IUT de Troyes - Université de Reims-Champagne-Ardenne
DUT Génie Électrique et Informatique Industrielle

M3105C - Programmation Orientée Objet

Support complet de cours

F. Morain-Nicolier

Table des matières

1 Introduction générale

1.1 Introduction

1.1.1 Qui est-ce ?

Frédéric Morain-Nicolier
— `http://pixel-shaker.fr` et `http://sy24.postach.io`
— `frederic.nicolier@univ-reims.fr`
— IUT Troyes / Bureau C202 / 03 25 42 71 68

1.2 Description du cours

1.2.1 Description générale

M3105 C : Programmation orientée objet

Objectifs
Comprendre une démarche de conception orientée objet. Se familiariser avec un langage à objets.

Compétences visées :
— Découper une application en objets,
— Exprimer un cahier des charges en UML,
— Utiliser un paquetage de classes pour construire un objet composite, Utiliser le polymorphisme.
— Programmer en langage objet.

Contenu

— Penser objet : définir une **classe**, définir un **objet**, établir des liaisons entre objets, **constructeurs**, **destructeurs**, **interfaces**, **méthodes**, propriétés, objets internes.

— Construire une application en langage objet.

— Les API standard.

1.2.2 Progression

CM / TD / TP

— 3 CM

— 12 TD (avec 2 évaluations)

— 4 TP (-> projet)

Notions de l'orienté objet

— env. 4 séances TD + 1 évaluation

— Processing

— Notions pour débuter : classe, méthode, attribut, instance, constructeur, destructeur, héritage, notion d'UML.

Orienté object en C++

— env. 2 séances TD

— C++

— Syntaxe du C++

— Notion de référence

Applications en C++

— env. 4 séances TD + 1 évaluation

— C+

— Qt : notion de d'évènement

— LED / Boutons tactiles - Timers - MLI

Projet

— sur deux ou trois jours

— en autonomie partielle

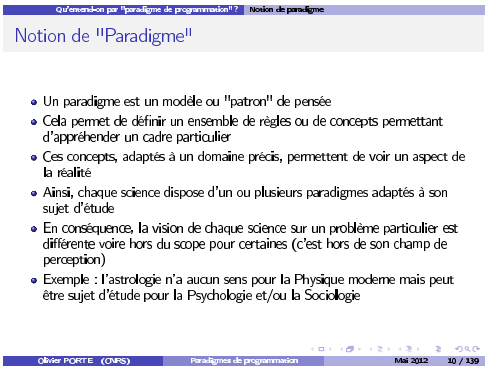
— C++

— Raspberry Pi

2 Notions d'orienté objet

2.1 Paradigmes de programmation

▶ extraits de « Paradigmes de programmation - Une introduction » Olivier Porte - CNRS



2.2 Raisonner OO

2.2.1 Objets

« Programmation » du réveil d'un étudiant :

— se reveiller
— s'habiller,
— prendre un petit déjeuner,
— sortir,
— prendre le bus

▶ Quel est l'« objet » qui est concerné ici ?

▶ Un humain.

▶ Qui a certaines propriétés :

— couleur des cheveux, des yeux
— taille, poids,
— ...

▶ Et qui sait faire certaines actions :

— se réveiller,

— dormir,
— manger,
— résoudre des intégrales,
— ...

En programmation
— Les propriétés sont analogues aux variables.
— Les actions sont analogues aux fonctions.

▶ La programmation orienté objet consiste à réunir variables et fonctions ensemble.

2.2.2 Attributs et méthodes

Définissons un humain :
Variables → **Attributs**
— Taille
— Poids
— Genre
— Couleur des yeux
— Couleur des cheveux

Fonctions → **Méthodes**
— Dormir
— Se réveiller
— Manger
— Prendre le bus

▶ Ceci décrit ce qu'est un humain (simplifié).

