ircam Centre Pompidou

UPMC SORBONNE UNIVERSITÉS

JAVA
- Héritage -

Ninon Devis: ninon.devis@ircam.fr

Philippe Esling: esling@ircam.fr

License 3 Professionnelle - Multimédia

Plan du cours

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

Rappels

Notions de base

- Qu’est ce qu’un membre de classe ?
  - *Attributs et méthodes*
- Qu’est ce qu’un membre précédé de “public” ?
  - *Il est utilisable par d’autres objets, visible partout.*
- Qu’est ce qu’un membre précédé de “private” ?
  - *Il est caché et n’est visible que depuis les instances de la classe à laquel il appartient. On ne peut pas l’utiliser ou le voir en dehors de la classe qui le définit.*

Rappels

Déclarer une classe

- Par convention, que déclare t-on et dans quel ordre ?
  - *Constantes, énumérations, attributs puis méthodes.*
- Comment est identifié un attribut ?
  - [portée] [type de retour] [identifiant]
- Comment est identifiée une méthode ?
  - [portée] [type de retour] [identifiant] ([liste des paramètres]) {...}

```java
public class Voiture {

  public String marque;
  public float vitesse;

}
```

```java
public class Voiture {

  private float vitesse;

  public float getVitesse() {
    return vitesse;
  }

  public void accelerer(float deltaVitesse) {
    vitesse = vitesse + deltaVitesse;
  }
}
```

Rappels

Les constructeurs

- Qu'est-ce qu'un constructeur et quelle est sa signature ?
  - Il initialise un objet nouvellement créé, c'est une méthode sans type de retour.
  - [portée] [nom de la classe] ([liste des paramètres]) {...}

```java
public class Voiture {

  private String marque;
  private float vitesse;

  public Voiture(String marque) {
    this.marque = marque;
  }

  public Voiture(String marque, float vitesseInitiale) {
    this.marque = marque;
    this.vitesse = vitesseInitiale;
  }

}
```

Rappels

Les constructeurs

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

- méthode du même nom que la classe
- sans type de retour (même pas *void*)
- Le constructeur est appelé chaque création d'un nouvel objet (appelé l'allocation) avec l'utilisation de new.

Si vous avez besoin de détails sur toutes les notions vues en cours un site clair avec des exemples: https://gayerie.dev/docs/java/index.html

Packages

Regroupement de classes dans un archivage

- Les packages sont représentés sur le disque par des répertoires qui permettent de classer des fichiers.
- Pourquoi regrouper des classes sous forme de package ?
  - Éviter les collisions de noms de classes.
- Pour qu'une classe appartienne à un package il suffit que le fichier source commence par: `1 package [nom du package];`
- Exemple: Où se trouve cette classe sur le disque ?

```
1 package monapplication;
2
3 public class TextEditor {
4
5 }
```

➔ Dans le fichier TextEditor.java,
Ce fichier se trouve dans un répertoire nommé “monapplication”

Packages

Regroupement de classes dans un archivage

- Si rien n'est précisé, le programme est considéré comme faisant partie du "paquetage anonyme". => Mauvaise idée
- Par défaut, une classe a également accès au package java.lang qui fournit les classes fondamentales (Object, Math, String…)
- Comme pour les répertoires il peut y avoir des sous-packages:

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

- Comment appeler une classe d'un certain package dans une autre classe ?

Exemple:

```
1 package mesPackages.sousPackage1;
2
3 public class Point { ...
```

- Que mettre au début d'une autre classe pour pouvoir utiliser la classe Point ?

```
import mesPackages.sousPackages1.Point;
import mesPackages.sousPackages1.*;
```

Autorise les références abrégées

Packages

La portée de niveau package

- En plus de la portée “public” et “private”, il existe une portée “package”:
  - Une classe, une méthode ou un attribut avec cette portée n’est accessible *qu’aux membres du même package.*

```java
package monpackage;

class Algorithm {

  public Algorithm() {
    // ...
  }
}
```

➔ Comment désigner la portée package ?

```java
package monpackage;

public class Library {

  private Library() {
  }

  public static byte[] library(byte[] msg) {
    Algorithm algo = new Algorithm();
    return algo.library(msg);
  }
}
```

Rappels & Quiz

```java
package pobj.cours3;

public class Point {
  // attributs
  private double x, y;
  // constructeurs
  public Point(double a, double b){x=a;y=b;}
//  public Point(){x=0;y=0;}
  // accesseurs
  public double getX(){return x;}
  public double getY(){return y;}
  // me'thodes
  private void moveto (double a, double b){x=a;y=b;}
  public void rmoveto (double dx, double dy){x+=dx;y+=dy;}
  public double distance(){
    double x = this.getX();
    double y = this.getY();
    return Math.sqrt(x*x+y*y);
  }
  // me'thodes pre'de'finies (standards)
  public String toString(){return ("("+x+","+y+")");}
}
```

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?

