LA PROGRAMMATION ORIENTÉ OBJET EN JAVA

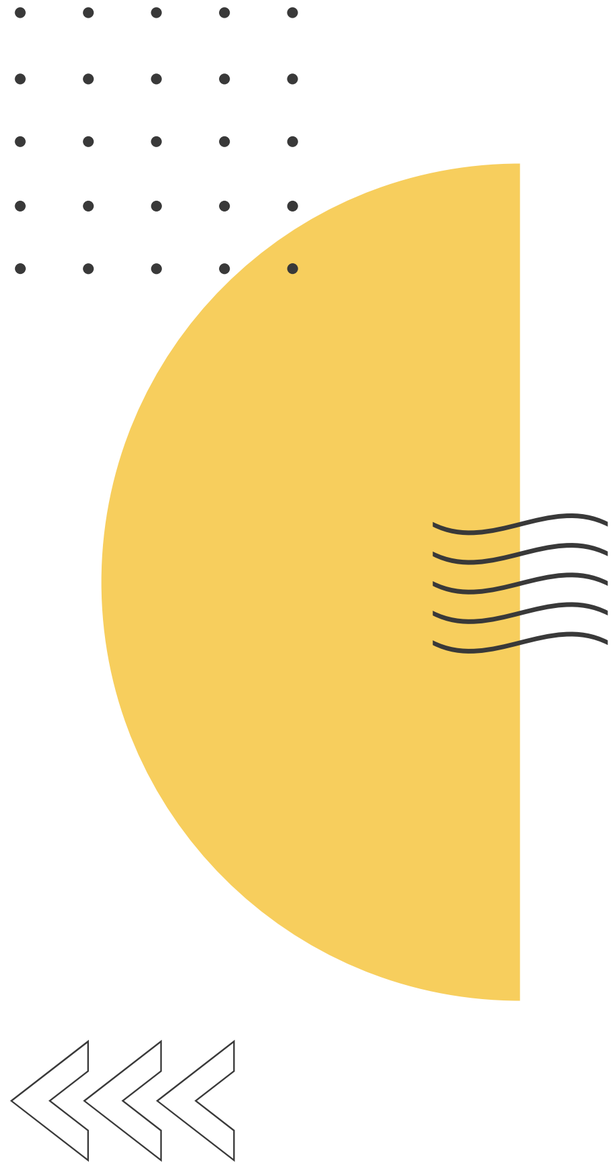
Philémon Globléhi
Développeur Back-end

Sommaire

01- Définition

La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement.

source: https://fr.wikipedia.org/wiki/Programmation_oriente_objet



02 - Création de classe et d'objet

Une classe est une entité logicielle caractérisée par des propriétés et des savoir-faire (méthodes)

Un objet est une entité logicielle caractérisée par des propriétés, des savoir-faire (méthodes) ET un état (valeurs à un instant t des propriétés de la classe dont est issue l'objet)

- Classe = fichier (Ex : Employe.class) → statique
- Objet = espace mémoire → dynamique
- Créer un Objet = instancier une classe (via new)
- Employe employe1 = new Employe("Philémon","GLOBLEHI") ;

Exemple 01

Employe.java

```java
package domaine;


public class Employe {

    private String nom; // propriété
    private String prenoms; // propriété

    public void poserConges() { // méthode (savoir-faire)
        System.out.println("Un employé pose des congés");
    }

}

```

Exemple 02

Employe2.java

```java
package domaine;


public class Employe2 {

    private String nom;
    private String prenoms;

    public void poserConges() {
        System.out.println("L'employé " + this.nom + " " + this.prenoms + "pose un congé");
    }

}
```

03 - Encapsulation

L’encapsulation consiste à réduire la visibilité des propriétés et méthodes d'une classe, afin de n'exposer que les fonctionnalités réellement nécessaires pour les classes utilisatrices.

Cette réduction de visibilité est rendue possible avec
– private : visibilité interne à la classe
– public : visibilité pour toutes les classes
– protected : visibilité pour les classes enfants
– default (aucun des modes ci-dessus n’est spécifié) : visibilité pour les classes du même package

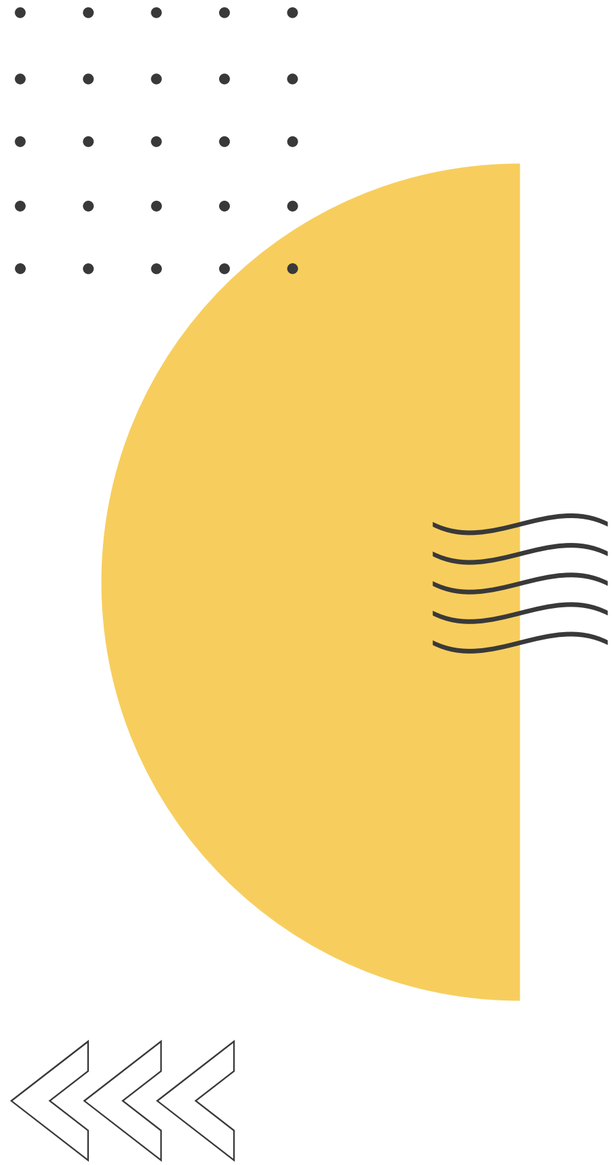
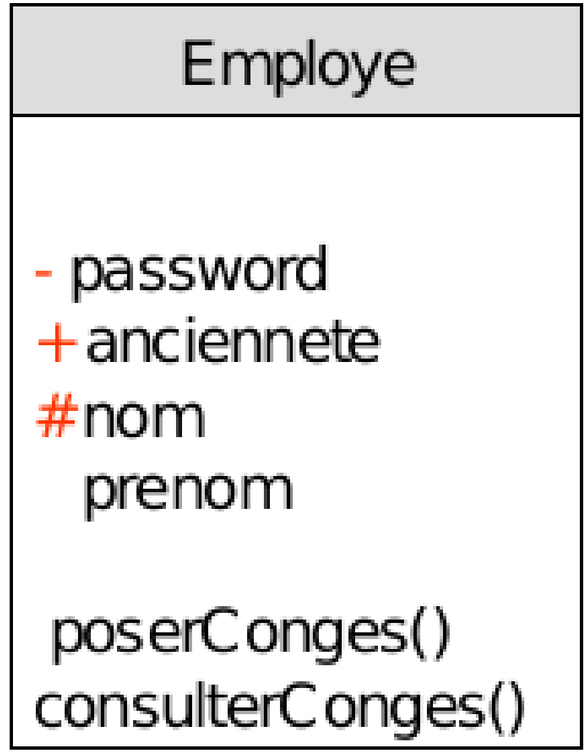


