



COURS 5: L'HÉRITAGE EN JAVA

Par Aïcha El Golli

aicha.elgolli@essai.ucar.tn



RAPPEL : CLASSE

- **Classe**: représente une « famille » d'objets partageant les mêmes propriétés et supportant les mêmes opérations. Elle sert à définir les caractéristiques des objets d'un type donné.
 - Décrit l'ensemble des données (attributs, caractéristiques, **variables d'instance**) et des opérations sur données (**méthodes**)
 - Sert de « modèle » pour la création d'objets (**instances** de la classe)

```
public class Point {
```

```
    private static final Point ORIGINE = new Point(0,0);
```

```
    private int x; // abscisse du point
    private int y; // ordonnée du point
```

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

```
    public void translate(int dx, int dy){
        x = x + dx;
        y = y + dy;
    }
```

```
    // calcule la distance du point à l'origine
    public double distance() {
        return Math.sqrt(x * x + y * y);
    }
}
```

Variable
de classe

Variables
d'instance

Constructeur

Méthodes

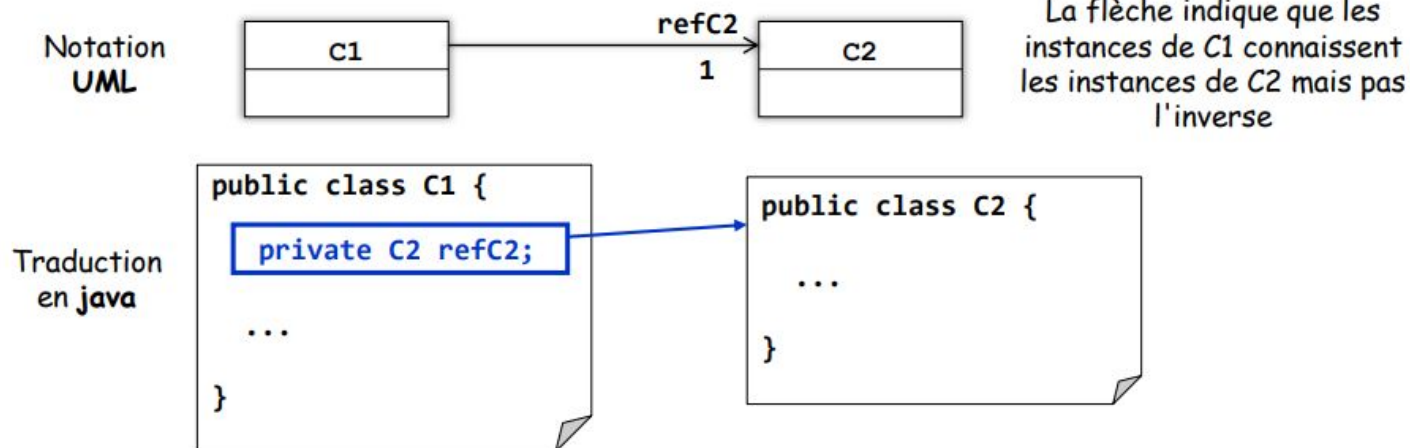
Membres

RÉUTILISATION

- Comment utiliser une classe comme brique de base pour concevoir d'autres classes ?
- Dans une conception objet on définit des associations (relations) entre classes pour exprimer la réutilisation.
- UML (Unified Modelling Language <http://uml.free.fr>) définit toute une typologie des associations possibles entre classes. Dans cette introduction nous nous focaliserons sur deux formes d'association
 - *Un objet peut faire appel à un autre objet : **délégation***
 - *Un objet peut être créé à partir du « moule » d'un autre objet : **héritage***

DÉLÉGATION

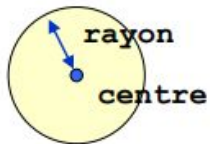
- Un objet **o1** instance de la classe **C1** utilise les services d'un objet **o2** instance de la classe **C2** (**o1** délègue une partie de son activité à **o2**)
- La classe **C1** utilise les services de la classe **C2**
 - *C1 est la classe cliente*
 - *C2 est la classe serveuse*



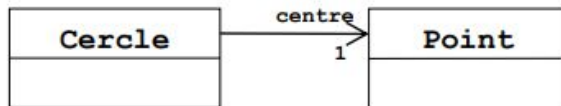
- La classe cliente (**C1**) possède une référence de type de la classe serveuse (**C2**)

DÉLÉGATION: EXEMPLE

- Exemple la classe **Cercle**



- rayon : double
- centre : deux doubles (x et y) ou bien **Point**



- L'association entre les classes **Cercle** et **PointCartesien** exprime le fait qu'un cercle **possède (a un)** un centre

```

public class Cercle {

    /**
     * centre du cercle
     */
    private Point centre;

    /**
     * rayon du cercle
     */
    private double r;

    ...

    public void translater(double dx, double dy) {
        centre.translater(dx,dy);
    }    le cercle délègue à son centre l'opération de translation
    ...
}
  
```

AGRÉGATION

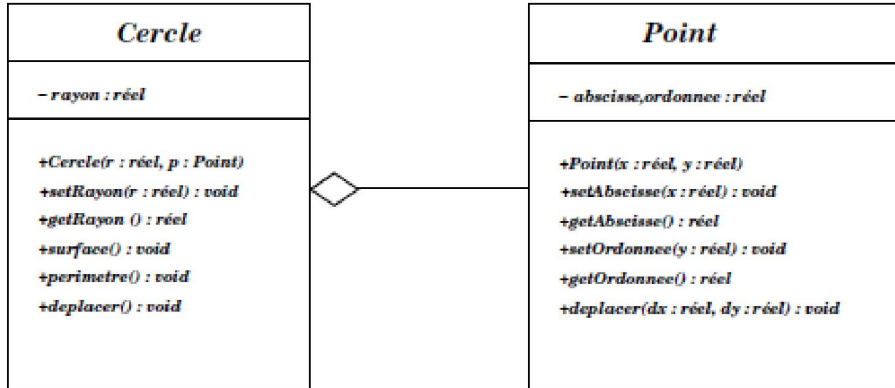
Relation particulière entre un objet et plusieurs objets le composant

exemple : un ordinateur est composé d'un processeur, un ordinateur est composé d'un disque dur, un ordinateur est composé d'un moniteur.

Soit une classe A :

une instance de la classe A est utilisée par une classe B à travers ses méthodes. L'instance B est autonome et indépendante de l'instance A.

