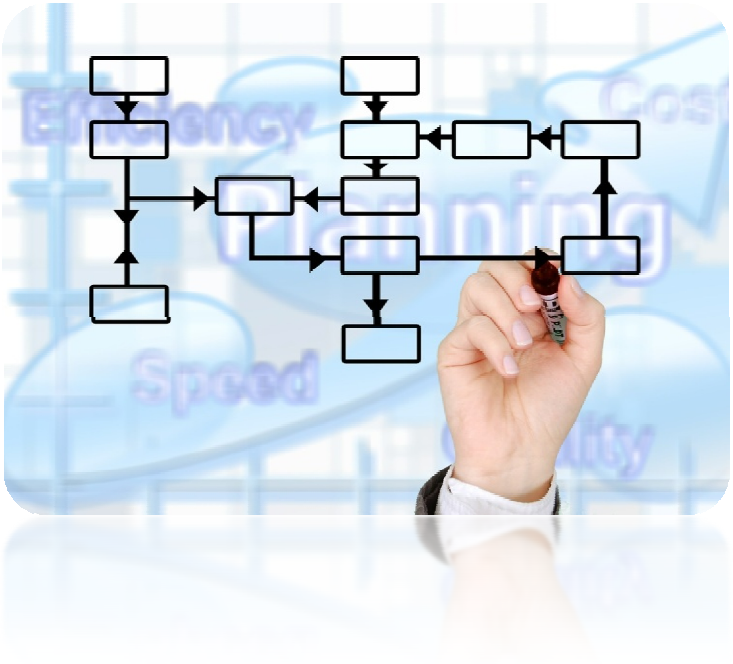
Introduction à la POO

Objectifs : comprendre la notion de classe et d'objet en programmation orientée objet.



1- La POO : généralités

2- Classes : attributs et méthodes

3- De la classe à l'objet

4- Créer une classe en Java

5- Encapsulation, getters et setters

6- Héritage : concept et exemple

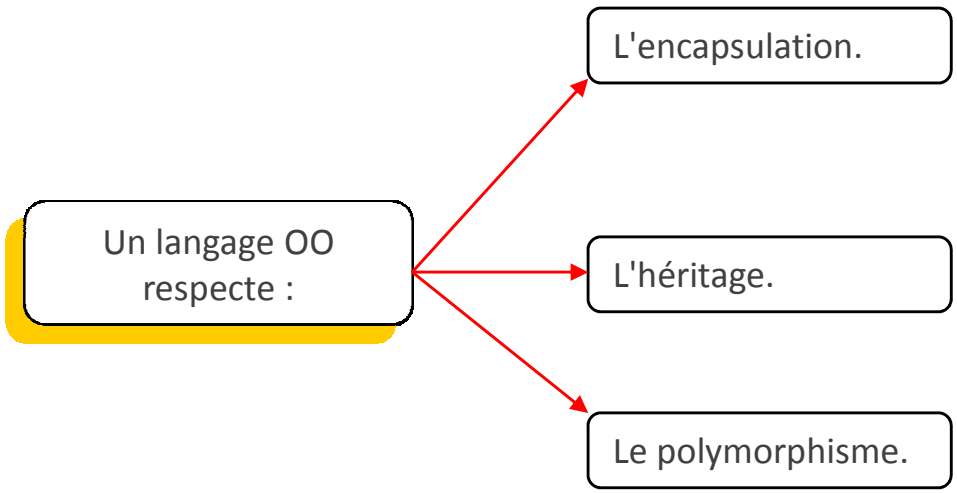
7- Algorithmique et POO

L'orientation objet met en œuvre plusieurs concepts.

1 Paradigme de la POO

Vous savez déjà écrire des programmes procéduraux **en Java**. Il faut donc maintenant des outils pour les ***organiser*** de **façon plus *efficace***. C'est l'un des objectifs de la notion d'***objet***.

Java respecte les 3 règles fondamentales de la POO.



L'encapsulation permet à un objet de protéger ses données de l'extérieur.

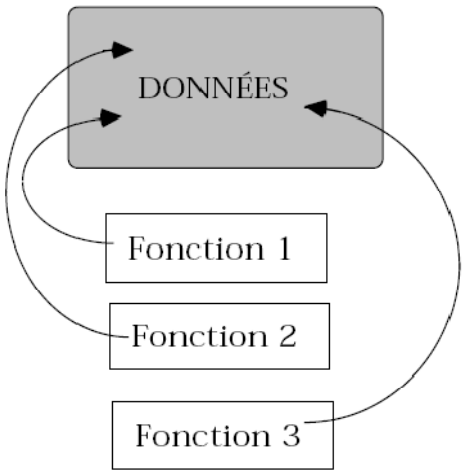
L'héritage est un mécanisme permettant d'étendre certaines propriétés et fonctions.

Le polymorphisme est un mécanisme permettant de gérer le contexte des objets.

En procédural, données et traitements sont séparés.

2 Programmation procédurale vs POO

En procédural, un **programme = suite d'instructions** exécutées par une machine. Son exécution = ces instructions agissent sur des données. Les notions de ***type de données*** et de ***fonctions*** sont **séparées**.



