École Doctorale en Sciences Economique, Gestion et Informatique
Faculté des Sciences Économiques et de Gestion de Sfax
Département Informatique

Auditoire

1er mastère professionnel Audit et Sécurité Informatique

Architectures des Systèmes d'Information (JavaEE)

Enseignante

AÏDA KHEMAKHEM

Année Universitaire

2019 – 2020

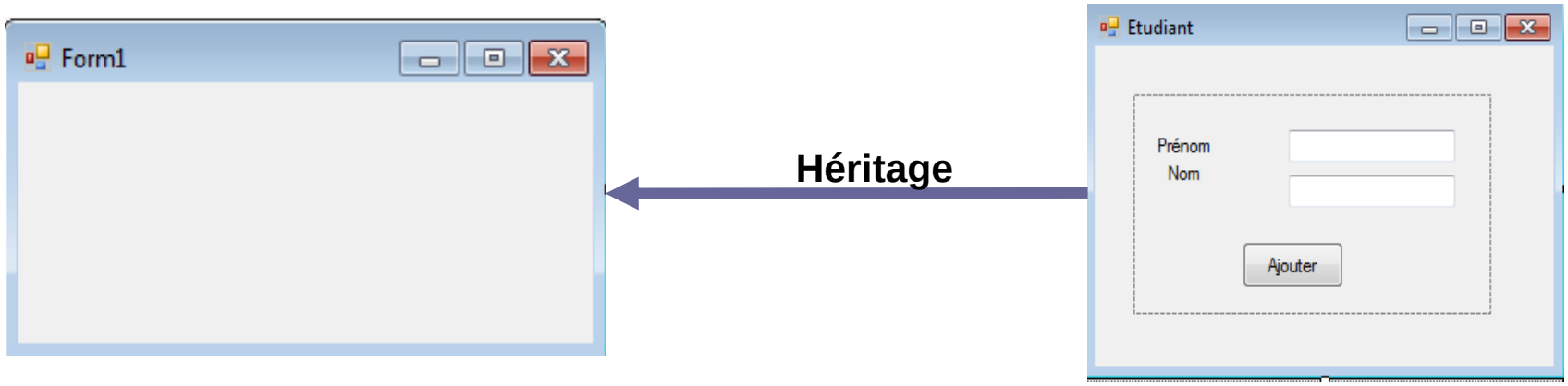
Rappel :

Concepts de POO

1. Programmation Orientée Objet : Classe et Objet
2. Interface
3. Collection

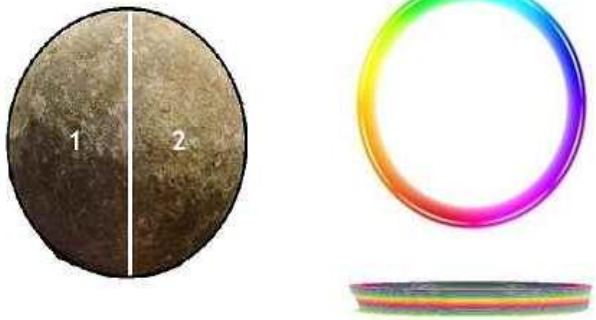
1. La Programmation Orientée Objet : POO

- Le concept de la POO est construit autour des notions d'objet et de classe
  - Une classe est l'implémentation d'un type de données
  - Un objet est une instance d'une classe
- Les avantages de la POO :
  - **Encapsulation** : c'est de réunir les données avec leurs traitements
  - **Héritage** : c'est la réutilisation des classes existantes



Domaine: Gestion d'un Cercle

CERCLE CERCLE



Périmètre = 2 PIERRES $\mathcal{P} = 2\pi r$

Aire = PIERRE^2 $\mathcal{A} = \pi r^2$

Programmation Orientée Objet

```
public class Position
{       private double x;
        private double y;
public Position(double x, double y)
   {        this.x = x;
            this.y = y;
}
  public String ToString()
   { return "X =" +this.x + "\nY =" +this.y;
    }       } //fin classe Position
```

```
public class Cercle
{   private double rayon;
    private Position centre;

public Cercle(double rayon, Position centre)
   {   this.rayon = rayon;
       this.centre = centre;
       }
  public String ToString()
   {  return "Cercle : Rayon =" + this.rayon +
"Centre =" + this.centre; }
 public double Perimetre()
{return 2.0 * Math.PI * rayon *this.rayon; }
  public double Aire()
    {  return Math.PI * rayon * rayon; }
   } //fin classe Cercle
```