AGRÉGATION



Le point représentant le centre a une existence autonome (cycles de vie indépendants) Il peut être partagé (à un même moment il peut être lié à plusieurs instances d'objets (éventuellement d'autres classes))

```
public class Cercle {
    /** centre du cercle */
    private Point centre;

    /** rayon du cercle*/
    private double r;

    public Cercle( Point centre, double r) {
        this.centre = centre;
        this.r = r;
    }
    ...

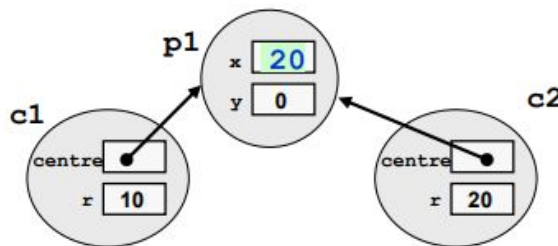
    public void translator(double dx, double dy) {
        centre.translater(dx,dy);
    }
}
```

c2.translater(10,0);

➡ Affecte aussi cercle c1
Après ces opérations le centre des 2 cercles est (20,0)

Affecte aussi le cercle c1
Après ces opérations le centre des 2 cercles est (20,0)

```
public static void main(String args[]) {
    Point p1 = new Point(10,0); /*il peut être utilisé en dehors du cercle dont il est le centre */
    Cercle c1 = new Cercle(p1,10)
    Cercle c2 = new Cercle(p1,20);
    ...
    c2.translater(10,0);
}
```



ASSOCIATION/COMPOSITION

Association: Relation établissant un lien fonctionnel entre deux classes, sans notion de composition :il faut choisir la classe de référence, l'implémentation est identique à l'agrégation, exemple : une Personne est associée à plusieurs Voiture, une Voiture est associée à une Personne

Composition

Une composition est une agrégation "contrainte" :

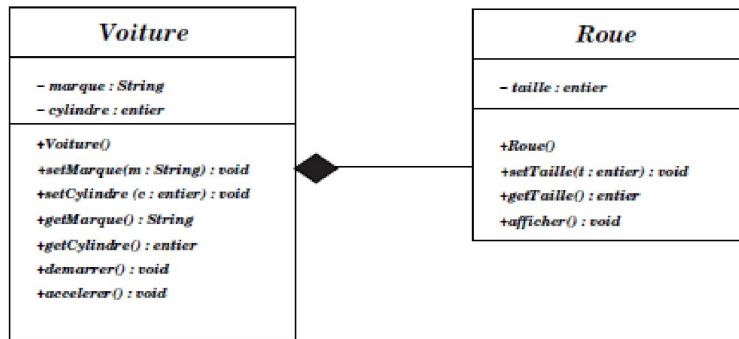
la contrainte est relative à la durée de vie des objets (les composants ont une existence qui dépend du composé),

exemple : une fenêtre est composée de boutons, de menus, ... qui sont détruits lorsque la fenêtre est détruite.

l'instance de la classe B est créée dans le constructeur de la classe A.

COMPOSITION

Classe Voiture :



```

public class Voiture {
    private String marque;
    private int cylindree;
    private Roue tab[ ];

```

```

    public Voiture () {
        tab=new Roue[4];
        tab[0] = new Roue() ; tab[1] = new Roue() ;
        tab[2] = new Roue() ;tab[3] = new Roue() ;
    }
    ...
}

```

COMPOSITION

```
public class Cercle {

    /** centre du cercle */
    private Point centre;

    /** rayon du cercle */
    private double r;

    public Cercle(Point centre, double r) {
        this.centre = new Point(centre);
        this.r = r;
    }
    ...
}
```

```
c2.translater(10,0);
```

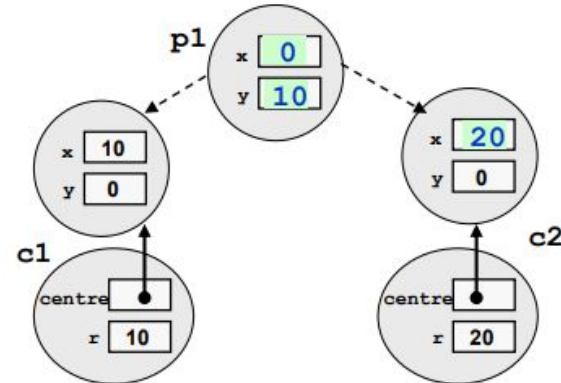
⇒ N'affecte que le cercle c2

```
p1.rotation(90);
```

⇒ N'affecte pas les cercles c1 et c2

- Le **Point** représentant le centre n'est pas partagé (à un même moment, une instance de **Point** ne peut être liée qu'à un seul **Cercle**)

```
Point p1 = new Point(10,0);
Cercle c1 = new Cercle(p1,10);
Cercle c2 = new Cercle(p1,20);
...
```



- les cycles de vies du **Point** et du **Cercle** sont liés : si le cercle est détruit (ou copié), le centre l'est aussi.

HÉRITAGE

- ✓ Permet de spécialiser ou d'étendre une classe
- ✓ exemple : un étudiant est une personne, un cercle est un objet graphique, un bouton est un objet graphique.
- ✓ En Java, la notion d'héritage multiple n'existe pas : une "classe fille" ne peut hériter que d'une seule "classe mère".
- ✓ La mise en œuvre d'une "classe fille" héritant d'une "classe mère" se fait par le mot-clé *extends*. il y a alors :
 - héritage des méthodes,
 - héritage des attributs.

VISIBILITÉ

Encapsulation des données

création des classes dont tous les attributs sont privés : les données d'une instance sont donc inaccessibles aux instances des autres classes.

utilisation d'attributs protégés : membre accessible uniquement par la classe et ses sous-classes : **protected**

les constructeurs de la classe fille auront accès à tous les attributs de la classe mère et pourront les initialiser.

Appeler des méthodes de la super-classe : **super** se réfère aux champs et aux méthodes définis dans sa super-classe :

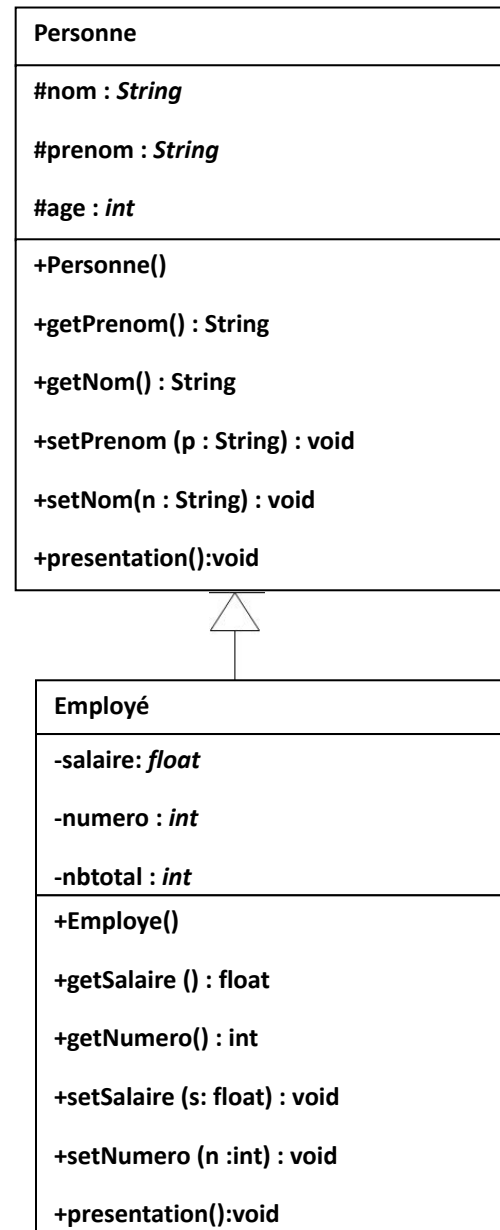
? recherche de methode() dans la super-classe : **super.methode();**

