
JAV – TD 3

bibliothèques standards
composition
communication entre objets
surcharge

Les tableaux

- Déclaration

```
int[] array_of_int; // équivalent à : int array_of_int[];  
Color rgb_cube[][][];
```

- Création et initialisation

```
array_of_int = new int[42];  
rgb_cube = new Color[256][256][256];  
int[] primes = {1, 2, 3, 5, 7, 7+4};  
array_of_int[0] = 3;
```

- Utilisation

```
int l = array_of_int.length; // l = 42  
int e = array_of_int[40];
```

Bibliothèques : d'où viennent les objets, les classes?

- Les objets n'apparaissent que
 - s'ils sont fabriqués par quelque chose...
 - et si l'on a obtenu une référence dessus (appel de méthode par exemple)
- Les classes
 - Soit on les écrit nous-même
 - Soit on nous les fournit (bibliothèque)
 - Java fournit plus de 10 000 classes dans le kit de base (bibliothèque standard)
 - La difficulté est de connaître leur existence

Bibliothèques standards

- Le Java Development Kit (JDK) comporte énormément de classes prédéfinies
 - plus de 10 000 éléments dans l'API (Application Programming Interface)
 - 1 archive (jdkxxx.jar) contient toutes les classes
- => organisation hiérarchique des classes en paquetages (packages)
 - correspondance directe package / répertoire

API du JDK

<http://tc-net.insa-lyon.fr>

java.lang : classes de bases (+reflect)

java.io : entrées/sorties

java.util : utilitaires (structures, dates, events) (+zip)

java.net : réseau

java.applet : gestion des applets

java.awt : interface graphique (image, +datatransfert, +event)

java.beans : définition de composants réutilisables

java.math : entier de taille variable

java.rmi : invocation distante (+dgc, +registry, +server)

java.security : (+acl, +interfaces)

java.sql : dialogue avec une base de données

java.text : traduction, chaine=f(langue)

etc...

Composition

- Une classe est un agrégat d'attributs et de méthodes : les *membres*
- **COMPOSITION** : quand un objet contient un **attribut** d'une autre classe

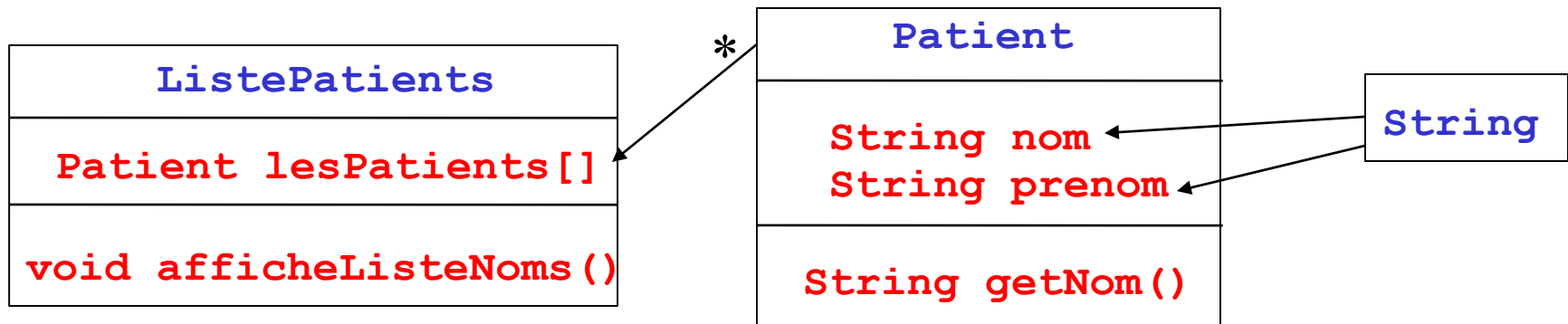
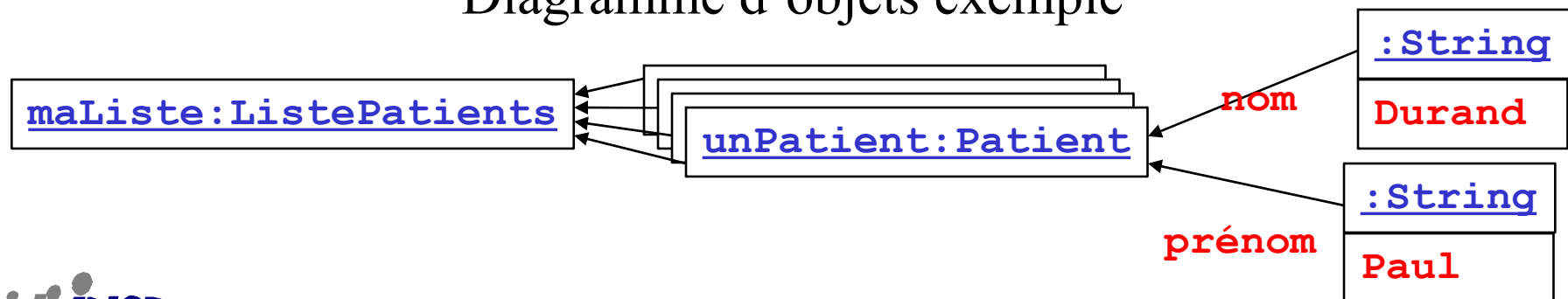


