qui implémente différemment la fonction **Compare**. Nous pouvons par exemple trier les personnes sur leur âge.

```

tabTrie=(Personne[])tri(tab,
// création d'une instance de classe implémentant
// l'interface Compareur
    new Compareur()
    // voici le code de la classe
    {
        // comme l'exige l'interface voici la méthode compare
        public int compare(Object o1, Object o2)
        {
            Personne p1,p2;
            if (o1 instanceof Personne & o2 instanceof Personne)
            {
                p1=(Personne)o1;
                p2=(Personne)o2;
            }
            else
            {
                return Classable.ERREUR;
            }
            if (p1.calculAge()<p2.calculAge())
            {
                return Classable.INFERIEUR;
            }
            if (p1.calculAge()>p2.calculAge())

```

```

        {
            return Classable.SUPERIEUR;
        }
        return Classable.EGAL;
    } // accolade de fermeture de la méthode

} // accolade de fermeture de la classe
); // fin de l'appel de la fonction de tri

for (int i=0;i<tabTrie.length;i++)
{
    System.out.println(tabTrie[i]);
}

```

Lors de la compilation de ce code, des fichiers .class sont générés pour chaque classe utilisée dans le code. Le compilateur génère automatiquement un nom pour le fichier de chaque classe anonyme. Il utilise pour cela le nom de classe container auquel il ajoute le suffixe \$ suivi d'une valeur numérique. La compilation de ce code va donc générer les fichiers suivants :

Principale.class : le fichier correspondant à la classe Principale.

Principale\$1.class : le fichier correspondant à la première classe interne anonyme.

Principale\$2.class : le fichier correspondant à la deuxième classe interne anonyme.

Le seul inconvénient des classes internes anonymes réside dans le fait qu'elles sont à usage unique. Si vous souhaitez la réutiliser plusieurs fois, il faut dans ce cas créer une classe nommée. Le nom attribué par le compilateur à la classe interne anonyme n'étant bien sûr pas accessible à partir du code (puisque'elle est anonyme !).

2.6. Expression lambda

Une expression lambda ressemble à une fonction mais à part qu'elle ne porte pas de nom. Elle est constituée d'un couple de parenthèses contenant les éventuels paramètres suivis des caractères -> et du bloc de code de l'expression délimité par des accolades.

L'appel à la fonction de tri peut donc se faire de la manière suivante, en utilisant une expression lambda à la place de l'instance de classe implémentant l'interface **Compareur**.

```

tabTrie=(Personne[])tri(tab,(Object o1,Object o2)->
{
    Personne p1,p2;
    if (o1 instanceof Personne & o2 instanceof Personne)
    {

```

```

        p1=(Personne)o1;
        p2=(Personne)o2;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (p1.getNom().compareTo(p2.getNom())<0)
    {
        return Classable.INFERIEUR;
    }
    if (p1.getNom().compareTo(p2.getNom())>0)
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
});

```

Ce code ressemble énormément à celui utilisé pour appeler la fonction de tri avec comme paramètre une instance de classe anonyme.

```

tabTrie=(Personne[])tri(tab,  new Compareteur()
{
    public int compare(Object o1, Object o2)
    {
        Personne p1,p2;
        if (o1 instanceof Personne & o2 instanceof Personne)
        {
            p1=(Personne)o1;
            p2=(Personne)o2;
        }
        else
        {
            return Classable.ERREUR;
        }
    }
}

```



```

        if (p1.getNom().compareTo(p2.getNom())<0)
        {
            return Classable.INFERIEUR;
        }

        if (p1.getNom().compareTo(p2.getNom())>0)
        {
            return Classable.SUPERIEUR;
        }

        return Classable.EGAL;
    }

}

);

```

La syntaxe reste cependant relativement complexe. Regardons maintenant comment simplifier un peu cela.

Dans une application, nous devons gérer un ensemble de personnes. Pour cette gestion nous regroupons nos personnes dans un tableau.

```

Personne[] tab;

tab=new Personne[5];

tab[0] = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
tab[1] = new Personne("McQueen", "Steeve ",LocalDate.of(1930,3,24));
tab[2] = new Personne("Lennon", "John ",LocalDate.of(1940,10,9));
tab[3] = new Personne("Gibson", "Mel ",LocalDate.of(1956,1,3));
tab[4] = new Personne("Willis", "Bruce ",LocalDate.of(1955,3,19));

```

Notre application doit proposer différentes fonctionnalités de recherche d'une personne dans le tableau.

- La recherche par nom
- La recherche par prénom
- La recherche par nom et prénom
- La recherche par âge

Nous devons donc créer quatre fonctions permettant d'effectuer ces recherches.

```

public static Personne rechercheParNom(Personne[] tablo,String nom)
{
    for(Personne p:tablo)

```

```

        {
            if (p.getNom().equals(nom))
            {
                return p;
            }
        }
        return null;
    }

    public static Personne rechercheParPrenom(Personne[] tablo,String prenom)
    {
        for(Personne p:tablo)
        {
            if (p.getPrenom().equals(prenom))
            {
                return p;
            }
        }
        return null;
    }

    public static Personne rechercheParAge(Personne[] tablo,int age)
    {
        for(Personne p:tablo)
        {
            if (p.calculAge()==age)
            {
                return p;
            }
        }
        return null;
    }

    public static Personne rechercheParNomPrenom(Personne[]
    tablo,String nom,String prenom)
    {

```

```

        for(Personne p:tablo)
        {
            if (p.getNom().equals(nom) && p.getPrenom().equals(prenom))
            {
                return p;
            }
        }
        return null;
    }
}

```

Ces quatre fonctions ont encore une fois énormément de ressemblances. Seule la ligne de code effectuant la comparaison change d'une version à l'autre. Pour nous permettre de « factoriser » notre code, il peut être intéressant d'extraire de la fonction le code de comparaison.

Commençons par définir une interface décrivant la signature que devra respecter la fonction chargée de vérifier l'égalité de deux personnes.

```

public interface CompareurPersonne
{
    boolean isIdentique(Personne p);
}

```

Il nous faut maintenant concevoir la nouvelle version de la fonction de recherche d'une personne qui utilise l'interface définie précédemment.

```

public static Personne recherchePersonne(Personne[] tablo,
CompareurPersonne cp)
{
    for(Personne p:tablo)
    {
        if (cp.isIdentique(p))
        {
            return p;
        }
    }
    return null;
}

```

Il n'y a plus aucune trace de critère de comparaison à l'intérieur de cette fonction. Celui-ci sera défini lors de l'appel de la fonction.

Le package **java.util.function** propose de nombreuses interfaces contenant des définitions de fonctions dont l'utilisation revient de manière récurrente dans une application. Pour que ces interfaces soient facilement utilisables, elles sont généralement génériques. En utilisant l'une de ces interfaces prédéfinies, notre fonction de recherche peut s'écrire :

```
public static Personne recherchePersonnePrd(Personne[]
tablo, Predicate<Personne>pr)
{
    for(Personne p:tablo)
    {
        if (pr.test(p))
        {
            return p;
        }
    }
    return null;
}
```

Appliquons maintenant notre première expérience d'écriture d'une expression lambda à cette nouvelle fonction.

```
System.out.println(recherchePersonne(tab, (Personne pe)->
{
    if(pe.getPrenom().equals("Bruce"))
        return true;
    else
        return false;
}
));
```

Cette syntaxe fonctionne correctement mais reste relativement verbeuse. Regardons maintenant comment simplifier cette expression.

La première simplification porte sur les paramètres de l'expression lambda. Nous ne sommes pas obligés de spécifier le type des paramètres et, en complément, si l'expression n'attend qu'un seul paramètre les parenthèses sont elles aussi facultatives.

```
System.out.println(recherchePersonne(tab, pe->
{
    if(pe.getPrenom().equals("Bruce"))
        return true;
}
```

```

        else
            return false;
    }
});

```

Cette syntaxe fonctionne correctement mais reste relativement verbeuse. Regardons maintenant comment simplifier cette expression.

La première simplification porte sur les paramètres de l'expression lambda. Nous ne sommes pas obligés de spécifier le type des paramètres et, en complément, si l'expression n'attend qu'un seul paramètre les parenthèses sont elles aussi facultatives.

```

System.out.println(recherchePersonne(tab, pe->
{
    if(pe.getPrenom().equals("Bruce"))
        return true;
    else
        return false;
}
));

```

La deuxième simplification possible porte sur le corps de l'expression lambda. Si celle-ci ne contient qu'une seule expression, les accolades sont facultatives ainsi que l'utilisation du mot clé `return`. Dans ce cas, l'expression est évaluée lors de l'exécution et la valeur générée est retournée au code appelant.

```

System.out.println(recherchePersonne(tab, pe->
pe.getPrenom().equals("Bruce")));

```

Si vous utilisez le mot clé `return` dans votre expression, vous devez obligatoirement placer celle-ci dans un bloc de code délimité par des accolades, même si elle ne contient qu'une seule expression.

L'expression lambda peut utiliser les variables disponibles dans le contexte où elle est définie.

```

BufferedReader br;

br=new BufferedReader(new InputStreamReader(System.in));
String prenom;
prenom=br.readLine();

System.out.println(recherchePersonne(tab, pe->
{
    if(pe.getPrenom().equals(prenom))
        return true;
}
));

```

```

        else
            return false;
    }
});

```

Toutefois, l'expression lambda ne peut pas modifier le contenu de la variable mais uniquement l'utiliser. La modification suivante génère une erreur lors de la compilation.

```

BufferedReader br;

br=new BufferedReader(new InputStreamReader(System.in));

String prenom;

prenom=br.readLine();

System.out.println(recherchePersonne(tab, pe->
{
    prenom=prenom.toLowerCase();
    if(pe.getPrenom().equals(prenom))
        return true;
    else
        return false;
}
));

```

```

Administrateur : invite de commandes
D:\java8\code\tests\src>javac Principale.java
Principale.java:132: error: local variables referenced from a lambda expression must be final
    prenom=prenom.toLowerCase();
    ^
Principale.java:132: error: local variables referenced from a lambda expression must be final
    prenom=prenom.toLowerCase();
    ^
Principale.java:133: error: local variables referenced from a lambda expression must be final
    if(pe.getPrenom().equals(prenom))
    ^
3 errors
D:\java8\code\tests\src>_

```

2.7. Référence de méthode

Lorsqu'une expression lambda devient trop volumineuse ou qu'elle doit être réutilisée à plusieurs emplacements dans l'application, il est préférable de déplacer son contenu dans une fonction et de simplement appeler cette fonction dans le corps de l'expression lambda.

Par exemple, l'expression lambda suivante :

```
tabTrie=(Personne[])tri(tab, (Object o1,Object o2)->
```

```

{
    Personne p1,p2;
    if (o1 instanceof Personne & o2 instanceof Personne)
    {
        p1=(Personne)o1;
        p2=(Personne)o2;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (p1.getNom().compareTo(p2.getNom())<0)
    {
        return Classable.INFERIEUR;
    }
    if (p1.getNom().compareTo(p2.getNom())>0)
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
});

```

peut facilement être externalisée dans une fonction.

```

public static int comparePersonne(Object o1,Object o2)
{
    Personne p1,p2;
    if (o1 instanceof Personne & o2 instanceof Personne)
    {
        p1=(Personne)o1;
        p2=(Personne)o2;
    }
    else
    {
        return Classable.ERREUR;
    }
}

```

```

    }

    if (p1.getNom().compareTo(p2.getNom())<0)
    {
        return Classable.INFERIEUR;
    }

    if (p1.getNom().compareTo(p2.getNom())>0)
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
}

```

Cette fonction est maintenant utilisable dans plusieurs expressions lambda. Le code de l'expression est d'ailleurs énormément simplifié par cette modification.

```

tabTrie=(Personne[])tri(tab,(Object o1,Object o2)->
comparePersonne(o1, o2));

```

La simplification peut être encore plus poussée en utilisant une référence de méthode. Cette solution consiste simplement à identifier la méthode à utiliser par son nom. L'opérateur `::` doit être utilisé dans cette situation. Il permet d'obtenir une référence vers la fonction à utiliser mais ne provoque pas l'appel de la fonction comme le ferait l'opérateur `.`

```

tabTrie=(Personne[])tri(tab,Principale::comparePersonne);

```

Cette syntaxe est possible car la fonction est déclarée avec le mot clé **static**. Il n'y a donc pas besoin d'avoir une instance de la classe dans laquelle elle est définie pour pouvoir l'utiliser.

Si la fonction n'est pas déclarée **static**, la règle est la même que pour tous les autres éléments disponibles dans une classe : il faut obligatoirement avoir une instance de la classe pour qu'elle soit utilisable.