TERMINOLOGIE

Héritage permet de reprendre les caractéristiques d'une classe *M* existante pour les étendre et définir ainsi une nouvelle classe *F* qui hérite de *M*.

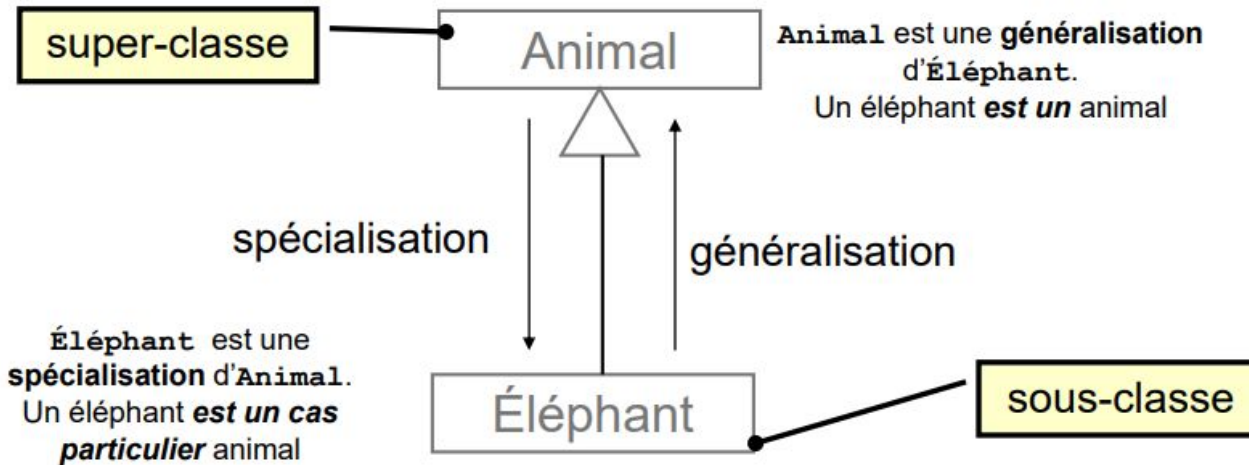
Les objets de *F* possèdent toutes les caractéristiques de *M* avec en plus celles définies dans *F*

- _ *Personne* est la classe mère et *Employé* la classe fille.
- _ la classe *Employé* hérite de la classe *Personne*
- _ la classe *Employe* est une sous-classe de la classe *Personne*
- _ la classe *Personne* est la super-classe de la classe *Employe*
- _ la relation d'héritage peut être vue comme une relation de “généralisation/spécialisation” entre une classe (la *super-classe*) et plusieurs classes plus spécialisées (ses *sous-classes*).



GÉNÉRALISATION/SPÉCIALISATION

- La généralisation exprime une relation “**est-un**” entre une classe et sa super-classe (chaque instance de la classe est aussi décrite de façon plus générale par la super-classe).



- La spécialisation exprime une relation de “**particularisation**” entre une classe et sa sous-classe (chaque instance de la sous-classe est décrite de manière plus spécifique)

GÉNÉRALISATION/SPÉCIALISATION

Héritage à travers tous les niveaux

- Pour résoudre un message, la hiérarchie des classes est parcourue de manière ascendante jusqu'à trouver la méthode correspondante.

```
public class A {
```

```
    public void hello() {
        System.out.println(«Hello»);
    }
}
```

```
public class B extends A {
```

```
    public void bye() {
        System.out.println(«Bye Bye»);
    }
}
```

```
public class C extends B {
```

```
    public void ouns() {
        System.out.println(«ouns!»);
    }
}
```

```
C c = new C();
```

```
c.hello();
```

```
c.bye();
```

```
c.ouns();
```

REDÉFINITION DES MÉTHODES

- ✓ une sous-classe peut **ajouter** des variables et/ou des méthodes à celles qu'elle hérite de sa super-classe.
- ✓ une sous-classe peut **redéfinir** (override) les méthodes dont elle hérite et fournir ainsi des implémentations spécialisées pour celles-ci.
- ✓ **Redéfinition d'une méthode (method overriding)** : *lorsque la classe définit une méthode dont le nom, le type de retour et les arguments sont identiques à ceux d'une méthode dont elle hérite*
- ✓ Lorsqu'une méthode redéfinie par une classe est invoquée pour un objet de cette classe, c'est la nouvelle définition et non pas celle de la super-classe qui est invoquée.

REDÉFINITION DES MÉTHODES

```
public class A {
```

```
    public void affiche() {  
        System.out.println("Je suis un A");  
    }
```

```
    public void hello() {  
        System.out.println("Hello");  
    }
```

```
}
```

```
public class B extends A {
```

```
    public void affiche() {  
        System.out.println("Je suis un B");  
    }
```

```
}
```

```
A a = new A();
```

```
B b = new B();
```

```
a.affiche(); → Je suis un A
```

```
a.hello(); → Hello
```

```
b.hello(); → Hello
```

```
b.affiche(); → Je suis un B
```

la méthode **affiche()** est redéfinie

c'est la méthode la plus spécifique qui est exécutée

REDÉFINITION DES MÉTHODES

- Ne pas confondre **redéfinition** (*overriding*) avec **surcharge** (*overloading*)

```
public class A {
    public void methodX(int i) {
        ...
    }
}
```

Surcharge

```
public class B extends A {
    public void methodX(Color i) {
        ...
    }
}
```

B possède deux
méthodes **methodX**
(**methodX(int)** et **methodX(Color)**)

Redéfinition

```
public class C extends A {
    public void methodX(int i) {
        ...
    }
}
```

C possède une seule
méthode **methodX**
(**methodX(int)**)

