51BE01IF
Introduction à la programmation 2

Support du cours-td

2014-2015

Table des matières

Présentation de l'enseignement

Principe du cours

Ce cours est centré sur le développement du logiciel de dessin `tagO2` permettant de faire un tag sur une image de fond. Le code complet du logiciel est disponible sur didel. Il ne s'agit pas pour vous de comprendre tout le code (notament pas les affichages et la gestion des clics de souris), mais de donner un objectif aux constructions faites en cours et en tp.
Chaque séance est construite autour d'une ou plusieurs classes dont le code complet sera donné sur didel. Cependant il est possible que le code de `tagO2` ne corresponde pas tout à fait, soit parce que le cours présente plusieurs versions, soit parce que ce qui est présenté en cours correspond aux constructions java connues, mais est facilement perfectible avec d'autres constructions.

Objectifs et organisation

L'objectif de cet enseignement est de se familiariser avec quelques structures de données simples : leur principe théorique, comment les implémenter et comment utiliser l'implémentation fournie en java.
Cet enseignement fait suite à l'enseignement IP1 "Introduction à la programmation 1" qui est un pré-requis fort. Par ailleurs il a lieu parallèlement à l'enseignement CI2 sur lequel nous pourrons parfois nous appuyer.

L'organisation générale est structurée de la façon suivante :

Cours le vendredi de 14h30 à 16h30, en amphi 2A,

TD un créneau de deux heures hebdomadaire,

TP un créneau de deux heures hebdomadaire.

En complément, un système de soutien est mis en place :

Tutorat du lundi au vendredi de 12h30 à 14h30 au SCRIPT

Commission de suivi sur rendez-vous au département sciences exactes

Contrôle de connaissances

L'évaluation se fait au cours de tests (sur papier et/ou machine) et d'examens (sur papier) :

- **Td** résultat des tests effectués en TD
- **Tp** note de contrôle continu TP
- **E0** partiel
- **E1** examen session 1
- **E2** examen session 2

Les notes finales sont calculées selon le principe suivant :

Contrôle continu	**Cc** = 1/3**Tp** + 2/3**Td**
Note d'écrit	**Ne** = max (**E1**, (**E0** + **E1**) / 2)

Note session 1	15%**Cc**+85% **Ne**
Note session 2	max (**E2**, 15%**Cc**+85% **E2**)

Attention : si vous n'avez pas de note de **Tp** ou pas de note de **Td**, vous ne pourrez pas avoir de note finale en session 1.

Chapitre 1

Classes

Attention : ce chapitre n'est pas un cours de programmation objet. On y aborde la notion de classe et d'objets de manière très superficielle et limitée, uniquement dans le but de définir de nouveaux types références.

Note : nous n'aborderons pas la notion de *package* dans ce cours. Les divers fichiers utilisés dans un même programme devront être mis dans le même répertoire, qui sera le répertoire courant au moment de la compilation et de l'exécution.

Ce chapitre s'articule autour de l'étude de la classe `Couleur`. Sont affichés au fur et mesure des extaits de la classe illustrant le propos. La classe complète se trouve en section 1.7. Elle pourra vous être utile pour situer les bouts de code au sein de la classe.

Au cours du premier semestre, ont été introduits et manipulés divers types :

- des types primitifs :
  - **boolean**,
  - **char**,
  - **byte**, **short**, **int**, **long**,
  - **float**, **double**,
- un type référence : `String`.

Pour rappel, un *type* désigne à la fois un ensemble de valeurs possibles et un ensemble d'opérations possibles. Ont également été introduits les types "tableau sur un type `T`", qui permettent de stocker plusieurs valeurs d'un même type (ici `T`), en nombre fixé au moment de l'allocation du tableau (**new**).

Une des utilisations des *classes* est de pouvoir stocker plusieurs valeurs de types divers (différents ou non) et d'y avoir accès à travers une seule variable. Par exemple, si on veut manipuler les points du plan dans un programme, il est plus agréable d'avoir un type représentant un point. On peut imaginer que l'utilisateur de ce type ignore si ce sont les coordonnées cartésiennes (abscisse+ordonnée) ou les coordonnées polaires (module+argument) qui sont stockées dans le type `Point`, mais peu importe du moment qu'il peut le manipuler comme il le souhaite. Si ensuite on veut manipuler des cercles du plan, il sera plus agréable d'avoir un type représentant un cercle qui contient à la fois son centre (qui est un point) et son rayon. On peut également considérer qu'un point doit pouvoir afficher ou fournir ses coordonnées cartésiennes ou polaires, ou qu'un cercle doit pouvoir fournir sa circonférence.

