

M2103 (POO)



# Bases de la programmation orientée objet

**Responsable : Cristina Onete**

cristina.onete@gmail.com

<https://onete.net/teaching.html>

# Dans les cours précédents

- ▶ Classe et objet
  - ▶ Attributs, méthodes, méthode principale (main)
  - ▶ Méthodes spéciales : toString(), equals()
- ▶ La programmation de base
  - ▶ Les types de variables
  - ▶ Les tableaux
  - ▶ Les instructions de base
  - ▶ Les boucles : while et for

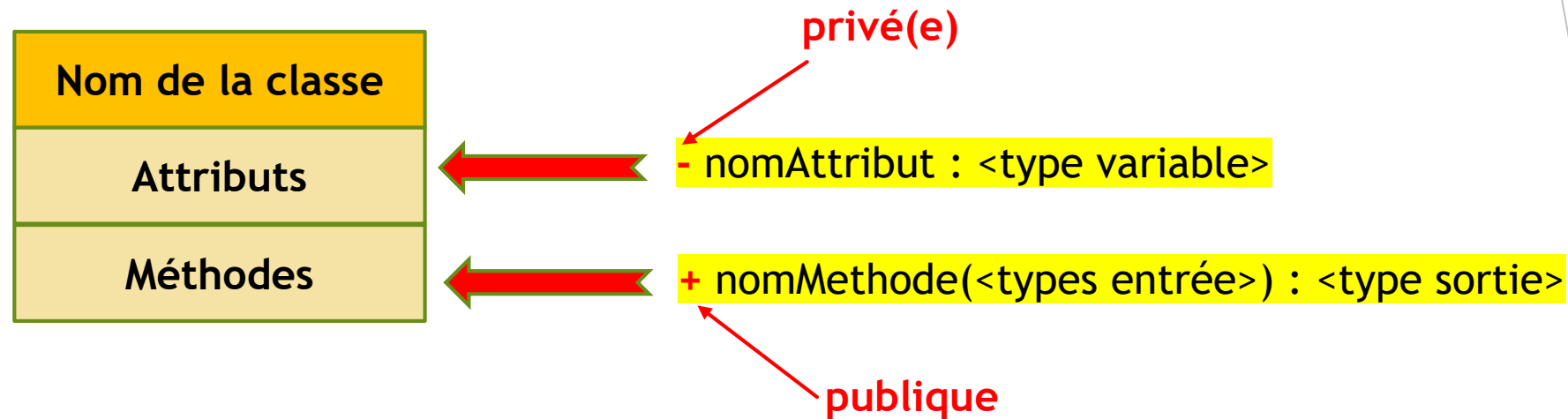
Aujourd'hui : diagrammes de classe, héritage, polymorphisme, composition

# Les diagrammes de classe



# Une façon de représenter une classe

- ▶ Une diagramme de classe décrit les attributs et méthodes d'une classe



- ▶ Pourquoi voudrions-nous utiliser une telle representation ?

Pour avoir un recul sur le code

Pour bien planifier son code

Pour bien spécifier son code

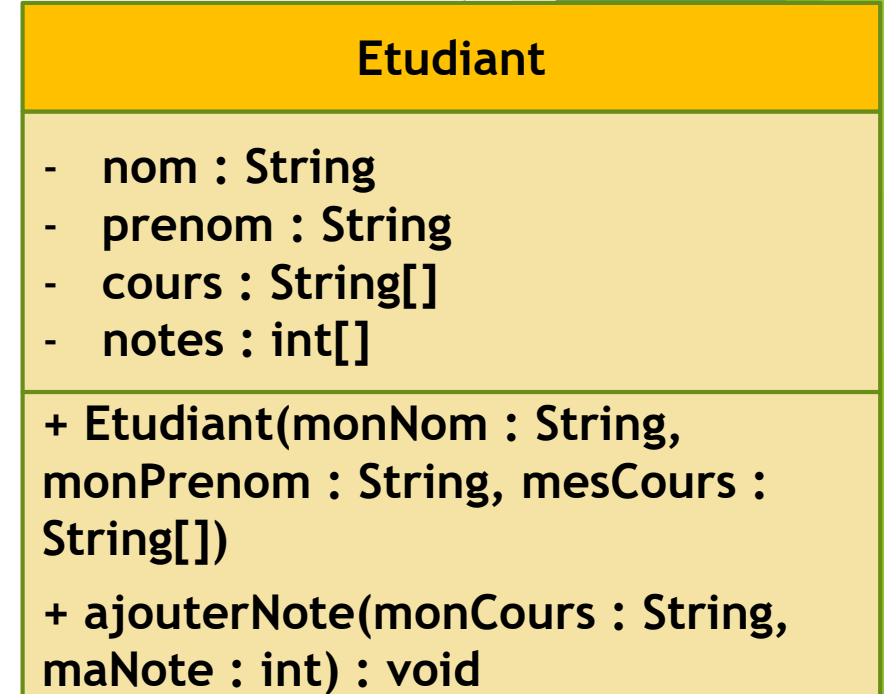
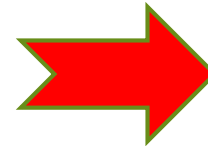
Pour savoir comment bien tester son code