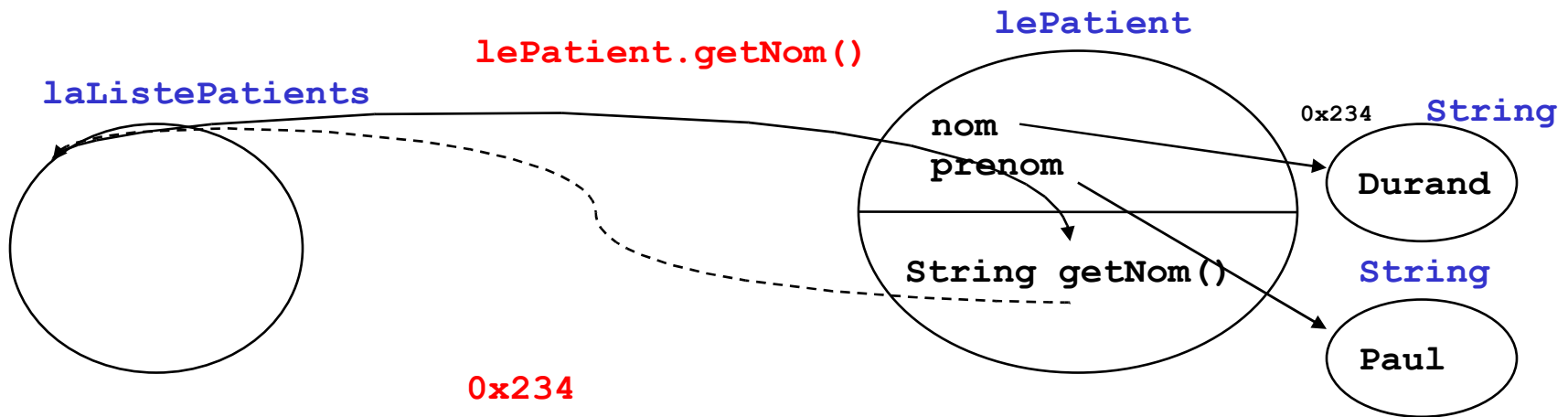
Diagramme de classes

Diagramme d'objets exemple



Communication entre objets

- message=appel de fonction d'un objet sur un autre
 - on dit « Invoquer une méthode »
 - ==> Unique manière de communiquer entre objets
- **Syntaxe** : `référenceObjet.nomMethode(parametres)` ;



Référence et this

- Pour pouvoir accéder à un objet, il faut avoir une référence sur cet objet (attribut ou variable)
 - La référence est le nom qu'on lui donne localement
- Si on veut se parler à soi-même, on utilise « this »

```
//suite class Circle
public void afficheToi () {
    System.out.println(
        "Je suis un cercle de surface"+this.getArea() ;
    }
}
```

- NB : les méthodes statiques peuvent-elles accéder à `this` ?

Synthèse communication

- Aucune instruction en dehors d'une classe
- Toute la communication se fait par échange de messages entre les objets

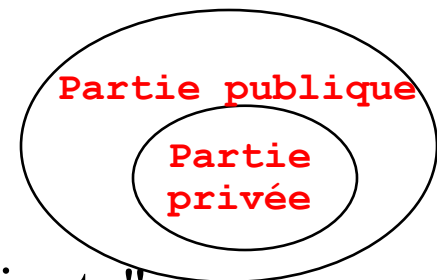
```
public class MaClasse{  
  
    int i=12;  
    String toto="Bonjour";  
  
    public static void main(String [] arg){  
        MaClasse unRepresentant=new MaClasse();  
        System.out.println("Je dis : ");  
        System.out.println(unRepresentant.disBonjour());  
    }  
  
    public String disBonjour(){return toto;}  
}
```