2.2.3 Notion de classe

Classe vs objet

— La définition précédente décrit ce qu'est être humain : il faut des cheveur, des yeux, ... et savoir faire certaines choses.
— Il s'agit de décrire le prototype d'humains particulier, c'est une **classe**.
— Un individu (ex. Albert Einstein, votre prof) est un **objet** particulier de la classe humain : c'est une **instance** de la classe **humain**.

Exercice

▶ Considérons la voiture sous l'approche orientée objet
— Attributs ?
— Méthodes ?
— Exemples d'instances ? (un piège ici)

2.2.4 Notions abordées

— Classe
— Objet
— Instance
— Attribut
— Méthode

2.3 Manipulation d'un objet

2.3.1 En prog classique

Une voiture

Considérons une voiture que l'on souhaite afficher et déplacer à l'écran.

Variables globales

- `car_color`
- `car_x_pos`
- `car_y_pos`
- `car_x_speed`

Setup

- Initialiser `car_color`
- Initialiser position : `car_x_pos` et `car_y_pos`
- Initiatliser `car_x_speed`

Draw

- Remplir le fond
- Afficher la voiture à sa position et avec sa couleur
- Incrémenter la position par la vitesse

code Processing

```
color c;
int xpos;
int ypos;
int xspeed;

void setup() {
  size(200,200);
  c = color(255);
  xpos = width/2;
  ypos = height/2;
  xspeed = 1;
}
```

code Processing

```
void draw() {
  background(0);
  display();
  drive();
}
void display () {
  rectMode(CENTER);
  fill(c);
  rect(xpos,ypos,20,10);
}
void drive () {
  xpos = xpos + xspeed;
  if (xpos > width) {
    xpos = 0;
  }
}
```

code Processing

(demo)

2.3.2 En OO

pseudo-code

— L'OO consiste à réunir les variables et les fonctions dans un objet **Car**.

— Cela simplifie le pseudo-code.

Variables globales

— un objet de classe **Car**

Setup

— Instancier (= initialiser) la voiture

Draw

— Remplir le fond

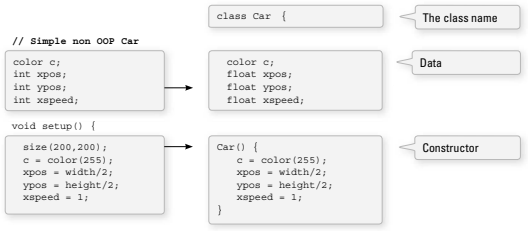
— Afficher la voiture

— Déplacer la voiture

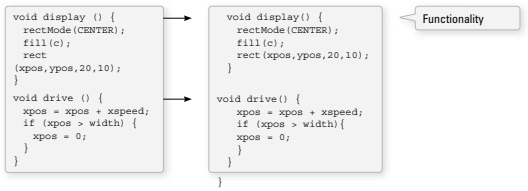
Code Processing

```
Car myCar;

void setup() {
  myCar = new Car(); // Un objet myCar
}

void draw() {
  background(0);
  myCar.drive();
  myCar.display();
}
```

Écriture de la classe `Car`



Écriture de la classe `Car`



(demo)

2.3.3 UML

Définition et usage

UML = *Unified Modeling Language* (Langage de Modélisation Unifié)

— modélisation graphique

— à base de pictogrammes

— ⇒ visualisation de la conception d'un système OO

— Diagrammes (entre autres) :

  — **de classes** : représentation des classes du projet

  — **d'objets** : représentation des instances utlisées dans le système

  — de comportement et d'interactions

Un exemple

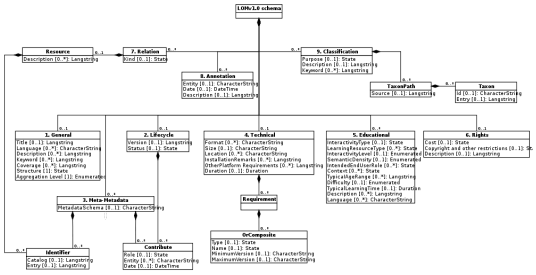


Diagramme de classe

Nom de la Classe
Attributs
Méthodes

— Nom de la Classe : dans le rectangle du haut

— Attributs : Visibilité nomAttribut : typeAttribut = Initialisation

— Méthodes : Visibilité nomFonction(directionParamètreN nomParamètreN : typeParamètreN) : typeRetour