1.1 Classes

Une *classe* est un modèle de donnée qui permet de regrouper des valeurs (les *attributs*) et des actions applicables à ces valeurs (les *méthodes*).
Un *objet* est une *instance* de cette classe s'il en possède les attributs et les méthodes.

1.1.1 Attributs

Si on suit la convention RGB [1], la classe décrivant une couleur contient naturellement trois valeurs entières.

Couleur.java

```
class Couleur{
    int r, g, b;
}
```

La classe `Couleur` étant définie, on peut s'en servir comme de n'importe quel type.
On peut déclarer une variable de type `Couleur` :

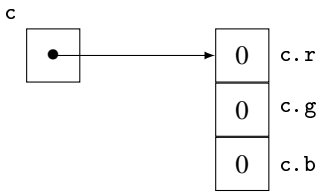
```
Couleur c;
```

Au moment de cette déclaration, est créé en mémoire l'emplacement pour cette variable, mais non pour ses attributs :

Pour créer l'emplacement mémoire de ses attributs, il faut créer un nouvel *objet* représentant une couleur, ce qui se fait par l'opérateur **new** suivi du nom de la classe et de parenthèses :

```
Couleur c = new Couleur();
```

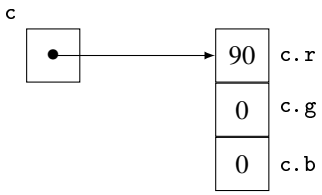
L'objet ainsi créé est une *instance* de la classe `Couleur`.
L'opérateur **new** affecte par défaut aux attributs la "valeur nulle" associée à leurs types respectifs.



On accède à un attribut d'une variable de type `Couleur` en écrivant le nom de cette variable, un point (`.`) et le nom de l'attribut, sans espace :

MainCouleur3.java

```
    Couleur c = new Couleur();
    c.r = 90;
    System.out.println("(" + c.r + "," + c.g + "," + c.b + ")");
```



1. Une couleur est alors composée de 3 coefficients entiers entre 0 et 255, représentant respectivement le rouge, le vert et le bleu.

1.1.2 Constructeurs

Jusqu'à présent, on a toujours créé un nouvel objet de la façon suivante :

```
new MaClasse()
```

En fait, au moment de la création, on peut vouloir spécifier des données liées à l'objet. Par exemple on peut souhaiter créer une instance de couleur représentant le turquoise.
Pour l'instant, on a la possibilité de modifier les attributs de l'objet après sa création, mais il serait plus naturel de pouvoir écrire :

```
Couleur turquoise = new Couleur(37,253,233);
   // codage rgb du turquoise
```

Les constructeurs servent (entre autres) à cela. Un constructeur s'écrit, dans le corps de la classe, de la manière suivante :

- une en-tête comportant :
  - des qualificatifs divers,
  - le nom exact de la classe,
  - une liste de paramètres (types et noms, séparés par des virgules) ;
- un corps dans lequel on spécifie les opérations à effectuer au moment de la création de l'objet.

L'objet qui est en construction se désigne par le mot-clef **this** dans le constructeur.
Exemple : constructeurs de la classe `Couleur` — un pour une couleur dont on spécifie le codage rgb, un autre pour un niveau de gris.

Couleur.java

```
class Couleur{
    int r, g, b;
    public Couleur(int r, int g, int b){
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public Couleur(int gris){
        this.r = this.g = this.b = gris;
    }
}
```

On peut alors créer une instance de la classe `Couleur` en spécifiant les valeurs de ses attributs :

```
Couleur turquoise = new Couleur(37,253,233);
```

Attention : par défaut chaque classe possède un constructeur sans argument qui se contente d'allouer la mémoire nécessaire aux attributs[2]. Si on écrit explicitement un constructeur, ce constructeur par défaut n'existe plus ; il faut donc le réécrire si on souhaite l'avoir :

Couleur.java

```
   /** la couleur par defaut est le blanc */
    public Couleur(){
        this(255, 255, 255);
    }

    public Couleur(int r, int g, int b){
```

2. En fait il commence par appeler le constructeur vide de la classe mère, notion qui ne sera pas abordée avant le cours de POO.

```
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public Couleur(int gris){
        this.r = this.g = this.b = gris;
    }
```

Comme montré sur l'exemple précédent, on peut invoquer un constructeur à l'intérieur d'un autre constructeur (de la même classe) :

— uniquement en première instruction,
— par l'appel à **this**`(...)` (avec les bons arguments).