REDÉFINITION DES MÉTHODES

Redéfinition des méthodes (method **overriding**) : *possibilité de réutiliser le code de la méthode héritée (**super**)* .

```
public class Personne {  
    protected String nom;  
    protected String prenom;  
    ...  
    public void presentaion()  
    {  
        System.out.println("La personne s'appelle : " + nom + prénom);  
        System.out.println("il est de sexe "+ sexe+ " il a " + age+" ans");  
    ...  
    }  
    ...  
}
```

```
public class Employe extends Personne{  
    float salaire;  
    ...  
    public void presentation()  
    {  
        super.presentation ();  
        System.out.println("Son salaire est de : "+  
salaire);  
    ...  
    }  
}
```


PARTICULARITÉS DE L'HÉRITAGE EN JAVA

Héritage simple

une classe ne peut hériter que d'une seule autre classe

dans certains autres langages (ex C++) possibilité d'héritage multiple

La hiérarchie d'héritage est un arbre dont la racine est la classe Object (java.lang)

toute classe autre que **Object** possède une super-classe

toute classe hérite directement ou indirectement de la classe **Object**

par défaut une classe qui ne définit pas de clause **extends** hérite de la classe **Object**

```
public class Point extends Object {  
  
    int x; // abscisse du point  
    int y; // ordonnée du point  
  
    ...  
}
```

RÉUTILISATION DES CONSTRUCTEURS

Redéfinition des méthodes (method overriding) :

*possibilité de réutiliser le code de la méthode héritée (**super**)*

De la même manière il est important de pouvoir réutiliser le code des constructeurs de la super classe dans la définition des constructeurs d'une nouvelle classe

invocation d'un constructeur de la super classe :

super(paramètres du constructeur)

*utilisation de **super(...)** analogue à celle de **this(...)***

Exemple :

```
public Employe(String nom, String prenom, int a, float salaire, int numero) {  
    super(nom,prenom,a);  
    this.salaire=salaire;  
    this.numero=numero;  
}
```

Appel du constructeur de la super-classe.
Cet appel si il est présent doit **toujours** être
la première instruction du corps du constructeur.

CHAÎNAGE DES CONSTRUCTEURS

appel à un constructeur de la super classe doit **toujours** être la première instruction dans le corps du constructeur

*si la première instruction d'un constructeur n'est pas un appel explicite à l'un des constructeur de la superclasse, alors JAVA insère implicitement l'appel **super()***

*chaque fois qu'un objet est créé les constructeurs sont invoqués en remontant en séquence de classe en classe dans la hiérarchie jusqu'à la classe **Object***

*c'est le corps du constructeur de la classe **Object** qui est toujours exécuté en premier, suivi du corps des constructeurs des différentes classes en redescendant dans la hiérarchie.*

garantit qu'un constructeur d'une classe est toujours appelé lorsqu'une instance de l'une de ses sous classes est créée

un objet c instance de C sous classe de B elle même sous classe de A est un objet de classe C mais est aussi un objet de classe B et de classe A. Lorsqu'il est créé c doit l'être avec les caractéristiques d'un objet de A de B et de C.

CONSTRUCTEUR PAR DÉFAUT

- Lorsqu'une classe ne définit pas explicitement de constructeur, elle possède un constructeur par défaut :
 - *sans paramètres*
 - *de corps vide*
 - *inexistant si un autre constructeur existe*

```
public class Object {  
  
    public Object()  
    {  
        ...  
    }  
    ...  
}
```

```
public class A extends Object {  
  
    // attributs  
    String nom;  
  
    // méthodes  
    String getNom() {  
        return nom;  
    }  
    ...  
}
```

```
public A() {  
    super();  
}
```

Constructeur
par défaut
implicite

Garantit chaînage
des constructeurs

REDÉFINITION DES ATTRIBUTS (SHADOWED VARIABLES)

- Lorsqu'une sous classe définit une variable d'instance dont le nom est identique à l'une des variables dont elle hérite, **la nouvelle définition masque la définition héritée**
 - *l'accès à la variable héritée se fait au travers de **super***

```
public class ClasseA {  
    int x;  
}
```

```
public class ClasseB extends Classe A {  
    double x;  
}
```

```
public class ClasseC extends Classe B {  
    char x;  
}
```

en général ce n'est pas une très bonne idée de masquer les variables

`((ClasseA) this) .x`

~~`super.super.x`~~

`super.x`

`x` ou `this.x`

Dans le code de ClasseC

VISIBILITÉS DES MÉTHODES ET VARIABLES

principe **d'encapsulation** : les données propres à un objet ne sont accessibles qu'au travers des méthodes de cet objet

sécurité des données : elles ne sont accessibles qu'au travers de méthodes en lesquelles on peut avoir confiance

masquer l'implémentation : l'implémentation d'une classe peut être modifiée sans remettre en cause le code utilisant celle-ci.

En JAVA possibilité de contrôler l'accessibilité (visibilité) des membres (variables et méthodes) d'une classe

public accessible à toute autre classe

private n'est accessible qu'à l'intérieur de la classe où il est défini

protected est accessible dans la classe où il est défini, dans toutes ses sous-classes et dans toutes les classes du même package

package (visibilité par défaut) n'est accessible que dans les classes du même package que celui de la classe où il est défini

LES PAQUETAGES

Les paquetages de Java permettent de structurer les grosses applications en groupes de classes liées les unes aux autres.

Par ailleurs, un paquetage définit un espace de noms : deux classes (ou interfaces) peuvent porter le même nom dès lors qu'elles sont dans des paquetages différents. Pour lever toute ambiguïté éventuelle, il suffit de préfixer le nom de la classe ou de l'interface par le nom du paquetage qui la contient.

Constitution des paquetages

S'il est présent, le mot clef `package` suivi du nom du paquetage doit figurer en **premier** dans un fichier source Java. Toutes les classes de ce fichier font alors partie de ce paquetage, **mais** une seule de ces classes peut être publique et doit alors donner son nom au fichier.

Par exemple, le source suivant

```
package desCompteurs ;
```

```
public class Compteur { ... }
```

indique que la classe `Compteur` appartient au paquetage `desCompteurs` est qu'elle est utilisable à l'extérieur car publique.

LES PAQUETAGES

Plusieurs fichiers sources peuvent participer au contenu d'un même paquetage, ceci permet d'éviter des fichiers monstrueux et des compilations trop longues, et aussi de faire qu'un paquetage propose plusieurs classes publiques. Si on veut que le paquetage `desCompteurs` propose aussi la classe `CompteurMod` on écrit un autre fichier :

```
package desCompteurs ;
```

```
public class CompteurMod extends Compteur { ... }
```

Les classes d'un fichier qui ne propose pas explicitement de nom de paquetage appartiennent à un paquetage anonyme (souvent assimilé au répertoire courant lors de la compilation).

La structuration en paquetages est hiérarchique et utilise une notation pointée, chaque point séparant deux niveaux consécutifs de la hiérarchie, par exemple les paquetages `java.lang` et `java.awt.event`.

