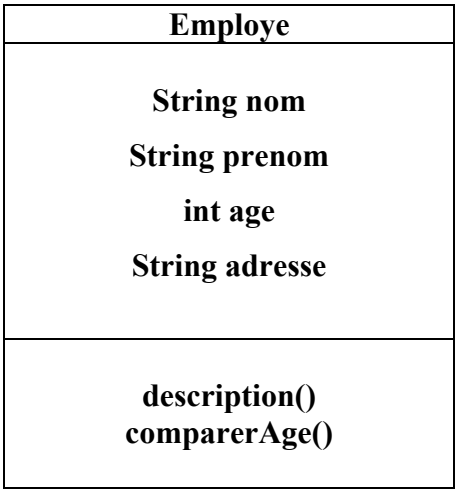
Programmation Orienté Objet (POO)

La programmation orientée objet est une technique de programmation célèbre qui existe depuis des années maintenant. Elle va vous aider à mieux organiser votre code, à le préparer à de futures évolutions et à rendre certaines portions réutilisables pour gagner en temps et en clarté. C'est pour cela que les développeurs professionnels l'utilisent dans la plupart de leurs projets. Parmi les avantages de la programmation orienté objet, on trouve :

- **Une clarté dans la façon de programmer** : en procédural, toutes les variables sont typiques, on ne peut pas avoir une variable ou un enregistrement qui permet de stocker par exemple une chaine de caractères, un entier, une autre chaine de caractères…etc. Cela est possible avec la POO en utilisant les objets.
- **La facilité de travailler en groupe** : le fait de travailler avec des classes séparées permet de faciliter le travail en groupe, si par exemple 4 développeurs travaillent sur un projet, il suffit que chacun travaille sur une classe et le regroupement des classes est très facile à faire.
- **La réutilisabilité** : l'un des fondements de la POO est **l'héritage**, ce dernier permet à une classe (classe fille) d'utiliser toutes les fonctions et variables de la classe dont elle hérite (classe mère) sans pour autant réécrire le code source.
- **La POO est adéquate pour une conception UML** : la POO complète une conception UML et il y'a mêmes des outils qui permettent de générer du code source à partir d'un diagramme de classe. Une table en UML se transforme en classe en POO, les attributs d'une tables en UML se transforment en variables d'instances en POO, les fonctions en UML sont des méthodes d'instances en POO…etc.

Pour faire un rappel sur la programmation orienté objet, nous allons traiter un exemple d'un projet de gestion de personnels dans une entreprise. Soit le digramme de Classes suivant :



Supposant qu'un employé dans une entreprise est représenté par son nom, son prénom, son âge et son adresse.

La première étape d'implémentation de ce diagramme de Classes est la création d'une Classe dans un projet Java, cette classe doit porter le même nom que la Classe UML. Par convention le nom des Classes débute toujours avec une lettre majuscule.

Pour bien suivre cette partie, de préférence créez un projet avec une classe Main et ensuite créez une Classe Employe (simple non exécutable).

Création d'une classe

```
public class Employe {

}
```

Public & private

Ici, la classe Employe est précédée du mot clé **public**. Vous devez savoir que lorsque nous créons une classe comme nous l'avons fait, NetBeans nous facilite la tâche en ajoutant automatiquement ce mot clé, qui correspond à la portée de la classe.

Une méthode marquée comme **public** peut donc être appelée depuis n'importe quel endroit du programme.

Nous allons voir une autre portée : **private**. Elle signifie que notre méthode ne pourra être appelée que depuis l'intérieur de la classe dans laquelle elle se trouve ! Les méthodes déclarées **private** correspondent souvent à des mécanismes internes à une classe que les développeurs souhaitent « cacher » ou simplement ne pas rendre accessibles de l'extérieur de la classe… Même chose pour les variables.