Dans ce cas, nous devons utiliser le nom de la variable contenant l'instance de la classe pour pouvoir faire référence à la méthode.

```

TrieusePersonne tp;

tp=new TrieusePersonne();

tabTrie=(Personne[])tri(tab,tp::comparePersonne);

```

2.8. Les génériques

Les types génériques sont des éléments d'un programme qui s'adaptent automatiquement pour réaliser la même fonctionnalité sur différents types de données. Lorsque vous créez un élément générique, vous n'avez pas besoin de concevoir une version différente pour chaque type de donnée avec lequel vous souhaitez réaliser une fonctionnalité.

Lorsque vous utilisez un type générique, vous le paramétrez avec un type de données. Ceci permet au code de s'adapter automatiquement et de réaliser la même action indépendamment du type de

données. Une alternative pourrait être l'utilisation du type universel **Object**. L'utilisation des types génériques présente plusieurs avantages par rapport à cette solution :

- Elle impose la vérification des types de données au moment de la compilation et évite les inévitables vérifications qui doivent être faites manuellement avec l'utilisation du type **Object**.
- Elle évite les opérations de conversion du type **Object** vers un type plus spécifique et inversement.
- L'écriture du code est facilitée dans certains environnements de développement avec l'affichage automatique de tous les membres disponibles pour un type de données particulier.
- Elle favorise l'écriture d'algorithmes qui sont indépendants des types de données.

Les types génériques peuvent cependant imposer certaines restrictions concernant le type de données utilisé. Ils peuvent par exemple imposer que le type utilisé implémente une ou plusieurs interfaces, soit un type référence ou possède un constructeur par défaut. Il est important de bien comprendre quelques termes utilisés avec les génériques.

- Le type générique : c'est la définition d'une classe, interface ou fonction pour laquelle vous spécifiez au moins un type de données au moment de sa déclaration.
- Le type de paramètre : c'est l'emplacement réservé pour le type de paramètre dans la déclaration du type générique.
- Le type argument : c'est le type de données qui remplace le type de paramètre lors de la construction d'un type à partir d'un type générique.
- Les contraintes : ce sont les conditions que vous imposez qui limitent le type argument que vous pouvez fournir.
- Le type construit : c'est la classe, interface, ou fonction construite à partir d'un type générique pour lequel vous avez spécifié des types argument.

Classes génériques

Une classe qui attend un type de paramètre est appelée classe générique. Vous pouvez générer une classe construite en fournissant à la classe générique un type argument pour chacun de ces types paramètre.

Définition d'une classe générique

Vous pouvez définir une classe générique qui fournit les mêmes fonctionnalités sur différents types de données. Pour cela, vous devez fournir un ou plusieurs types de paramètre dans la définition de la classe. Prenons l'exemple d'une classe capable de gérer une liste d'éléments avec les fonctionnalités suivantes.

- Ajouter un élément.
- Supprimer un élément.
- Se déplacer sur le premier élément.
- Se déplacer sur le dernier élément.
- Se déplacer sur l'élément suivant.
- Se déplacer sur l'élément précédent.
- Obtenir le nombre d'éléments.

Nous devons tout d'abord définir la classe comme une classe ordinaire.

```
public class ListeGenerique
```

```
{  
}
```

La transformation de cette classe en classe générique se fait en ajoutant un type de paramètre immédiatement après le nom de la classe.

```
public class ListeGenerique <T>  
{  
}
```

Si plusieurs types paramètres sont nécessaires, ils doivent être séparés par des virgules dans la déclaration. Par convention les types paramètres sont représentés par un caractère majuscule unique.

Si le code de la classe doit réaliser d'autres opérations que des affectations, vous devez ajouter des contraintes sur le type paramètre. Pour cela, ajoutez le mot clé **extends** suivi de la contrainte. La contrainte peut être constituée par une classe spécifique dont le type argument devra hériter ou par une ou plusieurs interfaces qu'il devra implémenter. Si plusieurs contraintes doivent s'appliquer, elles sont séparées par le caractère &. Il faut dans ce cas spécifier en début de liste la contrainte liée à une classe puis celles liées aux interfaces. Ci-dessous quelques exemples pour illustrer cela.

Classe générique exigeant que le type argument hérite de la classe `Personne` :

```
public class ListeGenerique <T extends Personne >  
{  
}
```

Classe générique exigeant que le type argument implémente l'interface **Classable** :

```
public class ListeGenerique <T extends Classable >  
{  
}
```

C'est le même mot clé **extends** qui est utilisé pour une contrainte liée à une classe ou à une interface.

Classe générique exigeant que le type argument hérite de la classe `Personne` et implémente les interfaces **Classable** et **Cloneable** :

```
public class ListeGenerique <T extends Personne & Classable & Cloneable >  
{  
}
```

S'il n'y a pas de contrainte spécifiée, les seules opérations autorisées seront celles supportées par le type **Object**.

Dans le code de la classe, chaque membre qui doit être du type argument doit être défini avec le type paramètre, T dans notre cas.

Voyons maintenant le code complet de la classe :

```
import java.util.ArrayList;
```

```

public class ListeGenerique <T>
{
    // pour stocker les éléments de la liste
    private ArrayList<T> liste;
    // pointeur de position dans la liste
    private int position;
    // nombre d'éléments de la liste
    private int nbElements;
    // constructeur avec un paramètre permettant de dimensionner
    // la liste
    public ListeGenerique(int taille)
    {
        liste=new ArrayList<T>(taille);
    }
    public void ajout(T element)
    {
        liste.add(element);
        nbElements = nbElements + 1;
    }
    public void insert(T element,int index)
    {
        // on vérifie si l'index n'est pas supérieur au nombre
        // d'éléments ou si l'index n'est pas inférieur à 0
        if (index >= nbElements || index < 0)
        {
            return;
        }
        liste.add(index,element);
        // on met à jour le nombre d'éléments
        nbElements = nbElements + 1;
    }

    public void remplace(T element,int index)
    {
        // on vérifie si l'index n'est pas supérieur au nombre

```

```

        // d'éléments ou si l'index n'est pas inférieur à 0
        if (index >= nbElements || index < 0)
        {
            return;
        }
        liste.set(index,element);
    }
    public void supprime(int index)
    {
        int i;
        // on vérifie si l'index n'est pas supérieur au nombre
        // d'éléments ou si l'index n'est pas inférieur à 0
        if (index >= nbElements || index < 0)
        {
            return;
        }
        liste.remove(index);
        // on met à jour le nombre d'éléments
        nbElements = nbElements - 1;
    }
    public T getElement(int j)
    {
        return liste.get(j);
    }
    public int getNbElements()
    {
        return nbElements;
    }
    public T premier() throws Exception
    {
        if (nbElements == 0)
        {
            throw new Exception("liste vide");
        }
        // on déplace le pointeur sur le premier élément

```

```

        position = 0;
        return liste.get(0);
    }

    public T dernier() throws Exception
    {
        if (nbElements == 0)
        {
            throw new Exception("liste vide");
        }

        // on déplace le pointeur sur le dernier élément
        position = nbElements - 1;
        return liste.get(position);
    }

    public T suivant() throws Exception
    {
        if (nbElements == 0)
        {
            throw new Exception("liste vide");
        }

        // on vérifie si on n'est pas à la fin de la liste
        if (position == nbElements - 1)
        {
            throw new Exception("pas d'element suivant");
        }

        // on déplace le pointeur sur l'élément suivant
        position = position + 1;
        return liste.get(position);
    }

    public T precedent() throws Exception
    {
        if (nbElements == 0)
        {
            throw new Exception("liste vide");
        }

        // on vérifie si on n'est pas sur le premier élément

```

```

        if (position == 0)
        {
            throw new Exception("pas d'élément précédent");
        }

        // on se déplace sur l'élément précédent
        position = position - 1;
        return liste.get(position);
    }
}

```

Utilisation d'une classe générique

Pour pouvoir utiliser une classe générique, vous devez tout d'abord générer une classe construite en fournissant un type argument pour chacun de ces types paramètre. Vous pouvez alors instancier la classe construite par un des constructeurs disponibles. Nous allons utiliser la classe conçue précédemment pour travailler avec une liste de chaînes de caractères.

```
ListeGenerique<String> liste = new ListeGenerique<String>(5);
```

Cette syntaxe peut être simplifiée en laissant le compilateur déterminer lui-même le type argument à utiliser lors de l'appel du constructeur. Il suffit simplement d'omettre le type argument entre les caractères < et >.

```
ListeGenerique<String> liste=new ListeGenerique<>(5);
```

Cette déclaration permet d'instancier une liste de cinq chaînes. Les méthodes de la classe sont alors disponibles.

```
liste.ajout("premier") ;
liste.ajout("deuxieme");
```

Le compilateur vérifie bien sûr que nous utilisons notre classe correctement notamment en vérifiant les types de données que nous lui confions.

Voici le code d'une petite application permettant de tester le bon fonctionnement de notre classe générique :

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class TestListeGenerique
{
    static ListeGenerique<String> liste = new
ListeGenerique<String>(5);

    public static void main(String[] args)
    {
        liste.ajout("premier");
    }
}

```



```

liste.dernier());

                break;

            }

        }

        catch (Exception e)

        {

            System.out.println(e.getMessage());

        }

        System.out.println("p (premier) < (précédent) >(suivant) d
(dernier) f (fin)");

    }

}

```

Nous pouvons également vérifier que notre classe fonctionne sans problème si nous lui demandons de travailler avec des personnes :

```

liste.ajout(new Personne("toto2","prenom2",LocalDate.of(1922,2,15)));
liste.ajout(new Personne("toto1","prenom1",LocalDate.of(1911,1,15)));
liste.ajout(new Personne("toto5","prenom5",LocalDate.of(1955,5,15)));
liste.ajout(new Personne("toto3","prenom3",LocalDate.of(1933,3,15)));
liste.ajout(new Personne("toto4","prenom4",LocalDate.of(1944,4,15)));

```

Méthodes génériques

Une fonction générique est une méthode définie avec au moins un type paramètre. Ceci permet au code appelant de spécifier le type de données dont il a besoin à chaque appel de la fonction. Une telle méthode peut cependant être utilisée sans indiquer d'informations pour le type argument. Dans ce cas le compilateur essaie de déterminer le type en fonction des arguments qui sont passés à la méthode. Pour illustrer la création de fonctions génériques, nous allons transformer la fonction de tri en version générique.

```

public static <T extends Classable> void
tri(ListeGenerique<T> liste) throws Exception
{

    int i,j;

    T c;

    for (i=0;i< liste.getNbElements()-1;i++)

    {

        for( j = i + 1; j<liste.getNbElements();j++)

        {

```



```

        if
(liste.getElement(j).compare(liste.getElement(i))==Classable.
INFÉRIEUR)

        {

            c = liste.getElement(j);

            liste.replace(liste.getElement(i), j);

            liste.replace(c,i);

        }

        else if
(liste.getElement(j).compare(liste.getElement(i))==Classable.
ERREUR)

        {

            throw new Exception("erreur pendant le tri");

        }

    }

}

```

Comme pour la création d'une classe générique le type paramètre est spécifié entre les caractères < et >. Si des contraintes doivent s'appliquer aux types arguments, vous devez les spécifier avec les mêmes règles que pour les classes génériques.

L'appel d'une fonction générique est semblable à l'appel d'une fonction normale hormis le fait que nous pouvons spécifier des types arguments pour chacun des types paramètre. Ceci n'est pas une obligation car le compilateur est capable d'inférer les types arguments au moment de la compilation.

L'appel de la fonction peut donc se faire avec les deux syntaxes suivantes :

```
TestListeGenerique.<Personne>tri(liste);
```

ou

```
TestListeGenerique.tri(liste);
```

Les génériques et l'héritage

L'utilisation combinée des génériques et de l'héritage peut parfois provoquer quelques soucis au développeur d'une application. Nous avons déterminé dans le paragraphe consacré à l'héritage qu'avec une variable d'un certain type l'on pouvait référencer une instance de classe de ce type mais aussi une instance de classe de n'importe lequel de ses sous-types. Le code suivant est donc parfaitement légal et fonctionne correctement.