Pour pouvoir être mentionnée (utilisée) à l'extérieur du paquetage qui la contient, une classe ou une interface doivent être déclarées public.

RÉCAPITULATION DES RÈGLES DE VISIBILITÉ

En ce qui concerne un membre de classe, il y a quatre possibilités suivantes qu'on fixe explicitement ou implicitement sa visibilité :

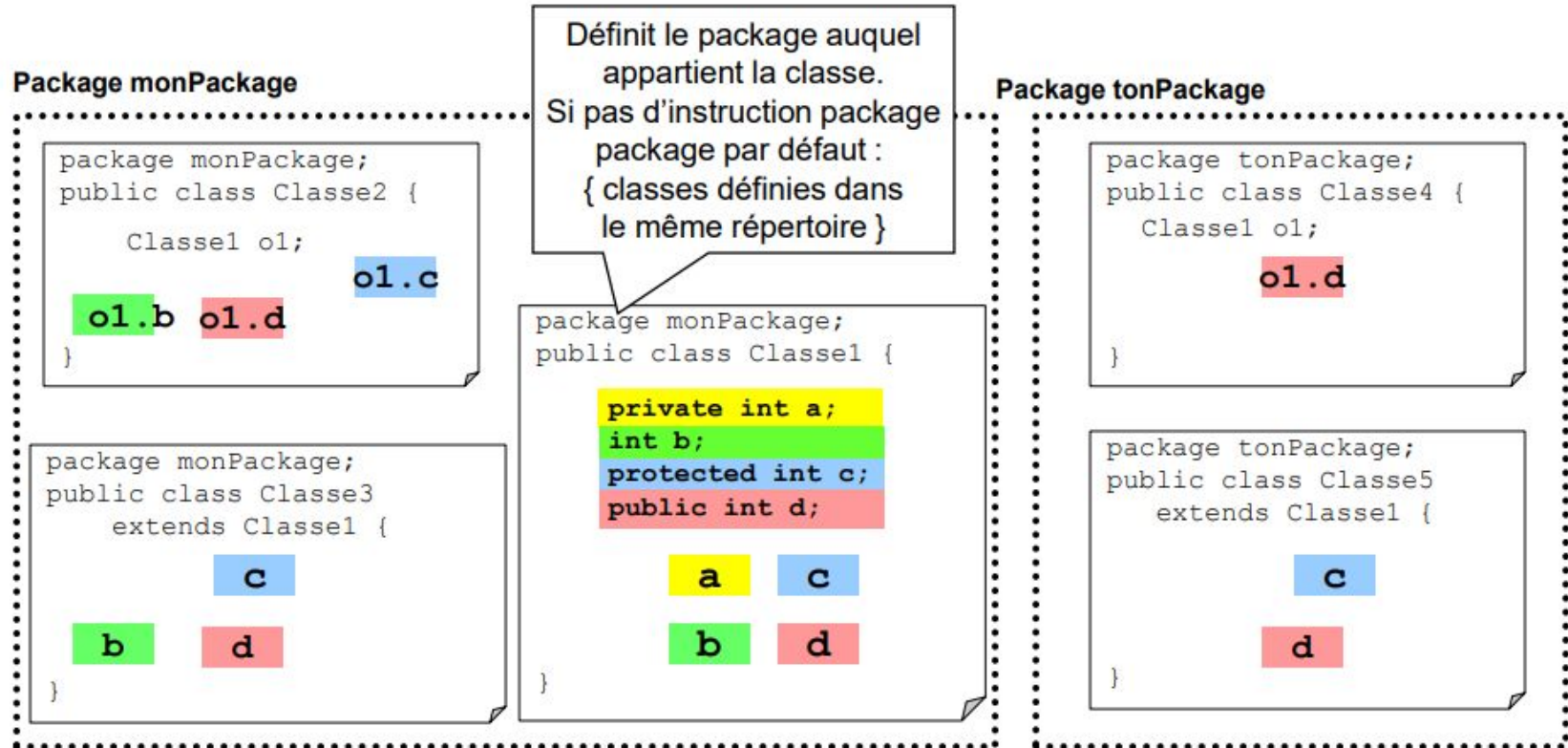
public le membre est partout visible,

protected le membre n'est visible qu'à l'intérieur du même paquetage et dans les sous-classes de la classe qui déclare ce membre (même si elles ne sont pas dans le même paquetage),

private le membre n'est visible que dans la classe qui le déclare,

implicitement le membre n'est visible qu'à l'intérieur du même paquetage.

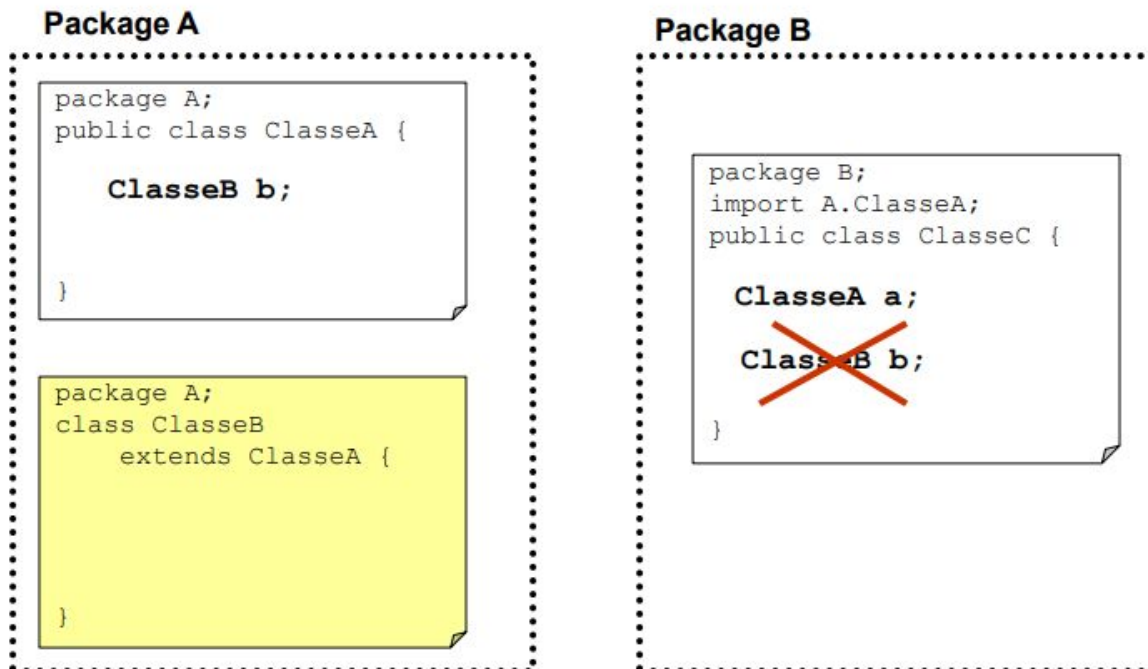
	private	- (package)	protected	public
La classe elle même	oui	oui	oui	oui
Classes du même package	non	oui	oui	oui
Sous-classes d'un autre package	non	non	oui	oui
Classes (non sous-classes) d'un autre package	non	non	non	oui