Objets & Classes

Retour sur “static”

- Le mot clef static permet d’économiser l’espace mémoire: il garantit qu’une seule instance de la variable ou méthode est créée en mémoire.
- S’applique à des variables, des méthodes ou des blocs.

```java
public class Foo
{
    // méthode static
    static void mymethod()
    {
        System.out.println("Méthode statique
    }

    public static void main(String[] args)
    {
        mymethode();
    }
}
```

Pour une méthode permet d’appeler:

```
maclasse.mamethode();
```

au lieu de:

```
maclasse m = new maclasse();
m.method();
```

```java
public class Foo
{
    // Variable statique
    static int x = 2;
    static int y;

    // Block statique
    static {
        System.out.println("Bloc statique initialisé.");
        y = x * 3;
    }

    public static void main(String[] args)
    {
       System.out.println("La valeur de x est : "+x);
       System.out.println("La valeur de y est : "+y);
    }
}
```

Objets & Classes

- Quelle est la différence entre variable d'instance et variable de classe ?
  - Variable d'instance: spécifique à une objet
  - Variable de classe: spécifique à une classe et est partagée par l'*ensemble des objets* d'une classe.

```java
package coursL3;

public class Personne {
    private String nom;
    private int age;

    static int nombrePersonnes;

    public Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
        nombrePersonnes ++;
    }

    public String getNom() {
        return nom;
    }

    public int getAge() {
        return age;
    }
}
```

```java
package coursL3;

public class Test {

    public static void main(String[] args) {

        Personne p1 = new Personne("Rick", 69);
        Personne p2 = new Personne("Morty", 12);
        System.out.println(Personne.nombrePersonnes);
        System.out.println(p1.getAge());
```

- Comment faire si nombrePersonnes est "private" ?

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 en un unique exemplaire
- sont accessibles par la notation “point” également.

```
public class Foo
{
    // method static
    static void mymethod()
    {
        System.out.println("Méthode statique de la classe Foo");
    }

    public static void main(String[] args)
    {
        mymethod();
    }
}
```

mymethod est déclarée en static

On peut donc y accéder avant la création des objets de sa classe et sans référence à aucun objet

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

Emploi de this

```java
package coursL3;

public class Personne {
    private String nom;
    private int age;

    static int nombrePersonnes;

    public Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
        nombrePersonnes ++;
    }
```