Diagramme de classe

— Visiblité (pour le moment) :

  — + : accès public (Toutes les autres classes ont accès à cet attribut)

  — - : accès privé (Seule la classe elle-même a accès à cet attribut)

— Direction du paramètre :

  — in : accès rentrant

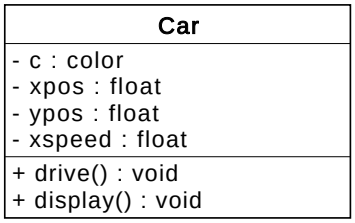
  — out : accès sortant

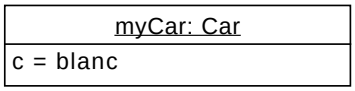
  — inout : accès rentrant et sortant

classe `Car`

Car
- c : color - xpos : float - ypos : float - xspeed : float
+ drive() : void + display() : void

Diagramme d'objets





2.4 Constructeur d'une classe

2.4.1 Plusieurs instances

— Dans l'exemple précédent, la voiture était créée à l'aide de l'opérateur `new` :

```
Car myCar = new Car();
```

— Cet exemple est limité car que ce passe-t'il si l'on souhaite deux voitures ?

```
Car myCar1 = new Car();
Car myCar2 = new Car();
```

— Avec le code actuel, `myCar1` et `myCar2` seront identiques.
— Ainsi, plutôt qu'écrire : *Crée une nouvelle voiture*,
— il faudrait pouvoir écrire : *Crée une nouvelle voiture rouge, à la position (0,10) et de vitesse 1*
— ce qui permettrait : *Crée une nouvelle voiture bleue, à la position (0,100) et de vitesse 2*
— et donc d'avoir des objets différents.

— Il faut donc pouvoir spécifier les attributs à la création de l'objet.
— Quelle méthode faut-il modifier ?

modification du constructeur

On souhaite pouvoir écrire par exemple :

```
Car myCar = new Car(color(0, 0, 100),
                    0, 100, 2);
```

Le code du nouveau constructeur est donc :

```
Car(color _C, float _Xpos, float _Ypos,
    float _Xspeed) {
  c = _C;
  xpos = _Xpos;
  ypos = _Ypos;
  xspeed = _Xspeed;
}
```

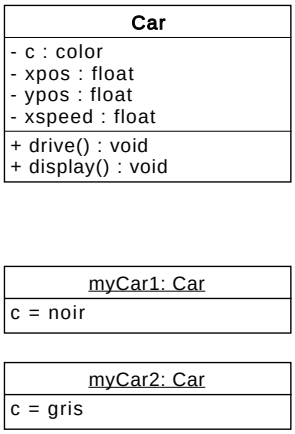
Utilisation

Il est alors facile de créer et d'animer deux voitures distinctes.

```
Car myCar1;
Car myCar2; // Deux instances !

void setup() {
  size(480, 270);
  // appel du constructeur avec
  // des arguments differents.
  myCar1 = new Car(color(51), 0, 100, 2);
  myCar2 = new Car(color(151), 0, 200, 1);
}
```

```
void draw() {
  background(255);
  myCar1.move();
  myCar1.display();
  myCar2.move();
  myCar2.display();
}
```

Diagrammes UML



(démonstration)

3 Appronfondissement

3.1 Encapsulation

3.1.1 Principe

Exemple de classe `voiture` (réelle) :

- `tourner()`
- `accelerer()`
- `freiner()`
- `radio()`
- ...