On peut écrire autant de constructeurs que souhaité, à condition que deux constructeurs quelconques aient des signatures différents, c'est-à-dire qu'ils aient toujours une liste de paramètres différents (nombre, types et/ou ordres des paramètres). On peut bien entendu appeler une méthode ou une fonction dans un constructeur.
On peut aussi mettre des valeurs par défaut pour les attributs :

```
class Couleur{
    int r=255, g=255, b=255;
}
```

Dans ce cas elles sont attribuées avant tout appel à un constructeur.

1.1.3 Attributs de classe

Un utilisateur qui veut utiliser du bleu devra écrire **new** `Couleur(0,0,255)` , s'il veut utiliser du orange ce sera **new** `Couleur(255,127,0)`. Il serait nettement plus agréable que certaines couleurs courantes soient prédéfinies.
Pour cela, on peut ajouter dans la classe la définition par défaut de certaines couleurs, par exemple :

```
// attention: tres mauvaise idee
    Couleur rouge = new Couleur(255,0,0);
    Couleur vert = new Couleur(0,255,0);
    Couleur bleu = new Couleur(0,0,255);
    Couleur orange = new Couleur(255,127,0);
```

Cette manière de faire pose un gros problème : à chaque création d'une nouvelle instance de `Couleur`, on crée 4 autres instances de `Couleur` (respectivement pour le rouge, le vert, le bleu et l'orange), ou plus si on veut prédéfinir plus de couleurs, et pour chacune de ces instances on recrée 4 instances de `Couleur`, etc. de manière récursive. Le programme finit par planter.
La solution consiste à créer un seul objet par couleur prédéfinie pour toute la classe : chaque instance de cette classe partagera l'objet. Cela se fait avec le mot clef **static** :

Couleur.java

```
    static Couleur rouge = new Couleur(255,0,0);
    static Couleur vert = new Couleur(0,255,0);
    static Couleur bleu = new Couleur(0,0,255);
    static Couleur orange = new Couleur(255,127,0);
```

Ces attributs sont accessibles directement à travers la classe `Couleur`, sans qu'il y ait eu nécessairement création d'une instance de cette classe : `Couleur.bleu` par exemple. Ils sont créés à la première "utilisation" de la classe, que ce soit pour accéder à l'un d'entre eux ou pour créer un nouvel objet.

1.2 Utilisation d'une classe comme type

1.2.1 Type d'attributs

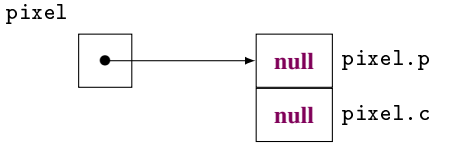
On peut maintenant utiliser le type `Couleur` comme n'importe quel type, notamment pour déclarer un attribut d'un nouveau type.
Pour représenter un pixel coloré, on définit les classes suivantes :

```
class Pixel{
    int x, y;

    public Pixel(int x, int y){
        this.x = x;
        this.y = y;
    }
}

class PixelColore {
    Pixel p;
    Couleur c;
}
```

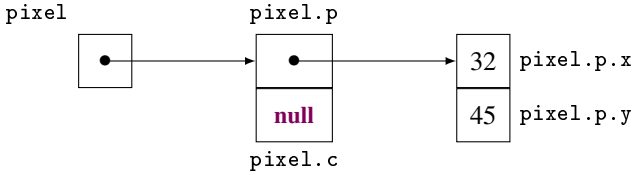
Pour créer un objet représentant un pixel bleu en $(32, 45)$, on écrit :

```
PixelColore pixel;
```

