

JAVA

- Héritage -

Ninon Devis: ninon.devis@ircam.fr

Philippe Esling: esling@ircam.fr

License 3 Professionnelle - Multimédia

Retour projet: **La Bataille**

Faire un programme :

1. Créer une classe Bataille qui contiendra le **main**
2. Créer une classe Carte
 - a. Contient un constructeur pour créer une carte
 - b. Attributs : couleur (parmi un tableau statique) et valeur (idem)
 - c. Méthode : accesseurs, modificateurs et comparateur
3. Créer une classe Joueur
 - a. Propriétés : tableau de cartes et compteur de points
 - b. Méthode : tire une carte et ajoute une carte
4. Ecrire le programme de jeu principal
 - a. Crée deux joueurs.
 - b. Initialise un paquet de cartes et le mélange aléatoirement (`Math.random`)
 - c. Effectue une boucle de jeu en affichant les scores (`System.out.println`)
 - d. Affiche le vainqueur de la partie

Rappels & Quiz

```
1 package pobj.cours3;
2
3 public class Point {
4     // attributs
5     private double x, y;
6     // constructeurs
7     public Point(double a, double b){x=a;y=b;}
8     // public Point(){x=0;y=0;}
9     // accesseurs
10    public double getX(){return x;}
11    public double getY(){return y;}
12    // me'thodes
13    private void moveto (double a, double b){x=a;y=b;}
14    public void rmoveto (double dx, double dy){x+=dx;y+=dy;}
15    public double distance(){
16        double x = this.getX();
17        double y = this.getY();
18        return Math.sqrt(x*x+y*y);
19    }
20    // me'thodes pre'de'finies (standards)
21    public String toString(){return (" "+x+", "+y+"");}
22 }
```

Rappels & Quiz

- Quel est le nom du paquetage de la classe?
- Quel est le nom de la classe?
- Quel est le nom du fichier? Son chemin d'accès?
- Quels sont les constructeurs et leur signature?
- Quelles sont les méthodes publiques et leur signature?
- Quelles sont les méthodes privées et leur signature?
- Quelles sont les méthodes prédéfinies?

Plan du cours

- I. Packages
- II. Objets et Classes
- III. UML
- IV. Héritage
- V. Interface
- VI. Classes Abstraites
- VII. Modificateurs
- VIII. Projet

Packages

Permettent le regroupement de classes et interfaces dans un archivage hiérarchique

(~ dossiers qui permettent de classer des fichiers)

- La déclaration d'un paquetage est la première instruction d'un fichier:

```
1 package nom.[nom]*;
```

- *Exemple:* pour inclure la classe Point dans l'extension 'mesPackages.sousPackage1', écrire au début du fichier Point.java :

```
1 package mesPackages.sousPackage1;
```

- Pour pouvoir utiliser la classe Point dans une autre classe, deux solutions:

```
1 import mesPackages.sousPackages1.Point;
```

```
1 import mesPackages.sousPackages1.*; ← Autorise les références abrégées
```

Packages

Quelques précisions:

- Collisions Interdites.
- Si rien n'est précisé, le programme est considéré comme faisant partie du “paquetage anonyme” par défaut.
- La hiérarchie des paquetages suit la hiérarchie des dossiers du système.
- L'appartenance à un paquetage modifie les règles de visibilité.

Objets & Classes

Java dynamique: programmation objet

Les variables et les méthodes d'instances:

- ne sont pas déclarées avec `static`
- sont allouées à la création d'une instance (avec le mot-clé `new`)
 - état local à l'instance pour les variables
 - table des méthodes commune pour toutes les instances de la même classe.
- existent pour chaque instance
- sont accessibles par la notation "point" si elles sont `public`.

`o.m(a)` = appel de méthode `m` avec la paramètre `a` sur l'objet `o` d'une classe `c` de construction. (c'est à dire un objet `o` créé par un constructeur défini dans la classe `c`)

Objets & Classes

Java statique: programmation modulaire

Les variables et les méthodes de classes:

- sont déclarées avec `static`
- existent dès le chargement de la classe (sans instance)
- existent pour en un unique exemplaire
- sont accessibles par la notation “point” également.