# Un exemple

```
public class Etudiant{
    private String nom;
    private String prenom;
    private String[] cours; // les cours auxquelles il est enregistré
    private int[] notes; // ses notes

    public Etudiant(String monNom, String monPrenom, String[]
mesCours){
        this.nom = monNom;
        this.prenom = monPrenom;
        this.cours = mesCours;
        notes = new int[mesCours.length()];
    }

    public void ajouterNote(String monCours, int maNote){
        // ajoute une note à la bonne position dans notes
    }
}
```

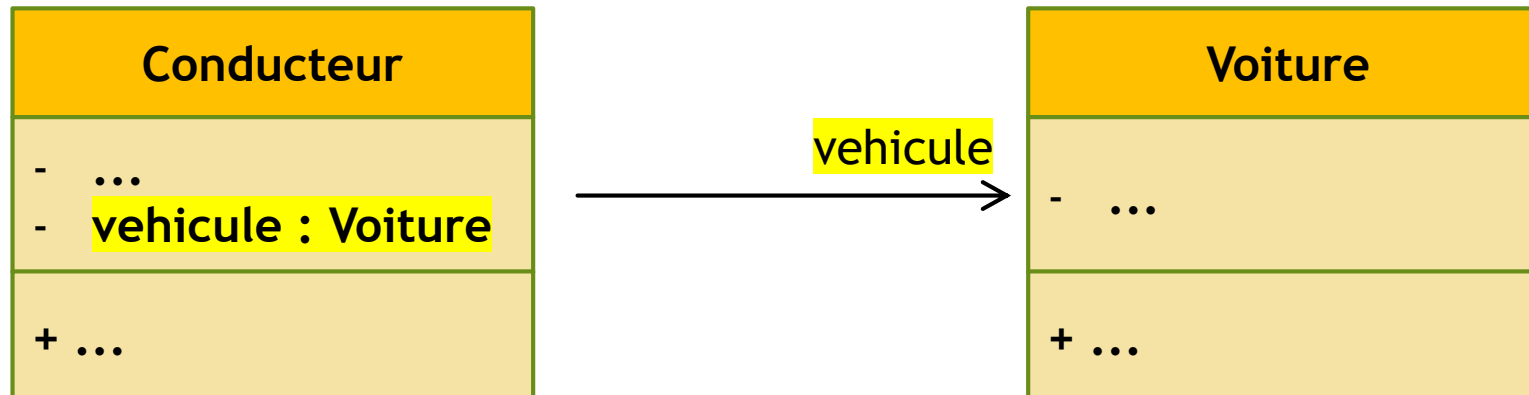


# L'association des classes

- Prenons l'exemple d'une voiture et un conducteur

```
public class Conducteur{  
    ...  
    // attributs : nom, prenom, type permis, # permis...  
    Voiture vehicule;  
  
    ...  
    // un constructeur, d'autres méthodes  
}
```