```

Personne p;

Client c;

c=new Client();

p=c;

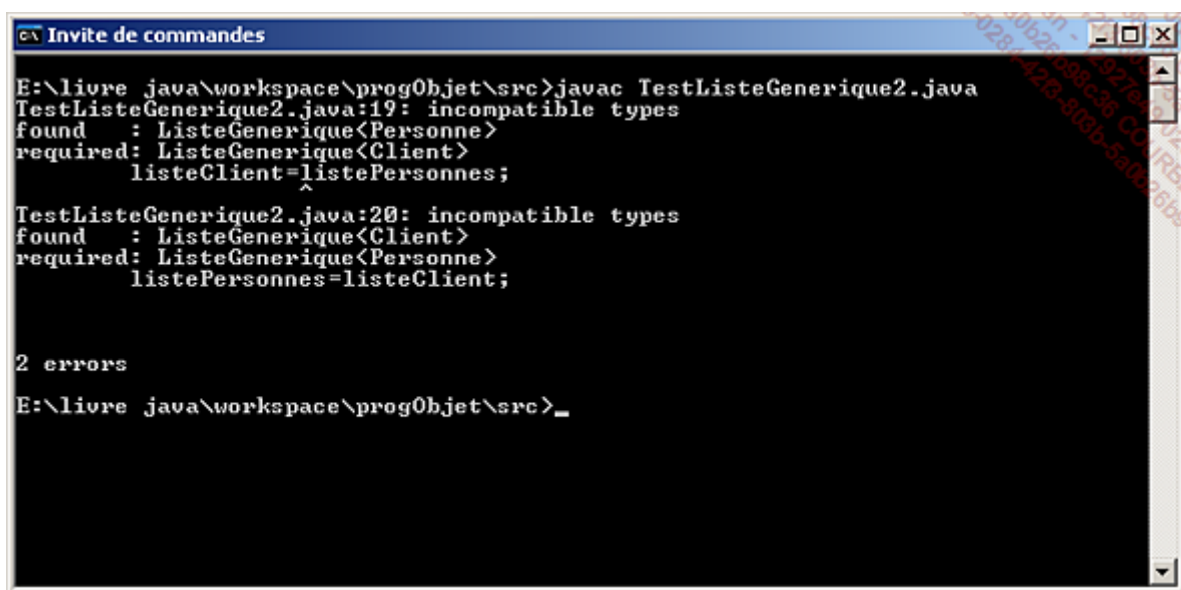
```

```
p.setNom("Dupont");  
p.setPrenom("paul");
```

Si nous tentons la même expérience avec les génériques en testant le code suivant.

```
ListeGenerique<Personne> listePersonnes;  
ListeGenerique<Client> listeClient;  
listeClient=new ListeGenerique<Client>(10);  
listePersonnes=listeClient;
```

Il est effectivement légitime de penser que, puisque l'on peut affecter à une variable de type `Personne` une référence vers une instance d'une de ses sous-classes (`Client`), la même opération est certainement réalisable avec une `ListeGenerique<Personne>` et une `ListeGenerique<Client>`. Pourtant le compilateur ne voit pas les choses de la même façon.



```
Invite de commandes  
E:\livre java\workspace\progObjet\src>javac TestListeGenerique2.java  
TestListeGenerique2.java:19: incompatible types  
found   : ListeGenerique<Personne>  
required: ListeGenerique<Client>  
    listeClient=listePersonnes;  
                        ^  
TestListeGenerique2.java:20: incompatible types  
found   : ListeGenerique<Client>  
required: ListeGenerique<Personne>  
    listePersonnes=listeClient;  
                        ^  
  
2 errors  
E:\livre java\workspace\progObjet\src>_
```

Cette limitation est en fait liée au mécanisme d'effacement de type utilisé par le compilateur. Son principal but est de rendre compatible avec les versions antérieures des machines virtuelles Java le code compilé.

En fait lors de la compilation d'une classe générique, le compilateur remplace le type paramètre par le type **Object** ou par un type correspondant à la contrainte placée sur le type paramètre. Le code de la classe **ListeGenerique** sera traité de la façon suivante par le compilateur.

```
import java.util.ArrayList;  
  
public class ListeGenerique <T>  
{  
    // pour stocker les éléments de la liste  
    private ArrayList<T Object> liste;  
    // pointeur de position dans la liste  
    private int position;  
    //nombre d'éléments de la liste
```

```

private int nbElements;

// constructeur avec un paramètre permettant
// de dimensionner la liste
public ListeGenerique(int taille)
{
    liste=new ArrayList<T Object>(taille);
}

public void ajout(T Object element)
{
    liste.add(element);
    nbElements = nbElements + 1;
}

public void insert(T Object element,int index)
{
    // on vérifie si l'index n'est pas supérieur au nombre
    // d'éléments ou si l'index n'est pas inférieur à 0
    if (index >= nbElements || index < 0)
    {
        return;
    }
    liste.add(index,element);
    // on met à jour le nombre d'éléments
    nbElements = nbElements + 1;
}

public T Object premier() throws Exception
{
    if (nbElements == 0)
    {
        throw new Exception("liste vide");
    }
    // on déplace le pointeur sur le premier élément
    position = 0;
    return liste.get(0);
}

public T Object dernier() throws Exception

```

```

{
    if (nbElements == 0)
    {
        throw new Exception("liste vide");
    }
    // on déplace le pointeur sur le dernier élément
    position = nbElements - 1;
    return liste.get(position);
}
...
...
}

```

Pour en être convaincu, essayons le code suivant dans la classe **ListeGenerique**.

```

public void ajout(T element)
{
    liste.add(element);
    nbElements = nbElements + 1;
}
public void ajout(Object element)
{
    liste.add(element);
    nbElements = nbElements + 1;
}

```

À la compilation, nous avons bien un message d'erreur nous indiquant que la méthode `ajout(Object)` est définie deux fois dans la classe.

Une autre opération réalisée par le compilateur consiste à effectuer une conversion pour les valeurs renvoyées par les fonctions de la classe générique. Le code suivant :

```

static ListeGenerique<Personne> liste = new
ListeGenerique<Personne>(5);
public static void main(String[] args) throws Exception
{
    liste.ajout(new Personne("toto2", "prenom2",
    LocalDate.of(1922, 2, 15)));
    liste.ajout(new Personne("toto1", "prenom1",
    LocalDate.of(1911, 1, 15)));
}

```

```

        liste.ajout(new Personne("toto5","prenom5",
LocalDate.of(1955,5,15)));

        liste.ajout(new Personne("toto3","prenom3",
LocalDate.of(1933,3,15)));

        liste.ajout(new Personne("toto4","prenom4",
LocalDate.of(1944,4,15)));

        Personne p;
        p=liste.getElement(0);
    }

```

sera en fait implicitement compilé sous cette forme :

```

static ListeGenerique<Personne> liste = new
ListeGenerique<Personne>(5);

public static void main(String[] args) throws Exception
{
    liste.ajout(new Personne("toto2","prenom2",
LocalDate.of(1922,2,15)));

    liste.ajout(new Personne("toto1","prenom1",
LocalDate.of(1911,1,15)));

    liste.ajout(new Personne("toto5","prenom5",
LocalDate.of(1955,5,15)));

    liste.ajout(new Personne("toto3","prenom3",
LocalDate.of(1933,3,15)));

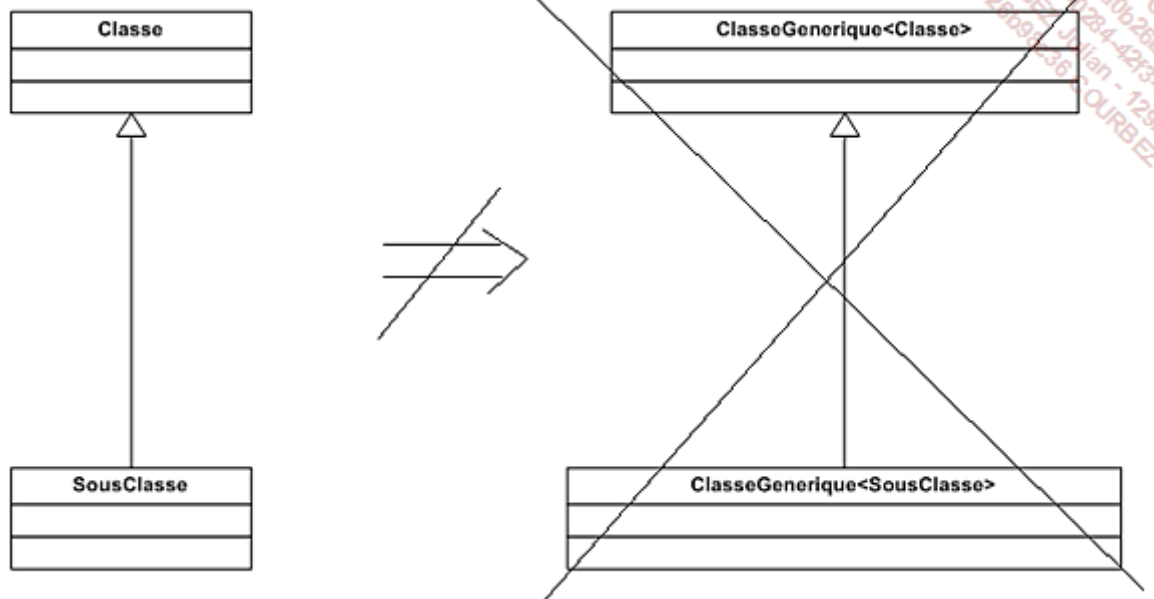
    liste.ajout(new Personne("toto4","prenom4",
LocalDate.of(1944,4,15)));

    Personne p;

    p=(Personne)liste.getElement(0);
}

```

Il faut donc retenir de ces expériences qu'une classe générique construite avec comme type paramètre une classe quelconque n'est pas la sous-classe d'une classe générique construite avec un type qui est le super-type de cette classe quelconque.



Cette limitation peut parfois être gênante. Si nous souhaitons créer une fonction qui accepte comme paramètre une **ListeGenerique** construite à partir de n'importe quel type de données, notre première intuition est d'écrire le code suivant :

```
public static void affichage(ListeGenerique<Object> liste)
{
}

```

Hélas, notre première intuition n'est pas bonne car la fonction `affichage` n'accepte, comme paramètre, avec cette syntaxe, que des instances de `ListeGenerique` d'`Object`.

Pour représenter le fait que la fonction `affichage` accepte comme paramètre une instance de **ListeGenerique** construite à partir de n'importe quel type, nous devons utiliser le caractère joker "?" dans la définition de la fonction `affichage`. Elle prend donc la forme suivante :

```
public static void affichage(ListeGenerique<?> liste)
{
}

```

Avec cette syntaxe nous pouvons utiliser la fonction `affichage` avec comme paramètre, une instance de **ListeGenerique** construite à partir de n'importe quelle classe. Par contre, dans le code de la fonction nous ne pourrions utiliser que les méthodes de la classe **Object** sur les éléments présents dans la liste. Nous pouvons ajouter des restrictions sur le type argument de la fonction en faisant suivre le caractère '?' par une contrainte définie avec exactement la même syntaxe que pour une classe générique. Pour que la fonction `affichage` accepte comme paramètre une **ListeGenerique** de `Personne` et de n'importe laquelle de ses sous-classes, nous devons utiliser la syntaxe suivante :

```
public static void affichage(ListeGenerique<? extends
Personne> liste)
{
    liste.premier();
}

```

```

    for (int i=0;i<liste.getNbElements();i++)
    {
        System.out.println(liste.getElement(i).getNom());
        System.out.println(liste.getElement(i).getPrenom());
        System.out.println("-----");
    }
}

```

À noter que dans ce cas les instances présentes dans la liste seront au moins des instances de la classe `Personne` et pourront être utilisées comme telles en toute sécurité à l'intérieur de la fonction.

Limitation des génériques

L'utilisation des types génériques impose quelques contraintes. Ces limitations sont liées au type argument utilisé et à l'usage qui en est fait.

a - Le type argument est obligatoirement un type par référence. Par exemple, il nous est impossible de créer une instance de **ListeGenerique** de type **int**.