Objets & Classes

Représentation des objets

Rappels:

- Un objet est une instance de classe et une classe peut avoir plusieurs instances
- Pour cela un objet est formé:
 - d'un état local contenant les valeurs des variables d'instance
 - + la valeur de lui-même (emploi du mot-clé `this`)
 - d'une table des méthodes d'instance contenant l'ensemble des méthodes d'instance de la classe et les méthodes prédéfinies.

Objets & Classes

Constructeur

Chaque classe possède au moins un constructeur pour **initialiser un nouvel objet** de ce type.

- Méthode au même nom que la classe
- sans type de retour
- appelé dès l'allocation, avec l'utilisation de `new`
- L'allocation par un constructeur est explicite en utilisant la construction `new`.

Objets & Classes

Emploi de this

- Correspond à une référence sur l'objet en cours d'exécution d'une méthode, permet de le référencer.

```
1 public double distance(Point p2){  
2     double dx = p2.getX() - this.getX();  
3     double dy = p2.getY() - this.getY();  
4     return Math.sqrt(dx*dx + dy*dy);  
5 }
```

- peut être utilisé en notation qualifiée

```
1 Point (double x, double y) {this.x = x; this.y = y;}
```

- ne peut pas être utilisé dans une méthode statique car il faut qu'un objet this ait été créé

Objets & Classes

Mémoire et égalité

- Du point de vue mémoire, **un objet est une référence sur une zone allouée** contenant les variables d'instance et la table des méthodes d'instance.
- La valeur **null**, le pointeur nul, est une valeur acceptable pour toute variable dont le type n'est pas primitif.
- C'est la valeur par défaut de toute variable du type d'une classe
 - **pointeur = référence = adresse mémoire**
 - pointeur nul = **null**
- Les objets et les tableaux **sont des références**. Pas les types de base.
- L'opérateur **==** teste uniquement les adresses, pas le contenu de la zone pointée.
- Utiliser la méthode **equals(o)** prédéfinie.

UML *Unified Modeling Language*: langage de modélisation graphique

- Notation semi-formelle standardisée de modélisation.
 - ➔ Développée par l'OMG (Object Management Group)
- Utile pour la rédaction des documents de travail, utilisation généralisée.
- Accent mis sur la *description*, pas sur la justification.
- Le système est segmenté en *vues*.
 - ➔ Deux types de vue: dynamique et statique.

Types de composants
<i>une classe</i>
une classe dans un état donné
un objet
un noeud ou une ressource logicielle
un rôle
<i>une interface</i>
un acteur
un cas d'utilisation
un sous-système

C'est une vue *statique* décrivant l'*organisation* des composants.

→ Une notation existe pour chacun de ces types de composants et de la relation qui les lie.

Relations entre composants
généralisation
<i>association</i>
<i>dépendance</i>
réalisation

Point

-x : double

-y : double

+getX() : double

+getY() : double

-movetoi(a : double, b : double)

+rmoveto(dx : double, dy : double)

+distance() : double

+distance(p2 : Point) : double

+toString() : String

→ **Nom de la classe** (en *italique* si abstraite)