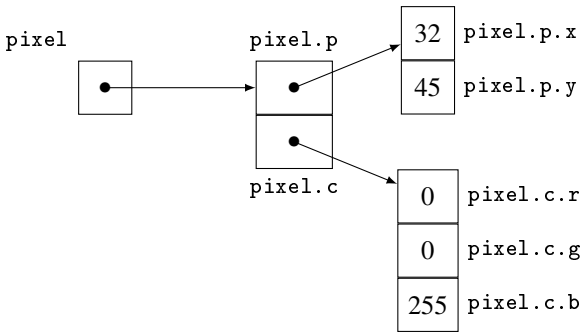

```
pixel = new PixelColore();
```




```
pixel.p = new Pixel(32, 45);
```




```
pixel.c = Couleur.BLEU;
```



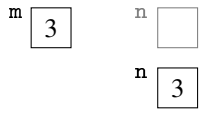
► **Exercice 1** :

Ecrire des constructeurs pour la classe `PixelColore`.

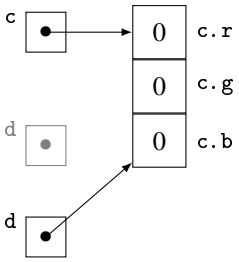
1.2.2 Valeur d'une variable, d'un attribut ou d'un paramètre

La valeur d'une variable, d'un attribut ou d'un paramètre de type `Couleur` est l'adresse de l'emplacement mémoire où se trouve l'instance de `Couleur`. Ainsi son affectation à une autre variable ou un autre attribut ne copie que cette adresse et pas le contenu de la variable, de l'attribut ou du paramètre.
Comparez :

```
int m = 3, n;
n = m;
```




```
Couleur c = Couleur.bleu, d;
```




```
d = c;
```

1.2.3 Paramètres et valeurs de retour des fonctions

Comme tous les types, les types `Couleur`, `Pixel` et `PixelColore` que nous venons de créer peuvent servir comme type de retour d'une fonction ou comme type de paramètre dans une fonction.
Les fonctions (telles que vues en IP1), sont des méthodes qualifiées de **static** : elles sont communes à toutes les instances de la classe.

Type de retour d'une fonction

Comme d'habitude, il faut annoncer le type de retour dans l'en-tête et mettre une expression compatible après le **return** :

```
static Pixel enHautAGauche(){
    return new Pixel(0, 0);
}
```

On peut affecter la valeur de retour de cette fonction à une variable de type compatible :

```
Pixel origine = enHautAGauche();
```

Type du paramètre d'une fonction

Les types `Couleur`, `Pixel` et `PixelColore` nouvellement créés peuvent aussi être utilisés pour passer des paramètres :

```
static boolean estHorizontal(Pixel extremite1, Pixel extremite2){
    return extremite1.y==extremite2.y && extremite1.x!=extremite2.x;
    // on verifie que les deux points sont distincts
    // et alignes horizontalement
}
```

On peut avoir des paramètres *et* un retour de types créés par nos soins, bien sûr :

```java
static Pixel milieu(Pixel extremite1, Pixel extremite2){
    return new Pixel((extremite1.x + extremite2.x) / 2,
                     (extremite1.y + extremite2.y) / 2);
}
```

On rappelle que les passages se font toujours *par valeur* en java. La valeur d'une variable de type référence (tableau ou classe) est l'adresse de l'emplacement mémoire où se trouve le contenu de cette variable. Une fonction (ou un constructeur) ne peut modifier la valeur de cette variable, mais peut tout à fait modifier son contenu.

1.3 Rappel : constantes

Les constantes se déclarent grâce au qualificatif **final**. Ce qualificatif peut aussi bien s'appliquer à une variable qu'à un attribut [3]. Une variable ou un attribut déclaré **final** ne peut recevoir qu'une affectation lors de l'exécution du programme.
Attention : on peut modifier le contenu d'une variable ou d'un attribut de type référence déclaré comme **final**, seule sa valeur (ie l'adresse de l'emplacement mémoire où se trouve son contenu) ne peut être modifiée.

1.4 Méthodes

On peut également ajouter des opérations qui pourront être *invoquées* sur une instance de cette classe : les *méthodes*. Une méthode est naturellement liée à une instance de la classe et son exécution n'a de sens que sur une telle instance :

Couleur.java

```java
    private final static double WR = 0.299, WB = 0.114, WG = 1-WR-WB;

    public double luminance(){
        return WR*r + WG*g + WB*b;
    } // pas d'ambiguite: on peut utiliser r ou this.r
```

L'objet sur lequel la méthode est invoquée est désigné par le mot-clef **this** à l'intérieur de la méthode. On peut l'omettre s'il n'y a pas d'ambiguïté.