Exo tableaux, composition, messages

- Créer une classe `TableauCircles` qui a
 - un attribut tableau de 10 objets de la classe `Circle` faite en TD1
 - des méthodes `addCircle`, `deleteCircle` et `AfficheCircles`
 - compiler, déboguer
- Faire une classe `MonProgTableauCircle` qui contient une méthode `main` qui:
 - remplit le tableau avec 5 cercles
 - prenez des $x/y/r$ en dur
 - demande l’affichage de l’ensemble des cercles
 - compiler, déboguer, lancer

Encapsulation

- Ne rendre visibles de l'extérieur que certaines parties de l'objet
 - L'utilisateur d'un objet n'accède qu'à sa partie publique. L'autre partie est dite privée.
- Intérêts
 - une classe fournit un ensemble de services. C'est son problème de choisir comment elle fait : les clients n'ont pas besoin de connaître les détails => lisibilité du code
 - Modification de la partie privée, tout en gardant la même définition publique
- D'une manière générale
 - Tous les attributs d'une classe sont "private"
 - Certaines méthodes sont "public", d'autres "private"



Exemple encapsulation

```
public class ClientWeb {  
  
    private void connect(String unServer){  
        /* ouvrir une connexion reseau */  
    }  
  
    private String getDocument(){  
        /* Demander au serveur la page */  
    }  
  
    public String getPageWeb(String server){  
        this.connect(server);  
        String tmp=this.getDocument();  
        return tmp;  
    }  
}
```

ClientWeb

- void connect(String)
- String getDocument()
- + String getPageWeb(String)

Exo encapsulation

- Reprendre la classe TableauCircles
 - ajouter un attribut private qui compte le nombre d'éléments dans le tableau et passez l'attribut tableau public (c'est interdit mais c'est pour la bonne cause)
 - modifier les fonctions addCircle, deleteCircle et AfficheCircles en conséquence
- Reprendre la classe MonProgTableauCircle
 - modifier son main pour ajouter elle-meme un objet Circle dans le tableau de TableauCircle
 - tester, expliquer les pb...

Surcharge de méthode

- La surcharge (overloading) permet à plusieurs méthodes ou constructeurs de partager le même nom.
- Pour que deux méthodes soient en surcharge il faut
 - qu'elles aient le même nom,
 - mais une signature différente (paramètres de la méthode)

Surcharge exemple

```
class Point {
    private double x,y;
    public Point (){ this.x=0.0; this.y=0.0;}
    public Point (double x, double y){this.x=x; this.y=y}
    //Calcule la distance entre moi et un autre point
    public double distance(Point autre){
        double dx=this.x-autre.getX();
        double dy=this.y-autre.getY();
        return Math.sqrt(dx*dx+dy*dy);
    }
    //Calcule la distance entre moi et une autre coordonnée
    public double distance(double x, double y){
        double dx=this.x-x;
        double dy=this.y-y;
        return Math.sqrt(dx*dx+dy*dy);
    }
    //Calcule la distance entre moi et une autre coordonnée
    public double distance(int x, int y){
        double dx=this.x-(double)x;
        double dy=this.y-(double)y;
        return Math.sqrt(dx*dx+dy*dy);
    }
    //Calcule la distance entre moi et l'origine
    public double distance(){
        return Math.sqrt(x*x+y*y);}}}
```

Surcharge exemple

- Appel
 - Quand une méthode est appelée, le nombre et le type des arguments permettent de définir la signature de la méthode qui sera invoquée.

```
Point p1=new Point();  
Point p2=new Point(20.0, 30.0);  
p2.distance(p1);  
p2.distance(50.0, 60.0);  
p2.distance(50, 60);  
p2.distance();
```


Passage de paramètres dans les méthodes

- Le mode de passage des paramètres dans les méthodes dépend de la nature des paramètres :
 - par *référence* pour les objets
 - les modifications faites par la méthode sont faites sur l'objet et sont donc persistantes
 - par *copie* pour les types primitifs
 - les modifications faites par la méthode sont faites sur la copie et ne sont donc pas visibles par l'appelante

```
public class C { //à taper , tester, valider!!
    public void methode1(int i, StringBuffer s) { i++; s.append("d"); }

    public void methode2() {
        int i = 0;
        StringBuffer s = new StringBuffer("abc");
        methode1(i, s);
        System.out.println("i=" + i + ", s=" + s); // i=0, s=abcd
    }
}
```

découverte de la méthode toString()

- Ecrire une classe Patient
 - avec des attributs String nom et prénom
- Ecrire une classe Client
 - avec une méthode main
 - avec une variable unPatient (Paul Durand)
 - que donne la ligne `System.out.println(unPatient);`
 - lancer, conclusion? Voir la documentation sur `println!`
- Ajouter une méthode dans la classe Patient
 - `public String toString()` qui retourne une chaîne comportant nom et prénom.
 - Relancer le client. Conclusion?