→ **Liste des attributs.**

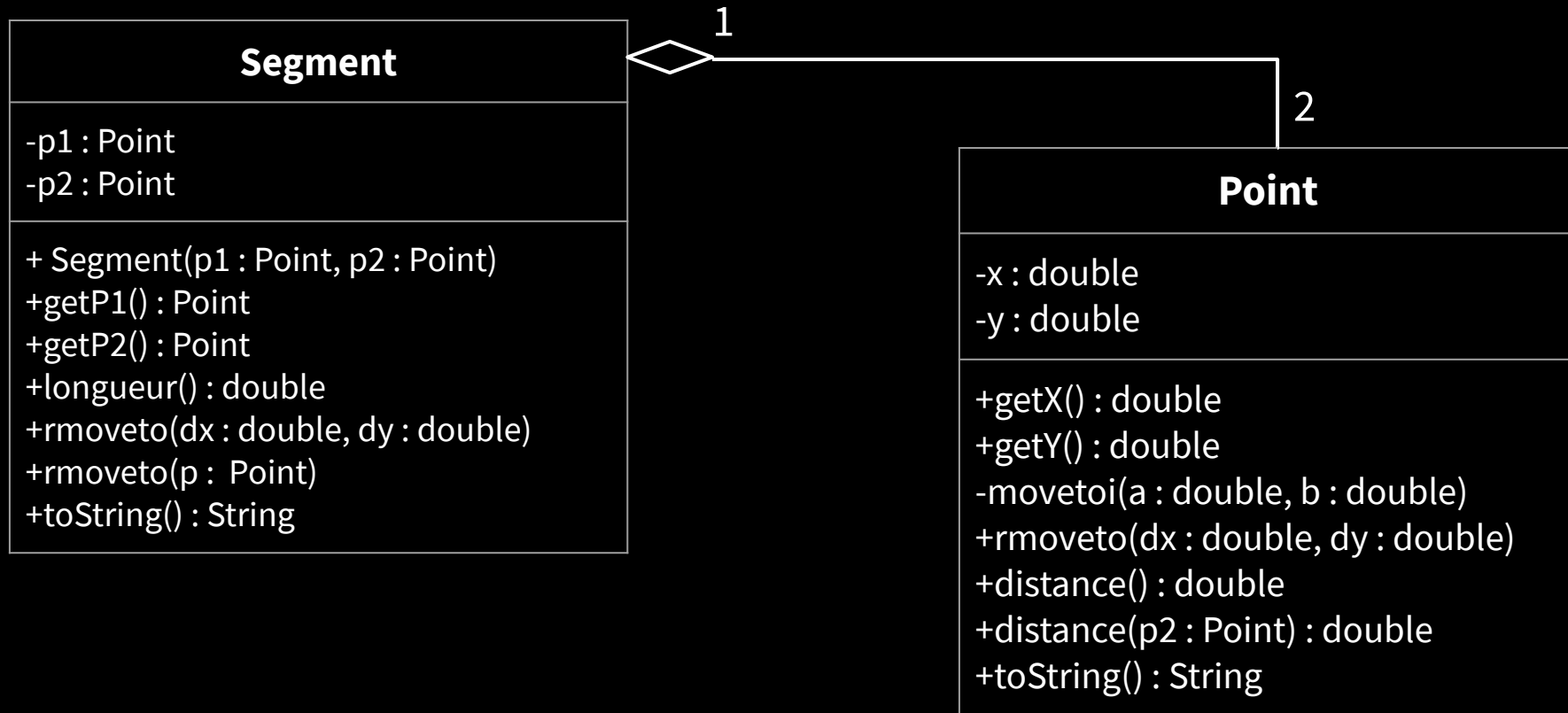
- un *modificateur*: + pour public, - pour privé, # pour protégé.
- le *nom* et le *type* de l'attribut.