Activité 1

Pour tester les nouveaux types (**class**), ajouter dans la **méthode** principale main les instructions nécessaires pour :

- Déclarer et instancier deux **objets** de type Position :

  p1 (10,20) et **p2** (15,25)

- Déclarer et instancier deux **objets** de type Cercle :

  c1 (**p1**, 9) et **c2** (**p2**,19)

- Afficher les caractéristiques du cercle c1
- Afficher le périmètre et l'aire du cercle c1

```
Position   p1, p2 ;
p1= new Position(10,20) ;
```

```
Cercle  c1, c2 ;
c1= new Cercle(9, p1) ;
```

```
Sysytem.out.println(" Cercle : Rayon = "+ c1.ToString() );
```

```
Sysytem.out.println("Le périmètre du Cercle = "+ c1.Perimetre ( ) ) ;
```

```
Sysytem.out.println("L'aire du Cercle = "+ c1.Aire ( ) ) ;
```

Manipulation des Objets

```
public class Test

{
public static void main(String[]
args) {



Position   p1, p2 ;
p1= new Position(10,20) ;

Cercle   c1, c2 ;
c1= new Cercle(9, p1) ;

System.out.println(" Cercle :
Rayon = "+ c1.ToString() );

System.out.println("Le périmètre
du Cercle = "+ c1.Perimetre ( ) )
;

System.out.println("L'aire du
Cercle = "+ c1.Aire ( ) ) ;
}}
```

Création des classes

```
public class Position
{       private double x;
        private double y;
public Position(double x, double y)
   {        this.x = x;
            this.y = y;
}
  public String ToString()
   { return "X =" +this.x + "\nY =" +this.y;
    }       } //fin classe Position
```

```
public class Cercle
{   private double rayon;
    private Position centre;

public Cercle(double rayon, Position centre)
   {   this.rayon = rayon;
       this.centre = centre;
       }
  public String ToString()
   {  return "Cercle : Rayon =" + this.rayon
+ "Centre =" + this.centre; }
 public double Perimetre()
{return 2.0 * Math.PI * rayon *this.rayon; }
  public double Aire()
    {  return Math.PI * rayon * rayon; }
   } //fin classe Cercle
```

2. Les interfaces

I. Présentation d'une interface

- Une **interface** représente une pure spécification qui définit les **méthodes** disponibles d'une famille d'objets, mais ne définit aucune implémentation (c'est une promesse de comportement).

- Du point de vue syntaxique, la définition d'une interface ressemble beaucoup à la définition d'une classe abstraite.

```
public interface MonInterface {
        public void meth();}
```

- Une déclaration d'interface définit un nouveau **type référence.**

II. Le contenu d'une interface

- Le corps d'une interface ressemble beaucoup à celui d'une classe abstraite avec cependant les différences suivantes :
  - Une interface ne peut définir que des méthodes abstraites. Le modificateur **abstract** n'est pas nécessaire.
  - Toutes les méthodes abstraites sont implicitement **publiques** même si le modificateur public est omis.
  - Une interface ne peut pas contenir de méthodes **statiques**.
  - Une interface ne peut pas définir de **champs d'instance** (attribut).
  - Une interface peut cependant contenir des **constantes** déclarées **static et final** (tous les champs sont implicitement static et final même si les modificateurs correspondants sont omis).
  - Une interface ne définit pas de constructeur (on ne peut pas l'instancier et elle n'intervient pas dans le chaînage des constructeurs).

III. L'implémentation d'une interface

- En Java, une classe ne peut hériter que d'une et d'une seule classe parente (héritage simple). Par contre, elle peut implémenter une ou plusieurs interfaces en utilisant la syntaxe suivante :

  class NomClasse implements interface1[, interface2]

- L'implémentation d'une ou de plusieurs interfaces (implements) peut être combinée avec l'héritage simple (extends). La clause implements doit suivre la clause extends.