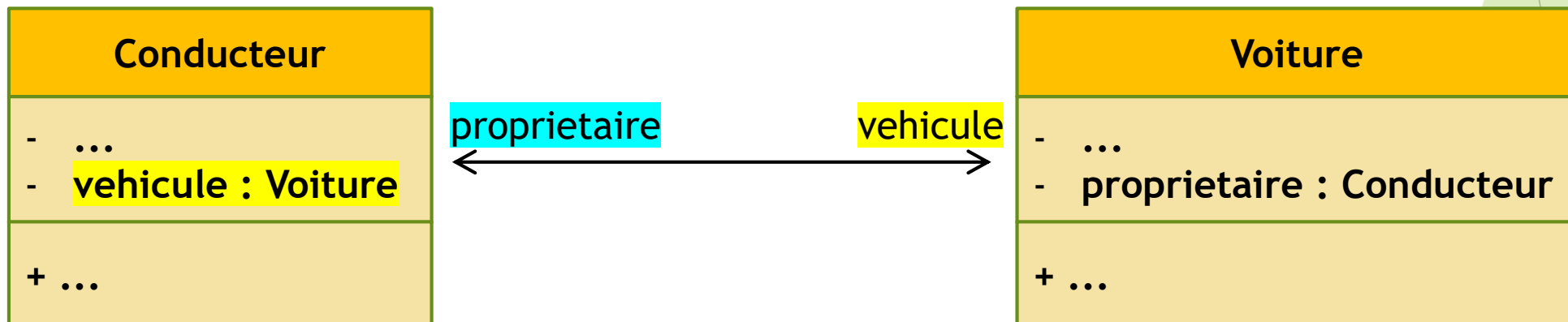
```
public class Voiture{  
    ...  
    // attributs : marque, modele, # immatriculation  
  
    ...  
    // un constructeur, d'autres méthodes  
}
```



# Une association réciproque

```
public class Conducteur{  
    ...  
    // attributs : nom, prenom, type permis, # permis...  
    Voiture vehicule;  
  
    ...  
    // un constructeur, d'autres méthodes  
}
```

```
public class Voiture{  
    ...  
    // attributs : marque, modele, # immatriculation  
    Conducteur proprietaire;  
  
    ...  
    // un constructeur, d'autres méthodes  
}
```

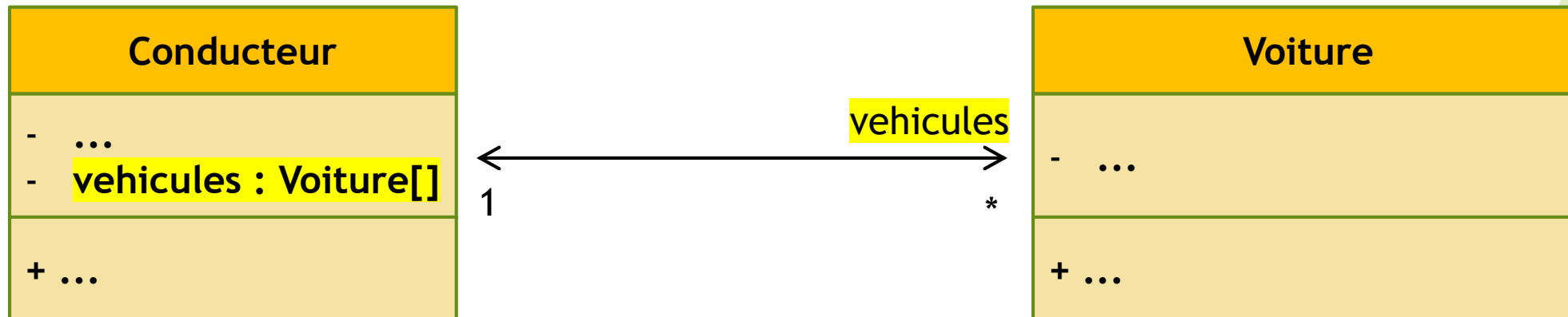


# La multiplicité

- Une association multiple : un conducteur, plusieurs voitures

```
public class Conducteur{  
    ...  
    // attributs : nom, prenom, type permis, # permis...  
    Voiture[] vehicules;  
  
    ...  
    // un constructeur, d'autres méthodes  
}
```

```
public class Voiture{  
    ...  
    // attributs : marque, modele, # immatriculation  
    ...  
    // un constructeur, d'autres méthodes  
}
```