→ **Liste des opérations.**

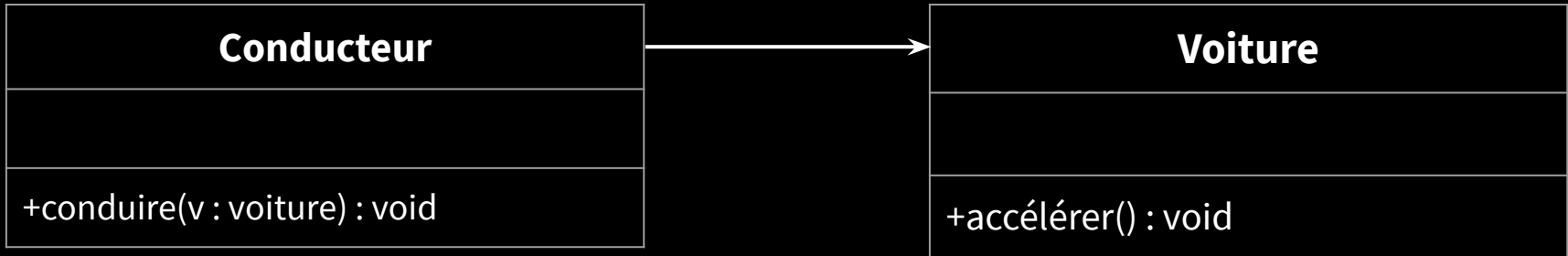
- un *modificateur*: + pour public, - pour privé, # pour protégé.
- le *nom* de l'opération, ses *arguments* et leur *type*, le *type de retour*.

- **Ligne pointillée** terminée par une flèche
- Dénote l'*existence nécessaire* de certains composants pour le bon fonctionnement d'un composant en particulier.
- *Différents types* de dépendances: call, bind, access, derive, friend, import, instantiate, parameter, realize, refine, send, trace, use.

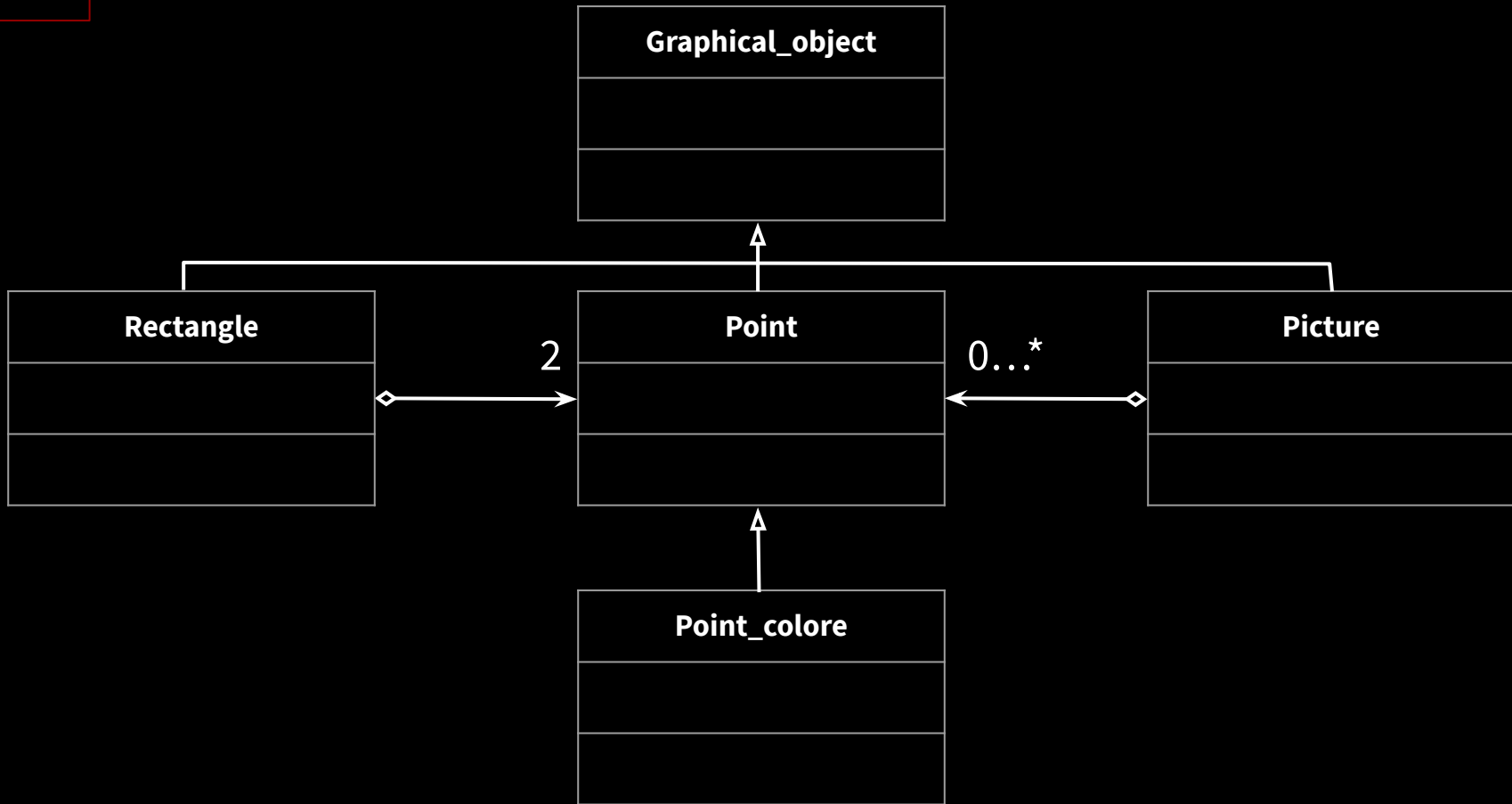
- **Ligne pleine** dont les extrémités sont optionnellement annotées par:
 - Une multiplicité:
 - * signifie un nombre indéterminé
 - $n \in \mathbb{N}$, un nombre n fixé
 - $m \dots n$, $(m, n) \in \mathbb{N}^2$, un nombre entre m et n
 - Un losange plein: signifie que l'association est une composition
 - Un losange vide: signifie que l'association est une agrégation
- **La composition**: B compose A si B “fait partie de” A
 - B ne peut pas exister sans A, si A disparaît B également.
- **L'agrégation**: relation “a un” entre deux classes. (peut être “a des”, “est composé de”...)
 - Voiture “a un” Moteur, Segment “a deux” Point