```
ListeGenerique<int> listeEntiers =new ListeGenerique<int>();
```

```

D:\java8\code\tests\src>javac Principale.java
Principale.java:228: error: unexpected type
    ListeGenerique<int> listeEntiers =new ListeGenerique<int>();
                   ^
    required: reference
    found:    int
Principale.java:228: error: unexpected type
    ListeGenerique<int> listeEntiers =new ListeGenerique<int>();
                   ^
    required: reference
    found:    int
2 errors
D:\java8\code\tests\src>_

```

Cette limitation peut être contournée en utilisant à la place les classes **wrapper** correspondant aux types simples. Dans notre cas, l'utilisation de la classe **Integer** résout le problème. Les mécanismes d'autoboxing et d'unboxing rendent transparente l'utilisation des types **wrapper** en lieu et place des types simples.

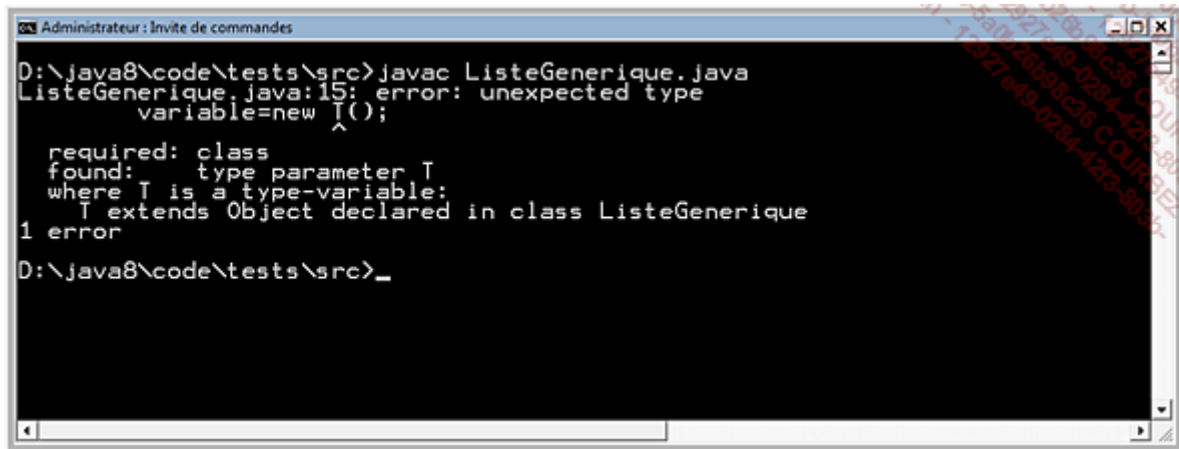
```

ListeGenerique<Integer> listeEntiers = new ListeGenerique<Integer>(5);

listeEntiers.ajout(10);
listeEntiers.ajout(7);
listeEntiers.ajout(19);
listeEntiers.ajout(45);
listeEntiers.ajout(8);

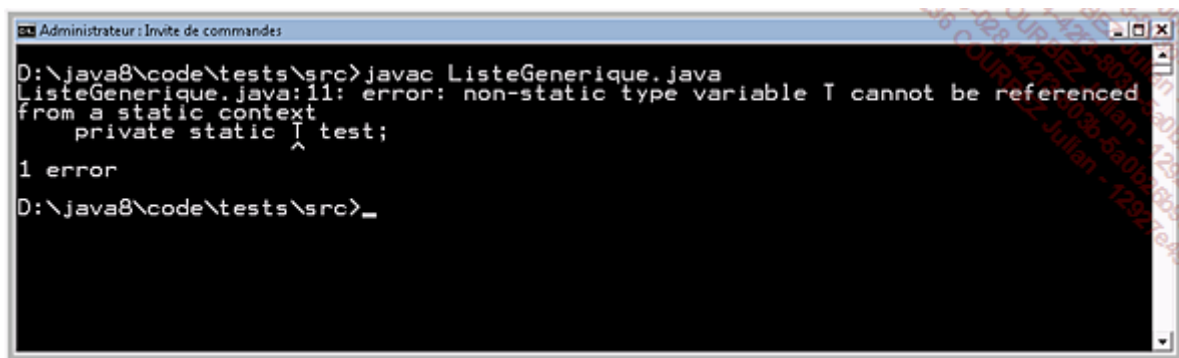
```

b - Le type paramètre ne peut pas être utilisé pour la création d'une instance de classe à l'intérieur d'un type générique. Le code suivant est par exemple interdit à l'intérieur de la classe **ListeGenerique**.



```
Administrateur : Invite de commandes
D:\java8\code\tests\src>javac ListeGenerique.java
ListeGenerique.java:15: error: unexpected type
    variable=new T();
                  ^
required: class
found:     type parameter T
where T is a type-variable:
    T extends Object declared in class ListeGenerique
1 error
D:\java8\code\tests\src>_
```

c - Le type paramètre ne peut pas non plus être utilisé pour déclarer des variables de classe (**static**) à l'intérieur d'un type générique.



```
Administrateur : Invite de commandes
D:\java8\code\tests\src>javac ListeGenerique.java
ListeGenerique.java:11: error: non-static type variable T cannot be referenced
from a static context
    private static T test;
                   ^
1 error
D:\java8\code\tests\src>_
```

La variable `test` étant unique pour toutes les instances de la classe **ListeGenerique**, son type ne peut pas changer en fonction du type argument utilisé pour la création de chacune des instances. Par exemple avec la ligne de code suivante la variable `test` serait de type `String`.

```
ListeGenerique<String> listeChaine=new ListeGenerique<String>(5);
```

Si une autre instance de la classe **ListeGenerique** est ensuite créée avec un type argument différent, il faudrait que la variable de classe `test` change de type, ce qui est bien sûr impossible.

d - L'opérateur **instanceOf** ne peut pas être utilisé avec des types génériques. Le mécanisme d'effacement de type faisant disparaître toute trace du type argument, après la compilation, le test suivant ne pourrait pas fonctionner.

```
public static <T extends Classable> void tri(ListeGenerique<T>
liste) throws Exception
{
    int i,j;
    T c;
    if (liste instanceof ListeGenerique<String>)
```



```
{
    ...
    ...
}
```

```
Administrateur : Invite de commandes
D:\java8\code\tests\src>javac Principale.java
Principale.java:374: error: illegal generic type for instanceof
    if (liste instanceof ListeGenerique<String>)
                                ^
1 error
D:\java8\code\tests\src>_
```

2.9. Les packages

Le but principal des packages est l'organisation et le rangement des classes et interfaces d'une application. Le principe est le même que dans la vie courante. Si vous avez un grand placard dans votre maison, il est évident que l'installation d'étagères dans ce placard va faciliter l'organisation et la recherche des objets qui y sont rangés par rapport à un stockage en "vrac". L'organisation des classes en package apporte les avantages suivants :

- Facilité pour retrouver et réutiliser un élément.
- Limitation des risques de conflits de noms.
- Création d'une nouvelle visibilité en plus des visibilités standards (**private**, **protected**, **public**).

Création d'un package

La première chose à faire lorsque l'on souhaite créer un package est de lui trouver un nom. Ce nom doit être choisi avec précaution pour permettre au package de remplir pleinement son rôle. Il doit être unique et représentatif des éléments stockés à l'intérieur.

Pour assurer l'unicité du nom d'un package, on utilise par convention son nom de domaine en inversant l'ordre des éléments comme première partie pour le nom du package.

Par exemple, si le nom de domaine est java.fr, la première partie du nom du package sera fr.java. Si le nom de domaine comporte des caractères interdits pour les noms de packages, ils sont remplacés par le caractère `_`. Ainsi pour le nom de domaine java-ecole.fr, la première partie du nom de package sera fr.java_ecole. Cette première partie du nom permet d'assurer l'unicité des éléments par rapport à l'extérieur. La suite du nom de package va assurer l'unicité des éléments à l'intérieur de l'application. Il faut bien sûr choisir des noms clairs et représentatifs des éléments du package. Nous pouvons par exemple avoir dans une application deux classes `Client` l'une représentant une personne ayant passé des commandes à notre entreprise, l'autre représentant un client au sens informatique du terme (client - serveur). Les noms de package suivant peuvent être utilisés pour héberger ces deux classes. `fr.java_ecole.compta` pour la classe représentant un client physique et `fr.java_ecole.reseau` pour la classe représentant un client logiciel. Par convention les noms de packages sont entièrement en lettres minuscules. Le nom choisi doit être placé dans le fichier

source précédé du mot clé **package**. La déclaration du package doit être obligatoirement la première ligne du fichier. Tous les éléments placés dans ce fichier source font désormais partie de ce package. Si aucune information n'est indiquée concernant le package dans le fichier source alors les éléments définis dans ce fichier seront considérés comme faisant partie du package par défaut qui ne possède pas de nom. Il convient d'être prudent avec le package par défaut car les classes qui y sont définies ne sont pas accessibles à partir d'un package nommé.

C'est entre autres pour cette raison que les concepteurs de Java recommandent de toujours placer une classe dans un package nommé.

L'utilisation des packages nous impose également une organisation spécifique pour l'enregistrement sur disque des fichiers sources et des fichiers compilés. Les répertoires où sont enregistrés les fichiers doivent respecter les noms utilisés pour les packages. Chaque composant du nom de package doit correspondre à un sous-répertoire. Les fichiers sources et les fichiers compilés peuvent être stockés dans des branches d'arborescence différentes.

Le respect de cette arborescence est impératif. Si le fichier d'une classe appartenant à un package n'est pas placé dans le bon répertoire, la machine virtuelle chargée d'exécuter l'application sera incapable de localiser cette classe. Pour localiser une classe, la machine virtuelle utilise la variable d'environnement **CLASSPATH** pour construire le chemin d'accès au fichier de cette classe. Elle concatène le chemin contenu dans cette variable avec le chemin correspondant au package dans lequel se trouve la classe. Si plusieurs répertoires sont indiqués dans la variable **CLASSPATH**, ils doivent être séparés par des points-virgules pour un environnement Windows et par des virgules pour un environnement Unix. Le répertoire courant fait partie par défaut du chemin de recherche.

Utilisation et importation d'un package

Lorsqu'une classe faisant partie d'un package est utilisée dans une application, le nom complet de la classe (nom du package + nom de la classe) doit être utilisé. Ceci provoque l'écriture de lignes de code relativement longues et difficiles à relire.

```
fr.java_ecole.compta.Client c=new fr.java_ecole.compta.Client
("dupont","paul",LocalDate.of(1953,11,8),'E');
```

Java propose l'instruction `import` pour indiquer au compilateur que certaines classes peuvent être utilisées directement par leur nom sans utilisation du nom de package. Cette instruction doit être placée en début de fichier aussitôt après une éventuelle déclaration de package mais avant la définition de la classe. La ligne de code précédente peut être considérablement abrégée avec la syntaxe suivante :

```
import fr.java_ecole.compta.Client;

...

Client c=new Client("dupont","paul", LocalDate.of(1953,11,8),'E');
```

Par contre, si beaucoup de classes du même package sont utilisées, la liste des importations va devenir volumineuse. Il est plus rationnel dans ce cas d'importer le package complet avec le caractère joker `*`.

Le code devient donc :

```
import fr.java_ecole.compta.*;
```

```
...  
...
```

```
Client c=new Client("dupont","paul", LocalDate.of(1953,11,8),'E');
```

Avec la possibilité d'utiliser toutes les classes du package `fr.java_ecole.compta` simplement en utilisant le nom de la classe. Cette solution peut parfois poser un problème si deux packages contenant des classes ayant des noms identiques sont importés. Il faudra dans ce cas, pour ces classes, utiliser leurs noms complets. Cette solution ne permet pas non plus l'importation de plusieurs packages simultanément. Les apparences étant parfois trompeuses, la structure de packages n'est pas hiérarchique et, malgré ce que pourrait laisser penser la structure des fichiers sur le disque, le package `fr.java_ecole.compta` n'est pas inclus dans le package `fr.java_ecole`. L'importation `import fr.java_ecole.*;` permet uniquement l'accès aux classes de ce package et absolument pas aux classes du package `fr.java_ecole.compta`. Si c'était le cas, on peut imaginer facilement les problèmes posés par les importations suivantes :

```
import com.*;  
import fr.*;
```

L'importation peut également simplifier l'écriture du code lors de l'utilisation de classes contenant de nombreuses méthodes **static**. Vous devez pour chaque appel d'une de ces méthodes la préfixer par le nom de la classe. Ceci a pour conséquence de réduire la lisibilité du code.

```
public class Distance  
{  
  
    public static double dist(double lat1,double lon1,double lat2,double lon2)  
    {  
  
        double deltaLat;  
        double abscurv;  
        deltaLat =lon2-lon1;  
        abscurv=Math.acos( (Math.sin(lat1)*Math.sin(lat2))+(Math.cos(lat1)  
*Math.cos(lat2)*Math.cos(deltaLat)));  
        return abscurv * 6371598;  
    }  
}
```

Il est possible de simplifier ce code en effectuant une importation **static** de la classe `Math` permettant l'utilisation de ces membres **static** sans les préfixer par le nom de la classe.