Exemple d'invocation de méthode :

```java
public static Couleur[][] noirEtBlanc(Couleur[][] image){
    Couleur[][] nEtB = new Couleur[image.length][image[0].length];
    for(int i=0; i<image.length; i++)
        for(int j=0; j<image[i].length; j++)
            nEtB[i][j] = new Couleur((int) image[i][j].luminance());
    return nEtB;
}
```

Dans une méthode, on désigne l'objet sur lequel la méthode est invoquée par le mot-clef **this** (comme dans un constructeur).
Attention 1 : On ne peut pas invoquer une méthode sur **null** !
Attention 2 : On ne peut utiliser le mot-clef **this** à l'intérieur d'une fonction (méthode statique), car elle n'est pas invoquée sur un objet, mais sur une classe.

3. Et même à une classe, mais ce n'est pas l'objet de ce cours.

1.5 Qualificatifs private / public

Jusqu'à présent, on a toujours eu la possibilité de modifier l'attribut d'un objet en lui affectant une nouvelle valeur. Mais ce n'est pas très raisonnable de penser que l'utilisateur peut modifier le contenu de l'objet représentant une couleur prédéfinie. S'il veut définir une autre couleur, il peut créer une nouvelle instance de `Couleur`, mais `Couleur.bleu` doit représenter toujours la même couleur.
On a donc besoin de cacher à l'utilisateur les attributs correspondant aux composantes rgb. Pour cela, on qualifie ces attributs de **private** :

```
class Couleur{
    private int r, g, b;
}
```

Si malgré tout on veut que l'utilisateur puisse lire ces valeurs, on ajoute des *getters* qui le lui permettront :

```
class Couleur{
    private int r, g, b;

    public int getRed(){
        return r;
    }
    // et de meme pour les autres couleurs
}
```

Souvent le qualificatif **private** est utilisé pour "protéger" des attributs : ici, c'est pour que l'utilisateur ne puissent pas les modifier, mais cela peut aussi être quand le domaine de valeurs d'un attribut est restreint, par exemple un poids peut être représenté par un **double**, mais il est toujours positif ; dans ce cas, on crée un attribut

```
private double poids;
```

Et on crée un *setter* qui contrôle les valeurs qu'on peut donner à l'attribut :

```
void setPoids(double poids){
    this.poids = (poids>=0) ? poids : 0;
}
```

Quand un attribut est qualifié de **private**, il n'est pas visible ailleurs que dans la classe. L'utilisateur d'une telle classe n'est a priori pas même conscient de l'existence de l'attribut. C'est ce qu'on appelle l'*encapsulation*.

Le qualificatif **public** signifie que l'attribut peut être vu de l'extérieur [4].

On peut également qualifier une méthode ou un constructeur de **private** ou de **public**. La signification est la même.

1.6 Existant dans l'API

L'API [5] java fournit une classe pour la couleur : `java.awt.Color`.

4. Il existe d'autres qualificatifs d'accès, que nous ne verrons pas dans ce cours.
5. Application Programming Interface

1.7 La classe Couleur complète

Couleur.java

```
/** gestion des couleurs en RGB
  r: composante rouge
  g: composante verte
  b: composante bleue
*/

class Couleur{
    private int r, g, b;
    private final static double WR = 0.299, WB = 0.114, WG = 1-WR-WB;


    static Couleur rouge = new Couleur(255,0,0);
    static Couleur vert = new Couleur(0,255,0);
    static Couleur bleu = new Couleur(0,0,255);
    static Couleur orange = new Couleur(255,127,0);

   /** la couleur par defaut est le blanc */
    public Couleur(){
        this(255, 255, 255);
    }

    public Couleur(int r, int g, int b){
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public Couleur(int gris){
        this.r = this.g = this.b = gris;
    }

    public double luminance(){
        return WR*r + WG*g + WB*b;
    } // pas d'ambiguite: on peut utiliser r ou this.r

    public int getRed(){
        return r;
    }

    public int getGreen(){
        return g;
    }

    public int getBlue(){
        return b;
    }
}
```