Pour utiliser une voiture faut-il savoir comment :

- le moteur fonctionne-t'il ?
- l'embrayage permet-il de changer de vitesse ?
- l'essence arrive-t'elle aux pistons ?
- ...

Il est inutile de connaitre le fonctionnement interne d'une voiture pour l'utiliser.
▶ C'est le concept **d'encapsulation**.

Définition 3.1.1 (Encapsulation)**.** Masquer le fonctionnement interne d'un objet pour l'utilisateur de l'objet

En orienté objet,

- fonctionnement interne = données (variables de l'objet) et fonctions. Ensemble des méthodes destinées à la gestion interne de l'objet, auxquelles le développeur final n'aura pas à avoir accès
- utilisateur = le programmeur qui utilise la classe.

Pourquoi est-ce une bonne idée ?

- Modularité : décomposition une application en (petits) modules indépendants autonomes.
- Réusabilité : réutiliser le code produit par d'autres
  - réduction du tps de développement
  - moindre complexité de l'application
  - moindres compétences requises
  - plus grande sûreté de fonctionnement
- Encapsulation :
  - aide à organiser le code
  - assurer l'intégrité des données qui ne pourrons être accéder qu'au traver des méthodes visibles.
  - **se protéger d'erreurs**.

3.1.2 Exemple

Une classe `CompteBancaire` qui possède un attribut contenant le `solde`

```
CompteBancaire compte = new CompteBancaire(1000);
```

Si l'on souhaite retirer de l'argent de ce compte, le plus simple est d'écrire :

```
compte.solde = compte.solde − 100;
```

Mais il est souhaitable d'encapsuler l'attribut `solde` et de le garder cacher. Car s'il y a des frais pour opération bancaire, il faut empêcher la manipulation directe du solde :

```
compte.retrait(100);
```

Ce n'est donc pas à l'utilisateur de gérer les frais, mais au créateur de la classe :

```
void retrait(float montant} {
  float frais = 1.25;
  compte = compte − (montant + frais);
}
```

Avantage :

— Si la banque change ses frais, il suffit d'ajuster la valeur de `frais`.

— Mais cela ne change rien pour l'utilisateur.

3.1.3 Accesseurs et mutateurs

En principe, une encapsulation complète doit **interdire l'accès direct** des attributs.

— Les attributs ne doivent être accessibles que par des méthodes.

— Il est fréquent de voir des accesseurs et des mutateurs dans les classes.

— *getters* et *setters* : méthodes qui permettent d'extraire et changer la valeur des attributs.

Exemple d'une classe `Point` :

```
class Point {
  float x,y;

  Point(float tempX, float tempY) {
    x = tempX;
    y = tempY;
  }

  // Getters
  float getX() {
    return x;
  }

  float getY() {
    return y;
  }

  // Setters
  float setX(float val) {
    x = val;
  }

  float setY(float val) {
    if (val > height) val = height;
    y = val;
  }
}
```

Noter que le *setter* `setY()` permet d'implémenter une protection des valeurs de l'attribut `y` :

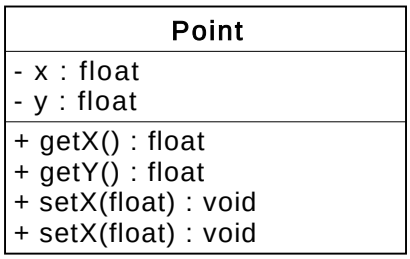
```
float setY(float val) {
  if (val > height) val = height;
  y = val;
}
```

Dans certains language (Java, c++), il est possible de forcer l'interdiction de visibilité par exemple par l'emploi du mot-clé `private` :

```
class Point {
private float x;
private float y;
}
```

Avec cette définition, une tentative d'accès à `x` ou `y` générera une erreur à la compilation

UML



Pour créer un point :

```
Point p = new Point(1.0, 2.0);
```

Pour modifier ce point :

```
p.setX(p.getY() + 1);
```

Encapsulation :

- Principe central de l'orienté-objet.
- Essentiel pour la conception d'application de grande envergure, avec plusieurs développeurs ⇒ encapsulation complète
- Mais parfois peu pratique pour des applications simples.
- Il n'est pas dramatique de pragmatiquement parfois permettre l'accès direct.