```
import static java.lang.Math.*;  
public class Distance  
{  
  
    public static double dist(double lat1,double lon1,double lat2,double lon2)  
    {
```

```

double deltaLat;

double abscurv;

deltaLat =lon2-lon1;

abscurv=acos((sin(lat1)*sin(lat2))+(cos(lat1)*cos(lat2)*cos(
deltaLat)));

return abscurv; /*6371598;

}

}

```

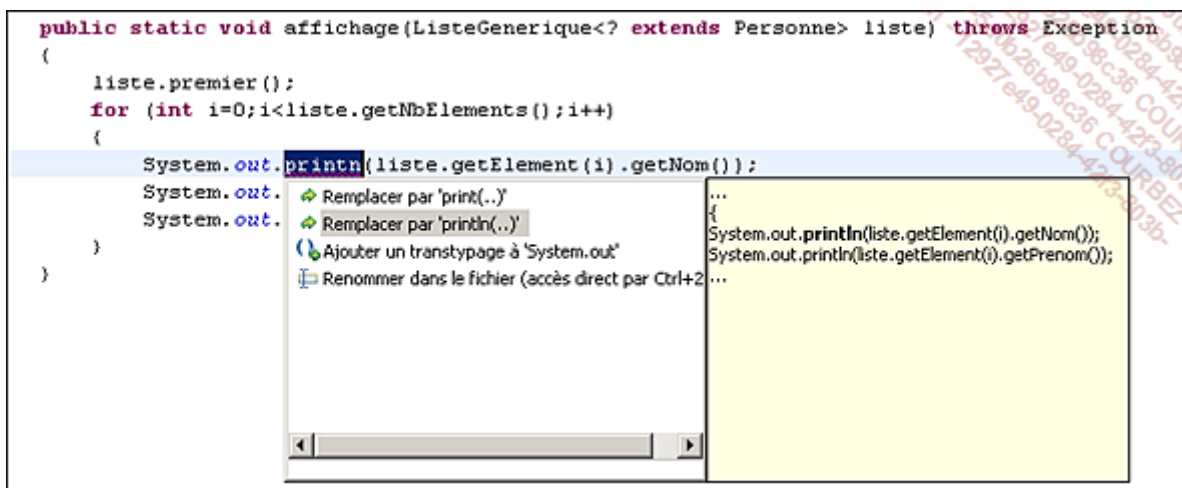
3. Gestion des exceptions

3.1. Les erreurs de syntaxe

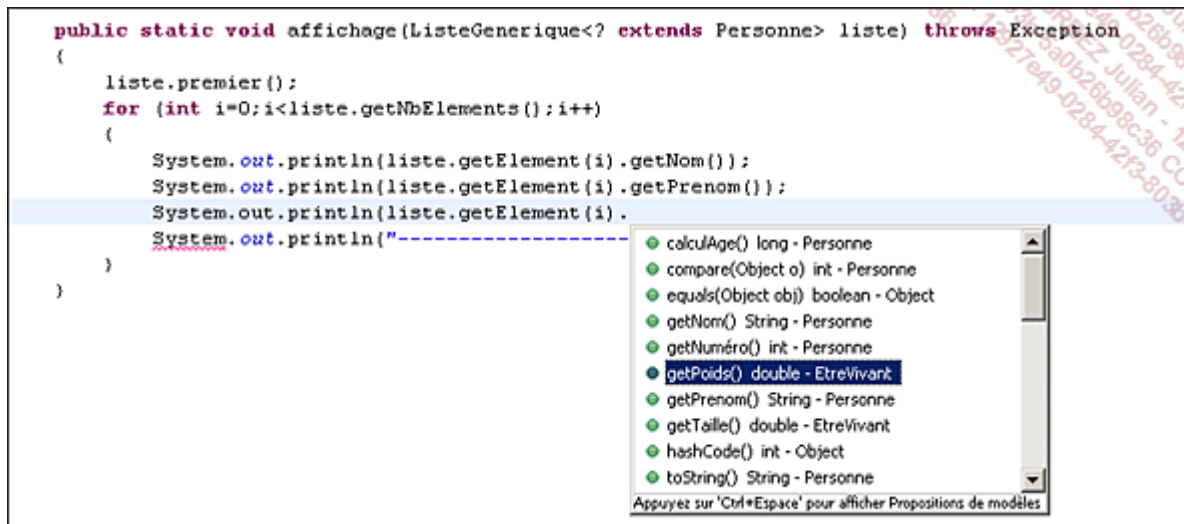
Ce type d'erreur se produit au moment de la compilation lorsqu'un mot clé du langage est mal orthographié. Très fréquentes avec les outils de développement où l'éditeur de code et le compilateur sont deux entités séparées, elles deviennent de plus en plus rares avec les environnements de développement intégré (Eclipse, NetBeans, Jbuilder...). La plupart de ces environnements proposent une analyse syntaxique au fur et à mesure de la saisie du code.

Les exemples suivants sont obtenus à partir de l'environnement Eclipse.

Si une erreur de syntaxe est détectée, alors l'environnement propose des solutions possibles pour corriger cette erreur.

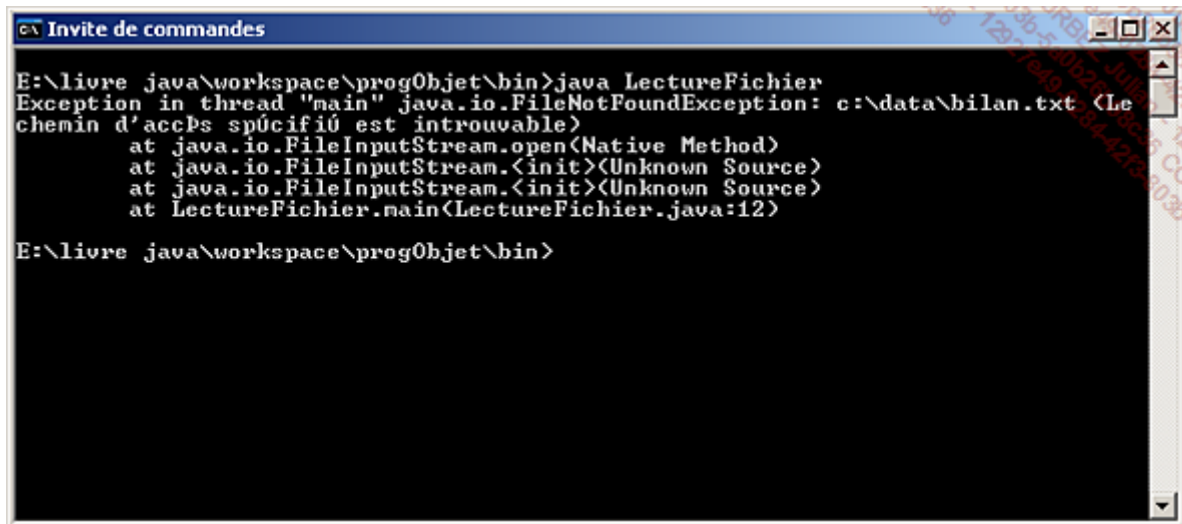


D'autre part, les "fautes d'orthographe" dans les noms de champs ou de méthodes sont facilement éliminées grâce aux fonctionnalités disponibles dans ces environnements.



3.2. Les erreurs d'exécution

Ces erreurs apparaissent après la compilation lorsque vous lancez l'exécution de votre application. La syntaxe du code est correcte mais l'environnement de votre application ne permet pas l'exécution d'une instruction utilisée dans votre application. C'est par exemple le cas si vous essayez d'ouvrir un fichier qui n'existe pas sur le disque de votre machine. Vous obtiendrez sûrement un message de ce type.



Ce type de message n'est pas très sympathique pour l'utilisateur !

Heureusement Java permet la récupération de ce type d'erreur et évite ainsi l'affichage de cet inquiétant message.

3.3. Les erreurs de logique

Les pires ennemis des développeurs. Tout se compile sans problème, tout s'exécute sans erreurs et pourtant "ça ne marche pas comme prévu" !!!

Il faut dans ce cas revoir la logique de fonctionnement de l'application. Les outils de débogage nous permettent de suivre le déroulement de l'application, de placer des points d'arrêt, de visualiser le contenu des variables, etc.

Ces outils ne remplacent cependant pas une bonne dose de réflexion.

Les exceptions

Lorsqu'une erreur se produit au cours de l'exécution d'une méthode, un objet **Exception** est créé pour représenter l'erreur qui vient de se produire. Cet objet contient de nombreuses informations concernant l'erreur qui est survenue dans l'application ainsi que l'état de l'application au moment de l'apparition de l'erreur. Cet objet est ensuite transmis à la machine virtuelle. C'est le déclenchement de l'exception. La machine virtuelle doit alors rechercher une solution pour la gérer. Pour cela, elle explore les différentes méthodes ayant été appelées pour atteindre l'emplacement où l'erreur s'est produite. Dans ces différentes méthodes, elle recherche un gestionnaire d'exceptions capable de traiter le problème. La recherche débute par la méthode où l'exception a été déclenchée puis remonte jusqu'à la méthode main de l'application si besoin. Lorsqu'un gestionnaire d'exception adapté est trouvé, l'objet **Exception** lui est transmis pour qu'il en assure le traitement. Si la recherche est infructueuse, l'application s'arrête.

Les exceptions sont en général classées en trois catégories :

- Les exceptions vérifiées correspondent à une situation anormale au cours du fonctionnement de l'application. Cette situation est en général liée à un élément extérieur à l'application, comme par exemple une connexion vers une base de données ou une lecture de fichier. Ces exceptions sont représentées par des instances de la classe **Exception** ou de l'une de ses sous-classes. Elles doivent obligatoirement être traitées par un bloc **try catch** ou propagées au code appelant avec le mot clé **throws** placé dans la signature de la fonction.
- Les erreurs correspondent à des conditions exceptionnelles extérieures à l'application que celle-ci ne peut généralement pas prévoir. Ces exceptions sont représentées par une instance de la classe **Error** ou de l'une de ses sous-classes. Ces exceptions ne sont pas obligatoirement traitées. Il faut d'ailleurs préciser que lorsqu'elles surviennent, le fonctionnement de l'application est généralement très fortement compromis. Si elles sont gérées, la plupart du temps le seul traitement consiste à afficher un message un peu moins effrayant pour l'utilisateur que celui proposé par défaut par la machine virtuelle Java. Dans pratiquement tous les cas l'arrêt de l'application est inéluctable.
- Les erreurs liées à une mauvaise utilisation d'une fonctionnalité du langage, ou à une erreur de logique dans la conception de l'application. L'erreur la plus fréquente que vous rencontrerez dans vos débuts avec Java sera certainement l'exception **NullPointerException** déclenchée lors de l'utilisation d'une variable non initialisée. Ces exceptions sont représentées par une instance de la classe **RuntimeException**. Bien que ces exceptions puissent être traitées par des blocs **try catch**, il est fortement déconseillé de le faire. Il est préférable d'analyser le code et de le modifier pour éviter qu'elles n'apparaissent.

Récupération d'exceptions

La gestion des exceptions donne la possibilité de protéger un bloc de code contre les exceptions qui pourraient s'y produire. Le code "dangereux" doit être placé dans un bloc **try**. Si une exception est déclenchée dans ce bloc de code, le ou les blocs de code **catch** sont examinés. S'il en existe un capable de traiter l'exception, le code correspondant est exécuté, sinon la même exception est déclenchée pour éventuellement être récupérée par un bloc **try** de plus haut niveau. Une

instruction **finally** permet de marquer un groupe d'instructions qui seront exécutées à la sortie du bloc **try** si aucune exception ne s'est produite ou à la sortie d'un bloc **catch** si une exception a été déclenchée. La syntaxe générale est donc la suivante :

```
try
{
    ...
    Instructions dangereuses
    ...
}
catch (exception1 e1)
{
    ...
    code exécuté si une exception de type Exception1 se produit
    ...
}
catch (exception2 e2)
{
    ...
    code exécuté si une exception de type Exception1 se produit
    ...
}
finally
{
    ...
    code exécuté dans tous les cas avant la sortie du bloc try ou
    d'un bloc catch
    ...
}
```

Cette structure a un fonctionnement très semblable au **switch case**.

Vous devez indiquer pour chaque bloc **catch** le type d'exception qu'il doit gérer.