Tableau de visibilté

Classe / Membre	la classe	une classe dans le même package	une classe qui hérite dans le package	une classe qui hérite hors package	une autre classe hors package
public	Oui	Oui	Oui	Oui	Oui
protected	Oui	Oui	Oui	Oui	Non
Ø	Oui	Oui	Oui	Non	Non
private	Oui	Non	Non	Non	Non

Notation UML

private
public
protected
default



Les getters et setters

Les attributs privés ne sont pas accessibles : il leur faut des accesseurs dédiés

– Lecture : « getter »

– Ecriture : « setter »

• Les noms de ces opérations suivent la convention suivante :

– Ils sont constitués de get ou set suivant le type d'accesseur, suivi du nom de l'attribut débutant par une majuscule :

• Exemple d'attribut:

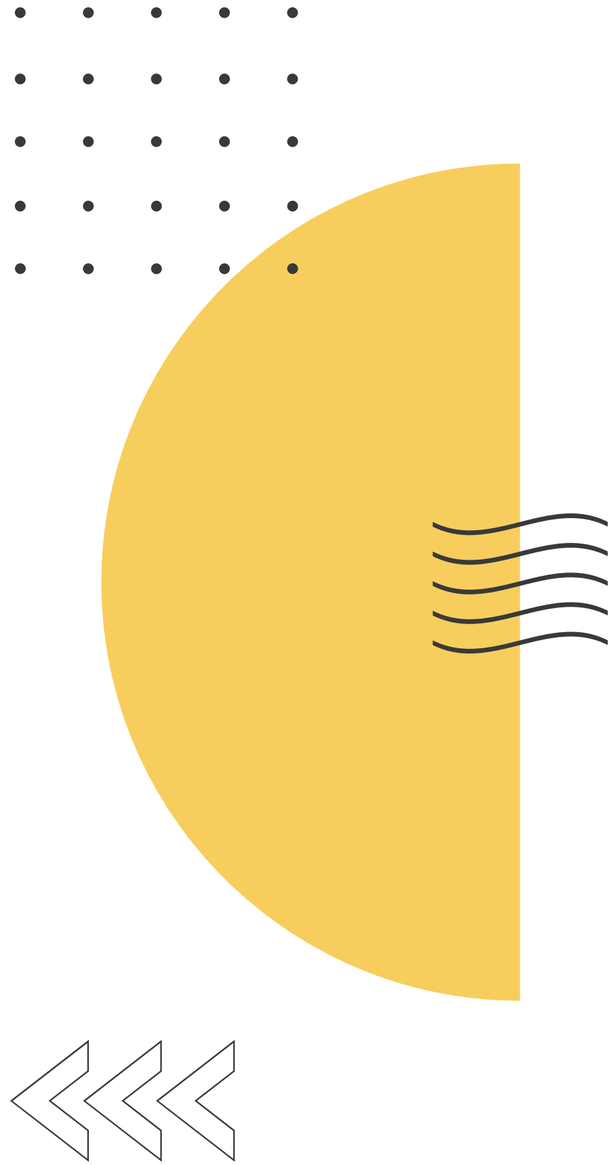
– private String password;

• Exemple d'accesseurs sur l'attribut password :

– public String getPassword()

– public void setPassword(String p)

- Bonne pratique : masquer les propriétés (private)
- On ne peut accéder 'directement' aux propriétés
• Passer par des méthodes publiques que la classe
  - expose dans sa grande bonté



Exemple 01

Employe3.java

```java
package domaine;


public class Employe3 {

    // Declaration propriétés
    private String nom;
    private String prenoms;

    // Delcaration des getters et setters
    public String getNom() {
        return this.nom;
    }

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

    public String getPrenoms() {
        return this.prenoms;
    }

    public void setPrenoms(String prenoms) {
        this.prenoms = prenoms;
    }

    public void poserConges() {
        System.out.println("L'employé " + this.nom + " " + this.prenoms + " pose des congés");
    }



}
```

