classe mère package MODEL, classe PERSONNAGE

```
package model;

public class Personnage {
    int lifePoints = 5;

    public void rencontrer(Personnage unPersonnage) {
        System.out.println("Salut le " + unPersonnage);
    }
}
```

classe fille1 package MODEL, classe MAGICIEN

```
package model;

public class Magicien extends Personnage {
    //METHODE HERITEE
    //méthode rencontrer est celle de sa classe mère Personnage
}
```

classe fille2 package MODEL, classe GUERRIER

```
package model;

public class Guerrier extends Personnage {
    private int lifePoints = 10;   //Shadowing -> un objet (une instance) de la classe Guerrier dispose de 2 attributs nommés LifePoints, celui de la classe guerrier(fille) et celui de la classe personnage (mère)
    //Le shadowing est peu utilisé car sujet à confusion mais peut parfois être utile dans des cas spécifique.

    //METHODE SPECIALISEE
    public void rencontrer(Personnage lePauvre) {   //Overriding -> redéfinition de la méthode = même nom, mais avec un corps et des paramètres qui peuvent être différents de la méthode de même nom de la classe mère

        //APPEL et APPLICATION DE LA METHODE HERITEE AU BESOIN AVEC super. AVANT D'EXECUTER LE CORPS DE LA METHODE SPECIALISEE
        super.rencontrer(lePauvre);
        //super.super est impossible comme syntaxe attention !
        //super est ici dans cet exemple pour appliquer une fois la méthode de la classe mère dans cette méthode de la classe fille redéfinie mais n'est aucunement indispensable : on peut redéfinir une méthode d'une classe fille sans utiliser super

        System.out.println("Bim ! Prend ça " + lePauvre);
    }

    public int getLifePoints() { //J'ai bien respecté l'encapsulation à savoir accès de l'extérieur à l'attribut privé par un accesseur public
        return lifePoints;
    }
}
```

classe principale package CONTROLLER, classe MAIN

```
package controller;

import model.Magicien;
import model.Guerrier;

public class Main {

    public static void main(String[] args) {
        Magicien merlin = new Magicien();
        Guerrier conan = new Guerrier();

        merlin.rencontrer(conan);
        conan.rencontrer(merlin); //Va appliquer une fois la méthode héritée puis une fois la méthode spécialisée (voir la classe guerrier)
        System.out.println(conan.getLifePoints()); //Va afficher 10 et non pas 5 car c'est bien l'attribut lifePoints de Guerrier et non de Personnage qui est utilisé à travers les méthodes de Guerrier
    }
}
```