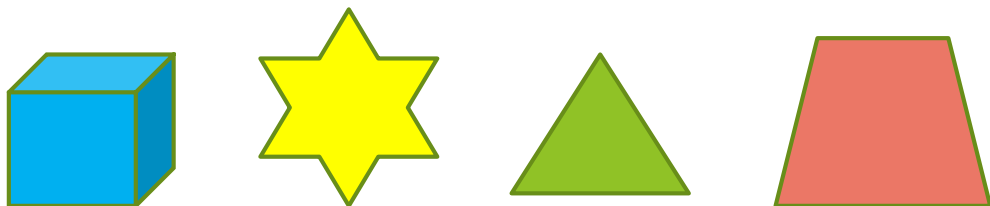


L'héritage

The background features abstract, overlapping geometric shapes in various shades of green, ranging from light lime to dark forest green. These shapes are primarily located on the right side of the frame, creating a dynamic, layered effect. The rest of the background is plain white.

# Des objets similaires

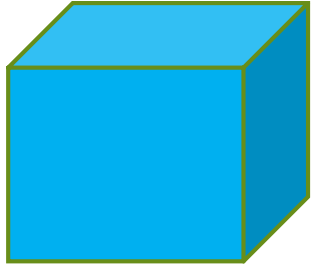
- ▶ Imaginons un jeu d'enfants avec des blocs :
  - ▶ Les enfants peuvent construire avec les blocs
  - ▶ Prenons les formes suivantes :



- ▶ Pour construire, l'enfant peut faire tourner ces formes
- ▶ Supposons aussi que certaines formes emettent également des sons

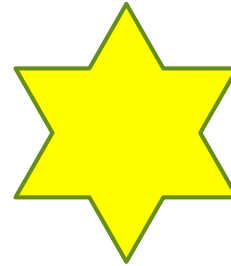


# En Java...



**Class Cube{**

```
...  
public void faireTourner (int x){  
    // roter le cube par x degrees  
...  
}  
}
```



**Class Etoile{**

```
...  
public void faireTourner(int x){  
    // roter le cube par x degrees  
... }  
public void emettreSon(int x){  
    // émet le son CLING CLING  
...  
}  
}
```

# Copier/coller du code

- ▶ ... est une mauvaise pratique
- ▶ Pourquoi ?

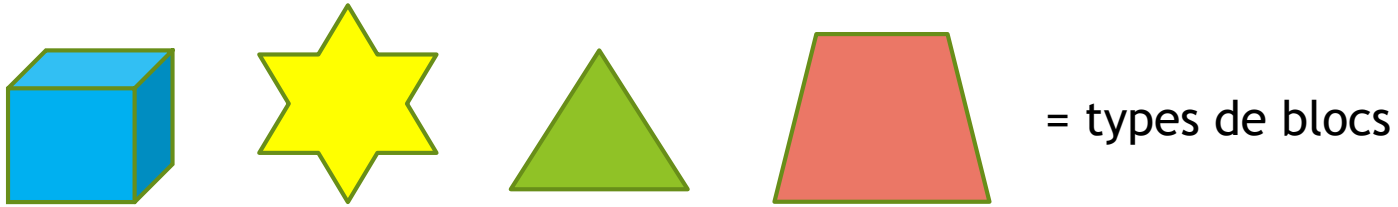
**L'incompatibilité** du code avec les variables déjà existents dans le code

Debugging : si le code a **des problèmes**, il faut les résoudre dans toutes les classes dans lesquelles ils apparaissent

**Inutilité**: Le code copié pourrait être daté ou résoudre un problème autrement résolu dans le même code

# L'heritage en Java

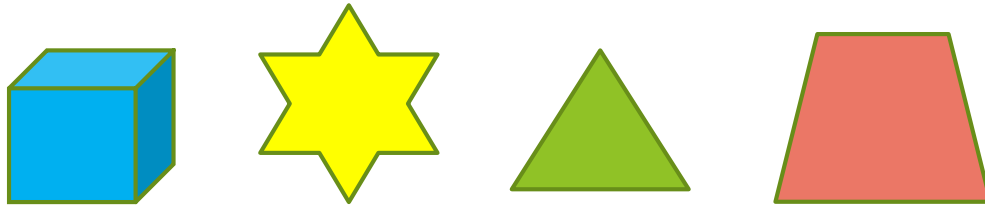
- ▶ Une façon de caractériser la relation "est un type de"
- ▶ On commence avec une super-classe : ici, la classe Bloc



