

Programmation Orientée Objet

Java

ENSA de Kenitra

2018 - 2019



ORACLE®



Pr. Aniss MOUMEN (ENSA – Kenitra)

amoumen@gmail.com

06.65.36.63.70

[Facebook](#) - [Linkedin](#) - [Researchgate](#)



ORACLE®



POO en Java : Partie II

- 1- UML**
- 2- Classe & Objet**
- 3- Héritage**
- 4- Exceptions**

Introduction à l'UML

Définition

❓ UML , "Unified Modeling Language" est un langage unifié pour la modélisation, Il fournit les fondements pour spécifier, construire, visualiser et décrire les artefacts d'un système logiciel. Modéliser, c'est décrire de manière visuelle et graphique les besoins et, les solutions fonctionnelles et techniques de votre projet logiciel.

- ❑ UML se base sur une sémantique précise et sur une notation graphique expressive. Il définit des concepts de base et offre également des mécanismes d'extension de ces concepts.
- ❑ UML est donc un langage visuel qui permet de **modéliser** et de communiquer à propos de systèmes, par l'intermédiaire de diagrammes et de texte.
- ❑ UML est un langage de modélisation objet. il facilite l'expression et la communication de modèles en fournissant un ensemble de symboles et de règles qui régissent l'assemblage de ces symboles.
- ❑ UML permet de modéliser de manière claire et précise la structure et le comportement d'un système indépendamment de toute méthode ou de tout langage de programmation.
- ❑ UML n'est pas un langage propriétaire et est accessible à tous.

