M3105C - Programmation Orientée Objet
2. Notions d'orienté objet

frederic.nicolier@univ-reims.fr

URCA - IUT Troyes - GEII

Plan général

1. Paradigmes de programmation
2. Raisonner OO
3. Manipulation d'un objet
4. Constructeur d'une classe

Plan

1.0 :

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

Notion de "Paradigme"

- Un paradigme est un modèle ou "patron" de pensée
- Cela permet de définir un ensemble de règles ou de concepts permettant d'appréhender un cadre particulier
- Ces concepts, adaptés à un domaine précis, permettent de voir un aspect de la réalité
- Ainsi, chaque science dispose d'un ou plusieurs paradigmes adaptés à son sujet d'étude
- En conséquence, la vision de chaque science sur un problème particulier est différente voire hors du scope pour certaines (c'est hors de son champ de perception)
- Exemple : l'astrologie n'a aucun sens pour la Physique moderne mais peut être sujet d'étude pour la Psychologie et/ou la Sociologie

Plan

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 ?

2.1 OBJETS :

- 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,
  - ...

2.1 OBJETS :

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 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).

2.3 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**.

2.3 NOTION DE CLASSE : EXERCICE

- Considérons le concept de *voiture* sous l'approche orientée objet
  - Attributs ?
  - Méthodes ?
  - Exemples d'instances ? (un piège ici)

2.4 NOTIONS ABORDÉES :

- Classe
- Objet
- Instance
- Attribut
- Méthode

Plan

3.1 EN PROG CLASSIQUE : UNE VOITURE

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

Variables globales

- `c` : couleur
- `xpos` : position x
- `ypos` : position y
- `xspeed` : vitesse de déplacement

Setup

- Initialiser la couleur `c`
- Initialiser position : `xpos` et `ypos`
- Initialiser la vitesse `xspeed`

A faire cycliquement

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

3.1 EN PROG CLASSIQUE : CODE C++

- Comment créer un projet dans QtCreator ?
(démo et explication des fichiers présents)
- Projet "minimal" avec appel cyclique (timer Qt)
(explication dans QtCreator)

3.1 EN PROG CLASSIQUE : CODE C++

- À déclarer (ici avec initialisation) :

```
int xpos;
int ypos;
int speed;
QColor col;
```

- Dessin de la voiture (dans `paintEvent`) :

```
painter.setPen(QPen(col, 2));
painter.drawRect(xpos, ypos, 20, 10);
```

- actualisation cyclique (dans `update`) :

```
xpos = xpos + speed;
if (xpos > 400) xpos = 0;
```

3.1 EN PROG CLASSIQUE : CODE PROCESSING

(demo)

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

- un objet, instance de la classe **Car**

Initialisation

- Instancier (= initialiser) la voiture

Actions à réaliser

- Afficher la voiture
- Déplacer la voiture

3.2 EN OO : CODE C++

- Initialisation et instanciation :

```
Car mycar; // dans mainwindow.h
mycar = Car(); // dans mainwindow.cpp (facultatif)
```

- Dessiner l'objet

```
mycar.draw(...);
```

- déplacer la voiture

```
mycar.drive();
```

3.2 EN OO : ÉCRITURE DE LA CLASSE CAR

- Dans le projet : ajout des fichiers `car.h` et `car.cpp`
- Déclaration (méthodes et attributs) dans `car.h` :

```
public:
    void drive();
    void draw(QPainter &p);

    int xpos = 10;
    int ypos = 100;
    int speed = 3;
    QColor c = Qt::blue;
```

3.2 EN OO : ÉCRITURE DE LA CLASSE CAR

- Méthodes dans car.cpp :

```
Car::Car() {
}

void Car::drive() {
    xpos = xpos + speed;
    if (xpos > 400) xpos = 0;
}

void Car::draw(QPainter &p) {
    p.setPen(QPen(c, 2));
    p.drawRect(xpos, ypos, 20, 10);
}
```