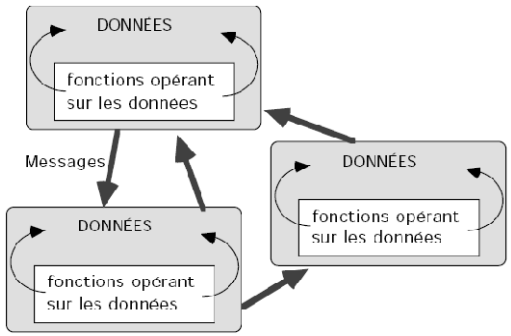
- Les fonctions et procédures travaillent ***"à distance"*** sur les données.
- Accent mis sur les actions. Il s'agit de répondre à la question: *Que veut on faire ?*
- Dissociation entre **données** et **fonctions :** problème lorsqu'on change les structures de données.

La POO vise à "responsabiliser" nos parties de programmes.

En POO, données et traitements sont liés.

2 Programmation procédurale vs POO : suite

On POO **une entité = un objet** qui prend en compte sa propre gestion (objet responsable).
L'exécution du programme = une collaboration entre entités pour résoudre le problème final. Les entités s'envoient des **messages**.



Il y a ***liaison entre données et traitements*** opérant sur ces données.

Exemple : modéliser un logiciel de trafic routier

- Les entités sont : - les feux tricolores - les carrefours - les véhicules - les agents de la circulation.

- Lorsqu'un feu tricolore passe au vert il envoie cette connaissance (= ce message) à l'agent posté à ce carrefour. L'agent prend une décision et en informe (envoi de messages) les chauffeurs des véhicules.

L'objet encapsule ses données et ses traitements.

3 POO : principe d'encapsulation

Le **principe d'*encapsulation*** consiste à **regrouper**, dans un même élément informatique, les aspects statique et dynamique (i.e. les données et les fonctions) spécifiques à une entité.

Cet élément informatique est appelé: **«objet»**.

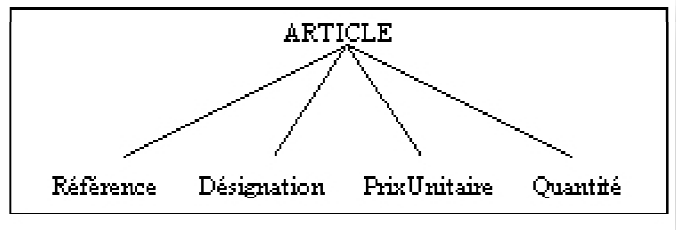
Capsule

- Les [structures de] données définies dans un objet sont appelées **les *attributs*** de l'objet.
- Les fonctions [de manipulation] définies dans un objet sont appelées **les *méthodes*** de l'objet.

OBJET = attributs + méthodes

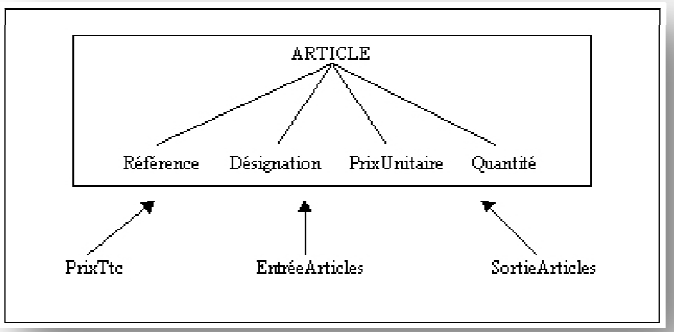
L'objet encapsule ses données et ses traitements.

3 POO : principe d'encapsulation : exemple



Un article en stock est un **« objet »** au sens courant du terme. Il est caractérisé par ses attributs.

Attributs :
Référence, Désignation, PrixUnitaire et Quantité sont les **attributs** de l'objet de type ARTICLE



De plus, un article est manipulé par des procédures ou ***méthodes*** possédant chacune un nom et des **paramètres d'appel et de retour**.

Méthodes :
PrixTtc() : calcule le prix TTC d'un article.
SortieArticle() : diminue la quantité en stock.
EntreeArticle() : augmente la quantité en stock

L'encapsulation, c'est la **fusion de méthodes et de données** ainsi réalisée. Elle permet aussi de définir différents niveaux de perception des données (***private***, ***public***, etc : voir plus loin).

Pour créer des objets, on utilise une classe.

1 Abstraction : vers la notion de classe

La notion d'objet doit permettre un certain degré d'abstraction. Soient des figures géométriques, l'objet **«rectangle»** ne sera intéressant que si l'on peut lui associer des propriétés générales, vraies pour l'ensemble des rectangles, et non pas uniquement pour un rectangle particulier.