- Exemple :

public class Cercle **extends** Forme **implements** Imprimable {...}

IV. L'héritage entre les interfaces

- Les interfaces peuvent avoir des sous-interfaces (tout comme les classes peuvent avoir des sous-classes). Comme pour les classes, le mot-clé extends est utilisé pour créer une sous-interface.

  public interface MonInterface extends TonInterface {...}

- Une sous-interface hérite de toutes les méthodes abstraites et de toutes les constantes de son interface parente et peut définir de nouvelles méthodes abstraites ainsi que de nouvelles constantes.

- Contrairement aux classes, une interface peut posséder plusieurs interfaces parentes (héritage multiple).

```
public interface MonInterface extends TonInterface, SonInterface
  {...}
```

Remarques

- Une classe qui implémente une sous-interface doit **implémenter** les méthodes abstraites définies directement par l'interface ainsi que les méthodes abstraites héritées de toutes les interfaces parentes de la sous-interface.

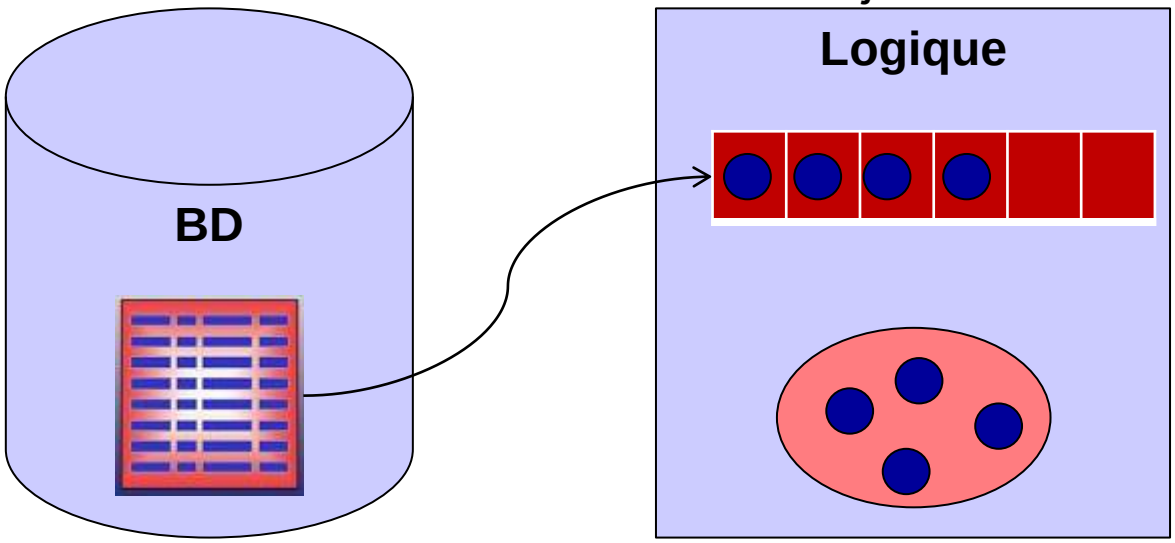
- Des conflits de noms (collision) peuvent se produire lorsqu'une classe implémente plusieurs interfaces comportant des noms de méthodes ou de constantes identiques

- Il faut distinguer plusieurs situations :
  - Plusieurs méthodes portent des signatures identiques :
    Pas de problème, la classe doit implémenter cette méthode
  - Mêmes noms de méthodes mais profils de paramètres différents : Implémentation de deux ou plusieurs méthodes surchargées
  - Mêmes noms de méthodes, profils de paramètres identiques, mais types des valeurs de retour différents :
    Pas possible d'implémenter les deux interfaces (cela provoquera une erreur à la compilation : Interface-Collision)
  - Noms de constantes identiques dans plusieurs interfaces :
    Doivent être accédées en utilisant la notation qualifiée (Interface.Constante)

3. Les collections

I. Pourquoi les collections ?

- **Tableau**
  - accès par index
  - insertions et suppressions peu efficaces
  - défaut majeur : nombre d'éléments borné



Table

Objet

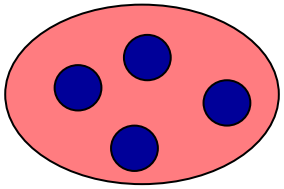
Interface Graphique

Objet

- **Collection (ArrayList)**
  - accès séquentiel : premier, suivant
  - insertions et suppressions efficaces
  - avantage majeur : nombre d'éléments modifiable