La première étape après la création d'une classe, est la définition des variables d'instances (celles qui identifient les objets).

```
public class Employe {

    String nom;
    String prenom;
    int age;
    String adresse;

}
```

Les types de variables

1. Les variables d'instance : ce sont elles qui définiront les caractéristiques de notre objet.
2. Les variables locales : ce sont des variables que nous utilisons à l'intérieur d'un bloc.
3. Les variables globales : ce sont des variables qu'on peut utiliser dans toute la classe
4. Les variables de classe : celles-ci sont communes à toutes les instances de votre classe et peuvent même être utilisées en dehors de la classe où elles sont déclarées.

Pour pouvoir instancier des objets employés, il faut déclarer les constructeurs dans la classe **Employe**.

Les constructeurs

Il y a deux types de constructeurs : le constructeur par défaut et le constructeur avec paramètres.

Constructeur par défaut

```
public Employe(){
    this.nom = "inconnu";
    this.prenom = "inconnu";
    this.age = 0;
    this.adresse = "nulle part";
}
```

Instanciation d'un objet (Dans une autre classe)

```
public static void main(String[] args) {
    // TODO code application logic here
    Employe e = new Employe();

}
```

Constructeur avec paramètres

```
public Employe(String n, String p, int a, String adr) {
    this.nom = n;
    this.prenom = p;
    this.age = a;
    this.adresse = adr;
}
```

Pour instancier un objet avec le constructeur avec paramètres, il faut respecter l'ordre et les types de variables.

```
Employe a = new Employe ("Mira","Abderrahmane",58,"Béjaia");
```

Accès aux variables d'instances via une autre classe

```
Employe e = new Employe();

Employe a = new Employe ("Mira","Abderrahmane",58,"Béjaia");

System.out.println(e.age);
e.age= -100;
System.out.println(e.age);

System.out.println(a.age);
a.age= -100;
System.out.println(a.age);
```

Vous voyez bien qu'il n'y a aucun contrôle sur les données affectées aux objets, comme l'exemple le montre, on peut même attribuer des valeurs négatives à la variable **age** ce qui est non cohérent. Cet accès aux variables est dit accès direct sans aucun contrôle, il faut toujours éviter ce genre de risque dans vos codes.

C'est pour cela qu'il faut d'abord protéger les variables d'instance en les déclarants **private**, comme ceci :

```
private String nom;
private String prenom;
private int age;
private String adresse;
```

Une fois que vous déclarez vos variables d'instances avec le mot clé **private**, vous allez remarquer qu'il y a des erreurs dans l'autre classe (là où vous avez instancié vos objets)

```
19          Employe e = new Employe();
20
21          Employe a = new Employe ("Mira","Abderrahmane",58,"Béjaia");
22
            System.out.println(e.age);
            e.age= -100;
            System.out.println(e.age);
26
            System.out.println(a.age);
            a.age= -100;
            System.out.println(a.age);
```

Ces erreurs sont dues à la protection des variables (Encapsulation) et désormais vous avez empêché l'accès direct à vos variables d'instances qui ne sont plus accessibles en dehors de la classe où elles sont déclarées.

Nous allons maintenant voir comment accéder tout de même à nos données d'une façon **correcte et contrôlée**. Ceci en utilisant les accesseurs (getters) et mutateurs (setters).

Accesseurs et Mutateurs (Getters & Setters)

Un accesseur est une méthode qui va nous permettre d'accéder aux variables de nos objets en lecture, et un mutateur nous permettra d'en faire de même en écriture ! Grâce aux accesseurs, vous pourrez afficher les variables de vos objets, et grâce aux mutateurs, vous pourrez les modifier.

```
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 int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getAdresse() {
    return adresse;
}

public void setAdresse(String adresse) {
    this.adresse = adresse;
}
```

Les accesseurs sont bien des méthodes (fonctions), et elles sont **public** pour que vous puissiez y accéder depuis une autre classe que celle-ci : depuis le main, par exemple. Les accesseurs sont du même type que la variable qu'ils doivent retourner. Les mutateurs sont, par contre, des procédures (void).