3.2 EN OO :

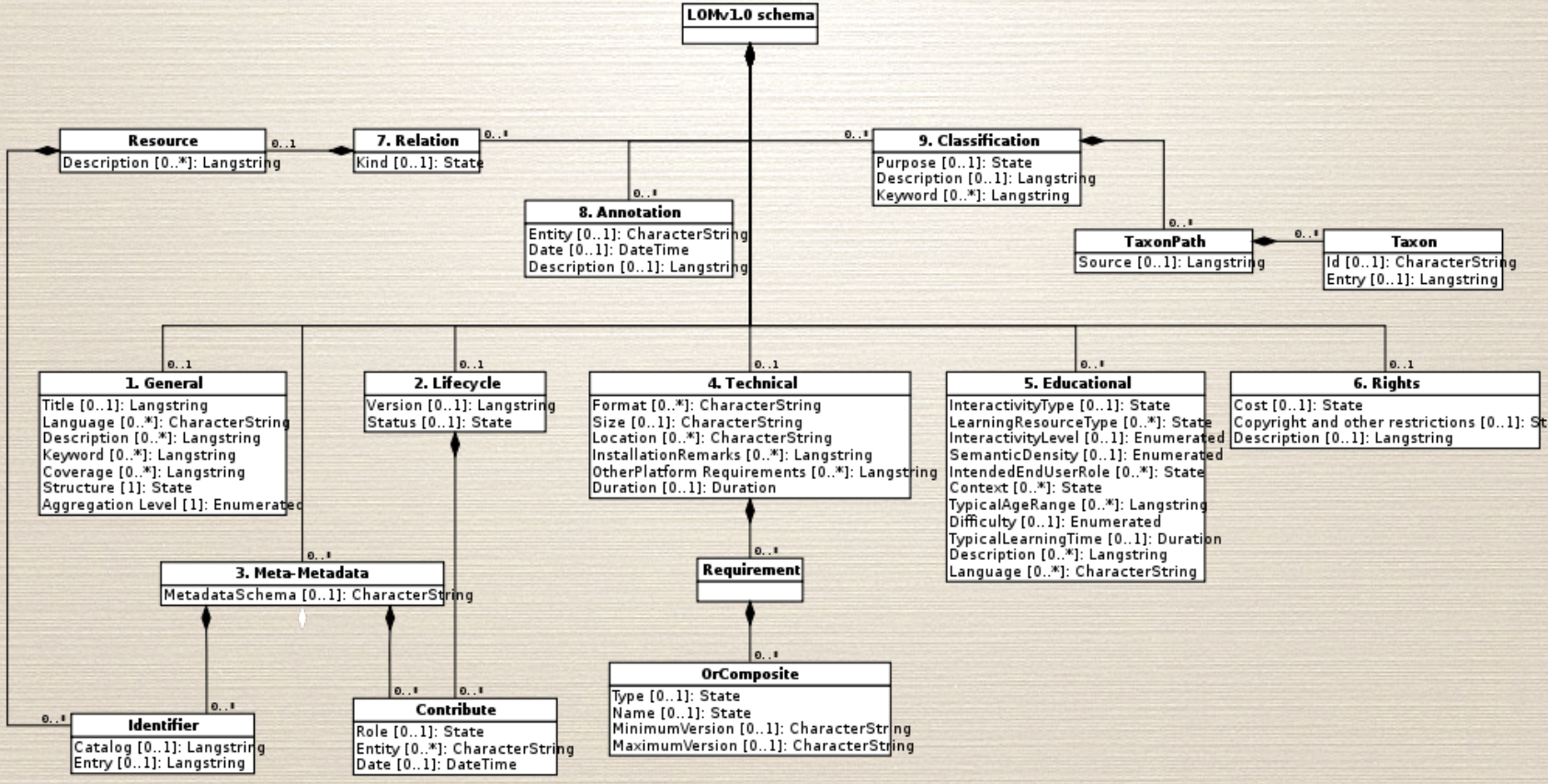
(demo)

3.3 UML : DÉFINITION ET USAGE

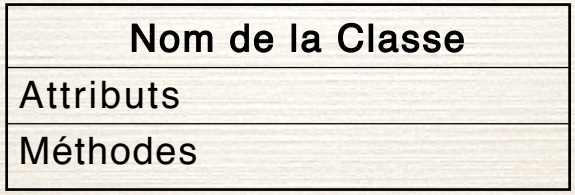
UML = *Unified Modeling Language* (Langage de Modélisation Unifié)

- modélisation graphique
- à base de pictogrammes
- $\Rightarrow$ 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