Les mêmes règles de visibilité s'appliquent aux méthodes

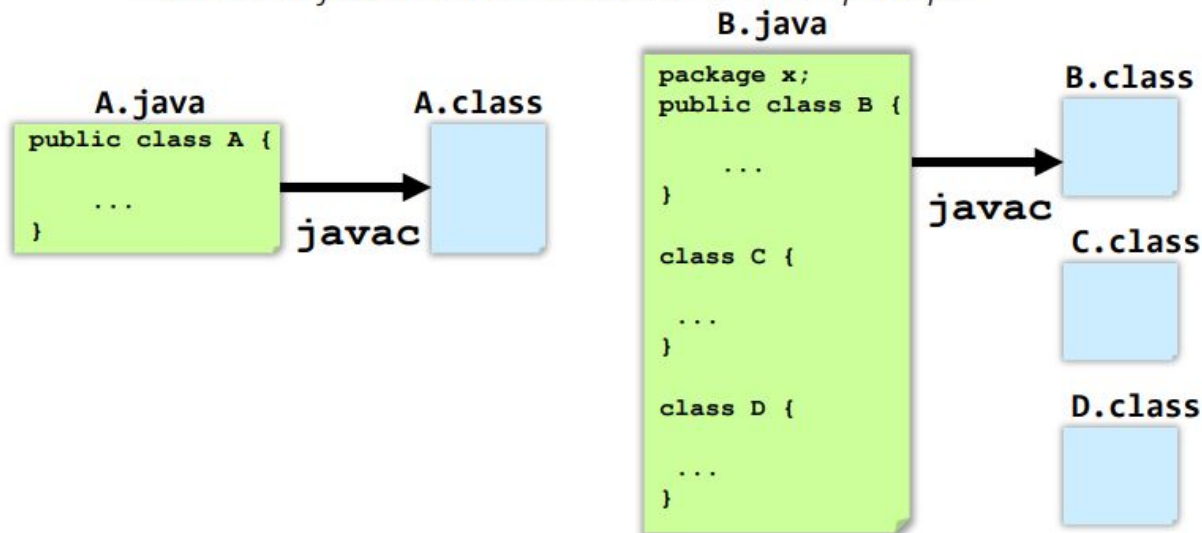
VISIBILITÉ DES CLASSES

- Deux niveaux de visibilité pour les classes :
 - **public** : la classe peut être utilisée par n'importe quelle autre classe
 - - (**package**) : la classe ne peut être utilisée que par les classes appartenant au même package



VISIBILITÉ DES CLASSES

- Jusqu'à présent on a toujours dit :
 - *une classe par fichier*
 - *le nom du fichier source : le nom de la classe avec extension **.java***
- En fait la vraie règle est :
 - *une classe **publique** par fichier*
 - *le nom du fichier source : le nom de la classe publique*



MÉTHODES FINALES

public **final** void méthodeX(...) {.... }

« verrouiller » la méthode pour interdire toute éventuelle redéfinition dans les sous-classes

efficacité : quand le compilateur rencontre un appel à une méthode finale il **peut** remplacer l'appel habituel de méthode (empiler les arguments sur la pile, saut vers le code de la méthode, retour au code appelant, dépilement des arguments, récupération de la valeur de retour) par une copie du code du corps de la méthode (inline call).

? si le corps de la méthode est trop gros, le compilateur est censé ne pas faire cette optimisation qui serait contrebalancée par l'augmentation importante de la taille du code.

? Mieux vaut ne pas trop se reposer sur le compilateur :

? utiliser **final** que lorsque le code n'est pas trop gros ou lorsque l'on veut explicitement éviter toute redéfinition

méthodes **private** sont implicitement **final** (elles ne peuvent être redéfinies)

CLASSES FINALES

Classes finales : Une classe peut être définie comme finale

```
public final class UneClasse {  
...  
}
```

interdit tout héritage pour cette classe qui ne pourra être sous-classée

toutes les méthodes à l'intérieur de la classe seront implicitement finales (elles ne peuvent être redéfinies)

*exemple : la classe **String** est finale*

Attention à l'usage de **final**, prendre garde de ne pas privilégier une supposée efficacité au détriment des éventuelles possibilités de réutiliser la classe par héritage.

EXEMPLE

Pour les classes A et B définies comme suit:

```
class A {  
    public int x;  
    public A( ) {x=5; }  
    public String toString(){return ("la valeur x est: "+x);}  
}  
class B extends A {  
    public B( ) {x++;}  
    public B(int i){this( ); x=x+i; }  
    public B(String s){super( ); x- -; }  
    public String toString(){return (super.toString()+ " ciao");}}
```

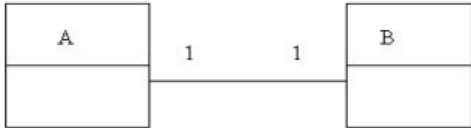
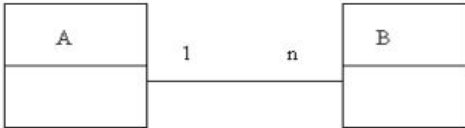

qu'affichera le code suivant?

```
B b1=new B( ); B b2 =new B(2003); B b3= new B("Bonjour");  
System.out.println(b1) ;  
System.out.println(b2.x + " et encore " + b3.x );
```