Exemple d'utilisation d'accesseurs et mutateurs

```
Employe a = new Employe ("Mira","Abderrahmane",58,"Béjaia");

  System.out.println(a.getAge());
  a.setAge(-100);
  System.out.println(a.getAge());
```

Vous remarquez que l'accès aux variables d'instance se fait à l'aide des accesseurs et mutateurs que vous avez définis. Néanmoins, le contrôle sur les valeurs de la variable **age** n'est pas assuré. C'est pourquoi il faut ajouter un **contrôle** dans le mutateur (setter) pour empêcher les valeurs négatives pour la variable **age**. De même, il faut faire la même chose pour les autres variables selon le besoin.

```
public void setAge(int a) {
    if (a>=0){
        this.age = a;
    }
    else{
        this.age = a * (-1);
    }
```

Avec ce contrôle dans le setter de la variable **age**, on est sûr qu'à la modification de la variable **age**, les valeurs négatives ne seront pas prise en compte. Ceci n'est pas suffisant pour assurer que tous les objets employés auront un **age** positif, parce qu'à l'instanciation on peut faire :

```
Employe a = new Employe ("Mira","Abderrahmane",-100,"Béjaia");
```

Dans cet exemple, l'objet a un âge négatif bien que nous avons fait un contrôle dans le setter. Pour empêcher ce cas de figure, il faut ajouter un autre contrôle dans le constructeur comme suit :

```
public Employe(String n, String p, int a, String adr) {
    this.nom = n;
    this.prenom = p;

    if (a >= 0) {
        this.age = a;
    } else {
        this.age = a * (-1);
    }

    this.adresse = adr;
}
```

Avec ce deuxième contrôle dans le constructeur, on est sûr maintenant que tous les objets instanciés auront toujours un âge positif.

Si vous avez d'autres contrôles pour les autres variables, vous procédez de la même façon.

Méthodes d'instances (fonctions et procédures)

Les méthodes d'instances sont les méthodes qui manipulent des objets.

L'appel des méthodes d'instance se fait par des instances (des objets).

Pour notre exemple nous avons deux méthodes **description() et comparerAge().**

- La méthode description() permet de décrire les employés comme suit :

```
Employe a = new Employe ("Mira","Abderrahmane",50,"Béjaia");
a.description();
```

doit retourner :

```
run:
l'employé s'appelle Mira Abderrahmane et il a 50 ans et il habite à Béjaia
```

Pour implémenter cette méthode, vous pouvez la déclarer comme fonction ou procédure.

```
public void description(){
    System.out.println( "l'employé s'appelle "+this.nom+" "+this.prenom+" et il a "+this.age+" ans et il habite à "+this.adresse);
}
```

- La méthode **comparerAge()** permet de comparer l'âge entre deux employés comme suit :

```
Employe a = new Employe ("Mira","Abderrahmane",50,"Béjaia");
Employe b = new Employe ("Abane","Ramdane",55,"Tizi Ouzou" );
a.comparerAge(b);
```

doit retourner comme résultat :

```
run:
Abane Ramdane est plus âgé que Mira Abderrahmane
```

Pour implémenter cette méthode, vous pouvez la déclarer comme fonction ou procédure.

```
public void comparerAge(Employe x) {
    if (x.age > this.age) {
        System.out.println(x.nom + " " + x.prenom + " est plus âgé que " + this.nom + " " + this.prenom);
    } else {
        if (x.age < this.age) {
            System.out.println(x.nom + " " + x.prenom + " est plus jeune que " + this.nom + " " + this.prenom);
        } else {
            System.out.println(x.nom + " " + x.prenom + " a le même âge que " + this.nom + " " + this.prenom);
        }
    }
}
```

Le mot clé **this** fait référence à l'**objet courant**. L'objet courant est l'objet qui appelle la méthode. A l'exécution, le mot clé **this** sera remplacé par l'objet qui a appelé la méthode.

Variables de classes

Nous avons vu les variables d'instance qui forment la carte d'identité d'un objet ; maintenant, voici les variables de classe. La particularité de ce type de variable, c'est qu'elles seront communes à toutes les instances de la classe, et elles peuvent être visibles (utilisées) même en dehors de la classe où elles sont déclarées.