- ▶ La super-classe décrit tout ce que les formes ont en commun :
  - ▶ Des **attributs** (nom, couleur) ou des **méthodes** (faireTourner)
- ▶ Après, nous construirons des sous-classes, qui héritent de la super-classe : **Cube**, **Etoile**, **Triangle**, **Trapeze**

# Concevoir son code

- ▶ Ce que les blocs ont en commun :



- ▶ Attributs : un nom, une couleur
- ▶ Méthodes : faireTourner : tous les blocs peuvent être tournés

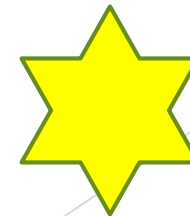
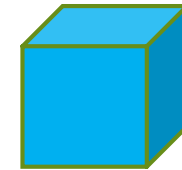
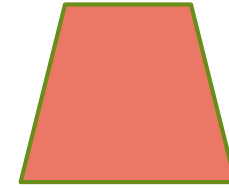
**Le cube, l'étoile, etc. héritent des caractéristiques générales des blocs**

- ▶ Et la méthode emettreSon ?

**La méthode emettreSon ne caractérise qu'un de ces blocs : le bloc Etoile**

# La superclasse

```
public class Bloc{  
    private String nom; // le nom de la forme  
    private String couleur; // sa couleur  
  
    public Bloc(String monNom, String maCouleur){  
        this.nom = monNom;  
        this.couleur = maCouleur;  
    }  
  
    public void faireTourner(int x){  
        // fait tourner la forme a x degrees  
        ...  
    }  
}
```



# Une sous-classe

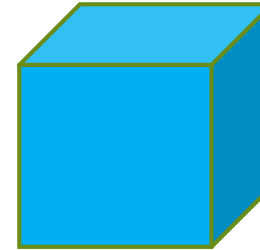
```
public class Cube extends Bloc{
```

Le mot "extends" déclare l'héritage

```
// la sous-classe hérite automatiquement tout attribut et  
méthode de la superclasse.
```

```
// Problème 1 : le constructeur
```

```
}
```



- ▶ Si une classe n'a pas de constructeur explicite, un constructeur implicite existe (hérité de la classe `Java.lang.Object`)
  - ▶ Ce constructeur crée un objet, sans initialiser ces attributs
  - ▶ Mais dans notre cas, la classe `Bloc` a bien un constructeur



# Le constructeur de la sous-classe

- ▶ Ce que nous voulons, c'est d'adapter le constructeur de la super-classe

```
public class Bloc{  
    ...  
    public Bloc(String monNom, String  
    maCouleur){  
        this.nom = monNom;  
        this.couleur = maCouleur;  
    }  
}
```

```
public class Cube extends Bloc{  
    public Cube(){  
        // nous faisons appel à la superclasse  
        super("cube", "bleu");  
    }  
}
```

Le mot "super" se réfère à la superclasse

**super("cube", "bleu")** appelle le constructeur Bloc(String, String)

À chaque fois qu'on veut utiliser la classe Bloc à l'intérieur de la classe Cube, nous utiliserons le mot **super** !

# L'héritage et la duplication de code

```
public class Bloc{
    private String nom; // le nom de la forme
    private String couleur; // sa couleur

    public Bloc(String monNom, String maCouleur){
        this.nom = monNom;
        this.couleur = maCouleur;
    }

    public void faireTourner(int x){
        // fait tourner la forme a x degrees
        ...
    }
}
```

```
public class Cube extends Bloc{
    public Cube(){
        super("cube", "bleu");
    }
}
```

## La classe Cube a :

- deux **attributs** hérités : nom, couleur
- un **constructeur**, signature Cube();
- une **méthode** faireTourner(int) héritée