3.3 UML : UN EXEMPLE



3.3 UML : DIAGRAMME DE CLASSE

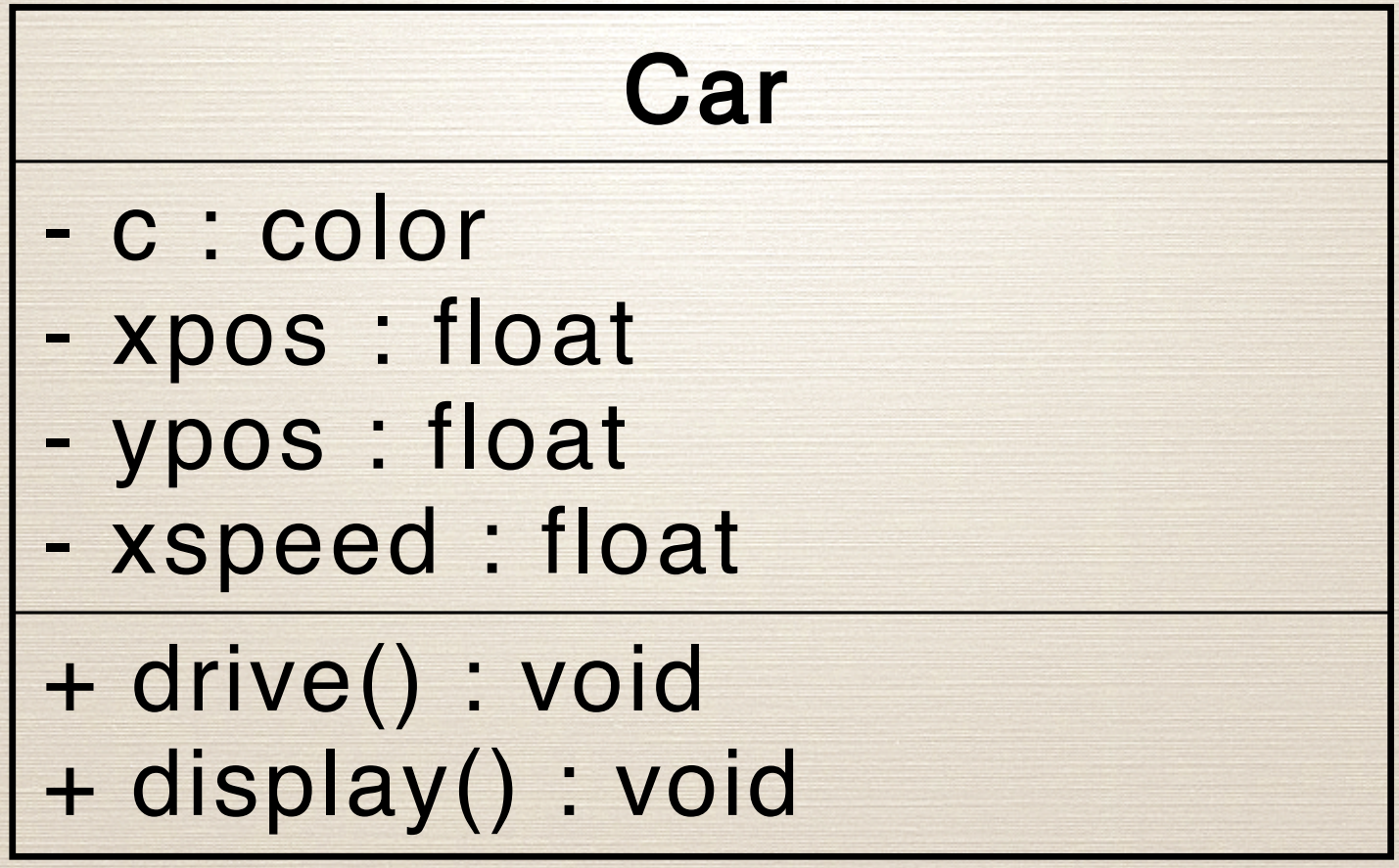


- Nom de la Classe : dans le rectangle du haut
- Attributs :
  Visibilité nomAttribut : typeAttribut = Initialisation
- Méthodes :
  Visibilité nomFonction(directionParamètreN
  nomParamètreN : typeParamètreN) : typeRetour