Rappel du polymorphisme et des constructeurs hérités : exemple complet sur des methodes différentes dans les classes Guerrier et Magicien de leur classe mere Personnage

```
package model;

public class Personnage {
    private int lifePoints;
    private String Nom;

    public void rencontrer(Personnage p) {
        System.out.println("Bonjour !");
    }

    //Constructeur de la classe mère
    public Personnage(int lifePoints, String nm) {
        this.lifePoints = lifePoints;
        this.Nom = nm;
    }

    //Attention ! lifePoints est un attribut commun a tout les personnages -> les Getters/setters ne sont pas redéfinis dans les classe filles
    //Pq ? -> parce qu'un Guerrier est aussi un Personnage je peux donc lui appeler un getter de la Classe personnage !
    public int getLifePoints() {
        return lifePoints;
    }

    public void setLifePoints(int lifePoints) {
        this.lifePoints = lifePoints;
    }

    //Getter et Setter communs a tous les personnages
    public String getNom() {
        return Nom;
    }

    public void setNom(String nom) {
        Nom = nom;
    }
}


package model;

public class Guerrier extends Personnage {
    //attributs
    private String prenom;

    //METHODE SPECIALISEE -> Spécialisation
    public void rencontrer(Personnage p) {
        System.out.println("Boum");
    }

    //Constructeur + appel du constructeur de la classe mère
    public Guerrier(int lifePoints, String nm, String pn) { //pn nouveau paramètre propre à la classe dérivée, les deux premiers appartenants a la classe mère
        super(lifePoints, nm);
        this.prenom = pn;
    }

    //Getter
    public String getNom() {
        return prenom;
    }
    //Setter
    public void setNom(String pn) {
        this.prenom = pn;
    }
}


package model;

public class Magicien extends Personnage {
    //Pas d'attributs supplémentaires

    public Magicien(int lifePoints, String nm) {
        super(lifePoints, nm);
    }

    //Pas d'attributs propres = pas de Getters ni de Setters

    //METHODE HERITEE (invisible)
    //méthode rencontrer est celle de sa classe mère Personnage
}


package controller;
import model.*;

public class Main {

    public static void main(String[] args) {
        Guerrier g = new Guerrier(10, "helmut", "longs");
        Magicien v = new Magicien(5, "saul");
        uneRencontre(g, v); //On passe une méthode ou le type dans les paramètres des instances(objets) est celui d'une classe mère -> POLYMORPHISME !
    }

    static void uneRencontre(Personnage a, Personnage b) { //POLYMORPHISME
        System.out.print(a.getNom()); //PIEGE ! on ne pourra pas faire a.getPrenom, car ce getter n'est pas dans la classe personnage
        System.out.print(" rencontre ");
        System.out.println(b.getNom() + " ! ");
        a.rencontrer(b); //POLYMORPHISME -> Résolution dynamique des liens ici, la méthode rencontrer est fonction du type effectif et non du type de variable
        //On execute les méthodes spécialisées dans les classes, pas les génériques !
        //C'est bien la méthode rencontrer de la classe guerrier qui sera executée..
    }
}

package model;

public abstract class Personnage { //Notez bien le rajout de ABSTRACT
    private int lifePoints;
    private String Nom;

    public abstract void afficher(); //Remarquez bien qu'il n'y a pas de {} car pas de corps pour une méthode abstraite dans la classe ou elle est introduite

    public void rencontrer(Personnage p) {
        System.out.println("Bonjour !");
    }
}

package model;

import java.util.ArrayList;

import model.*;

public class Jeu {
    private ArrayList<Personnage> perso;

    //La problématique ci-dessous est la suivante je veux une classe afficher différente pour chaque personnage "fille", mais la classe mère elle Personnage je ne sais pas comment lui construire une méthode afficher car elle en a pas besoin

    public void afficher() {    //ATTENTION je ne sais pas afficher un personnage : je sais afficher un guerrier magicien etc mais pas un personnage générique
        for(Personnage unPerso : perso) {
            unPerso.afficher(); //ERREUR -> Une méthode afficher doit être affichée dans chaque classe personnage
                                //       -> Chaque sous classe doit afficher sa méthode spécifique
                                //Comment faire si on ne sait pas afficher un personnage générique, comment forcer la redéfinition dans les sous classes ?

                                //SOLUTION 1 (MAUVAISE) -> Dans la classe personnage une méthode VIDE afficher(){} -> MAUVAIS -> Affichage uncorrect on affiche du vide et si une sous classe ne redéfinie pas la méthode on a des personnages fantômes, et ça n'oblige pas la sous classe à redéfinir
                                //SOLUTION 2 (BONNE) -> Déclarer la méthode afficher comme abstraite dans une classe OBLIGATOIREMENT abstraite -> La classe personnage devient abstract
        }
    }

    public void ajouterPersonnage() {

    }
}


abstract class FigureFermee {
public abstract double Surface(); //(1)Aucun des attributs qui rentre en paramètre de cette méthode n'est connaissable dans la classe mere
public abstract double perimetre(); //IDEM

public double volume(double hauteur){//Je peux mettre une hauteur en parametre sur un appel de cette méthode sur une instance de FigureFermee -> je ferais un appel de cette forme -> objet.volume(30.0) (attention l'objet ne pourra pas etre une instance DIRECTE de FigureFermee mais d'une classe fille)
return hauteur*surface();
}

}

FigureFermee rel = new Rectangle();
```

Attention

```
FigureFermee rel = new FigureFermee(); //INTERDIT c'est une classe abstraite

package model;

public abstract class Personnage { //Notez bien le rajout de ABSTRACT
    private int lifePoints;
    private String Nom;

    public abstract void afficher(); //Remarquez bien qu'il n'y a pas de {} car pas de corps pour une méthode abstraite dans la classe ou elle est introduite

    public void rencontrer(Personnage p) {
        System.out.println("Bonjour !");
    }

    //Constructeur de la classe mère
    public Personnage(int lifePoints, String nm) {
        this.lifePoints = lifePoints;
        this.Nom = nm;
    }

    //Attention ! lifePoints est un attribut commun a tout les personnages -> les Getters/setters ne sont pas redéfinis dans les classe filles
    //Pq ? -> parce qu'un Guerrier est aussi un Personnage je peux donc lui appeler un getter de la Classe personnage !
```

```
    public int getLifePoints() {
        return lifePoints;
    }

    public void setLifePoints(int lifePoints) {
        this.lifePoints = lifePoints;
    }

    //Getter et Setter communs a tous les personnages
    public String getNom() {
        return Nom;
    }

    public void setNom(String nom) {
        Nom = nom;
    }
}

package model;

public class Guerrier extends Personnage {
    //attributs
    private String prenom;

    //ERREUR ICI -> afficher() de la classe mère PERSONNAGE doit etre obligatoirement DEFINIE sinon cette classe est abstraite

    //METHODE SPECIALISEE -> Spécialisation
    public void rencontrer(Personnage p) {
        System.out.println("Roum");
    }

    //Constructeur + appel du constructeur de la classe mère
    public Guerrier(int lifePoints, String nm, String pn) { //pn nouveau paramètre propre à la classe dérivée, la deux premiers appartenants a la classe mère
        super(lifePoints, nm);
        this.prenom = pn;
    }

    //Getter
    public String getNom() {
        return prenom;
    }
    //Setter
    public void setNom(String pn) {
        this.prenom = pn;
    }
}

package model;

public class Jeu {
    Jeu juegos = new Jeu();
    Personnage war = new Guerrier(10, "erze", "mide"); //FAUX -> OBLIGATOIRE de redefinir l'ensemble des méthodes abstraites de la classe personnage

    public void ajouterPersonnage() {

    }
}
```