A noter que la classe `PVector` de Processing n'encapsule pas ses attributs :

```
PVector v1, v2;

void setup() {
  noLoop();
  v1 = new PVector(40, 20);
  v2 = new PVector(25, 50);
}

void draw() {
  ellipse(v1.x, v1.y, 12, 12);
  ellipse(v2.x, v2.y, 12, 12);
```

```
  v2.add(v1);
  ellipse(v2.x, v2.y, 24, 24);
}
```

3.2 Héritage

3.2.1 Principe

L'**héritage** permet de créer de nouvelles classes en se basant sur des classes existantes.

Prenons l'exemple d'une application peuplée d'animaux : chiens, chats, singes, pandas, ...

Programmons la classe `Chien`

```
class Chien {

int age;

Chien() {
  age = 0;
}
void manger() {
  // code
}
void dormir() {
  // code
}
void aboyer() {
  println("Ouaf !");
}
}
```

Classe `Chat` :

```
class Chat {
int age;

Chien() {
  age = 0;
}
void manger() {
  // code
}
void dormir() {
  // code
}
void miauler() {
  println("Miaou !");
}
}
```

Les classes `Chien` et `Chat` ont

- les même attributs : `age`
- et certaines méthodes en commun : `manger()` et `dormir()`.
- Les méthodes `aboyer()` et `miauler()` sont par contre spécifiques.

L'écriture de l'application va être pénible pour les classes `Singe`, `Panda`, ...

- Répétition de codes identiques à de nombreux endroits.

⇒ Écriture d'une classe `Animal` générale pour décrire nos animaux.

— Un `Chien` est un `Animal` et possède les propriétés d'un animal et peut faire tout ce qu'un animal peut faire. De plus, il peut `aboyer()`.

— Un `Chat` est un `Animal` et possède les propriétés d'un animal et peut faire tout ce qu'un animal peut faire. De plus, il peut `miauler()`.

L'héritage permet cela.

— Une classe peut **hériter** des attributs et méthodes d'une autre classe.

— Un `Chien` est une **sous-classe** de la classe `Animal`.

— Une **sous-classe** hérite de tous les attributs et des méthodes de sa **super-classe** (le parent).

— Une **sous-classe** peut avoir des attributs et des méthodes spécifiques que ne possède pas la **super-classe**.

— Une **sous-classe** peut être la **super-classe** d'une autre. Ex : la classe `Chien` pourrait hériter de la classe `Canide` qui pourrait hériter d'une classe `Mamifere`, ...

Voici comment traduire cela en processing :

```
class Animal {

int age;

Animal() {
   age = 0;
}
void manger() {
   // code
}
void dormir() {
   // code
}
}
```

La classe `Chien` hérite de la classe `Animal`

```
class Dog extends Animal {

Dog() {
  super();
}

void aboyer() {
  println("Ouaf !");
}
```

`super()` indique que l'on appelle le constructeur de la **super-classe**, donc de la classe `Animal` ici.

```
class Chat extends Animal {

Cat() {
  super();
}

void miauler() {
   println("Miaou !");
}
}
```

3.2.2 Deux nouveaux termes

— `extends` : indique le parent d'une classe. Une classe ne peut avoir qu'un seul parent direct (en processing)

— `super()` : appel du constructeur du parent. De cette façon, tout ce qui est fait dans le constructeur du parent sera fait dans le constructeur de l'enfant. **Optionnel mais fréquent**.