On les utilise souvent en programmation pour faire passer des paramètres d'une classe à une autre. Dans notre exemple, nous allons compter le nombre d'instances de notre classe Employe (Combien de fois on a instancié notre classe Employe).

Exemple avec Nombre d'instances

Afin qu'une variable soit une variable de classe, elle doit être précédée du mot clé **static**.

```
//Variables publiques qui comptent les instances
public static int nbreInstances = 0;
```

Pour calculer le nombre d'instances de la classe Employe, il faut incrémenter le nombre d'instance nbreInstances dans le constructeur par défaut et le constructeur avec paramètres comme suit :

```
public Employe(String n, String p, int a, String adr) {
    this.nom = n;
    this.prenom = p;

    if (a >= 0) {
        this.age = a;
    } else {
        this.age = a * (-1);
    }

    this.adresse = adr;

    nbreInstances=nbreInstances+1;
}

public Employe() {
    this.nom = "inconnu";
    this.prenom = "inconnu";
    this.age = 0;
    this.adresse = "nulle part";

    nbreInstances=nbreInstances+1;
}
```

Si on veut maintenant appeler cette variable via une autre classe (celle qui contient le main par exemple), l'appel se fait avec le nom de la classe .nom de variables. Dans notre exemple ça sera Employe.nbreInstances.

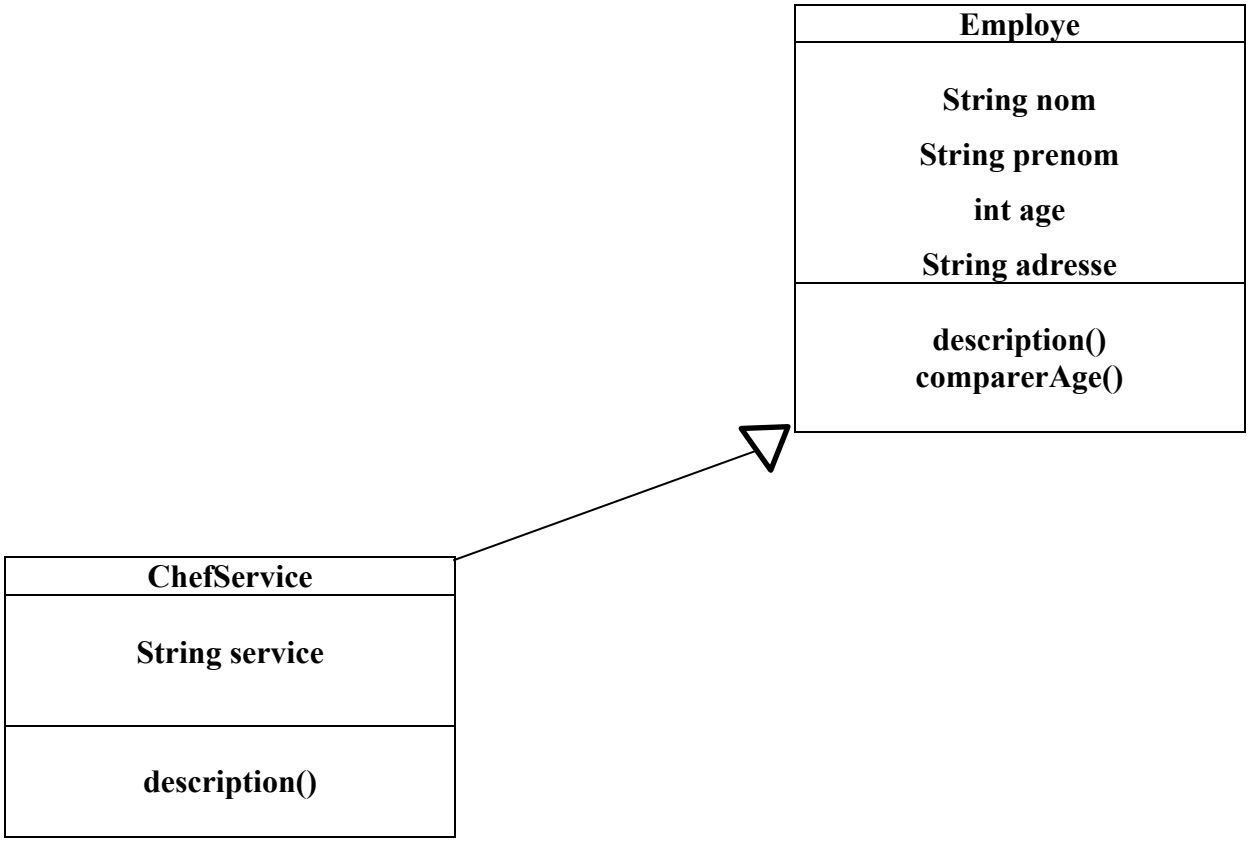
```
System.out.println("Le nombre d'instances de la classe Employe est "+Employe.nbreInstances);
```