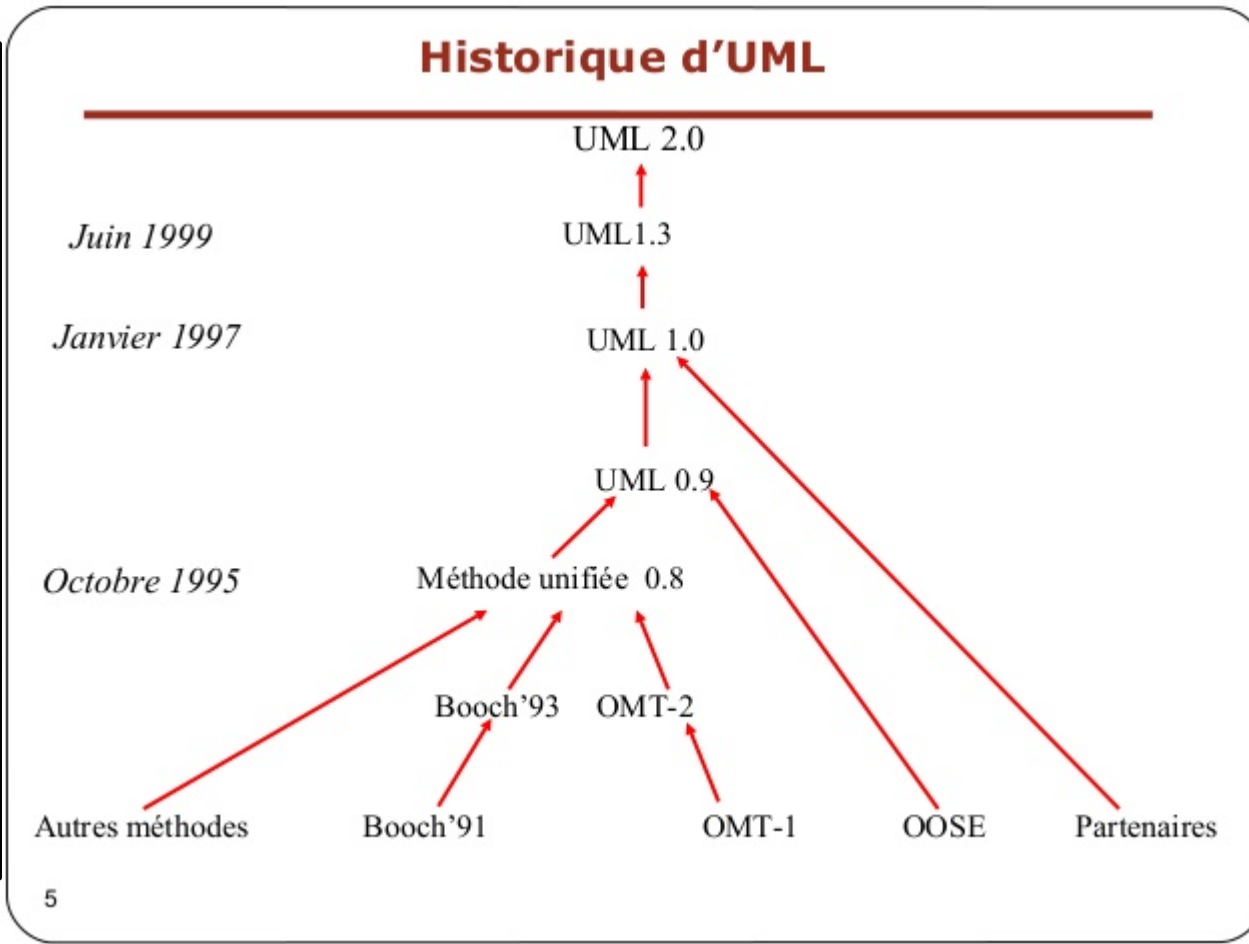
Introduction à l'UML

Historique

❓ **La méthode BOOCH** de Grady Booch mettait l'accent sur le design et la construction des systèmes logiciels.

❓ **La méthode OMT** (Objet Modeling Technique) de James Rumbaugh insistait sur l'analyse des systèmes logiciels.

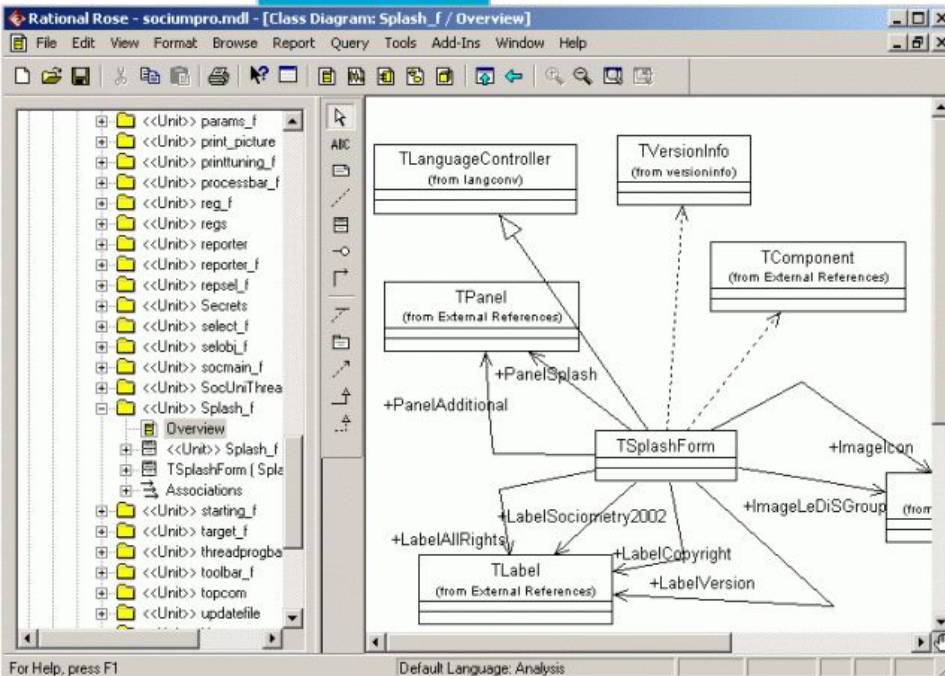
❓ **La méthode OOSE** (Object-Oriented Software Engineering) de Ivar Jacobson se consacrait à l'analyse des exigences.