Exemple 02

Lanceur.java

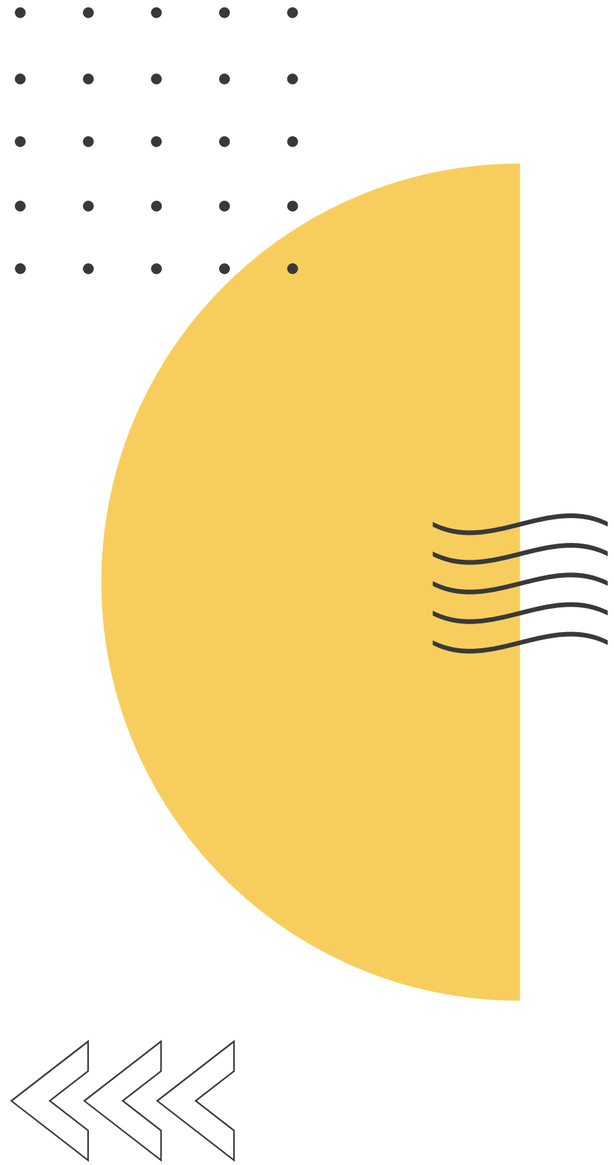
```java
package igs;

import domaine.Employe;

public class Lanceur {

    public static void main(String[] args) {

        System.out.println("Bienvenue dans mon application de gestion du personnel");

        Employe monEmploye = new Employe();

        monEmploye.setNom("Globléhi");
        monEmploye.setPrenoms("Philémon");

        System.out.println("L'employé " + monEmploye.getNom() + " " + monEmploye.getPrenoms() + " demande un congé");

    }

}
```

04 - Constructeur

Un constructeur est la méthode d'initialisation d'un objet
– Le nom de cette méthode est le même que le nom de la classe et elle n'a pas de type de retour
– Il sert à initialiser l'état de l'objet qu'il crée
– Une classe peut comporter plusieurs constructeurs ayant des arguments différents
• On fait appel au constructeur d'un objet avec le mot clé new
– new Employe("Globléhi") : indique que l'on appelle un constructeur de la classe Employe, qui prend un argument de type String

Le constructeur permet de définir un état de l'objet suite à sa création

- Cela permet de gérer l'état en plusieurs lignes (en passant par des setters)
- Instanciation se fait avec le mot clé new



Exemple 01

Employe.java

```java
package domaine;


public class Employe {

    private String nom;
    private String prenoms;


    public Employe() { // Constructeur sans argument
        super();
    }

    public Employe(String nom, String prenoms) { // Constructeur avec 2 arguments
        super();
        this.nom = nom;
        this.prenoms = prenoms;
    }


    // Getters et Setters
    public String getNom() {
        return nom;
    }

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

    public String getPrenoms() {
        return prenoms;
    }

```

Exemple 02

Lanceur.java

```java
package igs;

import domaine.Employe;

public class Lanceur {

    public static void main(String[] args) {

        System.out.println("Bienvenue dans mon application de gestion du personnel");

        Employe monEmploye = new Employe("Globléhi", "Philémon");

        System.out.println("L'employé " + monEmploye.getNom() + " " + monEmploye.getPrenoms() + " demande un congé");

    }

}
```

05 - Surcharge

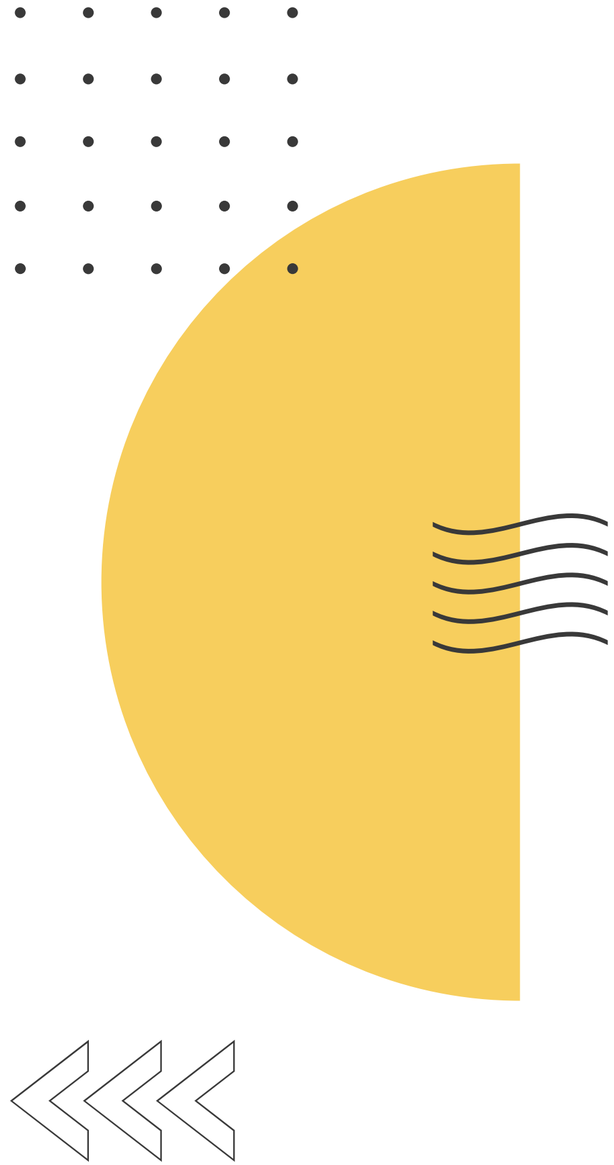
Surcharger une méthode = coder une méthode
qui a le même nom mais une signature différente

- Ex : surcharger un constructeur
- Ex : surcharger une méthode métier

signature de méthode: visibilite + type de retour + nom de la méthode + arguments

Avantage de la surcharge : offrir du choix au
client (developpeur) !