3.2.3 Sous-classe

Une sous-classe peut-être enrichie d'attributs et méthodes par rapport à sa super-classe. Par exemple :

```
Class Chien extends Animal {

color couleur_du_poil;

Chien() {
  super():
  couleur_du_poil = color (random(255));
}

void aboyer() {
  println("Ouaf !");
}

}
```

Il est possible de redéfinir une méthode pour la remplacer. Par exemple :

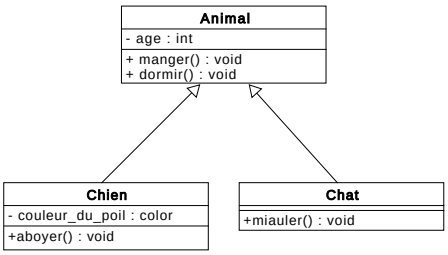
```
Class Chien extends Animal {

color couleur_du_poil;

Chien() {
  super():
  couleur_du_poil = color (random(255));
}

void manger() {
 // code specifique au repas d'un chien
}

void aboyer() {
  println("Ouaf !");
}

}
```

Une sous-classe peut exécuter le code d'une super-classe. Pratique pour ajouter des fonctionnalités aux méthodes de la super-classe. Par exemple :

```
Class Chien extends Animal {
color couleur_du_poil;

Chien() {
  super():
  couleur_du_poil = color (random(255));
}

void manger() {
  super.manger(); // methode manger() du constructeur
  println("Miam !"); // specifique a la classe Chien
}

void aboyer() {
  println("Ouaf !");
}
}
```

3.2.4 UML

Les relations d'héritage se représentent également dans un diagramme UML :



La flèche se lit « est un » : un `Chien` **est un** `Animal`.

3.2.5 Exemple

On souhaite avoir des formes graphiques qui tremblent à l'écran :

— à une position (x,y)

— d'une taille donnée

— et d'une méthode pour les afficher.

```
class Shape {
  float x;
  float y;
  float r;

  Shape(float x_, float y_, float r_) {
    x = x_;
    y = y_;
    r = r_;
  }

  void jiggle() {
    x += random(-1,1);
    y += random(-1,1);
  }

  // methode generique
  void display() {
    point(x,y); // a surcharger
  }
}
```

Créons une classe `Square` pour représenter des carrés.

— Elle doit hériter de tous les attributs et des méthodes de `Shape`.

— Créons un constructeur `Square()` qui appelle le constructeur de la super-classe.

```
class Square extends Shape {

  Square(float x_, float y_, float r_) {
    super(x_, y_, z_);
  }

  // herite de  jiggle() de Shape

...
```

— et surchargeons la méthode `display()` :

```
...

  // Surcharge
  void display() {
    rectMode(CENTER);
    fill(175);
    stroke(0);
    rect(x,y,r,r);
  }
}
```

Il est possible d'ajouter des fonctionnalités dans la sous-classe. Par exemple pour une classe `Circle` héritant de `shape` :

— un attribut de couleur (même si ici il serait plus logique de placer cet attribut dans la super-classe) ;

— ajout d'une méthode `changeColor` modifiant cet attribut ;

— modification de la méthode `jiggle()` qui fait varier le diamètre du cercle.

```
class Circle extends Shape {

  // heritage des attributs et ajout de la couleur
  color c;

  Circle(float x_, float y_, float r_, color c_) {
    super(x_,y_,r_); // appel du constructeur parent
    c = c_;          // initialisation de la couleur
  }
...
```

```
...
  // appel de jiggle du parent
  // mais ajout de fonctionalite
  void jiggle() {
    // Le tremblement du cercle modifie la position
    super.jiggle();
    r += random(-1,1); // et la taille.
    r = constrain(r,0,100);
  }

  // specifique a la classe Circle
  void changeColor() {
    c = color(random(255));
  }
...
```

```
...
  void display() {
    ellipseMode(CENTER);
    fill(c);
    stroke(0);
    ellipse(x,y,r,r);
  }
}
```