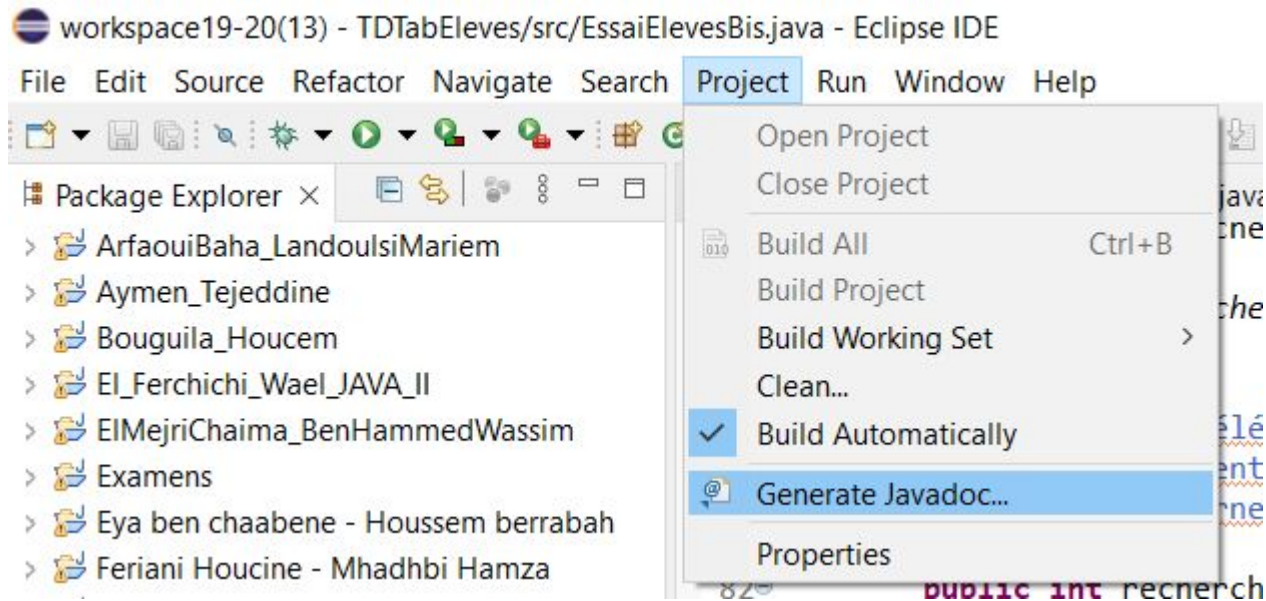
la valeur x est: 6 ciao

2009 et encore 4

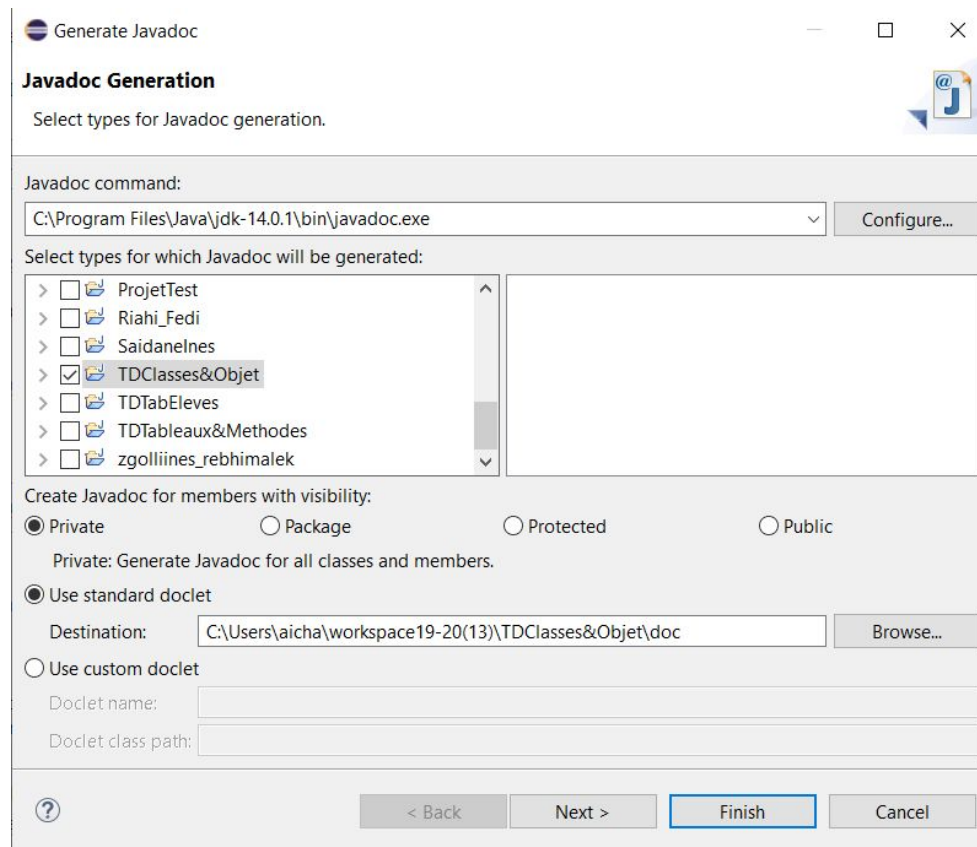
RÈGLES DE PASSAGE UML-JAVA

UML	JAVA
<p>Cas d'association un à un.</p>  <pre> classDiagram class A class B A "1" -- "1" B </pre>	<pre> public class A{ private B leB; } public class B{ private A leA; } </pre>
<p>Association de un à n.</p>  <pre> classDiagram class A class B A "1" -- "n" B </pre>	<pre> public class A{ private ArrayList lesB = new ArrayList(); } public class B{ private A leA; } </pre>
<p>Navigabilité restreinte.</p>  <pre> classDiagram class A class B A --> "n" B </pre>	<pre> public class A{ private ArrayList lesB = new ArrayList(); } public class B{ // pas de référence à un objet de la // classe A } </pre>

GÉNÉRATION LA DOC DE JAVA



GÉNÉRATION LA DOC DE JAVA



GÉNÉRATION LA DOC DE JAVA

Generate Javadoc

Javadoc Generation

Configure Javadoc arguments for standard doclet.

☒ Document title: Documentation projet TDClasses&Objets

Basic Options

- ☒ Generate use page
- ☒ Generate hierarchy tree
- ☒ Generate navigator bar
- ☒ Generate index
- ☒ Separate index per letter