Le processus d'abstraction consistera donc à identifier, pour un ensemble donné d'éléments:

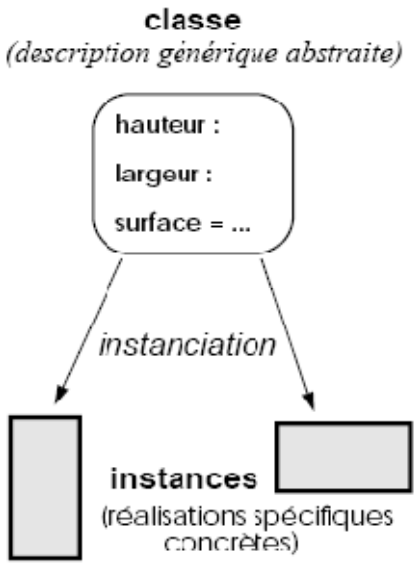
- des **caractérisations valides** pour la totalité de ces éléments;
- des **mécanismes communs** à la totalité de ces éléments.

Par exemple, les notions de **«largeur»** et **«hauteur»** sont des propriétés communes à l'ensemble des rectangles, c'est à dire au *concept* ***abstrait*** **«rectangle»**, et il en va de même pour la relation qui existe entre la **«surface»** d'un rectangle et ses dimensions (surface = largeur x hauteur).

Pour créer des objets, on utilise une classe.

2 Qu'est-ce qu'une classe ?

En programmation orientée objet, nous appelerons classe le résultat du processus d'abstraction précédemment décrit.

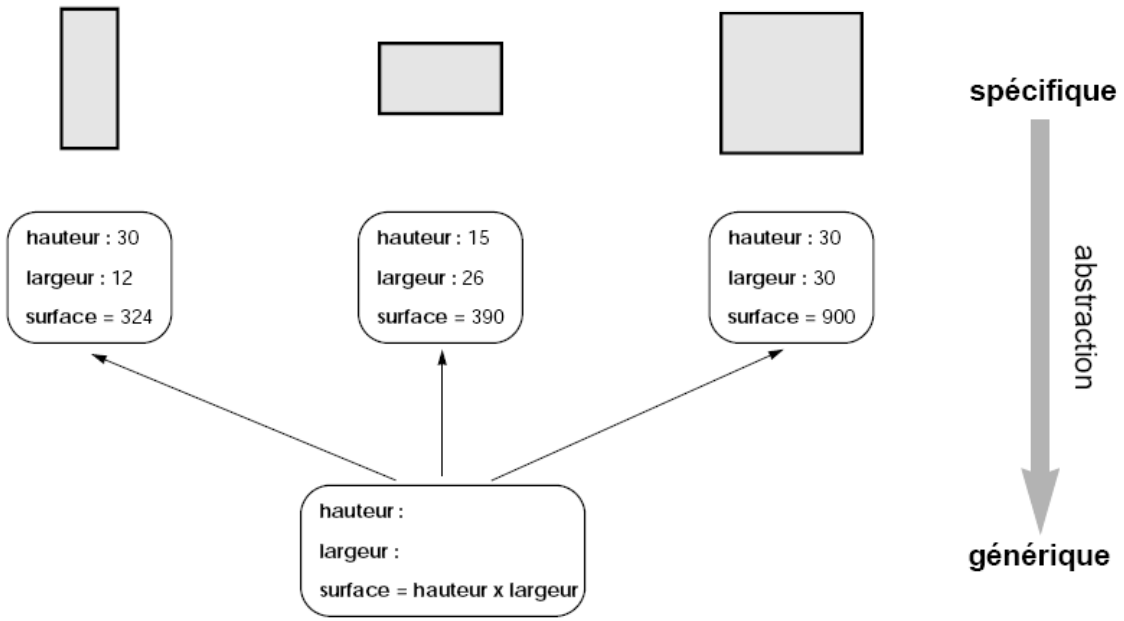


- Une «***classe***» = moule = patron **factorise les caractéristiques communes** des éléments qu'elle décrit pour la création des objets ou ***instances*** de la *classe*.
- Une classe est un **type** (complexe) et la «création» d'une instance se fait par le biais de la déclaration d'une variable du type représenté par la classe.
- Cette déclaration est aussi appelée ***instanciation*** et *l'instance* produite est donc une réalisation particulière de la classe.

Pour créer des objets, on utilise une classe.

3 Premier exemple de généralisation

D'un ensemble de figures particulières (carrés, rectangles), on définit un modèle générique (classe) rectangle dont les **attributs** sont la hauteur et la largeur, et la **méthode** la surface.



L'instanciation = créer un objet à partir d'une classe

4 Deuxième exemple de généralisation

Voici un autre exemple pour illustrer les notions d'objet et de classe. Nous considérons les objets réels que sont les chapeaux. Donnons-nous en un certain nombre.

Des objets chapeaux.

Couleur Taille Type

Les 3 objets ont les propriétés : couleur, taille et type. Ces propriétés caractérisent l'**état** d'un objet.