Introduction à l'UML

Historique

Rational software



Ivar Jacobson, Grady Booch, and James Rumbaugh
The Unified Software Development Process
Addison Wesley, 1999

1990: James Rumbaugh's OOMD

1992: Ivar Jacobson's Objectory

1993: Grady Booch's OOAD and diagrams

1995: Rational Software unites all three ("the three amigos")

- Definition of unified process
- Definition of UML (Unified Modeling Language)
- Rational Rose toolset

2003: IBM buys Rational

Rational Rose est un logiciel édité par l'entreprise *Rational Machines* (pour créer et éditer les différents diagrammes du modèle UML (**Unified Modeling Language**) d'un logiciel. Rational Software a été vendu pour 2,1 milliards de dollars à [IBM](#) le 20 février 2003. Rational Rose permet également de sauvegarder et d'imprimer ces diagrammes, ainsi que de [générer le code source Java](#) ou [C++](#) qui leur correspondent.

Introduction à l'UML

Langage UML

□ **L'alphabet d'UML** est composé essentiellement de formes géométriques et symboliques (rectangles, lignes, autres éléments graphiques) et de chaînes de caractères.

□ **Un mot** représente un groupe d'éléments issus de l'alphabet du langage, qui définit une unité de sens. En UML les mots appartiennent à **deux grandes catégories** :

Concepts : qui sont représentés par des rectangles ou des symboles avec un nom.

Relation entre concepts : ils sont illustrés par des lignes connectant les symboles entre eux.

□ **Une phrase** représente un groupe de mots issus du vocabulaire du langage, qui définit une unité de sens grammaticale contenant un sujet et une expression concernant ce sujet. La grammaire d'un langage spécifie les règles de combinaison des mots afin de former des phrases



Figure 2. Concepts et relations

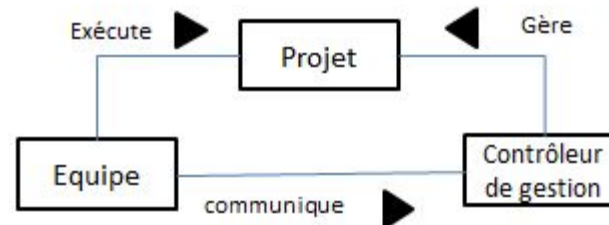
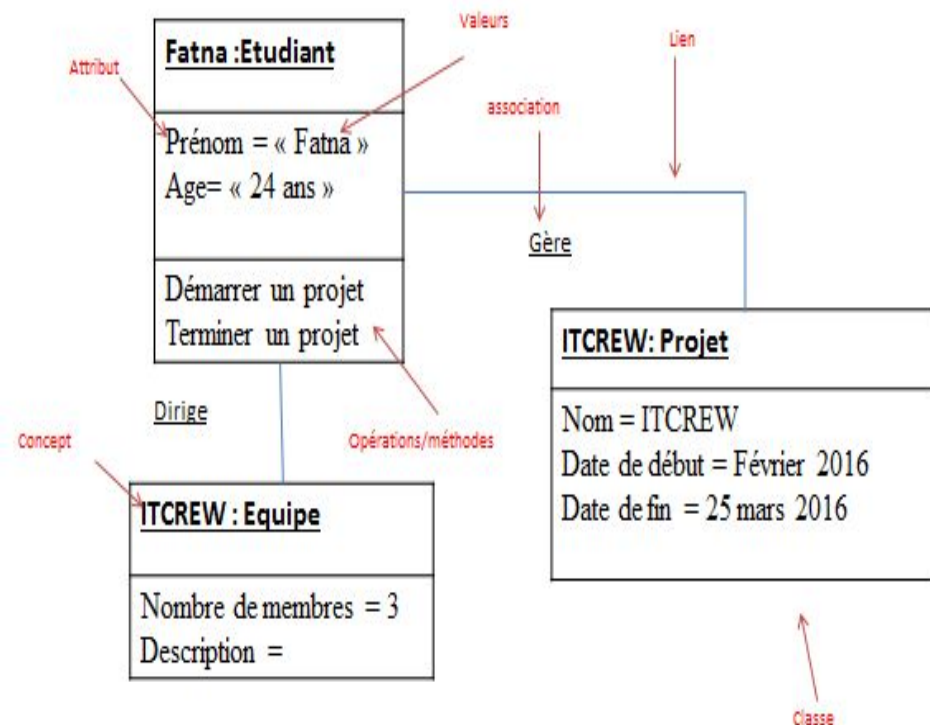