```
public void lireFichier(String nom)
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
```

```

try
{
    fichier=new FileInputStream("c:\\data\\bilan.txt");
}
catch (FileNotFoundException e)
{
    e.printStackTrace();
}
br=new BufferedReader(new InputStreamReader(fichier));
try
{
    ligne=br.readLine();
}
catch (IOException e)
{
    e.printStackTrace();
}
while (ligne!=null)
{
    System.out.println(ligne);
    try
    {
        ligne=br.readLine();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
}

```

Dans l'exemple précédent, chaque instruction susceptible de déclencher une exception est protégée par son propre bloc **try**. Cette solution présente l'avantage d'être extrêmement précise pour la gestion des exceptions au détriment de la lisibilité du code. Une solution plus simple consiste à regrouper plusieurs instructions dans un même bloc **try**. Notre exemple peut également être codé de la façon suivante :

```

public void lireFichier(String nom)

```



```

{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    try
    {
        fichier=new FileInputStream(nom);
        br=new BufferedReader(new InputStreamReader(fichier));
        ligne=br.readLine();
        while (ligne!=null)
        {
            System.out.println(ligne);
            ligne=br.readLine();
        }
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

```

Le code est beaucoup plus lisible, par contre nous perdons en précision car il devient difficile de déterminer quelle instruction a déclenché l'exception.

Il faut également être vigilant sur l'ordre des blocs **catch** et toujours les organiser du plus précis au plus général. Les exceptions étant des classes, elles peuvent avoir des relations d'héritage. Si un bloc **catch** est prévu pour gérer un type particulier d'exception, il est également capable de gérer toutes les types d'exceptions qui héritent de celle-ci. C'est le cas dans notre exemple puisque la classe **FileNotFoundException** hérite de la classe **IOException**. Le compilateur détecte une telle situation et génère une erreur. Si nous modifions notre code de la façon suivante :

```

public void lireFichier(String nom)
{
    FileInputStream fichier=null;

```

```

BufferedReader br=null;
String ligne=null;
try
{
    fichier=new FileInputStream(nom);
    br=new BufferedReader(new InputStreamReader(fichier));
    ligne=br.readLine();
    while (ligne!=null)
    {
        System.out.println(ligne);
        ligne=br.readLine();
    }
}
catch (IOException e)
{
    e.printStackTrace();
}
catch (FileNotFoundException e)
{
    e.printStackTrace();
}
}

```

Nous obtenons cette erreur au moment de la compilation.

Le code peut être encore plus concis en indiquant qu'un même bloc **catch** doit gérer plusieurs types d'exceptions. Les différents types d'exceptions qu'un bloc **catch** peut traiter doivent être indiqués dans la déclaration en les séparant par le caractère |.

```

public static void lireFichier(String nom)
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    double somme=0;

    try
    {

```

```

        fichier=new FileInputStream(nom);
        br=new BufferedReader(new InputStreamReader(fichier));
        ligne=br.readLine();

        while (ligne!=null)
        {
            System.out.println(ligne);
            ligne=br.readLine();
            somme=somme+Double.parseDouble(ligne);
        }
        System.out.println("total : " + somme);
    }

    catch (IOException | NumberFormatException e)
    {
        e.printStackTrace();
    }
}

```

Exceptions associées à des ressources

De nombreuses applications ont fréquemment besoin d'accéder à des ressources externes. Les fichiers et les bases de données sont certainement les exemples les plus courants. L'utilisation de ces ressources débute par une opération d'ouverture, l'exploitation de la ressource, puis la fermeture de la ressource. Fréquemment, les méthodes permettant d'exploiter ces ressources sont susceptibles de déclencher de nombreuses exceptions et de ce fait sont placées dans une structure **try-catch**. Celle-ci peut aussi se voir confier la fermeture de la ressource à la fin de l'exécution des instructions qu'elle contient. La ou les ressources doivent être déclarées et instanciées entre parenthèses après le mot clé **try**. Si le bloc **try** contient plusieurs déclarations, celles-ci doivent être séparées par un point-virgule.

À la fin de l'exécution du bloc **try** la méthode **close** est appelée sur toutes les ressources déclarées au niveau du mot clé **try**. Cet appel est toujours effectué avant l'exécution d'un bloc **catch** ou du bloc **finally**. Pour que ce mécanisme fonctionne, les classes correspondant aux ressources utilisées doivent implémenter l'interface **Closeable** ou **AutoCloseable**. Ces deux interfaces exigent l'existence de la méthode **close** dans la classe de la ressource utilisée dans le bloc **try**.

Dans l'exemple ci-dessous, l'objet **BufferedReader** est automatiquement fermé après l'exécution du bloc **try**.

```

String reponse="";

    try (BufferedReader br=new BufferedReader(new
InputStreamReader(System.in)))
    {

```

```

        while (!reponse.equals("fin"))
        {
            ...
            ...
            reponse=br.readLine();
        }
    }
    catch(IOException e )
    {
        e.printStackTrace();
    }
}

```

Le code de chaque bloc catch peut obtenir plus d'informations sur l'exception qu'il doit traiter en utilisant les méthodes disponibles dans la classe correspondant à l'exception. Les méthodes suivantes sont les plus utiles pour obtenir des informations complémentaires sur l'exception.

- **getMessage** : permet d'obtenir le message d'erreur associé à l'exception.
- **getCause** : permet d'obtenir l'exception initiale dans le cas où le chaînage d'exception est utilisé.
- **getStackTrace** : permet d'obtenir un tableau de **StackTraceElement** dont chaque élément représente une méthode appelée jusqu'à celle où est traitée l'exception. Pour chacune d'elle nous pouvons obtenir les informations suivantes :
 - ✓ Le nom de la classe où se trouve la méthode : **getClassName**.
 - ✓ Le nom du fichier où se trouve cette classe : **getFilename**.
 - ✓ Le numéro de la ligne où l'exception a été déclenchée : **getLineNumber**.
 - ✓ Le nom de la méthode : **getMethodName**.

Ces informations peuvent être utilisées pour générer des fichiers historiques du fonctionnement de l'application. Voici un exemple d'enregistrement de ces informations dans un fichier texte :

```

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.GregorianCalendar;

public class LectureFichier
{

```

```

public static void main(String args[])
{
    try {
        lireFichier("bilan.txt");
    }
    catch (CaMarchePasException e)
    {
        FileWriter log;
        BufferedWriter br;
        try
        {
            log=new FileWriter("historique.txt",true);
            br=new BufferedWriter(log);
            br.write("----->" + new GregorianCalendar().
getTime()+" <-----\r\n");
            br.write("erreur : " + e.getMessage()+"\r\n");
            for (int i=0;i<e.getStackTrace().length;i++)
            {
                br.write("dans le fichier "
+e.getStackTrace()[i].getFileName());
                br.write(" à la ligne "
+e.getStackTrace()[i].getLineNumber());
                br.write(" dans la méthode "
+e.getStackTrace()[i].getMethodName());
                br.write(" de la classe "
+ e.getStackTrace()[i].getClassName()+ "\r\n");
            }
            br.close();
            log.close();
        }
        catch (IOException ex)
        {
            System.out.println("erreur dans l'application");
        }
    }
}

```

```

    }
}

    public static void lireFichier(String nom) throws
CaMarchePasException
    {
        FileInputStream fichier=null;
        BufferedReader br=null;
        String ligne=null;
        try
        {
            fichier=new FileInputStream(nom);
            br=new BufferedReader(new InputStreamReader(fichier));
            ligne=br.readLine();
            while (ligne!=null)
            {
                System.out.println(ligne);
                ligne=br.readLine();
            }
        }
        catch (FileNotFoundException e)
        {
            throw new CaMarchePasException("le fichier n'existe
pas",e);
        }
        catch (IOException e)
        {
            throw new CaMarchePasException("erreur de lecture du
fichier",e);
        }
    }
}

```

Création et déclenchement d'exceptions

Les exceptions sont avant tout des classes, il est donc possible de créer nos propres exceptions en héritant d'une des nombreuses classes d'exception déjà disponibles. Pour respecter les conventions, il est conseillé de terminer le nom de la classe par le terme `Exception`. Nous pouvons par exemple écrire le code suivant :

```
public class CaMarchePasException extends Exception
{
    public CaMarchePasException()
    {
        super();
    }
    public CaMarchePasException(String message)
    {
        super(message);
    }
    public CaMarchePasException(String message, Throwable cause)
    {
        super(message, cause);
    }
    public CaMarchePasException(Throwable cause)
    {
        super(cause);
    }
}
```

La surcharge des constructeurs de la classe de base est fortement conseillée pour conserver la cohérence entre les classes d'exception.

Cette classe peut ensuite être utilisée pour le déclenchement d'une exception personnalisée. Pour déclencher une exception, il faut au préalable créer une instance de la classe correspondante puis déclencher l'exception avec le mot clé **throw**. Le déclenchement d'une exception dans une fonction avec le mot clé **throw** provoque la sortie immédiate de la fonction. Le code suivant déclenche une exception personnalisée dans les blocs **catch**.

```
public static void lireFichier2(String nom) throws CaMarchePasException
{
    FileInputStream fichier=null;
    BufferedReader br=null;
    String ligne=null;
    try
```