Tourner chapeau Porter chapeau Enlever chapeau

Un chapeau peut être porté, tourné ou enlevé. Ces actions sont appelées **comportements** de l'objet.

instanciation

généraliser

Chapeau
Couleur Taille Type
Tourner chapeau Porter chapeau Enlever chapeau

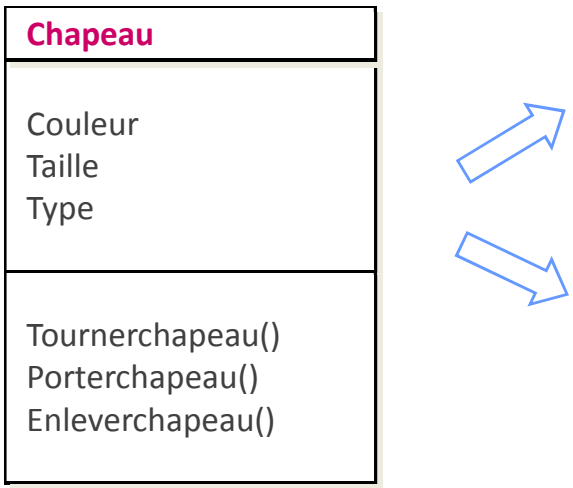
Classe Chapeau.

Classe = moule = modèle à partir duquel on peut créer un chapeau.

L'instanciation = créer un objet à partir d'une classe

5 Attributs et méthodes

Une classe sera caractérisée par ses **attributs ou variables** d'instances, et par ses **méthodes**. Attributs et méthodes forment les **membres** de la classe. Reprenons l'exemple de la classe **Chapeau** :



Classe Chapeau.

- **Les variables d'instance.**

  Les propriétés ***Couleur***, ***Taille*** et ***Type*** sont les **variables d'instance** de la classe. Elles caractérisent **l'état** d'un objet.

- **Les méthodes**

  Les actions ***Tournerchapeau()***, ***Porterchapeau()*** et ***Enleverchapeau()*** sont appelées **méthodes** de la classe. Elles caractérisent le **comportement** des objets de cette classe.

L'instanciation = créer un objet à partir d'une classe

1 Un objet est différent d'une classe

Il est important de bien saisir la différence entre les notions de classe et d'instance.

classe = *attributs* + *méthodes* + *mécanisme d'instanciation*

membres = *attributs* et *méthodes* . L'*instanciation* est le mécanisme qui permet de créer des instances dont les traits sont ceux décrits par la classe.

instance = valeurs d'*attributs* + accès aux *méthodes*

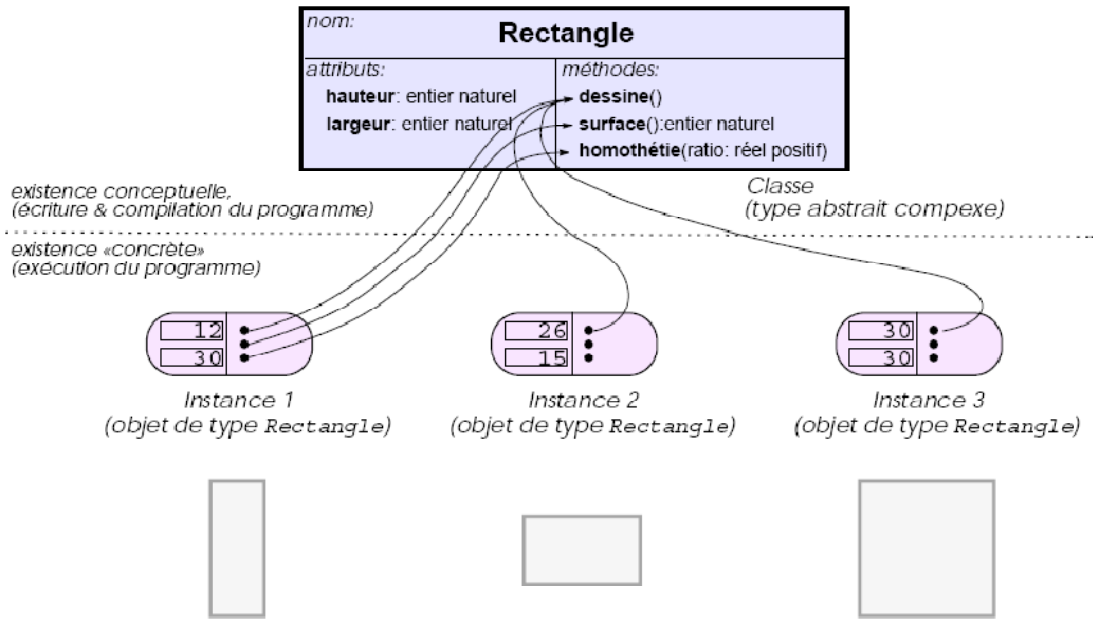
Les termes *instance* et *objet* sont pratiquement synonymes, d'où la reformulation de l'équation fondamentale de départ :

OBJET = [valeurs d']attributs + [accès aux] méthodes

L'instanciation = créer un objet à partir d'une classe

2 Exemple illustratif de l'instanciation

De la classe **Rectangle**, on crée quelques objets.



La classe User

1 Illustration : la classe User

On veut créer une classe **User** contenant deux attributs **nom** et **age** et une méthode ***getInfos()***. La classe est représentée sous la forme de diagramme UML (Unified Modeling Language) simplifié ci-dessous.