- public void poserConges (String dateDebut, String dateFin)
- public void poserConges (DemandeConges demande)



Exemple 01

```
 9
10    public Employe() { // Constructeur sans argument
11        super();
12    }
13
14    public Employe(String nom) { // Constructeur avec 1 seul argument
15        super();
16        this.nom = nom;
17    }
18
19    public Employe(String nom, String prenoms) { // Constructeur avec 2 arguments
20        super();
21        this.nom = nom;
22        this.prenoms = prenoms;
23    }
24
```

Exemple 02

```
42
43      // Methodes metiers
44
45      public void poserConges() {
46          System.out.println("Un employé pose des congés");
47      }
48
49      public void poserConges(String dateDebut, String dateFin) {
50          System.out.println("L'employé " + this.nom + " " + this.prenoms + " pose des congés de " + dateDebut + " a " + dateFin);
51      }
52
53  }
```

06 - Héritage

L’héritage est une notion qui implique :

– Une seule classe mère ou super-classe

• Pas d'héritage multiple (de code) en Java

• La classe mère de toutes les classes Java est la classe java.lang.Object

– Une ou plusieurs classes dites 'spécialisées' ou 'sous-classes'

• Elles héritent des attributs et méthodes public et protected de leur classe mère, par héritage direct ou en cascade (plusieurs niveaux d'héritage)

• Elles peuvent

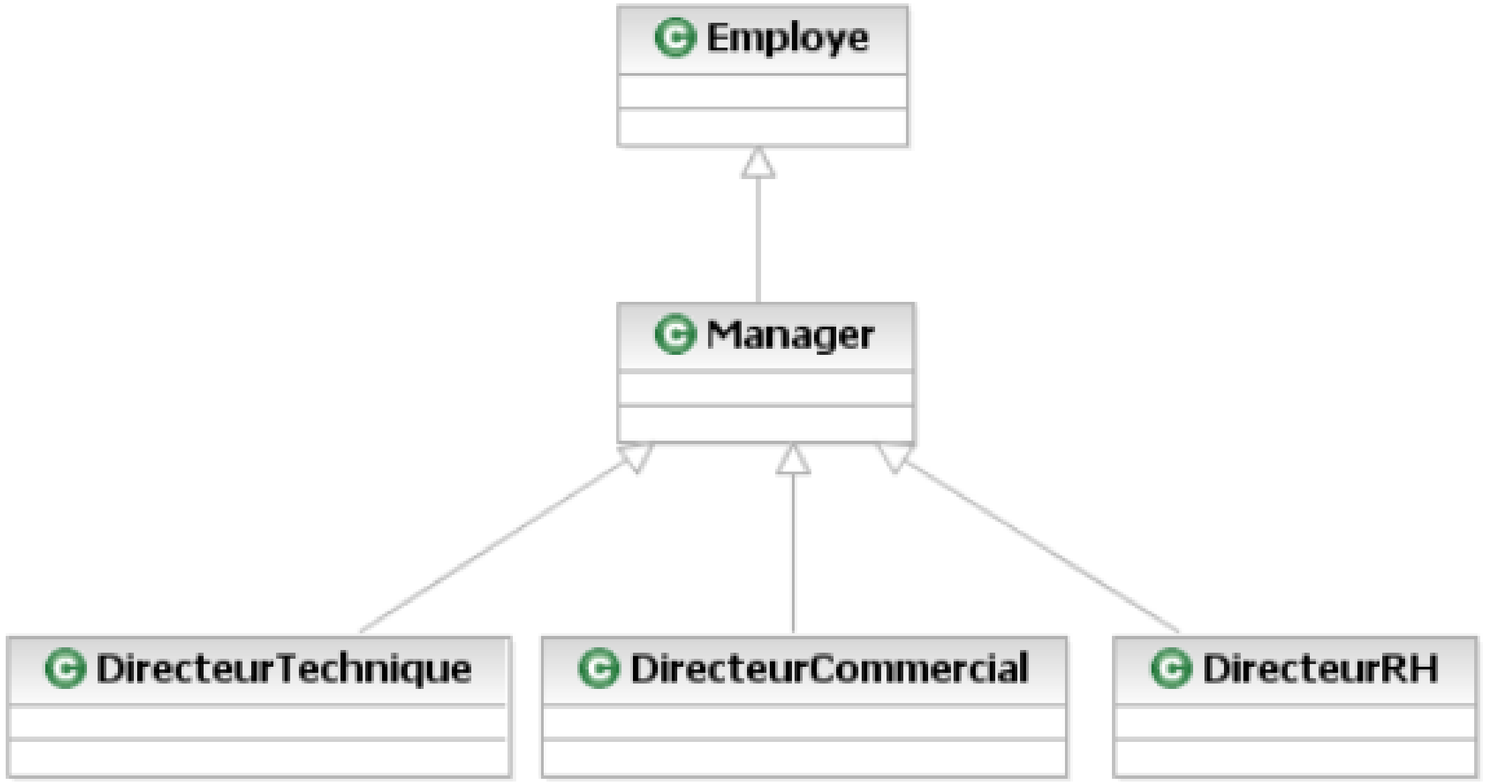
– Redéfinir les méthodes de la classe mère

– Enrichir le comportement en ajoutant d’autres méthodes

L'héritage s’implémente avec le mot clé extends suivi du nom de la classe mère

```
public class Manager extends Employe {
}
```

Notation UML



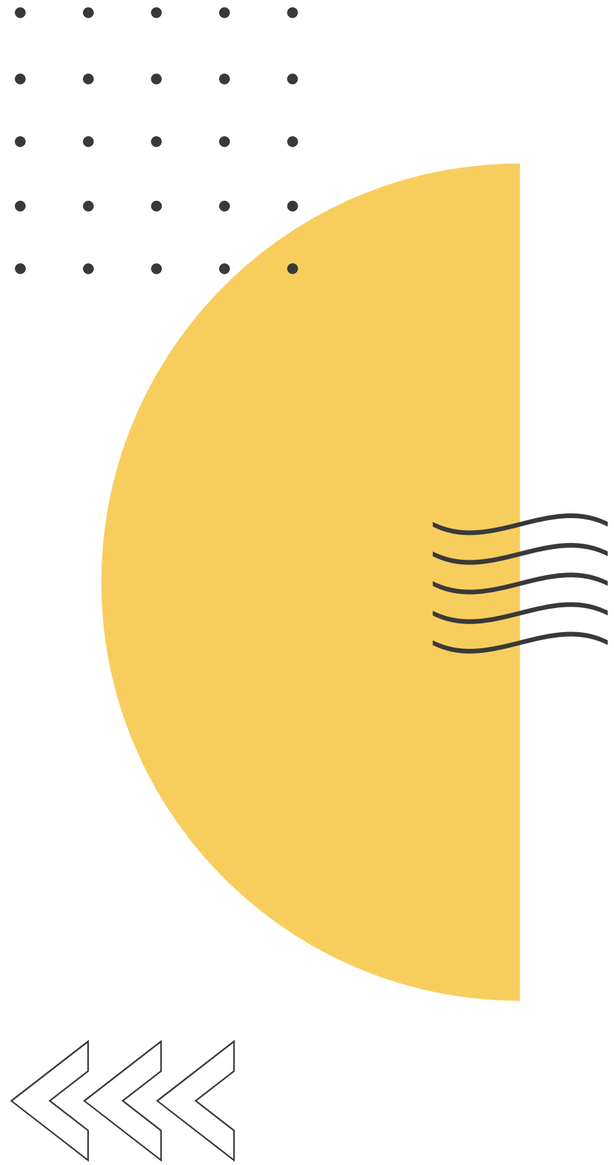
07 - Polymorphisme

Lorsque vous construisez une classe héritant d'une autre classe, vous avez la possibilité de redéfinir certaines méthodes de la classe mère. Il s'agit de remplacer le comportement de la fonction qui a été définie par la classe mère.C'est le concept de polymorphisme. L'idée étant de pouvoir utiliser le même nom de méthode sur des objets différents. Et bien sûr, cela n'a de sens que si le comportement des méthodes est différent.