L'héritage

La notion d'héritage est l'un des fondements de la programmation orientée objet. Grâce à elle, nous pourrons créer des classes héritées (aussi appelées *classes dérivées*) de nos classes mères (aussi appelées *classes de base*). Nous pourrons créer autant de classes dérivées, par rapport à notre classe de base, que nous le souhaitons. De plus, nous pourrons nous servir d'une classe dérivée comme d'une classe de base pour élaborer encore une autre classe dérivée.

Supposant que vous avez dans votre conception une classe ChefService qui hérite de la classe Employe.

Un chef service est un employé (il a un nom, un prénom, un âge et une adresse) et il gère un service dans l'entreprise (Service personnel, Service de la paie…etc).



Pour implémenter cette classe **ChefService**, il faut créer une classe dans le même package qui contient la classe **Employe**. Ensuite rajouter le mot clé **extends** pour que **ChefService** soit une classe héritée (Classe fille) de la classe **Employe**.

```
public class ChefService extends Employe{

}
```

Instanciez un objet de la classe **ChefService** et utilisez la méthode **description()** de la classe Employe.

```
ChefService c = new ChefService();
c.description();
```

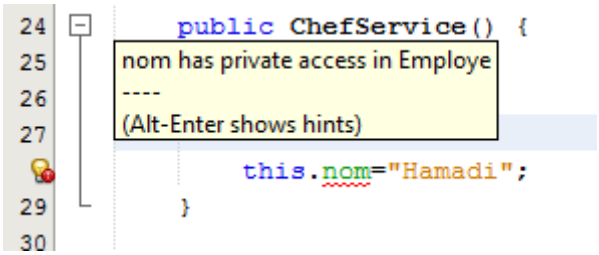
En exécutant, vous aurez comme résultat :

```
l'employé s'appelle inconnu inconnu et il a 0 ans et il habite à nulle part
```

En fait, lorsque vous déclarez une classe, si vous ne spécifiez pas de constructeur, le compilateur (le programme qui transforme vos codes sources en byte code) créera, au moment de l'interprétation, le constructeur par défaut. En revanche, dès que vous avez créé un constructeur, n'importe lequel, c'est celui-ci qui sera utilisé. Notre classe ChefService hérite de la classe Employe, par conséquent, le constructeur de notre objet appelle, de façon tacite, le

constructeur de la classe mère. C'est pour cela que les variables d'instance ont pu être initialisées.

Utilisation des variables d'instances de la classe mère (Employe) dans la classe fille (ChefService)



On ne peut pas accéder directement aux variables d'instances de la classe mère parce qu'elles sont déclarées **private.**

Pour remédier à ça en gardant les variables d'instances protégées (encapsulées), il faut utiliser le mot clé **protected.**

En fait, seules les méthodes et les variables déclarées **public** ou **protected** peuvent être utilisées dans une classe héritée ; le compilateur rejette votre demande lorsque vous tentez d'accéder à des ressources privées d'une classe mère.

```
protected String nom;
protected String prenom;
protected int age;
protected String adresse;
```

Là vous pouvez utiliser ces variables dans la classe fille (ChefService).