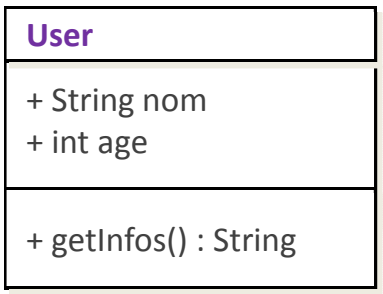


Diagramme UML simplifié

- Nous allons écrire le code de la classe **User** en Java, en mettant en place le fichier **User.java**.
- Créons ensuite un fichier de test nommé **TestUser.java**. C'est dans ce fichier que nous allons instancier la classe **User**.
- Compilons ces deux fichiers, et exécutons le fichier de test.

La classe User

2 Créer le fichier User.java

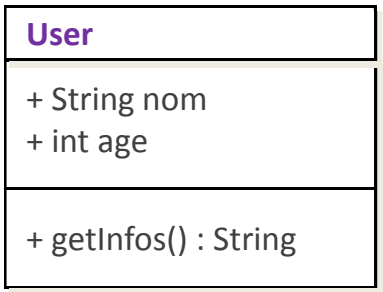


Diagramme UML simplifié

- Tapez le code ci-dessous.
- Enregistrez-le sous **User.java**.

```
public class User {

    // Attributs publics (= données)
    public int age;
    public String nom;

    // Une méthode
    public String getInfos() {
        return "Je suis " + this.nom + " et j'ai " + this.age + " ans.";
    }
}
```

Fichier User.java

La classe User

3 Créer le fichier TestUser.java

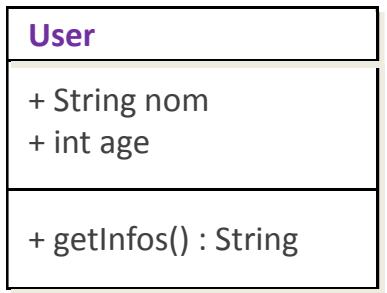


Diagramme UML simplifié

- Tapez le code ci-contre.
- Enregistrez-le sous **TestUser.java**.
- Compilez **User.java** et **TestUser.java** et exécuter.

```java
public class TestUser {

    // Méthode de lancement
    public static void main(String[] args) {

        // Créer un objet (utilisateur) = instancier
        User tintin  = new User();

        // Stocker dedans "TINTIN" et 40
        tintin.nom = "TINTIN";
        tintin.age = 40;


        // Afficher : nom, age
        System.out.println("Je suis : " + tintin.nom);
        System.out.println("Et j'ai : " + tintin.age + " ans.");

        // Afficher les infos
        System.out.println(tintin.getInfos());
    }
}
```

Fichier TestUser.java

La classe User

4 Modifiez User.java : ajout d'un constructeur

User
+ String nom + int age
+ getInfos() : String

Diagramme UML simplifié

- Modifiez **User.java** en ajoutant un constructeur.
- Enregistrez les modifications.

Ajoutez-ceci

```
public class User {

    // Attributs publics (= données)
    public int age;
    public String nom;

    // Un constructeur
    public User(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // Une méthode
    public String getInfos() {
        return "Je suis " + this.nom + "et j'ai " + this.age + " ans.";
    }
}
```

Fichier User.java

La classe User

5 Modifiez le fichier TestUser.java

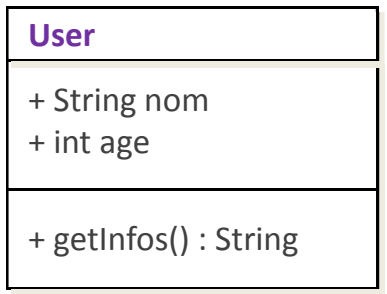


Diagramme UML simplifié

- Modifiez le code comme ci-contre.
- Enregistrez les modifications.
- Compilez **User.java** et **TestUser.java** et exécuter.

```java
public class TestUser {

    // Méthode de lancement
    public static void main(String[] args) {

        // Créer un objet : constructeur personnalisé
        User tintin  = new User("TINTIN", 50);

        // Afficher : nom, age
        System.out.println("Je suis : " + tintin.nom);
        System.out.println("Et j'ai : " + tintin.age + " ans.");

        // Afficher les infos
        System.out.println(tintin.getInfos());
    }
}
```

Fichier TestUser.java

La classe User

6 Modifiez User.java : données statiques

Un attribut est statique si on n'a nul besoin d'une instance pour l'utiliser. De la même manière, une méthode est statique si on n'a nul besoin d'une instance pour l'utiliser.

- Modifiez **User.java** en ajoutant des données statiques.
- Enregistrez les modifications.

Ajoutez-ceci

```
public class User {

    // Attributs publics (= données)
    public int age;
    public String nom;

    // Attribut statique ( = de classe)
    static int dpt = 77;

    // Un constructeur
    public User(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // Une méthode
    public String getInfos() {
        return "Je suis " + this.nom + " et j'ai " + this.age + " ans.";
    }

    // Une méthode statique (= de classe)
    public static String afficheDpt() {
        return "Departement: " + User.dpt;
    }
}
```

Fichier User.java

La classe User

7 Modifiez le fichier TestUser.java : données statiques

Pour atteindre un *attribut* statique (ou pour utiliser une *méthode* statique), il suffit de le (la) préfixer avec le nom de la classe : ***User.dpt*** et ***User.afficheDpt()*** respectivement.