Considérons le code ci-dessous, considérons la méthode déplacer() dans la classe mère Animal :

```
class Animal {
   void deplacer() {
   System.out.println("Je me déplace");
}
```

Appliquons le principe de polymorphisme pour cette méthode dans les différentes classes filles Chien, Oiseau et Pigeon :



```
class Chien extends Animal {
   void deplacer() {
      System.out.println("Je marche");
   }
}
```

```
class Oiseau extends Animal {
   void deplacer(){
      System.out.println("Je vole");
   }
}
```

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

   Animal a1 = new Animal();
   Animal a2 = new Chien();
   Animal a3 = new Pigeon();

   a1.deplacer();
   a2.deplacer();
   a3.deplacer();
   }
}
```

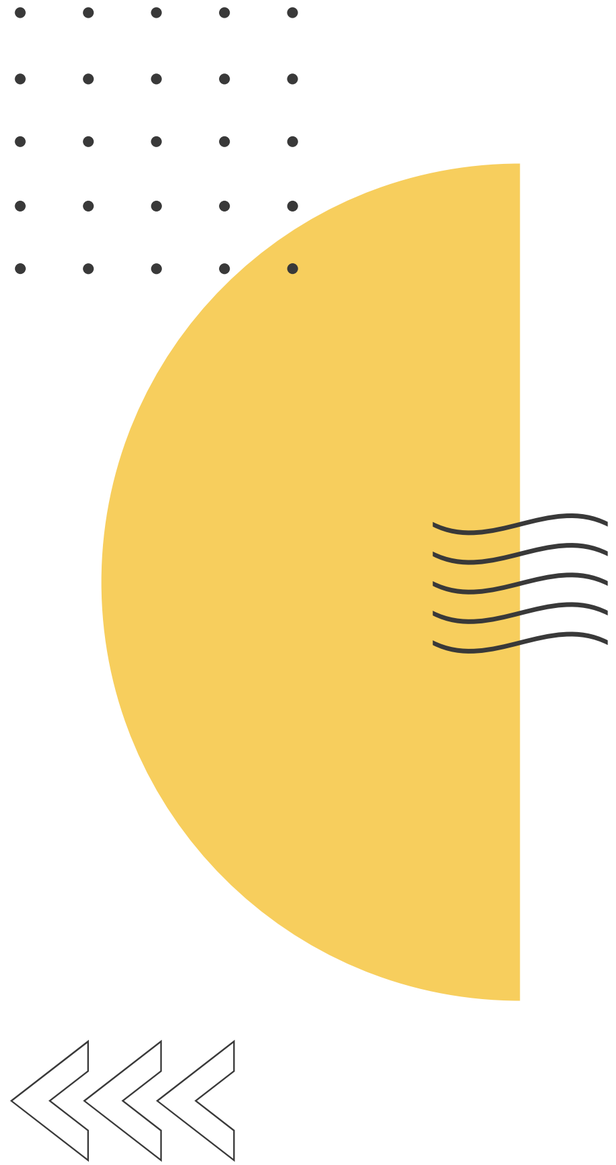
Et à l'exécution, ça donne :

```
Je me déplace
Je marche
Je vole surtout en ville
```

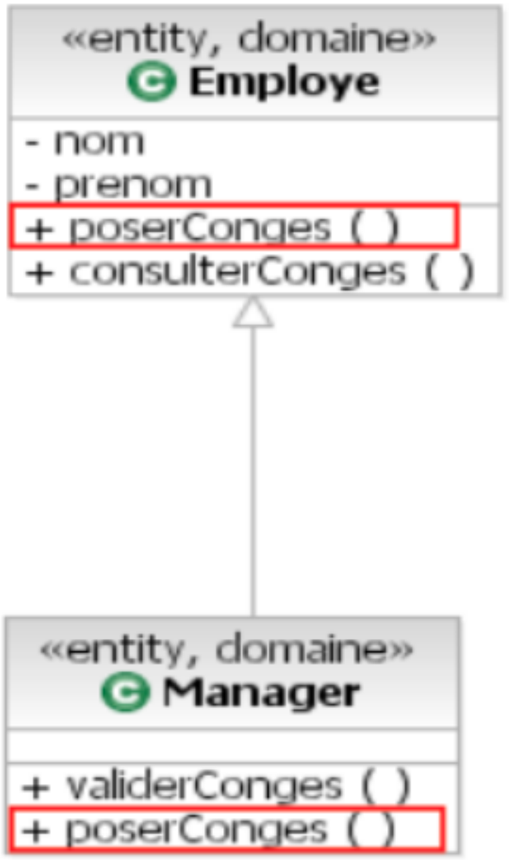
08 - Redéfinition

Redéfinir une méthode dans une classe fille c'est coder dans la classe fille une méthode qui a la même signature que dans la classe mère, mais dont l'implémentation est différente de celle de la classe mère.

signature de méthode: visibilite + type de retour + nom de la méthode + arguments



Contexte


«entity, domaine»
Employe
- nom
- prenom
+ poserConges ( )
+ consulterConges ( )
«entity, domaine»
Manager
+ validerConges ( )
+ poserConges ( )

Exemple 01

Employe.java

```java
package domaine;


public class Employe {

    // propriétés
    private String nom;
    private String prenoms;

    public Employe(String nom, String prenoms) { // Constructeur
        super();
        this.nom = nom;
        this.prenoms = prenoms;
    }

    // Getters et Setters
    public String getNom() {
        return nom;
    }

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

    public String getPrenoms() {
        return prenoms;
    }

    public void setPrenoms(String prenoms) {
        this.prenoms = prenoms;
    }

    // Methodes metiers
    public void poserConges() {
        System.out.println("L'employé " + this.nom + " " + this.prenoms + " pose des congés");
    }