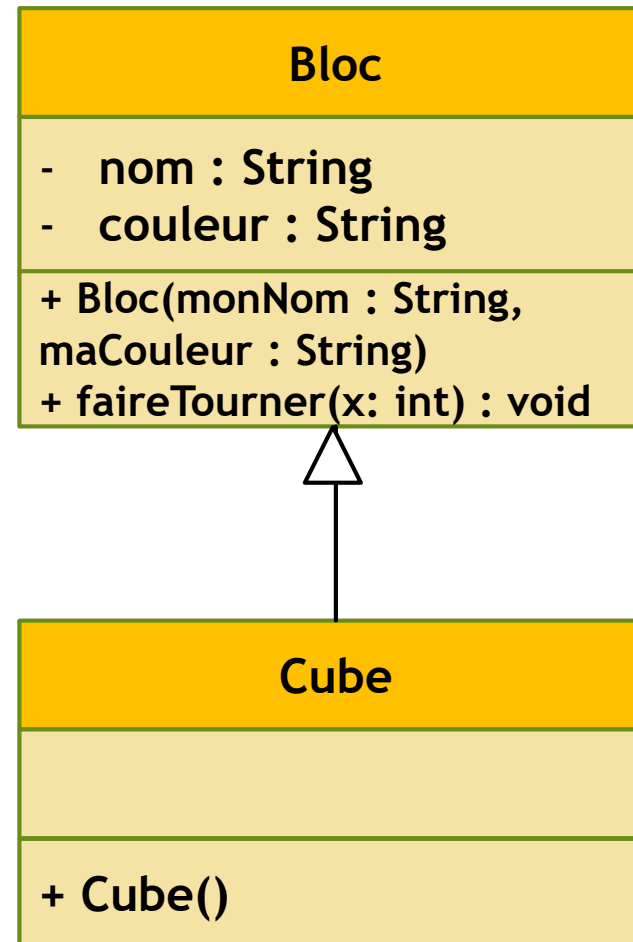
Tout(e) attribut/méthode non ré-écrit(e) dans la sousclasse se comporte comme spécifié dans la superclasse

# L'héritage & les diagrammes de classe

- La relation d'héritage est indiquée dans les diagrammes de classe

```
public class Bloc{  
    private String nom;  
    private String couleur;  
    public Bloc(String monNom, String maCouleur){  
        ... // constructeur  
    }  
    public void faireTourner(int x){  
        ... // fait tourner la forme a x degrees  
    }  
}
```

```
public class Cube extends Bloc{  
    public Cube(){  
        ... // le constructeur  
    }  
}
```



# Le polymorphisme

The background of the slide is white with abstract green geometric shapes on the right side. These shapes are composed of overlapping triangles and polygons in various shades of green, ranging from light lime to dark forest green. The shapes are positioned on the right edge, creating a modern, minimalist aesthetic.

# Une utilisation de l'heritage

- ▶ Un cube est un type de bloc
- ▶ Une étoile est un type de bloc
- ▶ Disons qu'on veut construire avec trois boîtes de blocs.
  - ▶ Chaque boîte contient plus ou moins blocs en fonction de sa taille
  - ▶ Les blocs sont choisis aléatoirement parmi les formes disponibles
- ▶ Nous voulons construire un objet BoiteABlocs à partir des classes suivantes :
  - ▶ la super-classe Bloc
  - ▶ les sous-classes Cube, Etoile, Trapeze, Triangle

# Le polymorphisme

```
public class BoiteABlocs{
    private String taille; // avec des valeurs "petite" ou "large"
    private Bloc[] blocs;
    public BoiteABlocs(String maTaille){
        // simplification : on aurait du verifier si maTaille = "petite" ou "large"
        this.taille = maTaille;
        if (maTaille.equals("petite")){ // une petite boite a 10 blocs
            this.blocs = new Bloc[10];
            double random = 0;
            // nous allons choisir 10 blocs aleatoirement
            for (int i = 0; i<blocs.length(); i++){
                random = Math.random() * 4;
                if (random <= 1){
                    Bloc[i] = new Cube();
                }
                else { ... // ici les autres cas
            }
        }
        else { ... // une large boite a 20 blocs, adapter code
    }
}
```