- Modifiez **TestUser.java** en ajoutant le code ci-contre.
- Enregistrez les modifications.
- Compilez **User.java** et **TestUser.java** et exécuter.

Ajoutez-ceci →

```
public class TestUser {

    // Méthode de lancement
    public static void main(String[] args) {

        // Créer un objet : constructeur personnalisé
        User tintin  = new User("TINTIN", 50);

        // Afficher : nom, age
        System.out.println("Je suis : " + tintin.nom);
        System.out.println("Et j'ai : " + tintin.age + " ans.");

        // Afficher les infos
        System.out.println(tintin.getInfos());

        // Utiliser un attribut et une méthode statiques
        System.out.println(User.afficheDpt());
        System.out.println("Numero dpt :" + User.dpt);
    }
}
```

Fichier TestUser.java

Protéger les données

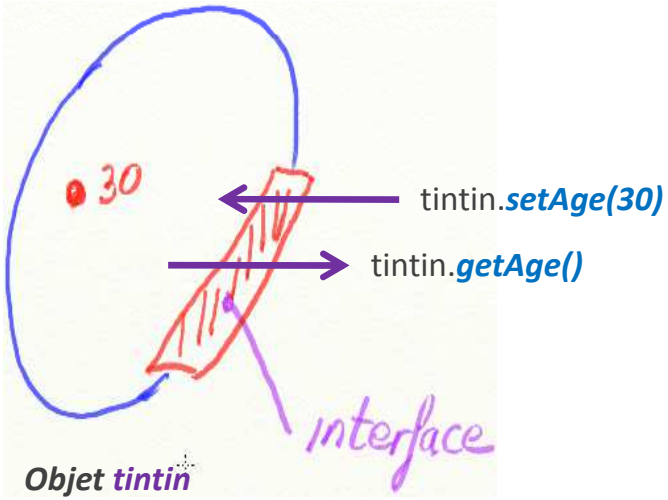
1 Protection des données

L'objet peut protéger ses données en les rendant privées (mot clé **private**). Dans ce cas, elles sont accessibles dans la classe mais inaccessibles en dehors.

- Pour les rendre accessibles à l'extérieur, la classe se munit d'une interface (=ensemble de fonctions) composée de méthodes publiques **accesseur (=getters)** et **modifieur (=setters**) à travers lesquelles les données pourront être modifiées ou lues.

- Les setters permettent de **valider les données** selon des règles métiers éventuelles.

- Les méthodes peuvent aussi être protégées.



Protéger les données

2 Exemple : modifiez User.java

Rendez maintenant l'âge privé dans la classe **User** et créez les méthodes ***getAge()*** pour **l'accesseur** et ***setAge(.)*** pour le **modifieur**.

- Modifiez **User.java** en utilisant l'encapsulation.
- Enregistrez les modifications.

User
+ String nom - int age
+ getAge() : int + setAge(int age) : void

Diagramme

Ajoutez-ceci

```java
public class User {

    // Attributs
    private int age;
    public String nom;

    // Un constructeur
    public User(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // Un getter
    public int getAge() {
        return this.age;
    }

    // Un setter
    public void setAge(int age) {
        if (age > 0){
            this.age = age;
        }
    }
}
```

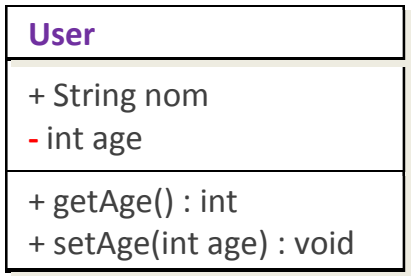
Fichier User.java

Protéger les données

3 Exemple : modifiez TestUser.java

Rendez maintenant l'âge privé dans la classe **User** et créez les méthodes ***getAge()*** pour **l'accesseur** et ***setAge(.)*** pour le **modifieur**.

- Modifiez **TestUser.java** en utilisant l'accesseur et le modifieur.
- Enregistrez les modifications.



Diagramme

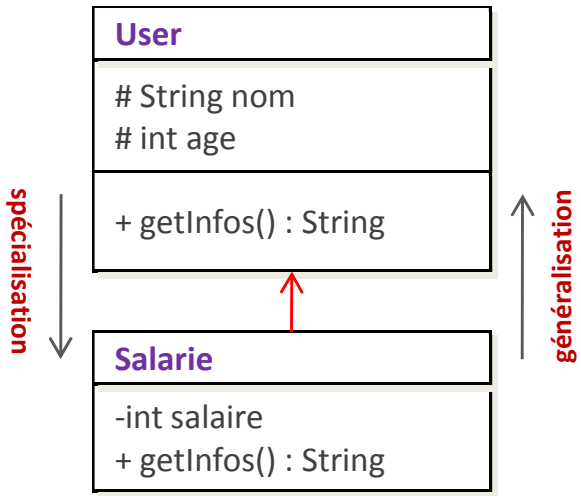
```java
public class TestUser {

    // Méthode de lancement
    public static void main(String[] args) {

        // Créer un objet : constructeur personnalisé
        User tintin  = new User("TINTIN", 50);

        // Modifier les données
        // tintin.age = 30; // Erreur compilation: age est private
        tintin.setAge(30);
        tintin.nom = "TOTO";

        // Affichage : utiliser un getter
        System.out.println("Je suis : " + tintin.nom);
        System.out.println("Et j'ai : " + tintin.getAge() + " ans.");
    }
}
```

Fichier TestUser.java

Père et fils

1 Notion d'héritage : définition

L'héritage est un procédé permettant de réutiliser du code. Une classe **Père** peut avoir une classe **Fille**. Si on peut mettre en évidence dans le cahier des charges une relation de type **« Fille » EST UNE SORTE** de **« Père »**, alors on met en évidence l'héritage.