Programme principal

```
Square s;
Circle c;

void setup() {
  size(480, 270);
  // A square and circle
  s = new Square(280, 180, 40);
  c = new Circle(360, 180, 80, color(175, 150));
}

void draw() {
  background(255);
  s.jiggle();
  c.jiggle();
  s.display();
  c.display();
}
```

(démo)

3.3 Surcharge

3.3.1 Définition

Définition 3.3.1. (Surcharge) Mécanisme permettant de définir plusieurs fois une même méthode avec des prototypes différents.

Par exemple :

```
fill(255);
fill(255,0,255);
fill(0,0,255,150);
```

▶ Le langage recherche la version où les arguments correspondent (en nombre et types).

Par exemple, si l'on crée une classe `Fish`

```
class Fish {
  float x;
  float y;
```

On peut vouloir construire des objets `Fish` avec une position aléatoire :

```
  Fish() {
    x = random(0,width);
    y = random(0,height);
  }
```

ou bien en spécifiant la position initiale :

```
  Fish(float tempX, float tempY) {
    x = tempX;
    y = tempY;
  }
}
```

On définit alors deux constructeurs différents :

```
Fish fish1 = new Fish();
Fish fish2 = new Fish(100,200);
```

3.4 Polymorphisme

3.4.1 Exemple

Munis du concept d'héritage, nous pouvons créer un programme peuplé d'animaux :

```
Dog[] dogs = new Dog[100];
Cat[] cats = new Cat[101];
Turtle[] turtles = new Turtle[23];
Kiwi[] kiwis = new Kiwi[6];
```

Les classes `Dog`, `Cat`, `Turtle` et `Kiwi` héritent d'une super-classe `Animal`.

Les tableaux étant de différentes tailles, plusieurs boucles sont nécessaires pour instancier :

```
for (int i = 0; i < dogs.length; i++) {
  dogs[i] = new Dog();
}
for (int i = 0; i < cats.length; i++) {
  cats[i] = new Cat();
}
for (int i = 0; i < turtles.length; i++) {
  turtle[i] = new Turtle();
}
for (int i = 0; i < kiwis.length; i++) {
  kiwis[i] = new Kiwi();
}
```

De même lors de l'appel des méthodes des objets créés :

```
for (int i = 0; i < dogs.length; i++) {
  dogs[i].eat();
}
for (int i = 0; i < cats.length; i++) {
  cats[i].eat();
}
for (int i = 0; i < turtles.length; i++) {
  turtles[i].eat();
}
for (int i = 0; i < kiwis.length; i++) {
  kiwis[i].eat();
}
```

Tout ceci fonctionne mais :

— beaucoup de boucles quasiment identiques, qui augmentent en nombre si l'on crée d'autres sous-classes de `Animal`.

Il faut garder à l'esprit que :

— Toutes nos créatures sont des animaux (classe `Animal`) et possèdent donc la méthode `eat()`.

— Il suffit donc de placer tous nos objets dans un tableau `Animal[]`.

Le programme devient plus simple :

```
Animal[] kingdom = new Animal[1000];
for (int i = 0; i < kingdom.length; i++) {
  if (i < 100) kingdom[i] = new Dog();
  else if (i < 400) kingdom[i] = new Cat();
  else if (i < 900) kingdom[i] = new Turtle();
  else kingdom[i] = new Kiwi();
}
for (int i = 0; i < kingdom.length; i++) {
  kingdom[i].eat();
}
```

Un objet de type `Dog` peut donc être vu comme une instance

— de la classe `Dog`

— ou de la classe `Animal`.

— C'est le principe du **polymorphisme**.

3.4.2 Définition

Définition 3.4.1. (Polymorphisme) capacité d'un objet à prendre plusieurs formes.
Le système choisit dynamiquement la méthode qui correspond au type de données effectif de l'objet en cours de manipulation.