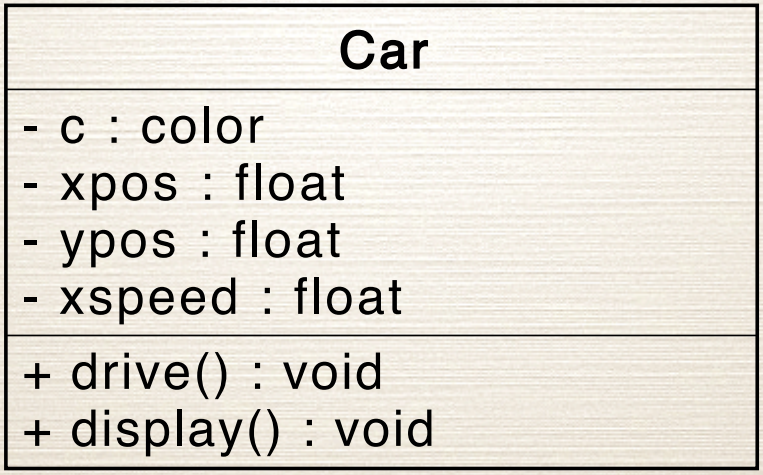
3.3 UML : DIAGRAMME DE CLASSE

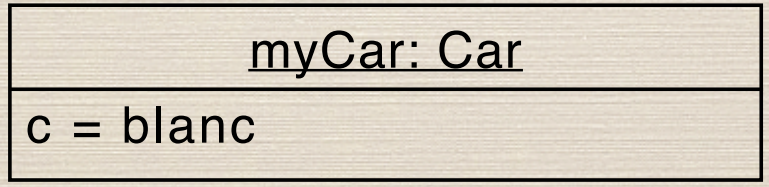
- Visiblité (pour le moment) :
  - + : accès *public* (toutes les autres classes ont accès à cet attribut)
  - - : accès *private* (seule la classe elle-même a accès à cet attribut)
  - # : accès *protected* (idem *private*, mais donne l'accès aux objets des classes filles)
- Direction du paramètre :
  - in : accès rentrant
  - out : accès sortant
  - inout : accès rentrant et sortant

3.3 UML : CLASSE CAR



3.3 UML : DIAGRAMME D'OBJETS





Plan

4.1 PLUSIEURS INSTANCES :

- Dans l'exemple précédent, la voiture était créée de la façon suivante `new` :

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

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

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

$\Rightarrow$ À votre avis ?

$\Rightarrow$ Testons !

4.1 Plusieurs instances :

- Avec le code actuel, `myCar1` et `myCar2` auront un aspect et un comportement 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.

4.1 PLUSIEURS INSTANCES :

- Il faut donc pouvoir spécifier les attributs à la création de l'objet.
- Quelle méthode faut-il modifier ?
  $\Rightarrow$ À votre avis ?

4.1 PLUSIEURS INSTANCES : MODIFICATION DU CONSTRUCTEUR

On souhaite pouvoir écrire par exemple :

```
Car myCar = Car(Qt::blue, 0, 100, 2);
```

Le code du nouveau constructeur est donc :

```
Car::Car(QColor _c, int _xpos, int _ypos, int _speed)
{
    c = _c;
    xpos = _xpos;
    ypos = _ypos;
    speed = _speed;
}
```

4.1 PLUSIEURS INSTANCES : UTILISATION

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

- `mainwindow.h` :

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

- `mainwindow.cpp` :

```
mycar1 = Car(); // facultatif
mycar2 = Car(Qt::red, 100, 300, 2);
```

4.1 PLUSIEURS INSTANCES :

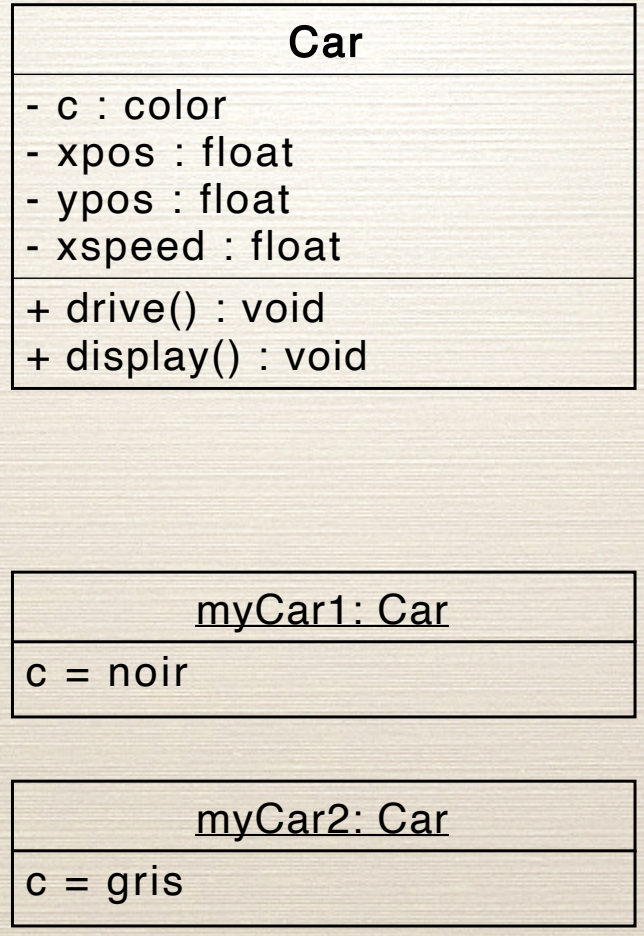
- Ne pas oublier d'ajouter l'appel aux méthodes `draw` et `drive`
- Dans `MainWindow::paintEvent`

```
mycar1.draw(painter);
mycar2.draw(painter);
```

- Dans `MainWindow::update`

```
mycar1.drive();
mycar2.drive();
```

4.1 PLUSIEURS INSTANCES : DIAGRAMMES UML



4.1 PLUSIEURS INSTANCES :

(démonstration)