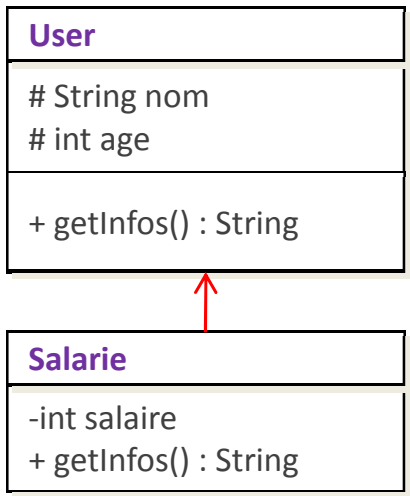
Diagramme

- Héritage : **« Salarie » EST UNE SORTE** de **« User »**.
- **« Salarie »** hérite des attributs et méthodes de **« User »**.
- Les attributs **protected** (signe **#**) sont disponibles dans **« Salarie »**.
- L'attribut **salaire** est propre à la classe Fille **« Salarie »**.

Père et fils

2 Notion d'héritage : exemple

On veut créer une classe **User** contenant deux attributs protégés **nom** et **age** et une méthode ***getInfos()***. Puis la classe **Salarié** ayant l'attribut privé **salaire** et la méthode ***getInfos()***. On va aussi générer des getters et setters.

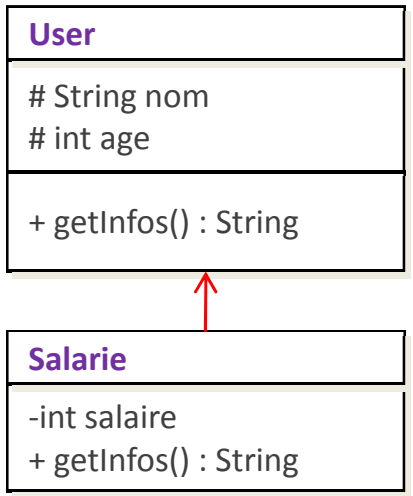


Diagramme

- Ecrire le code de la classe **User** dans **User.java** et celui de **Salarie** dans **Salarie.java**.
- Créer le fichier de test **TestUser.java**. C'est dans ce fichier que nous allons instancier les 2 classes.
- Compilez ces 3 fichiers, et exécutez.

Père et fils

3 Héritage : le fichier User.java



Diagramme

- Tapez le code ci-contre.
- Enregistrez le sous **User.java**.

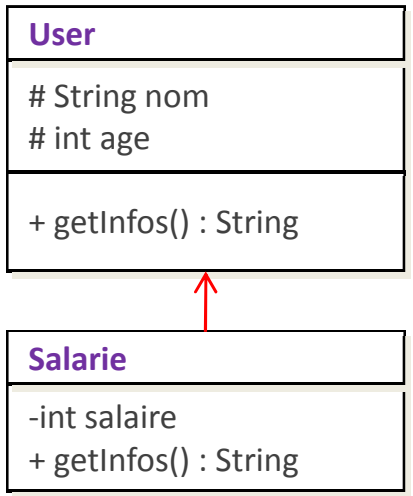
Attributs protégés

```
public class User {

    // Attributs
    protected int age;
    protected String nom;

    // Un constructeur
    public User(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // Getters
    public int getAge() {return this.age;}
    public String getNom() {return this.nom;}

    // Setters
    public void setAge(int age) {this.age = age;}
    public void setNom(String nom) {this.nom = nom;}

    // Une méthode
    public String getInfos() {
        return "Je suis " + this.nom + ", j'ai " + this.age + " ans";
    }
}
```

Fichier User.java

Père et fils

4 Héritage : le fichier Salarie.java



Diagramme

- Tapez le code ci-contre.
- Enregistrez le sous **Salarie.java**.