}

```

Exemple 02

Manager.java

```java
package domaine;


public class Manager extends Employe {

    public Manager(String nom, String prenoms) {
        super(nom, prenoms);
    }

    public void poserConges() {
        System.out.println("Les congés du manager " + getPrenoms() + " " + getNom() + " sont systématiquement acceptés");
    }

    public void validerConges() {
        System.out.println("Le manager " + getPrenoms() + " " + getNom() + " valide les demandes de ses employés");
    }

}

```

Exemple 03

Lanceur.java

```java
package igs;

import domaine.Employe;
import domaine.Manager;

public class Lanceur {

    public static void main(String[] args) {

        System.out.println("Bienvenue dans mon application de gestion du personnel");

        Employe monEmploye = new Employe("N'Guessan", "Yannick");
        Manager monManager = new Manager("Globléhi", "Philémon");

        monEmploye.poserConges();
        monManager.poserConges();

    }

}
```

09 - Interface

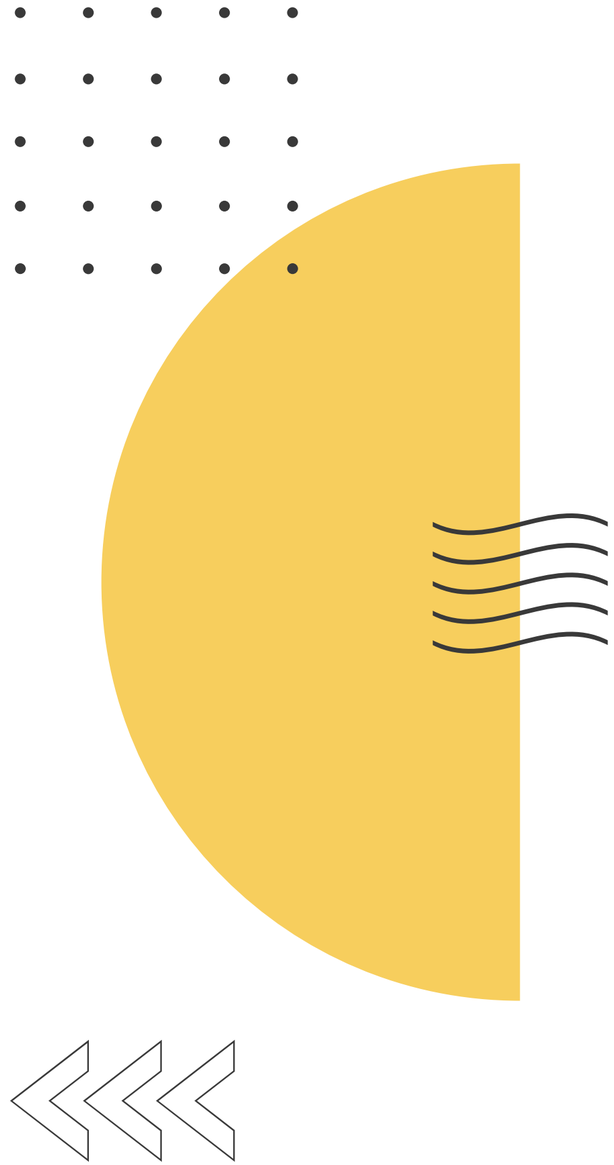
Interface = fichier .java

- Interface = contrat = 1 ou plusieurs méthodes
- On déclare la signature des méthodes dans l'interface
- Pas d'implémentation des méthodes dans l'interface !
- Une interface peut avoir des attributs
- Ces attributs sont alors des constantes

Une interface n’est pas une classe

- Impossible de créer d'instance interface !