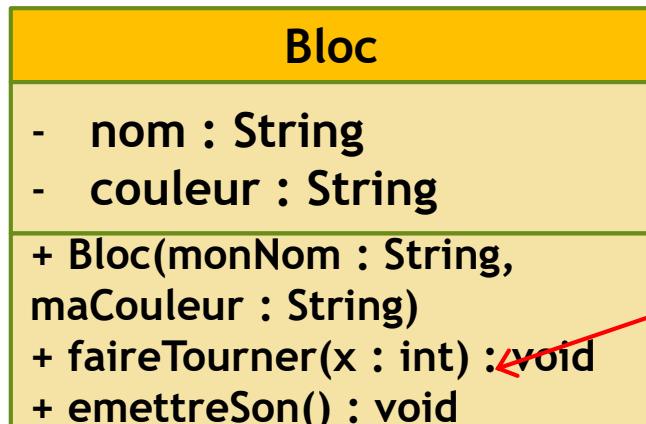
tableau polymorphe

10 blocs, peu importe la forme exacte

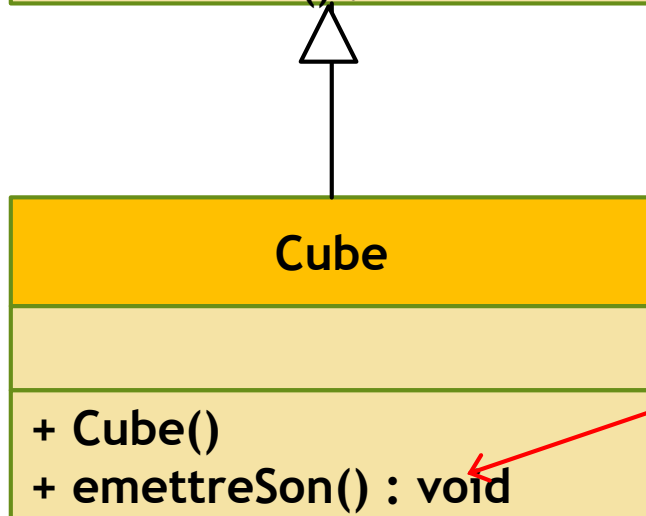
Ce bloc est un Cube  
Un autre pourrait-être un Triangle

# Un autre exemple

- Disons maintenant que tout Bloc fait un son différent :



```
public class Bloc{
    ...
    public void emettreSon(){
        // une méthode déclarée et vide
    }
}
```



```
public class Cube extends Bloc{
    ...
    public void emettreSon(){
        System.out.println("Ding dong!");
    }
}
```

# Le polymorphisme avec des sons

- ▶ Voici les sons emis par les blocs :
  - ▶ Cube : Ding Dong; Etoile : Cling Cling; Triangle : Ding Ling; Trapeze : Clang
- ▶ Prenons cette méthode principale, dans une autre classe

```
public class Musique{  
    ...  
    public static void main(String[] args){  
        Bloc[] mesBlocs = new Bloc[3];  
        mesBlocs[0] = new Bloc("Cube", "bleu");  
        mesBlocs[1] = new Cube();  
        mesBlocs[2] = new Etoile();  
        for (int i = 0; i<mesBlocs.length(); i++){  
            mesBlocs[i].emettreSon();  
        }  
    }  
}
```

tableau polymorphe

Bloc[0] est un Bloc  
Bloc[0].Son fait appel à la méthode  
emettreSon de la classe Bloc  
Bloc[1] appelle la méthode emettreSon  
de la classe Cube  
Bloc[2] appelle la méthode emettreSon  
de la classe Etoile



# Les classes & méthodes abstraites



# Les classes abstraites

- ▶ Abstraite = non-instanciable;
- ▶ une classe abstraite = une classe qui ne peut pas être instanciée

**A quoi peut bien servir une classe qui n'est jamais instanciée ?**

- ▶ On peut hériter même d'une classe abstraite !
  - ▶ Une superclasse abstraite aura des attributs et des méthodes
    - ▶ Des méthodes abstraites et/ou concrètes
    - ▶ Y compris un constructeur si on veut, mais celui-ci ne sera jamais utilisé
  - ▶ Les sousclasses personnalisent ces méthodes et sont instanciables
  - ▶ Avantage : moins de code à écrire dans la superclasse
  - ▶ Inconvénient : nous ne pourrons jamais instancier la superclasse

# Les méthodes abstraites

- ▶ Méthode abstraite = non instanciable
- ▶ À écrire : juste une signature suivi de ";"
- ▶ Une classe qui contient au moins une méthode abstraite doit être une classe abstraite

```
abstract class Forme{  
    private String nom;  
    private String couleur;  
    public abstract double perimetre();  
    public void emettreSon(){  
        System.out.println("Cette forme ne produit aucun son.");  
    }  
}
```

mod dédié : abstract

méthode abstraite  
public abstract + signature + ;

méthode concrète  
elle peut être utilisée directement ou  
on peut la ré-écrire dans les sousclasses