Figure 3. Concepts et relations, sens de phrase

Introduction à l'UML

Associations, classes, objets, liens, attributs et méthodes

- Les concepts qui expriment les phrases s'appellent des **classes** et les relations générales s'appellent des **associations**.
- Les **concepts** sont alors appelés **objets**, et les relations **liens**.
- Une **classe** définit un type **d'objet** et ses **caractéristiques**.
- Un **attribut** est un élément connu par un objet et représente essentiellement une **donnée**. Une classe définit des attributs et un objet possède les valeurs pour ces attributs.
- La manière dont l'objet réalise le **traitement** correspondant à une opération donnée correspond à la **méthode** ou **implémentation** de l'opération. Une classe définit les méthodes qui s'appliquent à ses objets.



Introduction à l'UML

Diagrammes

UML définit 9 types de diagrammes dans deux catégories de vues, les vues statiques et les vues dynamiques :

Vues statiques:

Les diagrammes de cas d'utilisation décrivent le comportement et les fonctions d'un système du point de vue de l'utilisateur

Les diagrammes de classes décrivent la structure statique, les types et les relations des ensembles d'objets

Les diagrammes d'objets décrivent les objets d'un système et leurs relations

Les diagrammes de composants décrivent les composants physiques et l'architecture interne d'un logiciel

Les diagrammes de déploiement décrivent la répartition des programmes exécutables sur les différents matériels

Vues dynamiques :

Les diagrammes de collaboration décrivent les messages entre objets (liens et interactions)

Les diagrammes d'états-transitions décrivent les différents états d'un objet

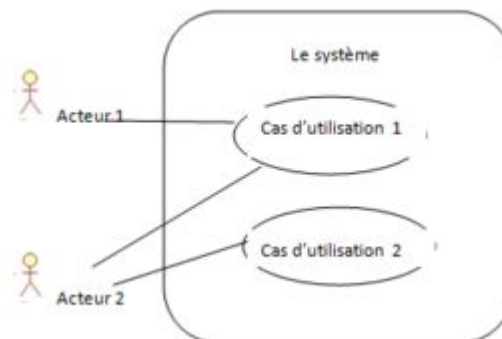
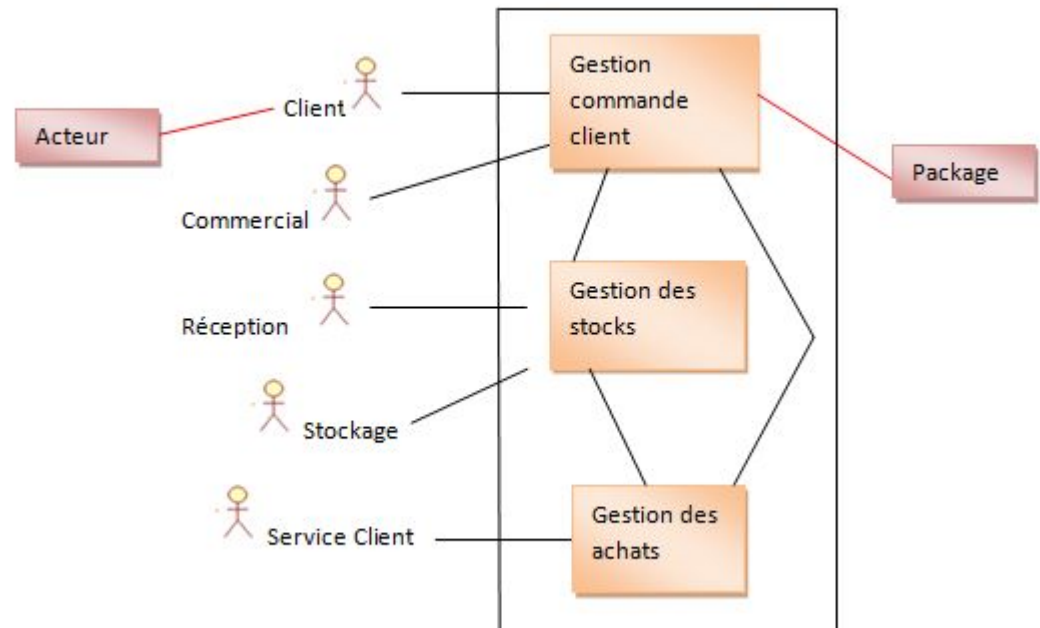
Les diagrammes d'activités décrivent les comportements d'une opération (en termes d'actions)