Par exemple :

— Un `Dog` est un `Dog`,

— mais puisque `Dog` hérite de `Animal`, il peut être considéré comme un `Animal`.

En code, ces deux déclarations sont légales :

```
Dog rover = new Dog();
Animal spot = new Dog();
```

— On crée un objet de type `Dog` que l'on stocke dans une variable de type `Animal`.

— Il est possible d'appeler les méthodes d'`Animal` sur l'objet `spot` (règle d'héritage).

— Si la classe `Dog` surcharge une méthode (comme `eat()`) : la langage déterminera dynamiquement l'identité véritable et éxécutera la version appropriée de la méthode.

— Particulièrement pratique avec des tableaux.

3.4.3 Exemple 2

Reprenons l'exemple précédent (`Circle` et `Square`).
Version non polymorphique :

```
Square[] s = new Square[10];
Circle[] c = new Circle[20];

void setup() {
  size(200, 200);
  smooth();
  for (int i = 0; i < s.length; i++) {
    s[i] = new Square(100, 100, 10);
  }
  for (int i = 0; i < c.length; i++) {
    c[i] = new Circle(100, 100, 10,
                      color(random(255), 100));
  }
}
```

```
void draw() {
  background(100);
  for (int i = 0; i < s.length; i++) {
    s[i].jiggle();
    s[i].display();
  }
  for (int i = 0; i < c.length; i++) {
    c[i].jiggle();
    c[i].display();
  }
}
```

Version polymorphique (plus simple, factorisée) :

```
Shape[] shapes = new Shape[30];
void setup() {
  size(200, 200);
  smooth();
  for (int i = 0; i < 10; i++) {
    shapes[i] = new Square(100, 100, 10);
  }
  for (int i = 10; i < 30; i++)
    shapes[i] = new Circle(100, 100, 10,
                           color(random(255), 100));
  }
}
void draw() {
  background(100);
  for (int i = 0; i < shapes.length; i++) {
    shapes[i].jiggle();
    shapes[i].display();
  }
}
```

3.5 Notion de référence

3.5.1 Exemple

Reprenons la classe `Car` :

```
Car myCar1;
Car myCar2;

void setup() {
  myCar1 = new Car(color(51), 0, 100, 2);
  myCar2 = new Car(color(151), 0, 200, 1);
}
void draw() {
  myCar1.move();
  myCar1.display();
  myCar2.move();
  myCar2.display();
}
```

Que se passe t'il si l'on affecte `myCar1` à `myCar2` ?

```
Car myCar1;
Car myCar2;

void setup() {
  myCar1 = new Car(color(51), 0, 100, 2);
  myCar2 = myCar1;
}
void draw() {
  myCar1.move();
  myCar1.display();
  myCar2.move();
  myCar2.display();
}
```

Et en modifiant l'attribut `y` de `myCar2` ?

```
Car myCar1;
Car myCar2;

void setup() {
  myCar1 = new Car(color(51), 0, 100, 2);
```

```
  myCar2 = myCar1;
  myCar2.y = 200;
}
void draw() {
  myCar1.move();
  myCar1.display();
  myCar2.move();
  myCar2.display();
}
```

Pour les objets, l'affectation porte sur la référence à l'objet et non sur la valeur.

— Une **référence** n'est pas la donnée elle-même mais seulement une information sur sa localisation

— Exemple avec des `PVector` :

— Quel intérêt à votre avis ?

— Comment faire si l'on souhaite créer un objet presque similaire à un autre ?

Pour copier un objet, il faut faire appel à une méthode spécifique :

```
void testPVector() {
  PVector v1 = new PVector(1,2,3);
  PVector v2 = v1.copy();
  v2.x = 10;
  println(v1);
  println(v2);
}
```

▶ **qui n'existe pas forcément !**

▶ Écrivons une méthode `copy()` pour la classe Car.