- Un Conducteur “conduit” une voiture
- sens de l'association “conduit”



- Ligne pleine se terminant par une flèche.
- Entre classes, elle dénote qu’une classe est plus *générale* qu’une autre.
- Soient A et B deux classes, elle exprime que “B est un A”
→ Un objet de la classe B peut être utilisé en lieu et place d’un objet de la classe A.
- Principe de *subsomption*: relation d’inclusion entre des concepts.



Héritage

Concept primordial et l'un des mécanismes les plus puissants de la programmation orientée objet

- L'héritage est la définition d'une classe par extension des caractéristiques d'une autre classe.
 - les variables et les méthodes décrites par la classe originale sont toujours utilisées
- Permet de reprendre les membres d'une classe (appelée *super-classe* ou *classe mère*) dans une autre classe (nommée *sous-classe*, *classe fille* ou *classe dérivée*) qui en hérite.
- En JAVA, ce mécanisme est mis en oeuvre au moyen du mot-clé `extends`

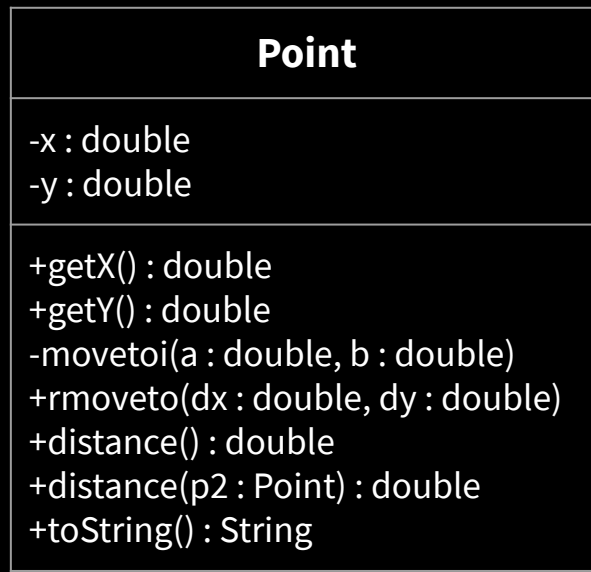
- Exemple:

```
1 public class Vehicule
2 {
3     public int vitesse;
4     public int nombre_de_places;
5 }
6
7 public class Automobile extends Vehicule
8 {
9     public Automobile()
10    {
11        this.vitesse = 90;
12        this.nombre_de_places = 5;
13    }
14 }
```