```

{
    fichier=new FileInputStream(nom);
    br=new BufferedReader(new InputStreamReader(fichier));
    ligne=br.readLine();
    while (ligne!=null)
    {
        System.out.println(ligne);
        ligne=br.readLine();
    }
}
catch (FileNotFoundException e)
{
    throw new CaMarchePasException("le fichier n'existe pas",e);
}
catch (IOException e)
{
    throw new CaMarchePasException("erreur de lecture
du fichier",e);
}
}

```

Lorsqu'une fonction est susceptible de déclencher une exception, vous devez le signaler dans la signature de cette fonction avec le mot clé **throws** suivi de la liste des exceptions qu'elle peut déclencher. Lorsque cette fonction sera ensuite utilisée dans une autre fonction, vous devrez obligatoirement tenir compte de cette, ou de ces éventuelles exceptions. Vous devrez donc soit gérer l'exception avec un bloc **try ... catch** soit la propager en ajoutant le mot clé **throws** à la déclaration de la fonction. Il faut cependant être prudent et ne pas propager les exceptions au-delà de la méthode **main** car dans ce cas, c'est la machine virtuelle Java qui les récupère et arrête brutalement l'application.

4. Les collections

Les applications ont très fréquemment besoin de manipuler de grandes quantités d'informations. De nombreuses structures sont disponibles pour faciliter la gestion de ces informations. Elles sont regroupées sous le terme collection.

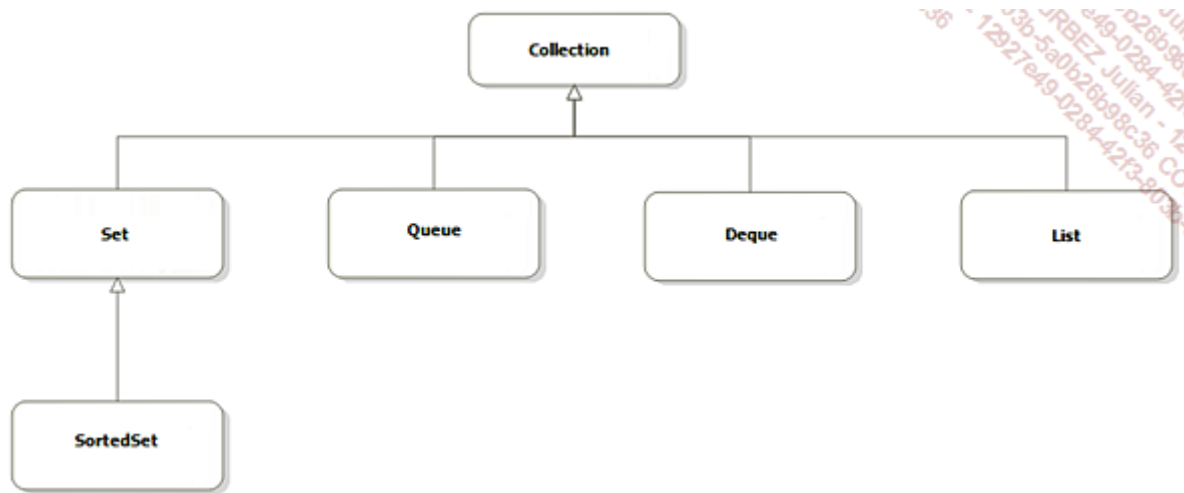
Le langage Java propose différentes classes permettant de mettre en place plusieurs modes de gestion.

La première solution pour gérer un ensemble d'éléments est l'utilisation de tableaux. Bien que simple à mettre en œuvre, cette solution n'est pas très souple. Son principal défaut tient au caractère fixe de la taille d'un tableau. S'il n'y a plus de place pour stocker des éléments

supplémentaires, il faut créer un nouveau tableau plus grand et y transférer le contenu du tableau précédent. Cette solution, lourde à mettre en œuvre, est consommatrice de ressources.

Le langage Java propose une vaste palette d'interfaces et de classes pour gérer facilement des ensembles d'éléments. Les interfaces décrivent les fonctionnalités disponibles alors que les classes implémentent et fournissent réellement ces fonctionnalités. Suivant le mode de gestion des éléments souhaité, on utilisera bien sûr la classe la plus adaptée. Cependant, les mêmes fonctionnalités de base doivent être accessibles quel que soit le mode de gestion. Pour assurer la présence de ces fonctionnalités indispensables dans toutes les classes, celles-ci ont été regroupées dans l'interface **Collection** qui est par la suite utilisée comme interface de base.

La hiérarchie des interfaces est représentée sur le diagramme ci-dessous.



Toutes ces interfaces sont génériques afin de gérer des ensembles composés de n'importe quels types d'éléments tout en évitant les fastidieuses opérations de transtypage.

Pour chacune de ces interfaces, une ou plusieurs classes correspondantes sont disponibles. Ce sont bien sûr celles-ci qui vont nous intéresser.

4.1. La classe **ArrayList**

Cette classe est une implémentation de l'interface **List**. Elle permet la gestion des éléments que l'on y place de manière quasi similaire à celle disponible avec un tableau, avec en plus l'aspect dynamique. Après la création de l'instance de la classe **ArrayList**, les éléments peuvent y être ajoutés grâce aux méthodes **add** et **addAll**. Par défaut, ces deux méthodes ajoutent les éléments à la suite de ceux déjà présents. Une version surchargée de ces deux méthodes permet de choisir l'emplacement où les éléments sont placés. Pour cette version, il faut spécifier la position où doit avoir lieu l'ajout.

La première position est à l'index 0. Si des éléments sont déjà présents, ils sont simplement décalés.

L'accès aux éléments de la liste se fait via la méthode **get** à laquelle on indique simplement l'index de l'élément. À l'inverse, la méthode **set** permet de placer un élément à la position indiquée. Il n'y a pas dans ce cas d'insertion, mais remplacement de l'élément se trouvant déjà à cette position.

La suppression d'un élément est effectuée par la méthode **remove** à laquelle on indique soit l'index, soit l'élément que l'on souhaite supprimer. Dans ces deux cas, les éléments suivants sont décalés d'un cran (il n'y a jamais de "trou" dans une **ArrayList**). Les trois fonctions **indexOf**, **lastIndexOf** et **contains** effectuent la recherche d'un élément dans l'**ArrayList**.

Les deux premières retournent l'index où l'élément a été trouvé ou -1 si aucun élément n'est trouvé. La fonction **indexOf** commence la recherche par le premier élément de la liste, la fonction **lastIndexOf** débute la recherche par la fin de la liste. La fonction **contains** indique simplement si l'élément est présent dans la liste, sans indication sur sa position s'il existe.

Le parcours des éléments de la liste est possible avec une boucle **for** ou en utilisant l'objet **Iterator** fourni par la méthode **iterator** de la classe **ArrayList**. Avec ces deux solutions, le parcours se fait uniquement du premier vers le dernier élément de la liste, de plus le contenu de la liste ne doit pas être modifié pendant le parcours. Un objet **ListIterator**, retourné par la méthode **listIterator**, permet plus de souplesse puisqu'il autorise les déplacements dans la liste dans les deux sens et accepte également la modification de la liste pendant le parcours.

L'exemple de code ci-dessous illustre ces différentes fonctionnalités :

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;

public class testArrayList
{
    public static void main(String[] args)
    {
        ArrayList<Personne> liste1;
        ArrayList<Personne> liste2;
        // création des deux instances
        liste1=new ArrayList<Personne>();
        liste2=new ArrayList<Personne>();

        // création des personnes pour remplir la liste
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

        // ajout de quatre personnes à la liste
        liste1.add(p1);
```

```

liste1.add(p3);
liste1.add(p4);
liste1.add(p5);

// insertion d'une personne entre p1 et p3
// donc à la position 1 de la liste
liste1.add(1, p2);

// ajout du contenu d'une liste à une autre liste
// les deux listes contiennent maintenant les mêmes objets.
// !!!! ne pas confondre avec liste2=liste1; !!!
liste2.addAll(liste1);

// affichage du nombre d'éléments de la liste
System.out.println("il y a " + liste1.size() + " personne(s)
dans la liste");
// parcours de la première liste du début vers la fin
Iterator<Personne> it;
it=liste1.iterator();
Personne p;
// tant qu'il reste des éléments

while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom());
}

// parcours de la première liste de la fin vers le début
// récupération d'un ListIterator positionné après
// le dernier élément (le nombre d'éléments de la liste)

ListIterator<Personne> lit;

```

```

        lit=listel.listIterator(listel.size());
        // tant qu'il reste des éléments
        while (lit.hasPrevious())
        {
            // récupération de l'élément courant
            // en remontant dans la liste
            p=lit.previous();
            System.out.println(p.getNom());
        }

        // remplacement d'un élément de liste
        listel.set(2,new Personne("Grant",
"Cary",LocalDate.of(1904,1,18)));

        // affichage de l'élément à la troisième position de la liste
        System.out.println(listel.get(2).getNom());

        // recherche d'un élément dans la liste
        int position;
        position=listel.indexOf(p4);
        if(position==-1)
            System.out.println("non trouve dans la liste");
        else
            System.out.println(listel.get(position).getNom());

        // recherche d'un élément inexistant dans la liste.
        // John Lennon a été remplacé par Cary Grant
        // La recherche débute à la fin de la liste

        position=listel.lastIndexOf(p3);
        if(position==-1)
            System.out.println("non trouve dans la liste");
        else
            System.out.println(listel.get(position).getNom());

```

```

// suppression sélective de la liste
// l'expression lambda determine quels éléments seront supprimés
listel.removeIf((Personne pe) ->pe.getDate_nais().getYear()<1940);
// parcours de la liste pour voir le résultat
it=listel.iterator();
// tant qu'il reste des éléments
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom()+ " ne en " +
p.getDate_nais().getYear() );
}
// une autre manière de parcourir la liste
// l'expression lambda est exécutée pour chaque
// élément de la liste

listel.forEach((Personne per)->System.out.println(per.getNom()+ "
age " + per.calculAge()));

}
}

```

4.2. La classe HashSet

Cette classe est une implémentation de l'interface **Set**. Son fonctionnement est pratiquement similaire à celui de la classe **ArrayList**. La principale différence est liée à la gestion des doublons.

Contrairement à une **ArrayList**, un **HashSet** n'accepte pas le stockage de deux éléments identiques. Lorsqu'un élément est ajouté à un **HashSet** avec la méthode **add**, celle-ci vérifie s'il n'y a pas déjà un élément identique en comparant le **hashCode** de l'élément avec ceux des éléments déjà présents dans la liste. Si l'ajout est effectué, la méthode **add** retourne un booléen **true** et bien sûr **false** dans le cas inverse. Ceci nous impose donc de redéfinir la méthode **hashCode** de toutes les classes pour lesquelles nous envisageons de stocker des instances dans un **HashSet**.

Cette méthode doit respecter une règle bien précise : si l'on considère que deux objets sont identiques, la méthode **hashCode** doit renvoyer la même valeur pour chacun d'eux.

Pour conserver une cohérence, il est également indispensable de redéfinir la méthode **equals** de la classe avec les mêmes critères d'égalité.

Pour pouvoir stocker dans un **HashSet** des instances de la classe `Personne`, nous devons donc y ajouter ces deux méthodes.

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Personne implements Classable

{
    private String nom;
    private String prenom;
    private LocalDate date_nais=LocalDate.of(1963,11,29);

    public Personne()
    {
    }

    public Personne(String n,String p,LocalDate d)
    {
        this.nom=n;
        this.prenom=p;
        this.date_nais=d;
    }

    public String getNom()
    {
        return nom;
    }

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

    public String getPrenom()
    {
```

```

        return prenom;
    }

    public void setPrenom(String prenom)
    {
        this.prenom = prenom;
    }

    public LocalDate getDate_nais()
    {
        return date_nais;
    }

    public void setDate_nais(LocalDate date_nais)
    {
        this.date_nais = date_nais;
    }

    @Override
    public int hashCode()
    {
        // On choisit les deux nombres impairs
        int resultat = 7;
        final int multiplier = 17;

        // Pour chaque attribut, on calcule le hashcode
        // que l'on ajoute au résultat après l'avoir multiplié
        // par le nombre "multiplieur" :
        resultat = multiplier*resultat + (nom==null ? 0 :
nom.hashCode());

        resultat = multiplier*resultat + (prenom==null ? 0 :
prenom.hashCode());

        resultat = multiplier*resultat + (date_nais==null ? 0 :
date_nais.hashCode());
    }

```

```

        return resultat;
    }
@Override
    public boolean equals(Object obj)
    {
        // si le deuxième objet est null il ne
        // peut pas y avoir égalité
        if (obj == null)
        {
            return false;
        }
        // si les deux objets ne sont pas de même type
        // il ne peut pas y avoir égalité
        if (getClass() != obj.getClass())
        {
            return false;
        }
        // il faut maintenant vérifier l'égalité de chacun
        // des attributs
        Personne p = (Personne) obj;
        if (!nom.equals(p.getNom()))
            return false;
        if (!prenom.equals(p.getPrenom()))
            return false;
        if (!date_nais.equals(getDate_nais()))
            return false;
        return true;
    }

    public long calculAge()
    {
        return date_nais.until(LocalDate.now(), ChronoUnit.YEARS);
    }

```



```

public int compare(Object o)
{
    Personne p;
    if (o instanceof Personne)
    {
        p=(Personne)o;
    }
    else
    {
        return Classable.ERREUR;
    }
    if (getNom().compareTo(p.getNom())<0)
    {
        return Classable.INFERIEUR;
    }
    if (getNom().compareTo(p.getNom())>0)
    {
        return Classable.SUPERIEUR;
    }

    return Classable.EGAL;
}
}

```

Vérifions maintenant nos modifications de la classe Personne avec le code suivant :

```

public static void main(String[] args)
{
    Personne p1,p2,p3;

    p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
    p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
    p3 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));

    System.out.println("hashCode de p1 : " + p1.hashCode());
    System.out.println("hashCode de p2 : " + p2.hashCode());
}

```

```

        System.out.println("hashCode de p3 : " + p3.hashCode());

        if (p1.equals(p2))
            System.out.println("p1 et p2 sont identiques");
        else
            System.out.println("p1 et p2 sont differents");

        if (p1.equals(p3))
            System.out.println("p1 et p3 sont identiques");
        else
            System.out.println("p1 et p3 sont differents");

    }

```

Nous obtenons le résultat suivant :

```

hashCode de p1 : -1636676846
hashCode de p2 : 633943827
hashCode de p3 : -1636676846
p1 et p2 sont differents
p1 et p3 sont identiques

```

ce qui valide nos modifications.

Nous pouvons maintenant tester les fonctionnalités de la classe **HashSet**. Elles sont pratiquement identiques à celles de la classe **ArrayList** à un petit détail près. En effet, dans un **HashSet** il n'est pas possible d'accéder à un élément particulier puisqu'il n'y a pas de notion d'index. La seule solution est d'utiliser l'objet **iterator** associé au **HashSet**.