II. Présentation de Collection

- Collection est objet qui regroupe de multiples éléments dans une seule entité



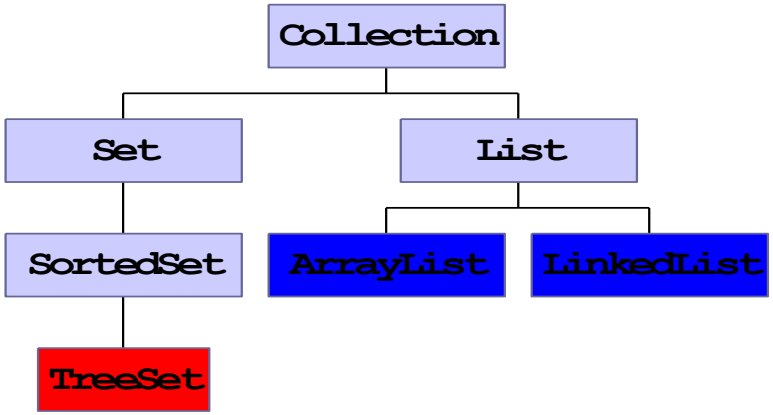
- Utilisation de collections pour
  - stocker, retrouver et manipuler des données
  - transmettre des données d'une méthode à une autre
- Exemples :
  - un dossier de courrier : collection de mails
  - un répertoire téléphonique : collection d'associations (noms, numéros de téléphone)
- Les collections sont de deux types :
  - dépourvus d'ordre (ex. les ensembles)
  - ordonnées (ex. vecteur ou liste chaînées).

III. Les collections dans Java

- Dans les premières versions de Java on trouve quelques implémentations pour différents types de collections comme les tableaux et la classe **Vector**
- Une **modification majeure** de Java 2 a été d'inclure un véritable **«framework»** pour la gestion des collections (package **java.util**)
  - Architecture unifiée pour représenter et manipuler les collections
  - Elle est composée de trois parties :
    - **deux hiérarchies d'interfaces** permettant de représenter resp. les **collections** et les **maps** sous forme de types abstraits
    - des **implémentations** de ces interfaces
    - des **algorithmes** réalisant des opérations fréquentes sur les collections (recherche, tri,…)

VI. Paquetage `java.util` de Java 2

- Interface **Collection**
- Interfaces **Set** et **List, SortedSet**
- Méthodes
  - **boolean add(Object o)**
  - **boolean remove(Object o)**
  - …
- Plusieurs implantations
  - tableau : **ArrayList**
  - liste chaînée : **LinkedList**
  - ensemble trié : **TreeSet**
- Algorithmes génériques : tri, maximum, copie ...
  - méthodes statiques de **Collections**



Collection non-générique	Collection générique

ArrayList

- Taille dynamique
- Les éléments de type Object
- Nécessite la conversion au type original

Exemple :

```
ArrayList ar1=new ArrayList();
ar1.Add( "Lundi" );
ar1.Add( new Date() );
String x=(String) ar1.get( 0 );
Date x=(Date) ar1.get( 1 );
```

ArrayList< ? >

- Taille dynamique
- Les éléments de même type qui doit être fixé avec la déclaration
- Plus performant

Exemple :

```
ArrayList<String> lst1;
lst1 = new ArrayList<String>();
ArrayList<Date> lst2 ;
lst2 = new ArrayList<Date>();
lst2.add( new Date());
lst2.add( "Lundi" ); //Erreur compilation
Date y= lst2.get ( 0 );
lst1.add("Lundi"); String x=lst1.get(0);
```

V. Méthodes communes de Collection

boolean add(Object) : ajouter un élément
boolean addAll(Collection) : ajouter plusieurs éléments
void clear() : tout supprimer
boolean contains(Object) : test d'appartenance
boolean containsAll(Collection) : appartenance collective
boolean isEmpty() : test de l'absence d'éléments
Iterator iterator() : pour le parcours (cf Iterator)
boolean remove(Object) : retrait d'un élément
boolean removeAll(Collection) : retrait de plusieurs éléments
boolean retainAll(Collection) : intersection
int size() : nombre d'éléments
Object[] toArray() : transformation en tableau

Object[] toArray(Object[] a) : tableau de même type que a

TP 1

- Les concepts de POO : classe, objet, constructeur…
- Les collections : Exemple Arraylist
- Les interfaces

Merci pour votre attention