} Le constructeur défini dans la classe Automobile peut permettre d'initialiser les attributs

Héritage

Exemple:



```
1 package pobj.cours3;
2
3 public class Point {
4     // Attributs
5     private double x, y;
6     // Constructeur
7     public Point(double a, double b){x=a; y=b}
8     // Accesseurs
9     public double getX(){return x;}
10    public double getY(){return y;}
11    // Méthodes
12    private void moveto (double a, double b){x=a; y=b}
13    public void rmoveto (double dx, double dy){x+=dx; y+=dy;}
14    public double distance(){
15        double x = this.getX();
16        double y = this.getY();
17        return Math.sqrt(x*x + y*y);
18    }
19    // Méthodes prédéfinies (standards)
20    public String toString(){return "(" + x + ", " + y + ")";}
21 }
```

```
1 package pobj.cours3;
2
3 public class PointColore extends Point {
4     private String couleur;
5     public PointColore (double x, double y, String c){
6         super(x, y);
7         this.couleur = c;
8     }
9     public PointColore(){couleur = "Indéfinie"}
10    public String getCouleur(){return couleur;}
11    public String setCouleur(){couleur = c;}
12    public String toString(){
13        return super.toString() + "-" + this.getCouleur();
14    }
15 }
```


Héritage

Super et this

- **this**: représente l'objet courant de la classe de définition.
- **super**: représente l'objet courant vu de la classe ancêtre.

```
1 package pobj.cours3;
2
3 public class PointCouleur extends Point {
4     private String couleur;
5     public PointCouleur (double x, double y, String c){
6         super(x, y);
7         this.couleur = c;
8     }
9     public PointCouleur(){couleur = "Indéfinie"}
10    public String getCouleur(){return couleur;}
11    public String setCouleur(){couleur = c;}
12    public String toString(){
13        return super.toString() + "-" + this.getCouleur();
14    }
15 }
```

- *super* permet d'accéder aux membres de la super-classe d'une classe, de la même manière que l'on accède aux attributs de la classe elle-même à l'aide de *this*.
- Permet de distinguer les redéfinitions (`super.toString()`) ou de nommer les constructeurs ancêtres (`super(x, y)`).

Héritage

Super et this

- La liaison avec super peut être résolue à la compilation (on connaît l'adresse de la méthode de la classe ancêtre)
- Ne marche qu'à un niveau: il n'y a pas de super.super.méthode()

```
12 public String toString(){  
13     return super.toString() + "-" + this.getCouleur();  
14 }
```

- Ce sera toujours le toString de Point qui sera appelé dans la méthode toString de PointCouleur.
- S'il n'y a pas d'appel explicite d'un constructeur de super, alors l'appel super(); est ajouté en première instruction du constructeur de la classe fille. Ainsi, écrire:

```
9 public PointCouleur(){couleur = "Indéfinie";}
```

Sera automatiquement remplacé par:

```
9 public PointCouleur(){super(); couleur = "Indéfinie";}
```

Héritage

Exécution de l'exemple dans le main:

```
1 package pobj.cours3;
2
3 class ExPoint {
4     public static void main(String[] args){
5         Point p0 = new Point();
6         Point p1 = new Point(2, 3);
7         PointColore pc0 = new PointColore();
8         PointColore pc1 = new PointColore(2, 3, "Rouge");
9
10        System.out.println(p0 + " " + p1);
11        System.out.println(pc0 + " " + pc1);
12    }
13 }
```

En sortie:

```
1 > java pobj/cours3/Expoints
2 (0.0, 0.0) (2.0, 3.0)
3 (0.0, 0.0)- Indéfinie (2.0, 3.0)- Rouge
```