Héritage

Sollicite le constructeur du parent

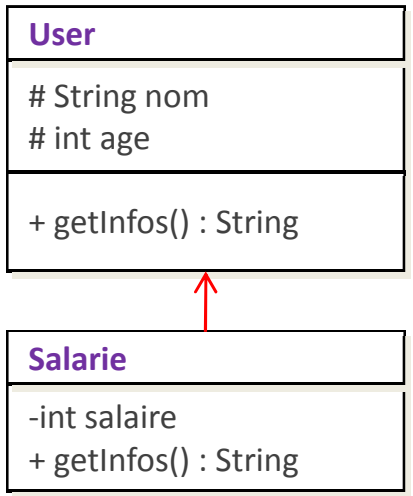
```
public class Salarie extends User {

    // Attributs
    private int salaire;

    // Un constructeur
    public Salarie(String nom, int age, int salaire) {
        // Appel du constructeur parent
        super(nom, age);

        this.salaire = salaire;
    }

    // Getter & setter
    public int getSalaire() {return this.salaire;}
    public void setSalaire(int salaire) {this.salaire = salaire;}

    // Une méthode
    public String getInfos() {
        return super.getInfos() + " et je gagne: " + this.salaire;
    }
}
```

Fichier Salarie.java

getInfos() du parent.

Père et fils

5 Héritage : le fichier TestUser.java



Diagramme

- Créer testUser.java (code ci-contre).
- Compilez **User.java**, **Salarie.java** et **TestUser.java**. Exécutez.

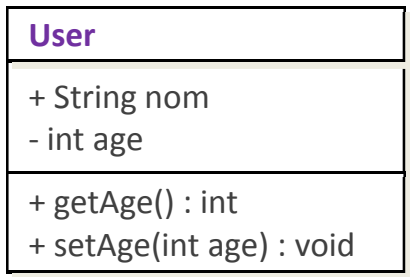
setAge(.) et ***getNom()*** du parent.

```
public class TestUser {

    // Méthode de lancement
    public static void main(String[] args) {

        // Créer un salarié
        Salarie dupont = new Salarie("DUPONT", 46, 3500);

        // Modifier les données
        dupont.setAge(30);
        dupont.setSalaire(8000);

        // Affichage : infos, nom, salaire
        System.out.println(dupont.getInfos());
        System.out.println("Nom :" + dupont.getNom());
        System.out.println("Salaire :" + dupont.getSalaire());
    }
}
```

Fichier TestUser.java

Algo vers classe

1 Mise en algorithme d'une classe



Diagramme

Nous partons des classes **User** et **TestUser** qui nous ont permis d'illustrer la POO pour créer le pseudo code d'une classe.

- Ecrire l'algorithme pour la classe **User**.

- Ecrire l'algorithme pour la classe **TestUser** (programme principal).

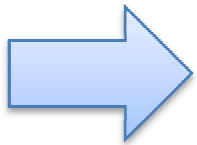
Algo vers classe

1 Mise en algorithme d'une classe : suite

Une classe est un type particulier.

```
TYPE
CLASSE User
    attributs
        nom : chaine
        age : entier

    methodes
        ! Constructeur
        PROCEDURE User(chaine: nom, entier : age)
        DEBUT
            this.nom <- nom
            .....
        FIN-PROC

        FONCTION getAge(): entier
        DEBUT
           .....
           Retourne this.age
        FIN-PROC

        PROCEDURE setAge(entier:age)
        DEBUT
        .....
        FIN-PROC
FIN-CLASSE
```

Algorithme : classe User

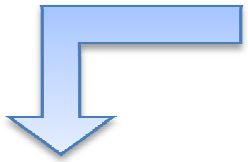


```java
public class User {

    // Attributs
    private int age;
    public String nom;

    // Un constructeur
    public User(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    // Un getter
    public int getAge() {
        return this.age;
    }

    // Un setter
    public void setAge(int age) {
        if (age > 0){
            this.age = age;
        }
    }
}
```

Fichier User.java

Algo vers classe

2 Test de la classe User



```
! Programme principal
PROGRAMME TESTUSER

VAR tintin : User

DEBUT
    ! Modifier des attributs
    tintin.nom <- 'TINTIN'

    ! Accéder aux attributs
    ECRIRE tintin.nom

    ! Appel de méthodes
    nom <- tintin.getNom()
    tintin.setNom('MARTIN')
FIN
```

Programme principal

```
public class TestUser {

    // Méthode de lancement
    public static void main(String[] args) {

        // Créer un objet : constructeur personnalisé
        User tintin  = new User("TINTIN", 50);

        // Modifier les données
        // tintin.age = 30; // Erreur compilation: age est private
        tintin.setAge(30);
        tintin.nom = "TOTO";

        // Affichage : utiliser un getter
        System.out.println("Je suis : " + tintin.nom);
        System.out.println("Et j'ai : " + tintin.getAge() + " ans.");
    }
}
```

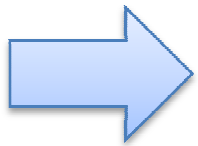
Fichier TestUser.java

Algo vers classe

3 Une autre écriture

On peut décrire une méthode hors de la définition de la classe. Il suffira d'utiliser la syntaxe **NomClasse:: nomMethode()**. On pourra ainsi gagner en clarté pour les algorithmes de méthodes un peu longues.

```
CLASSE User
    attributs
        ....

    methodes
        ....
        FONCTION getNom(): chaine
FIN-CLASSE
```



```
FONCTION User::getNom(): chaine
DEBUT
    ......
    RETOURNE  this.nom
FIN-FONCTION
```

Détails de la méthode

Nota :
On pourra écrire **Salarie HERITE DE User** pour l'héritage.