Java ne gère pas les héritages multiples : une classe dérivée (aussi appelée classe fille) ne peut hériter que d'une seule classe mère !

Si votre Classe fille comportera une variable (ou plus) que sa classe mère, il faut définir un constructeur par défaut et un constructeur avec paramètres. Vous pouvez toujours utiliser les variables de la classe mère grâce au mot clé **super()**

Pour notre exemple, la classe **ChefService** a un champ (une variable) de plus : **service**. Donc il faut d'abord déclarer cette variable ensuite l'utiliser dans les deux constructeurs comme suit :

```
private String service;

public ChefService() {
    super();
    service="inconnu";
}

public ChefService(String n, String p, int a, String adr,String ser) {
    super(n, p, a, adr);
    service=ser;
}
```

Getters et setters pour le champ « service »

```
public String getService() {
    return service;
}

public void setService(String service) {
    this.service = service;
}
```

- Pour les méthodes d'instances, la méthode **comparerAge()** de la classe **Employe** fonctionne également pour comparer l'âge entre deux chefs service. Elle fonctionne également pour comparer l'âge entre un chef service et un employé. C'est pourquoi ce n'est pas la peine de la redéfinir.
- Par contre pour la méthode **description()** de la classe **Employe**, quand on l'utilise avec un Chef service, elle affiche le nom, le prénom, l'âge et l'adresse mais elle n'affiche pas le **service**, c'est pourquoi il faut redéfinir la méthode **description()** dans la classe **ChefService** tout en utilisant les atouts de l'héritage pour ne pas réécrire tout le code comme suit :

```
public void description(){
    super.description();
    System.out.println("et il gère le service"+service);
}
```

Instanciation d'un Chef Service

```
ChefService c = new ChefService("Hamadi","Karim",55,"Alger","Personnel");
c.description();
```

En exécutant vous aurez comme résultat :

```
l'employé s'appelle Hamadi Karim et il a 55 ans et il habite à Alger
et il gère le servicePersonnel
```

Les exceptions

Une exception est une erreur se produisant dans un programme qui conduit le plus souvent à l'arrêt de celui-ci. Il vous est sûrement déjà arrivé d'obtenir un gros message affiché en rouge dans la console de NetBeans : eh bien, cela a été généré par une exception… qui n'a pas été *capturée*.

Pour remédier à ça, il faut capturer l'exception correspondante et ensuite de la traiter, c'est-à-dire d'afficher un message personnalisé et de continuer l'exécution.

Exemple de division sur 0

```
// Exemple de division sur 0
int j = 20, i = 0;
System.out.println(j / i);
System.out.println("Salam !");
```

Ça génère une exception et ça ne permet pas de continuer le programme et afficher le message.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at tp1.TpException.main(TpException.java:20)
Java Result: 1
```

Pour pouvoir capturer l'exception, on utilise le bloc **try{…} catch {…}**

```
int j = 20, i = 0;
try {
    System.out.println(j / i);
} catch (Exception e) {
    System.out.println("Division par zéro ! "+e);
}
System.out.println("Salam");
```

Prenons un cas de figure très simple : imaginons que vous souhaitez effectuer une action, qu'une exception soit levée ou non (nous verrons lorsque nous travaillerons avec les fichiers qu'il faut systématiquement fermer ceux-ci). Java vous permet d'utiliser une clause via le mot clé **finally**.

```
try {
    System.out.println(" =>" + (1 / 0));
} catch (Exception e) {
    e.printStackTrace();
} finally {
    System.out.println("action faite systématiquement");
}
```

Grâce à la clause **finally**, un morceau de code est exécuté quoi qu'il arrive. Cela est surtout utilisé lorsque vous devez vous assurer d'avoir fermé un fichier, clos votre connexion à une base de données ou un socket (une connexion réseau).