Héritage

Remarques importantes

- En Java, toute définition de classe étend une classe existante:
 - Si rien n'est précisé alors on étend la classe Object: c'est la racine de la hiérarchie de classe.
 - Lors de l'instanciation, la classe fille reçoit les caractéristiques héritées de sa classe mère ou super-classe, qui elle même reçoit celles de sa propre super-classe et ce récursivement jusqu'à la classe Object.
 - *Une classe hérite toujours d'une autre !*
- En Java, *l'héritage met en relation de généralisation* la sous-classe à la super-classe.
- Java *ne permet pas l'héritage multiple*: une classe dérive toujours d'une et une seule classe.

Interface

Une interface représente un ensemble d'opérations caractérisant un comportement.

- Liste des méthodes dont on donne seulement la *signature* (nom + liste des paramètres qu'elle accepte en entrée) et de déclaration de variables.
- Représente ce qu'on attend d'un objet.
- Peut être implémenté par une ou plusieurs classes qui doivent donner une implémentation de chacune des méthodes annoncées (et éventuellement d'autres).
- Une classe peut implémenter plusieurs interfaces (permettant ainsi l'héritage multiple).
- Une interface n'a pas de constructeurs.
- Elle n'est pas instanciable.

Interface

Exemple: interface copiable

```
1 interface Copiable{  
2     Object copier()  
3 }
```

Création de l'interface

```
1 class Point implements Copiable {  
2     ...  
3     public Object copier() {  
4         return new Point(this.x, this.y);  
5     }  
6 }  
7  
8 class PointCouleur extends Point {  
9     ...  
10    public Object copier() {  
11        return new PointCouleur(this.x, this.y, this.couleur);  
12    }  
13 }
```

Implémentation de l'interface par les deux classes: Point et PointCouleur.

→ Une classe qui hérite d'une classe qui implémente une interface, l'implémente aussi.

```
1 Copiable cp1 = new Point(10, 20);  
2 Copiable cp2 = new PointCouleur(1, 1, "Noir");
```

Instanciation des classes pour accéder à la méthode de l'interface.

Interface

Héritage et interface

Une interface peut hériter d'une autre interface et même de plusieurs interfaces. Les méthodes de cette interface correspondent à l'union des méthodes héritées et des méthodes déclarées.

Exemple:

```
1 public interface MouseInputListener extends MouseListener, MouseMotionListener
```

Les interfaces permettent d'introduire une **couche d'abstraction** supplémentaire à la programmation qui la rend ainsi **plus flexible**.

Classes Abstraites

- A mi-chemin entre les classes et les interfaces.
- Comme les interfaces, les classes abstraites ne sont **pas instanciables**.
- Les classes abstraites sont déclarées par le modificateur **abstract**.
- Intérêts à définir des classes abstraites:
 - faire de la factorisation de code avec des implémentations partielles.
 - permettre un maximum de partage du code
- Ce sont des classes dont certaines méthodes ne possèdent pas de corps.
 - ➔ Si une sous-classe d'une classe abstraite redéfinit toutes les méthodes de l'ancêtre alors elle devient concrète, sinon elle reste abstraite.

Classes Abstraites

Exemple

```
1 abstract class Forme {
2     public void affiche () {
3         System.out.println ("Je suis " + this.who_s_i_am ());
4     }
5     public abstract String who_s_i_am () ;
6 }
7
8 class Carre extends Forme {
9     public String who_s_i_am () { return ("un carre") ; }
10 }
11
12 class Cercle extends Forme {
13     public String who_s_i_am () { return ("un cercle") ; }
14 }
15
16 class TestForme {
17     public static void main (String[] args) {
18         Forme c1 = new Cercle();
19         Forme c2 = new Carre();
20         c1.affiche();
21         c2.affiche();
22     }
23 }
```

`affiche()` est
complètement définie
contrairement à
`who_s_i_am()`

Implémentation de la
méthode `who_s_i_am()`
par deux classes

Modificateurs

Autorisation d'accès par modificateur de visibilité

- **public**: accessibles pour tous les objets.
- **private**: accessibles dans la classe de définition.
- **protected**: accessibles dans les sous-classes et classes du même paquetage.