# Hériter d'une classe abstraite

- ▶ On part sur une classe abstraite `Forme`;
- ▶ Nous pouvons maintenant créer une sous-classe `Polygone` :
  - ▶ Si nous ne voulons pas instancier toutes les méthodes abstraites de la superclasse, alors la classe `Polygone` doit être abstraite

```
public abstract class Forme{  
    ... // ici le code de la super-classe  
}
```

```
public abstract class Polygone extends Forme{  
    ... // ici le code de la sous-classe abstraite  
}
```

- ▶ Nous pouvons également créer une sous-classe concrète `Cercle`
  - ▶ Celle-ci va instancier toutes les méthodes de la classe `Forme`
  - ▶ Mot-clé `@Override`

```
public class Cercle extends Forme{  
    ... // ici le code de la sous-classe  
  
    @Override  
    public void perimetre(){ ... // code concret}  
}
```

# Concrète vs. abstraite : le polymorphisme

- ▶ Supposons la hiérarchie de classes suivante :
  - ▶ Superclasse : `Forme`
  - ▶ Sous-classes de `Forme` : `Polygone`, `Cercle`
- ▶ Si `Forme`, `Polygone`, `Cercle` sont des classes concrètes :

```
public class JeuAuxFormes{  
    public static void main(String[] args){  
        Forme[] mesFormes = new Forme[3];  
        mesFormes[0] = new Forme();  
        mesFormes[1] = new Polygone();  
        mesFormes[2] = new Cercle();  
    }  
}
```

Polymorphisme  
3 Formes sont instantiées avec  
des divers constructeurs

# Concrète vs. abstraite : le polymorphisme

- ▶ Supposons la hiérarchie de classes suivante :
  - ▶ Superclasse : Forme
  - ▶ Sous-classes de Forme : Polygone, Cercle
- ▶ Si Forme est abstraite, Polygone et Cercle sont concrètes

```
public class JeuAuxFormes{  
    public static void main(String[] args){  
        Forme[] mesFormes = new Forme[3];  
        mesFormes[0] = new Forme();  
        mesFormes[1] = new Polygone();  
        mesFormes[2] = new Cercle();  
    }  
}
```

Ce n'est plus possible d'utiliser ce code

Pourquoi pas ?

# Concrète vs. abstraite : le polymorphisme

- ▶ Supposons la hiérarchie de classes suivante :
  - ▶ Superclasse : Forme
  - ▶ Sous-classes de Forme : Polygone, Cercle
- ▶ Si Forme est abstraite, Polygone et Cercle sont concrètes

```
public class JeuAuxFormes{  
    public static void main(String[] args){  
        Forme[] mesFormes = new Forme(3);  
        mesFormes[0] = null;  
        mesFormes[1] = new Polygone();  
        mesFormes[2] = new Cercle();  
        Polygone maForme = new Polygone();  
        mesFormes[0] = maForme;  
    }  
}
```

Nous pouvons initialiser un objet de la superclasse en tant que objet de la sousclasse

# La composition

The background features abstract, overlapping geometric shapes in various shades of green, ranging from light lime to dark forest green. These shapes are primarily located on the right side of the frame, creating a dynamic, layered effect. The rest of the background is plain white.



# La composition vs l'aggrégation

- ▶ La composition est une relation très forte entre deux classes, un composite et une composante
  - ▶ Elle reflète la relation "contient un(e)" ou "a un(e)"
  - ▶ De plus, le composant n'a pas de sens (n'existe pas) sans le composite
- ▶ Exemple : CorpsHumain et Main ou Jambe

```
public class CorpsHumain{  
    Jambe[] lesJambes;  
    Main[] lesMains;  
    ... // d'autres attributs  
    ...// des méthodes  
}
```

```
public class Jambe{  
    double taille;  
    ... // d'autres attributs  
    ...// des méthodes  
}
```

```
public class Main{  
    double taille;  
    ... // d'autres attributs  
    ...// des méthodes  
}
```

# Diagrammes de composition