```

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.ListIterator;

public class TestHashSet1
{
    public static void main(String[] args)
    {
        HashSet<Personne> hash1;
        HashSet<Personne> hash2;
    }
}

```

```

// création des deux instances
hash1=new HashSet<Personne>();
hash2=new HashSet<Personne>();

// création des personnes pour remplir le HashSet
Personne p1,p2,p3,p4,p5;
p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

// ajout de quatre personnes au HashSet
hash1.add(p1);
hash1.add(p3);
hash1.add(p4);
hash1.add(p5);

// ajout du contenu d'un HashSet à un autre HashSet
// les deux HashSet contiennent maintenant les mêmes
// objets.
// !!!! ne pas confondre avec hash2=hash1; !!!
hash2.addAll(hash1);

// affichage du nombre d'éléments du HashSet
System.out.println("il y a " + hash1.size() + " personne(s)
dans le HashSet");

// parcours du premier HashSet du début vers la fin
Iterator<Personne> it;
it=hash1.iterator();

// tant qu'il reste des éléments dans le HashSet
Personne p;

```

```

while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom());
}
// suppression sélective dans le HashSet
// l'expression détermine quels éléments seront supprimés
hash1.removeIf((Personne pe)->pe.getDate_nais().getYear()<1940);
// parcours du HashSet pour voir le résultat
it=hash1.iterator();
// tant qu'il reste des éléments dans le HashSet
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom()+ " ne en " +
p.getDate_nais().getYear() );
}
// une autre manière de parcourir le HashSet
// l'expression lambda est exécutée pour chaque
// élément du HashSet

hash1.forEach((Personne per)->System.out.println(per.getNom()+ "
age " + per.calculAge()));
}
}

```

Malgré cette petite restriction, les **HashSet** sont particulièrement efficaces pour réaliser des opérations ensemblistes, comme les unions et les intersections. Avant de tester ces opérations avec des **HashSet**, regardons leurs définitions théoriques.

Supposons que nous avons à notre disposition deux ensembles contenant des *Personnes*. L'un contient des chanteurs, l'autre contient des acteurs.

- Notion de sous-ensemble : on considère qu'un ensemble est un sous-ensemble d'un autre si tous ses éléments sont contenus dans celui-ci. Par exemple, on peut dire que chanteurs est un sous-ensemble d'acteurs si tous les chanteurs sont également acteurs.

- Notion d'union : l'union de deux ensembles est l'ensemble constitué par tous les éléments contenus dans chacun des deux ensembles. On peut par exemple définir un ensemble nommé artistes qui est l'union des acteurs et des chanteurs.
- Notion d'intersection : l'intersection de deux ensembles est constituée par les éléments contenus à la fois dans le premier et le deuxième ensemble. Dans notre cas, il contient les personnes étant à la fois acteurs et chanteurs.
- Notion de différence : la différence entre deux ensembles est constituée par tous les éléments présents dans l'un mais pas dans l'autre. Dans notre cas, ils représentent les personnes qui sont uniquement chanteurs et celles qui sont uniquement acteurs.

Quatre fonctions de la classe **HashSet** permettent de traduire très facilement ces notions. Il s'agit de la fonction **containsAll** pour la notion de sous-ensemble, de la fonction **addAll** pour l'union, de la fonction **retainAll** pour l'intersection et de la fonction **removeAll** pour la différence.

Ces fonctions modifient le contenu du **HashSet** sur lequel elles sont appelées, il est donc prudent d'en faire une copie et de travailler sur cette copie.

L'exemple de code suivant illustre ces différentes notions :

```
import java.time.LocalDate;
import java.util.HashSet;
import java.util.Iterator;

public class TestHashSet
{
    public static void main(String[] args)
    {
        HashSet<Personne> acteurs;
        HashSet<Personne> chanteurs;
        acteurs=new HashSet<Personne>();
        chanteurs=new HashSet<Personne>();

        // création des personnes pour remplir la liste
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

        acteurs.add(p1);
```

```

acteurs.add(p2);
acteurs.add(p4);
acteurs.add(p5);

chanteurs.add(p1);
chanteurs.add(p3);

// test si les chanteurs sont également acteurs
if (acteurs.containsAll(chanteurs))
    System.out.println("tous les chanteurs sont aussi acteurs");
else
    System.out.println("certains chanteurs ne sont pas aussi acteurs");
System.out.println("***** les artistes *****");
// création d'un HashSet artistes contenant chanteurs
// et acteurs
HashSet<Personne> artistes;
artistes=new HashSet<Personne>(chanteurs);
artistes.addAll(acteurs);

// parcours du premier HashSet des artistes
Iterator<Personne> it;
it=artistes.iterator();
// tant qu'il reste des éléments dans le HashSet
Personne p;
while (it.hasNext())
{
    // récupération de l'élément courant
    p=it.next();
    System.out.println(p.getNom());
}
System.out.println("***** chanteurs et acteurs *****");
// création d'un HashSet des personnes qui sont
// chanteurs et acteurs

HashSet<Personne> act_chant;

```

```

act_chant=new HashSet<Personne>(chanteurs);
act_chant.retainAll(acteurs);
    it=act_chant.iterator();
    // tant qu'il reste des éléments dans le HashSet
    while (it.hasNext())
    {
        // récupération de l'élément courant
        p=it.next();
        System.out.println(p.getNom());
    }
System.out.println("***** chanteurs uniquement *****");
// création d'un HashSet des personnes
// uniquement acteurs
HashSet<Personne> uniquementChanteurs;
uniquementChanteurs=new HashSet<Personne>(chanteurs);
uniquementChanteurs.removeAll(acteurs);
for(Personne pe:uniquementChanteurs)
{
    System.out.println(pe.getNom());
}
System.out.println("***** acteurs uniquement *****");
// création d'un HashSet des personnes
// uniquement acteurs
HashSet<Personne> uniquementActeurs;
uniquementActeurs=new HashSet<Personne>(acteurs);
uniquementActeurs.removeAll(chanteurs);
for(Personne pe:uniquementActeurs)
{
    System.out.println(pe.getNom());
}
}

```

4.3. La classe LinkedList

La classe **LinkedList** est très complète puisqu'elle implémente les interfaces **Iterable**, **Collection**, **Deque**, **List** et **Queue**. Grâce à l'implémentation des

interfaces **Collection**, **List** et **Iterable**, elle possède un fonctionnement identique à la classe **ArrayList**. L'implémentation des interfaces **Deque** et **Queue** apporte quelques fonctionnalités supplémentaires que nous allons étudier.

Cette classe est recommandée si vous avez besoin d'accéder aux éléments dans le même ordre que celui dans lequel ils ont été stockés ou dans l'ordre inverse. Ces mécanismes sont appelés *First in - First out* (FIFO) ou *Last in - First out* (LIFO).

L'ajout d'un élément s'effectue avec les méthodes `addFirst` ou `addLast` pour un ajout soit en début de liste soit en fin de liste.

Pour obtenir un élément présent dans la liste, deux options sont possibles :

- Obtenir l'élément en laissant celui-ci dans la liste, dans ce cas il faut utiliser les méthodes **peekFirst** ou **peekLast**.
- Obtenir l'élément et le retirer de la liste, dans ce cas il faut utiliser les méthodes **pollFirst** ou **pollLast**.

```
import java.time.LocalDate;
import java.util.LinkedList;

public class TestLinkedList
{
    public static void main(String[] args)
    {
        LinkedList<Personne> ll;
        ll=new LinkedList<Personne>();
        // création des personnes pour remplir le HashSet
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));
        // ajout des éléments dans la liste
        ll.addFirst(p1);
        ll.addFirst(p2);
        ll.addFirst(p3);
        ll.addFirst(p4);
        ll.addFirst(p5);
        Personne p=null;
```



```

        // extraction et suppression des éléments
        // de la liste en commençant par le plus ancien
        do
        {
            p=ll.pollLast();
            if (p!=null)
                System.out.println(p.getNom());
        } while(p!=null);

        ll.clear();
    }
}

```

4.4. Streams et pipelines

Généralement les collections sont utilisées pour stocker des informations pour pouvoir les récupérer ou effectuer des traitements dessus. La technique classique pour réaliser des opérations sur les éléments contenus dans une collection consiste à faire une boucle **for** ou à utiliser l'**iterator** associé à la collection. Cette solution peut être avantageusement remplacée par l'utilisation de **stream** et de **pipeline**. Pour illustrer le fonctionnement de ces deux éléments, on peut faire la correspondance avec le fonctionnement d'une usine. Notre usine reçoit la matière première, laquelle traverse différentes unités de transformation pour obtenir en sortie le produit fini. Dans notre cas, les données présentes dans la collection représentent la matière première. Celles-ci sont transportées par l'intermédiaire d'objets implémentant l'interface **Stream**. Les unités de transformation sont représentées par les **Pipelines**. Ceux-ci peuvent produire un résultat directement exploitable (le produit fini) ou un autre objet **Stream** qui va alimenter un nouveau **Pipeline**.

Toutes les classes implémentant l'interface **Collection** sont capables de fournir un objet de type **Stream**. C'est cet objet qui est responsable de l'itération sur les éléments présents dans la collection.

Les **Pipelines** sont un peu plus complexes puisqu'ils sont constitués de plusieurs éléments.

Le premier représente la source à partir de laquelle le **Pipeline** va obtenir les informations sur lesquelles il va effectuer le ou les traitements. C'est bien sûr un objet **Stream** qui sera la source d'un **Pipeline**. Les traitements sont représentés par les différentes méthodes définies dans l'interface **Stream**. Ces méthodes correspondent à différents types de traitement pouvant être réalisés sur les données. Par défaut, ces méthodes ne font rien. Leurs comportements doivent être adaptés en fonction du type des objets qu'elles auront à traiter. Une expression lambda ou une référence de méthode est généralement utilisée pour cela. Ces méthodes peuvent générer le résultat final, un autre objet **Stream** constitué d'objets de même type que les objets d'origine ou un autre objet **Stream** contenant un autre type de données. On peut par exemple citer les méthodes suivantes représentatives de chacun de ces cas de figure.

`boolean allMatch(Predicate<? super T> predicate)` : cette fonction retourne un booléen indiquant si tous les éléments répondent à la condition indiquée par l'objet **Predicate**.

`IntStream mapToInt(ToIntFunction<? super T> mapper)` : cette fonction retourne un objet **Stream** d'entiers générés par le résultat de la fonction **ToIntFunction**.

`Stream<T> filter(Predicate<? super T> predicate)` : cette fonction génère un nouvel objet **Stream** contenant uniquement les objets du flux d'origine remplissant la condition définie par l'objet **Predicate**.

L'interface **Stream** contient plusieurs dizaines de méthodes répondant aux besoins les plus couramment rencontrés. Il est donc primordial de consulter la documentation la concernant.

L'exemple ci-dessous présente un petit aperçu des possibilités disponibles :

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;

public class TestStream
{
    public static void main(String[] args)
    {
        ArrayList<Personne> liste;

        // création des deux instances
        liste=new ArrayList<Personne>();

        // création des personnes pour remplir la liste
        Personne p1,p2,p3,p4,p5;
        p1 = new Personne("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Personne("McQueen", "Steeve",LocalDate.of(1930,3,24));
        p3 = new Personne("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Personne("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Personne("Willis", "Bruce",LocalDate.of(1955,3,19));

        // ajout de cinq personnes à la liste
        liste.add(p1);
```

```

    liste.add(p2);
    liste.add(p3);
    liste.add(p4);
    liste.add(p5);

    // la fonction allMatch retourne true si tous les éléments
    // de la liste remplissent la condition exprimée
    // dans l'expression lambda

    if(liste.stream().allMatch(p->p.getDate_nais().getYear()>1945))
        System.out.println("toutes les personnes sont nées après 1945");
    else
        System.out.println("certaines personnes sont nées avant 1945");

    // filtrage de personnes nées au mois de mars
    // ce filtrage génère un nouveau Stream
    // forEach parcourt ce nouveau Stream et
    // exécute l'expression lambda pour chaque élément

    liste.stream().filter(p->p.getDate_nais().getMonthValue()
==3).forEach(p->System.out.println(p.getNom()));

    // recherche de la personne la plus âgée de la liste
    // l'expression lambda représente l'implémentation
    // de l'interface Comparator

    System.out.println(liste.stream().max((pe1,pe2)->
    {
        if (pe1.calculAge()>pe2.calculAge())
            return 1;

        if (pe1.calculAge()<pe2.calculAge())
            return -1;

        return 0;
    }).get().getNom());

```

```

        // calcul de l'âge moyen des personnes présentes dans la
        // liste mapToLong génère un nouveau Stream de type long
        // obtenu à partir de la référence de fonction
        // average calcule la moyenne de ce nouveau flux
        // getAsDouble la tranforme en type double
        double ageMoyen = liste
            .stream()
            .mapToLong(Personne::calculAge)
            .average()
            .getAsDouble();

        System.out.println("age moyen des personnes de la liste :
" + ageMoyen + " ans");
    }
}

```

5. Exercices

Exercice 1

Créer une classe représentant un article d'un magasin de vente par correspondance. Un article est caractérisé par sa référence, sa désignation, son prix. Créer ensuite une méthode `main` permettant de tester le bon fonctionnement de la classe précédente.

Exercice 2

Ajouter les deux classes `Livre` et `Dvd` héritant de la classe `Article`.

Un livre possède un numéro ISBN, contient un certain nombre de pages et à été écrit par un auteur, un Dvd a une certaine durée et a été produit par un réalisateur.

Ajouter les attributs nécessaires aux classes `Livre` et `Dvd` pour avoir le nom de l'auteur ou du réalisateur. Tester ensuite le fonctionnement de ces deux nouvelles classes.

Exercice 3

Modifier les classes `Livre` et `Dvd` pour avoir disponibles les informations suivantes concernant l'auteur ou le réalisateur :

- Son nom
- Son prénom
- Sa date de naissance

Indice : les auteurs et les réalisateurs sont des Personnes.

Exercice 4

Modifier le code précédent pour pouvoir obtenir rapidement la liste des articles concernant un auteur ou un réalisateur.