Remarques:

- Il n'y a qu'une seule classe publique dans un fichier .java: elle porte le nom de ce fichier!
- Généralement les attributs d'une classe sont déclarés en **private** et nécessite de créer des méthodes `get()` pour y accéder et `set()` pour les modifier.

Modificateurs

Autorisation d'accès par modificateur de visibilité

Modificateur	Classe	Paquetage	Sous-classe	Monde
public	oui	oui	oui	oui
protected	oui	oui	oui	
default	oui	oui		
private	oui			

- Une classe a toujours accès à tous ses champs.
- Les classes du même paquetage ont accès à tous les champs sauf les privés
- Une sous-classe d'un autre paquetage n'a pas accès aux champs protégés

Modificateurs

Modificateurs généraux

- **final**:
 - Devant une *variable* il la rend **immuable**.
 - Pour un *objet* il **fige la référence** et non la valeur de la référence (seule l'instanciation est figée).
 - Devant une *classe* il **empêche l'héritage**, cette classe ne peut pas avoir de sous-classe.
 - Devant une *méthode* il rend cette méthode **non modifiable** dans une classe dérivée.
 - **static**:
 - Pour une *méthode*: **static** indique qu'elle **peut être appelée sans instancier** sa classe à travers `Classe.method()`.
 - Devant un *attribut*: il s'agit d'un **attribut de classe**. Sa valeur est partagée entre les différentes instances
- **static final** produit une constante!

Mini-Projet

Lecteur de fichiers

- Fichiers = système de communication/spécification pour les projets collaboratifs.
- Simplification de l'architecture des IHM multi-composants, réduction des dépendances.
- 3 manières de gérer les fichiers. En Java, une logique de flux:
 - Approche générale, “à l'ancienne”
 - Lecture/Ecriture ASCII
 - Approche Objet
 - Serialization
 - Externalization
- Dans ce projet on privilégie la première approche

Mini-Projet

Lecteur de fichiers

- **Fichiers:** lire les noms, vérifier l'existence, vérifier la possibilité d'écriture
→ Équivalent des fonctions `dir`, `cd` ... mais à l'intérieur de Java
- Une fois le fichier ciblé, l'**ouvrir** et **lire** ce qu'il y a dedans.
- Créer un fichier et/ou écrire dedans.

La classe `File` permet de gérer et manipuler les fichiers:

- test d'existence
- distinction fichier / répertoire
- copie / effacement

Mini-Projet

Lecteur de fichiers

- File: désigner un fichier
- FileInputStream: création de cet objet = ouverture en lecture du fichier:

- Des exceptions à gérer
- Fermer les fichiers ouverts

```
1 FileInputStream in = null ;
2 File f = new File("xanadu.txt");
3 try {
4     in = new FileInputStream(f); // ouverture du fichier
5     // Throws : FileNotFoundException: => try/catch
6     // OPERATIONS DE LECTURE
7 }
8 finally {
9     if (in != null) {
10         in.close ();
11     }
12 }
```

- Faire un programme tel que:
 - classe LecteurFichier: lis le nom d'un fichier (donné au constructeur), l'affiche sur la console
 - classe LecteurReverse: Ne modifie que la fonction affiche pour afficher à l'envers

Mini-Projet

Lecteur de fichiers

Faire un programme:

1. Défini une interface de lecteur de fichiers
2. Plusieurs sous-classes pour différents types de fichiers
3. Utilise une classe abstraite principale pour définir les méthodes qui ne changeront pas d'un fichier à l'autre
4. Implémente une des classes qui affiche le fichier à l'envers sur l'écran en terme de lignes
5. Implémente une des classes qui affiche le fichier de manière palindromique (en terme de caractères).
6. Comparsateur de fichiers "diff"

45 minutes - Noté