```
public interface IGestionConges{
        public String titre = "Interface de gestion des congés";
        public void poserConges();
        public Collection consulterConges();
}
```

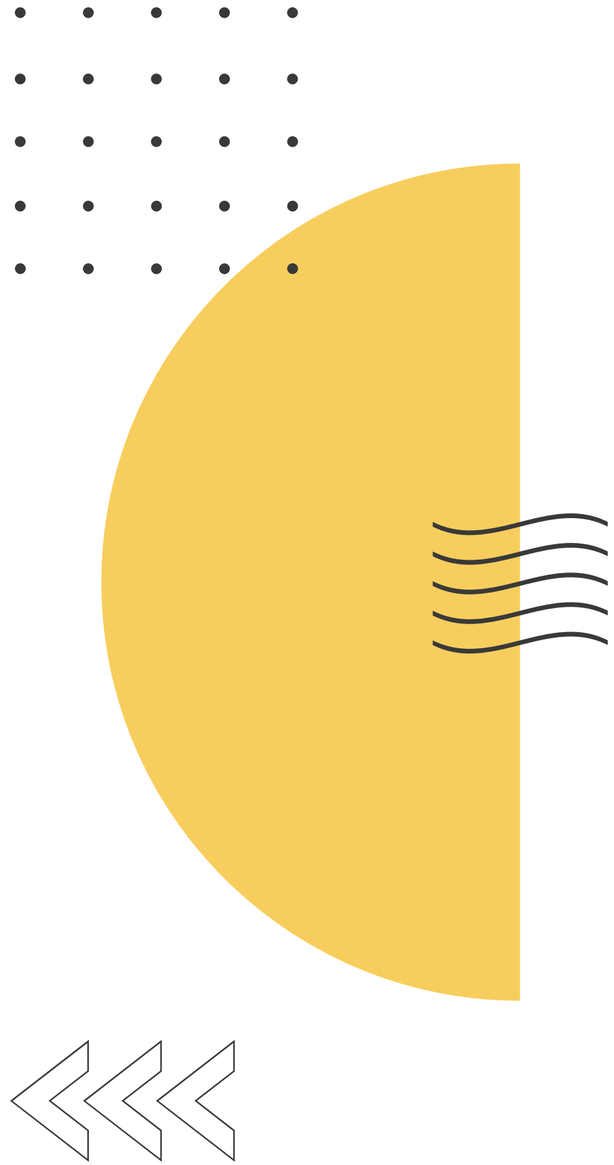


Implémentation

Une classe a la possibilité de dire qu'elle offre la totalité des savoir-faire d'une interface

- On dit que la classe 'implémente l'interface'
  - Mot clé 'implements'

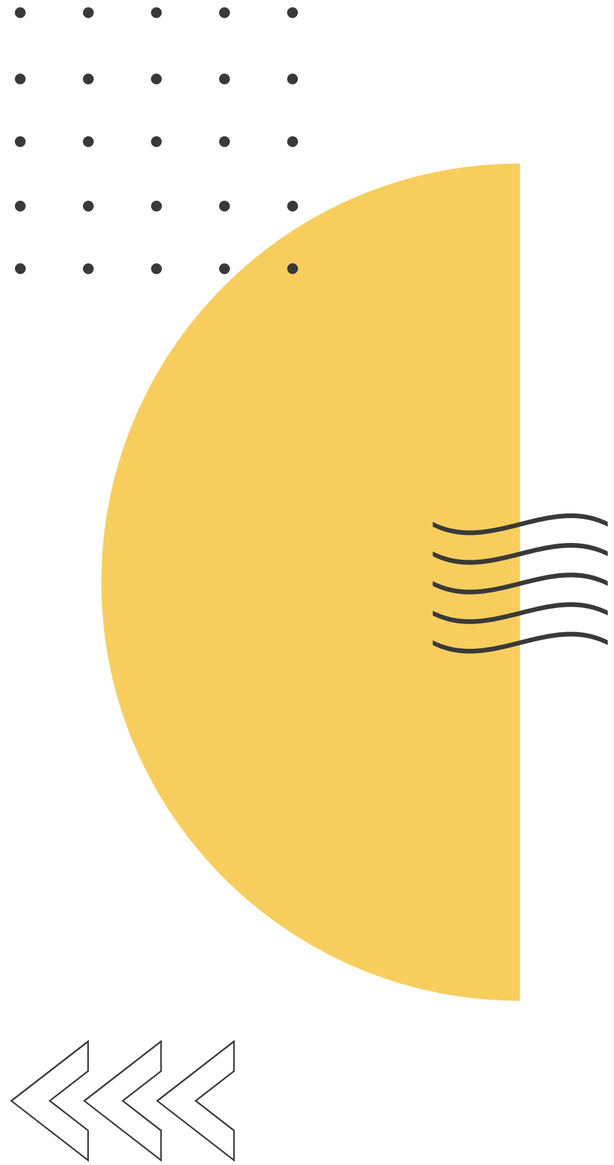
```
public class Employe implements IGestionConges
{
	public void poserConges() {
		//Fourniture d’une Implémentation en J ava
		//Redéfinition par rapport à la classe mère : polymorphisme
	}
	public void consulterConges() {
		// Fourniture d’une Implémentation en J ava
	}
}
```



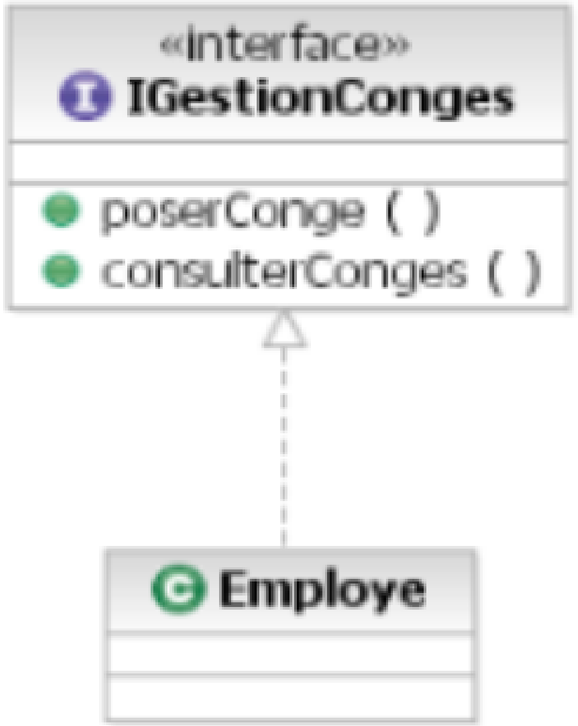
Implémentation

- Une classe peut implémenter plusieurs interfaces