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

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

- peut être utilisé en notation qualifiée

```java
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.
- 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.

UML

Diagramme de classes

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.

Types de composants
une *classe*
une classe dans un état donné
un objet
un noeud ou une ressource logicielle
un rôle
une *interface*
un acteur
un cas d’utilisation
un sous-système

Relations entre composants
généralisation
association
dépendance
réalisation

UML

Formalisme du composant “Classe”

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 méthodes.

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

UML

Formalisme de la relation “dépendance”

- 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`.

UML

Formalisme de la relation “association”

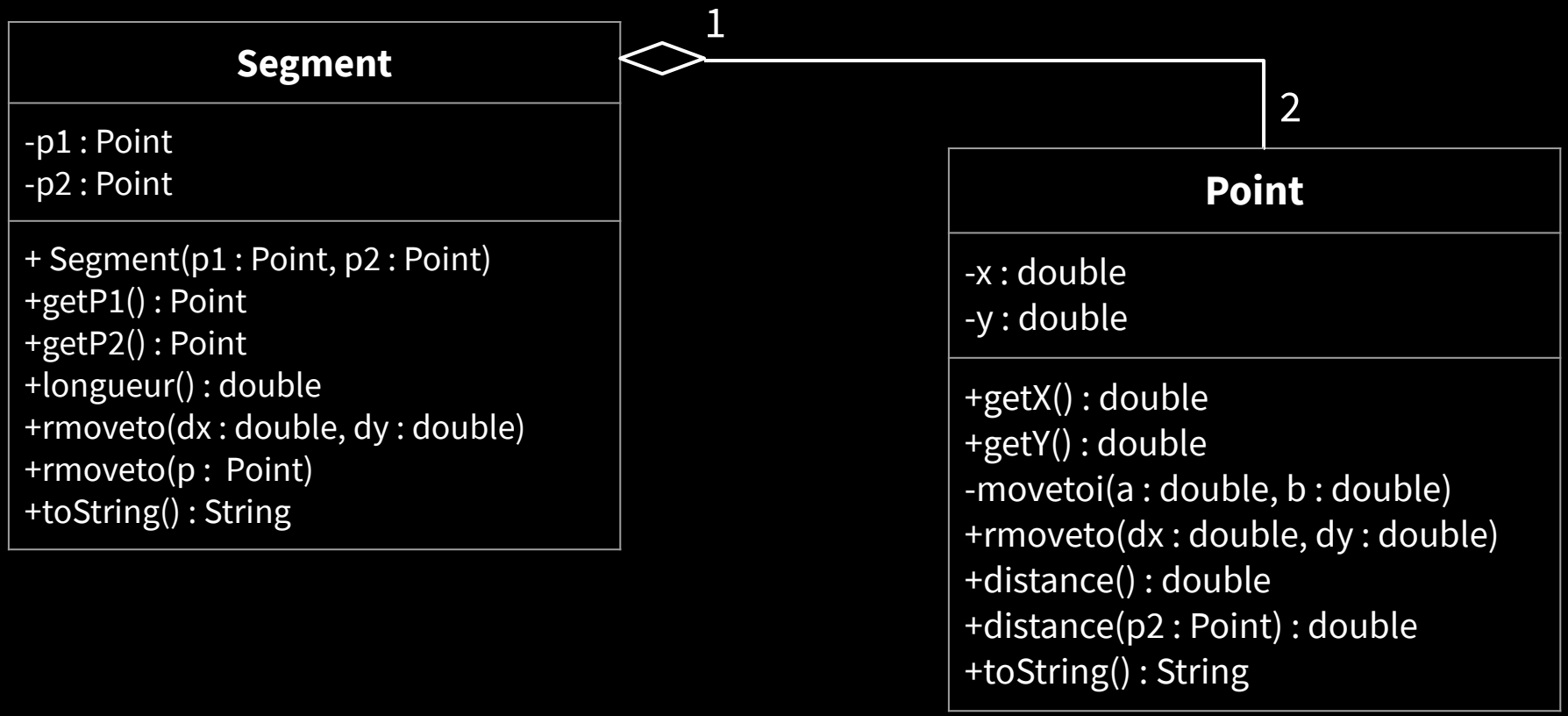
- Ligne pleine dont les extrémités sont optionnellement annotées par:
  - Une multiplicité:
    - * signifie un nombre indéterminé
    - $n \in N$, un nombre n fixé
    - m...n, $(m, n) \in N^2$, un nombre entre m et n
  - Un losange plein: signifie que l’association est une composition
  - Un losange 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 disparait 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

UML

Exemple de l’agrégation



UML

Formalisme de la relation “généralisation”

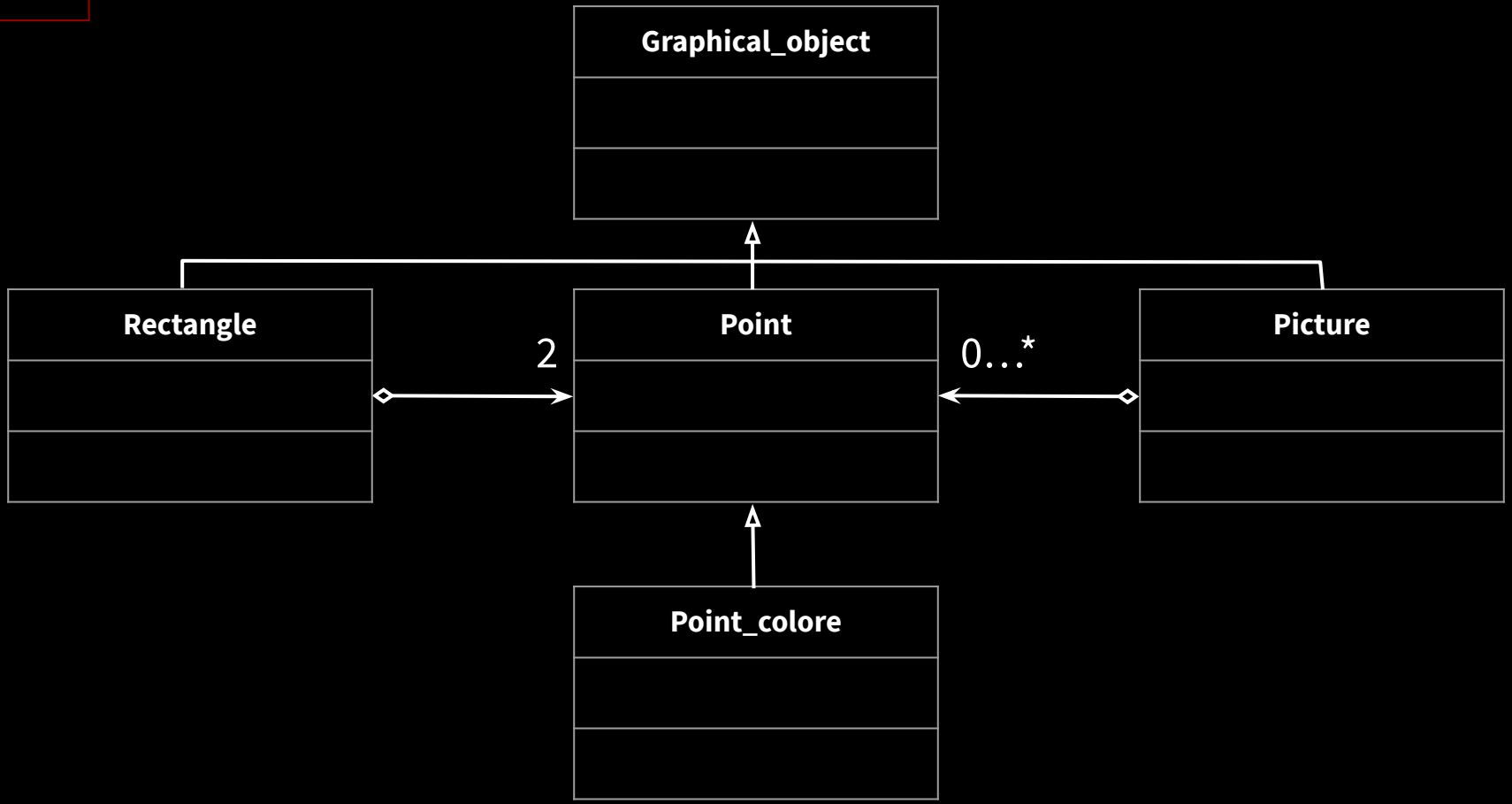
- Ligne pleine se terminant par une flèche.

- 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.

UML

Exemple de généralisation



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.

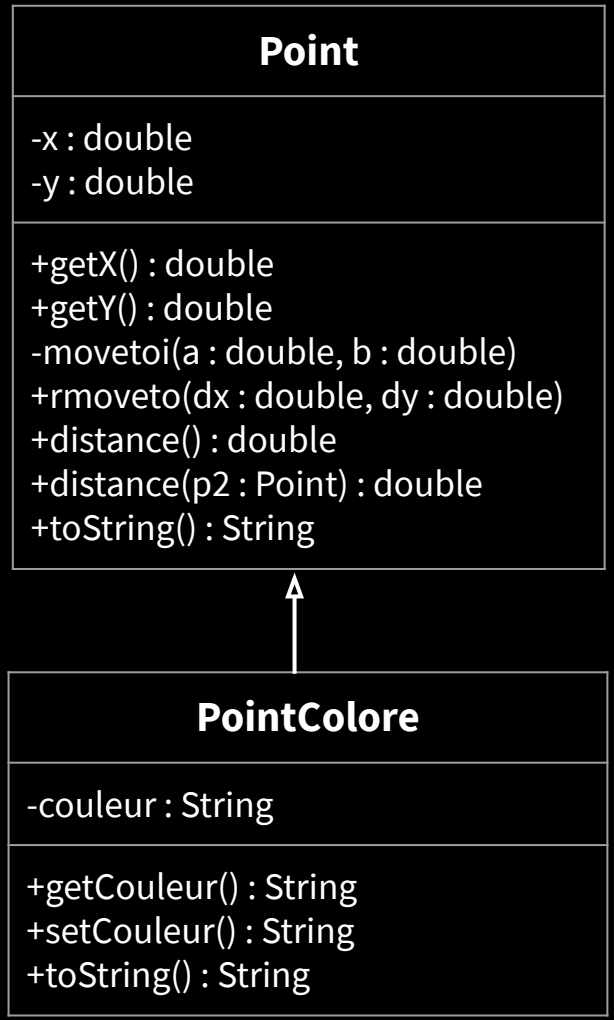
  ➔ les variables et les méthodes décrites par la classe originale sont toujours utilisées
- Permet de reprendre les membres d'une classe (appelée *super-classe* ou *classe mère*) dans une autre classe (nommée *sous-classe*, *classe fille* ou *classe dérivée*) qui en hérite.
- En JAVA, ce mécanisme est 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:




```java
package pobj.cours3;

public class Point {
    // Attributs
    private double x, y;
    // Constructeur
    public Point(double a, double b){x=a; y=b}
    // Accesseurs
    public double getX(){return x;}
    public double getY(){return y;}
    // Méthodes
    private void moveto (double a, double b){x=a; y=b}
    public void rmoveto (double dx, double dy){x+=dx; y+=dy;}
    public double distance(){
        double x = this.getX();
        double y = this.getY();
        return Math.sqrt(x*x + y*y);
    }
    // Méthodes prédéfinies (standards)
    public String toString(){return ("(" + x +", " + y +")");}
}
```

```java
package pobj.cours3;

public class PointColore extends Point {
    private String couleur;
    public PointColore (double x, double y, String c){
        super(x, y);
        this.couleur = c;
    }
    public PointColore(){couleur = "Indéfinie"}
    public String getCouleur(){return couleur;}
    public String setCouleur(){couleur = c;}
    public String toString(){
        return super.toString() + "-" + this.getCouleur();
    }
```

Héritage

Super et this

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

```java
package pobj.cours3;

public class PointColore extends Point {
    private String couleur;
    public PointColore (double x, double y, String c){
        super(x, y);
        this.couleur = c;
    }
    public PointColore(){couleur = "Indéfinie"}
    public String getCouleur(){return couleur;}
    public String setCouleur(){couleur = c;}
    public String toString(){
        return super.toString() + "-" + this.getCouleur();
    }
```

➔ *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 `PointColore.`
- 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 PointColore(){couleur = "Indéfinie";}
```

Sera automatiquement remplacé par:

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

Héritage

Exécution de l'exemple dans le main:

```java
package pobj.cours3;

class ExPoint {
    public static void main(String[] args){
        Point p0 = new Point();
        Point p1 = new Point(2, 3);
        PointColore pc0 = new PointColore();
        PointColore pc1 = new PointColore(2, 3, "Rouge");

        System.out.println(p0 + " " + p1);
        System.out.println(pc0 + " " + pc1);
    }
}
```

En sortie:

```
> java pobj/cours3/Expoints
(0.0, 0.0) (2.0, 3.0)
(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.

Héritage

Exercice d'application

- Reprendre le projet Point
- Ajouter une classe FormeGeometrique
  - Contient un centre (ou un point d'accroche)
- Créer un ensemble de sous-classes:
  - Rectangle
  - Carré
  - Triangle
- Définir un ensemble de fonctions: déplacer le centre et calculer l'aire.

➔ Quelles sont les méthodes héritées ou non ? Réfléchissez à comment vous organiser sur l'héritage !

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      Objet 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 PointColore extends Point {
9      ...
10     public Object copier() {
11         return new PointColore(this.x, this.y, this.couleur);
12     }
13 }
```

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

➔ 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 PointColore(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.qui_suis_je ());
4      }
5      public abstract String qui_suis_je () ;
6  }
7
8  class Carre extends Forme {
9      public String qui_suis_je () { return ("un carre'") ; }
10 }
11
12 class Cercle extends Forme {
13 public String qui_suis_je () { 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 qui_suis_je()

Implémentation de la méthode qui_suis_je() 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 nécessitent de créer des méthodes get() pour y accéder.

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.
- 2 manières de gérer les fichiers. En Java, une logique de flux:
  - Approche générale, “à l’ancienne”
    - Lecture/Ecriture ASCII
  - Approche Objet
    - Serialization

➔ Dans ce projet on privilégie la première approche

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 }
```

Mini-Projet

Lecteur de fichiers

Faire un programme:

1. Défini une interface de lecteur de fichiers
2. Plusieurs sous-classes abstraites pour différents types de fichiers (on s'intéresse surtout aux fichiers texte). Elles définissent les méthodes qui ne changeront pas.
3. Implémente une classe qui affiche le fichier à l’endroit.
4. Implémente une classe 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. Comparateur de fichiers “diff”.