Document these tags

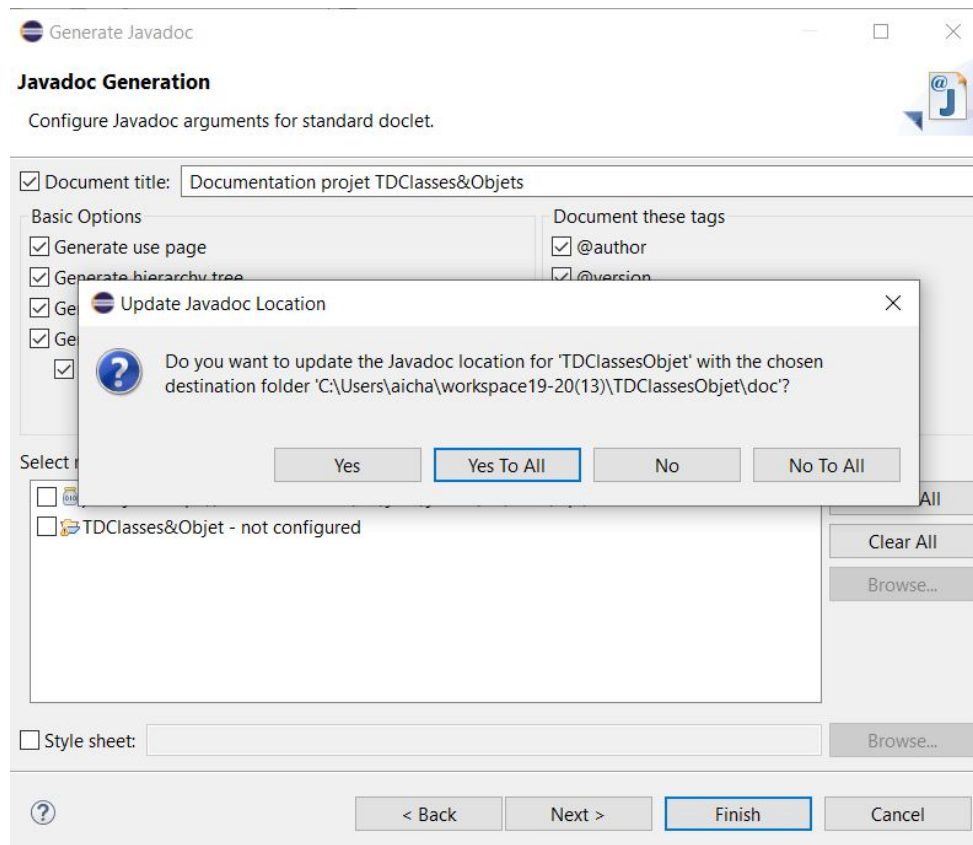
- ☒ @author
- ☒ @version
- ☒ @deprecated
- ☒ deprecated list

Select referenced archives and projects to which links should be generated:

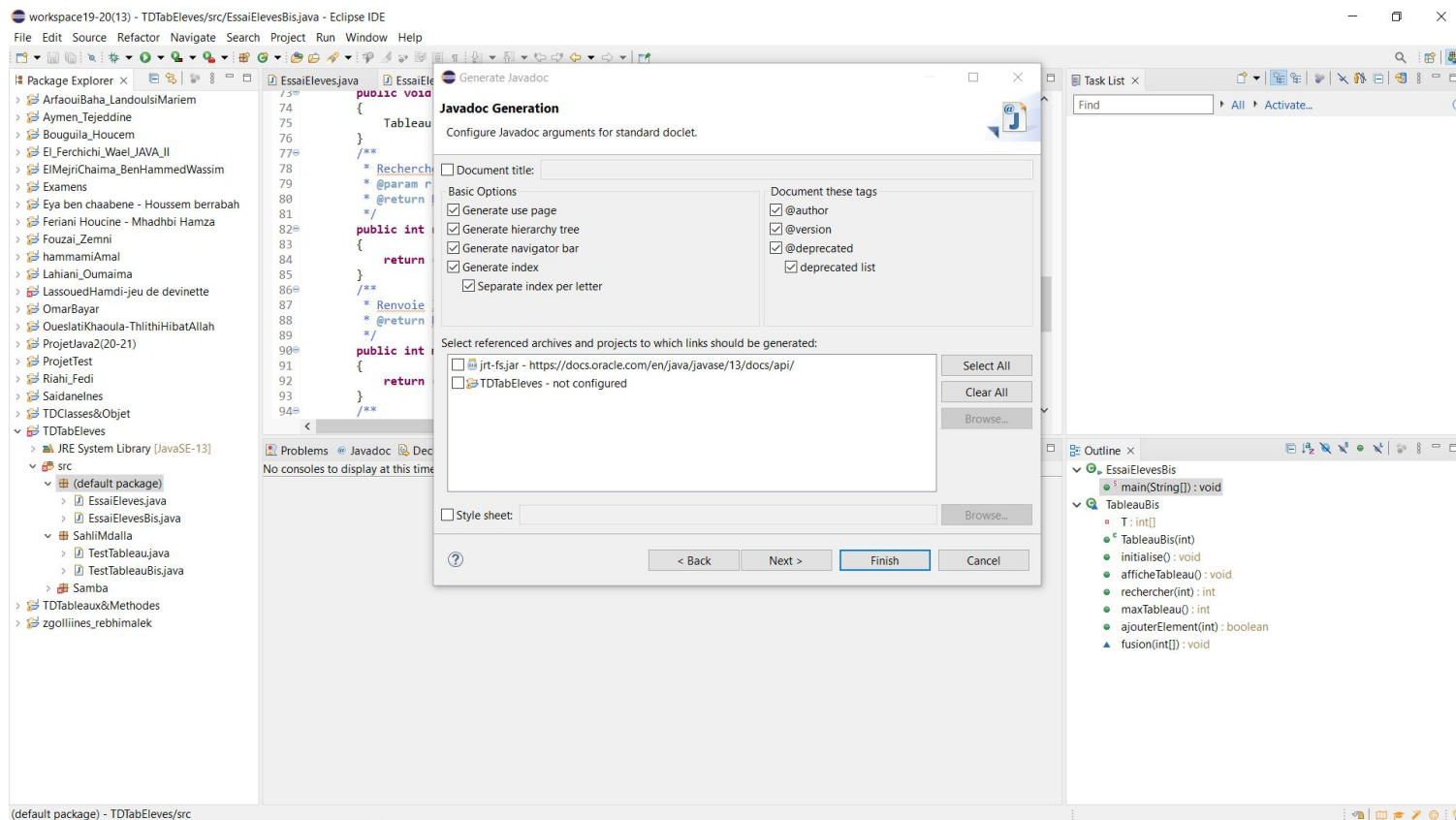
- ☐ jrt-fs.jar - <https://docs.oracle.com/en/java/javase/13/docs/api/>
- ☐ TDClasses&Objet - not configured

☐ Style sheet: Browse...

GÉNÉRATION LA DOC DE JAVA



GÉNÉRATION LA DOC DE JAVA



GÉNÉRATION LA DOC DE JAVA

- ▼ TDClasses&Objet
 - > JRE System Library [JavaSE-13]
 - ▼ src
 - ▼ (default package)
 - > Point2D.java
 - ▼ doc
 - > class-use
 - > index-files
 - > resources
 - > script-dir
 - allclasses-index.html
 - allpackages-index.html
 - constant-values.html
 - deprecated-list.html
 - element-list
 - help-doc.html
 - index.html**
 - member-search-index.js
 - member-search-index.zip
 - overview-tree.html
 - package-search-index.js
 - package-search-index.zip
 - package-summary.html
 - package-tree.html

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH:

Class Point2D

java.lang.Object
Point2D

```
public class Point2D
extends java.lang.Object
```

Field Summary

Fields		
Modifier and Type	Field	Description
protected double	x	abscisse du point
protected double	y	ordonnée du point

Constructor Summary

Constructors	
Constructor	Description