```
public class Employe implements IGestionConges, Interface2, Interface3
{
	public void poserConges() {
		//Fourniture d'une Implémentation en Java
		//Redéfinition par rapport à la classe mère : polymorphisme
	}
	public void consulterConges() {
		// Fourniture d'une Implémentation en Java
	}
	// autres méthodes issues de I2, I3
}
```



Notation UML



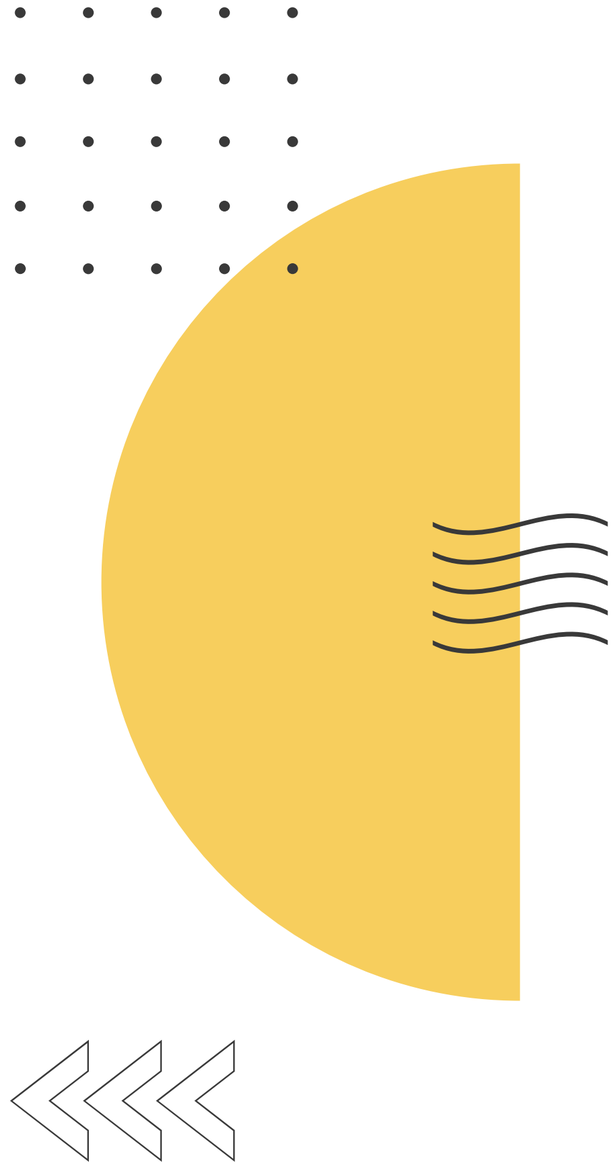
Ici , la classe Employe s'engage à fournir une implémentation de toutes les méthodes de

- l'interface IgestionConges.

Contrainte

Toute classe prétendant implémenter l'interface sera obligée par le compilateur de fournir des implémentations de ces méthodes.

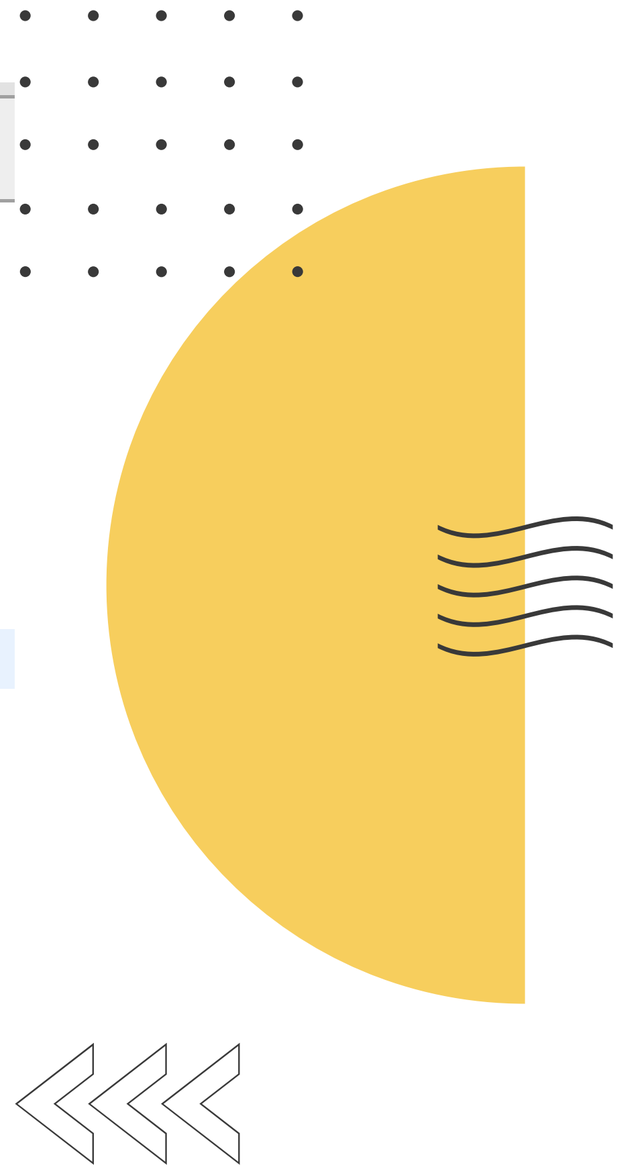
- Sinon le code ne compilera pas.



Exemple 01

IGestionConges.java

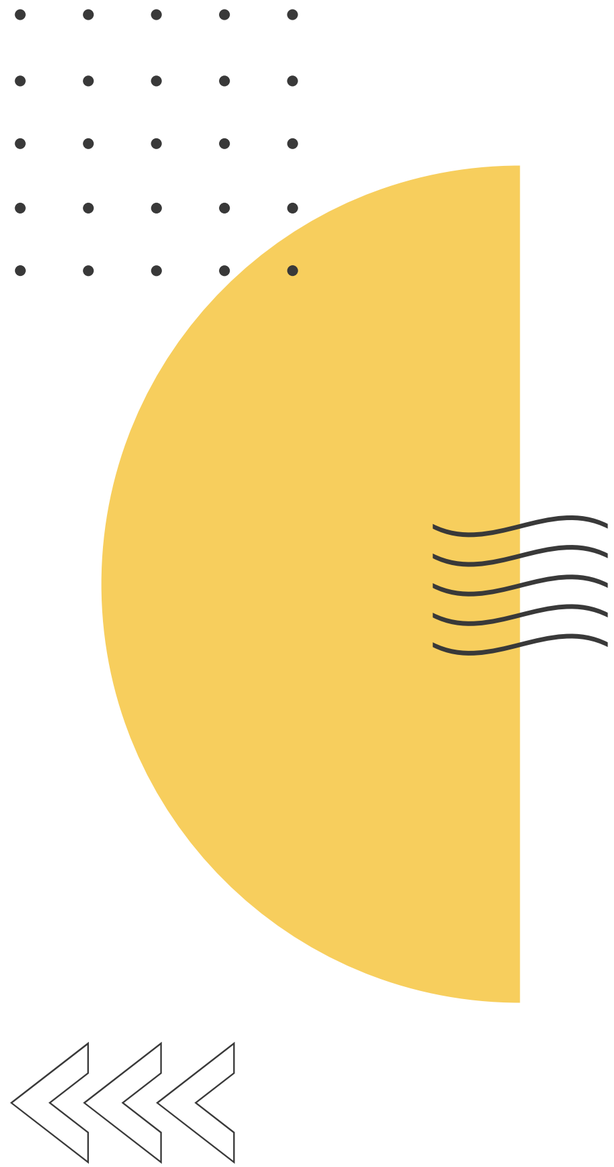
```java
package helpers;

public interface IGestionConges {

    // Declaration des signatures de methodes

    public void poserConges();

    public void consulterConges();

}
```



Exemple 02

IGestionConges.java | *Employe.java | Manager.java

```java
package domaine;

import helpers.IGestionConges;

public class Employe implements IGestionConges {

    @Override
    public void poserConges() {
        // TODO Auto-generated method stub

    }

    @Override
    public void consulterConges() {
        // TODO Auto-generated method stub

    }


}
```



10 - Classe abstraite

Une classe abstraite ne peut pas être instanciée. Il faudra l'étendre (héritage) et définir toutes les méthodes abstraites qu'elle

- contient pour pouvoir l'utiliser.

Une méthode abstraite, signalée par le modificateur abstract n'a alors qu'un prototype, c'est-à-dire son type de retour, suivi de son nom, suivi de la liste de ses paramètres entre

- des parenthèses, suivi d'un point-virgule.

On peut mélanger de l'abstrait avec du non abstrait ; dans la classe Forme figurent deux méthodes abstraites et une méthode non

- abstraite (coefficientEtalement()).

```
public abstract class Forme {
   public abstract float perimetre(); //methode abstraite
   public abstract float surface(); //methode abstraite

   public double coefficientEtalement() {
      double lePerimetre = perimetre();
      return 16 * surface() / (lePerimetre * lePerimetre);
   }
}
```

Merci