Les diagrammes de séquence décrivent de manière temporelle les interactions entre objets et acteur

Introduction à l'UML

Diagramme des cas d'utilisation

Un système est une boîte qui contient d'autres boîtes, appelées **packages**. Le contenu d'un package est illustré par différents diagrammes. Un de ces diagrammes représente les **cas d'utilisation**, c'est-à-dire les **fonctionnalités** ou lots d'actions que devront **réaliser nos acteurs**. Le diagramme de cas d'utilisation représente les fonctionnalités nécessaires aux utilisateurs.

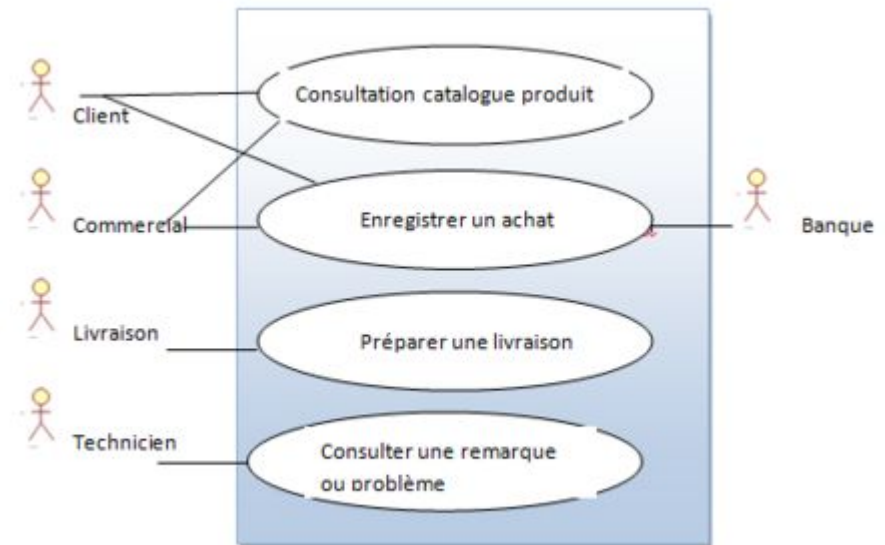


Introduction à l'UML

Diagramme des cas d'utilisation

Méthodologie :

- ❑ Défini le contexte du futur logiciel, en identifiant les différents **acteurs**;
- ❑ décomposé le système en **packages**, c'est-à-dire les familles de fonctionnalités;
- ❑ illustré les fonctionnalités principales pour un des packages, grâce aux **cas d'utilisation**.



Introduction à l'UML

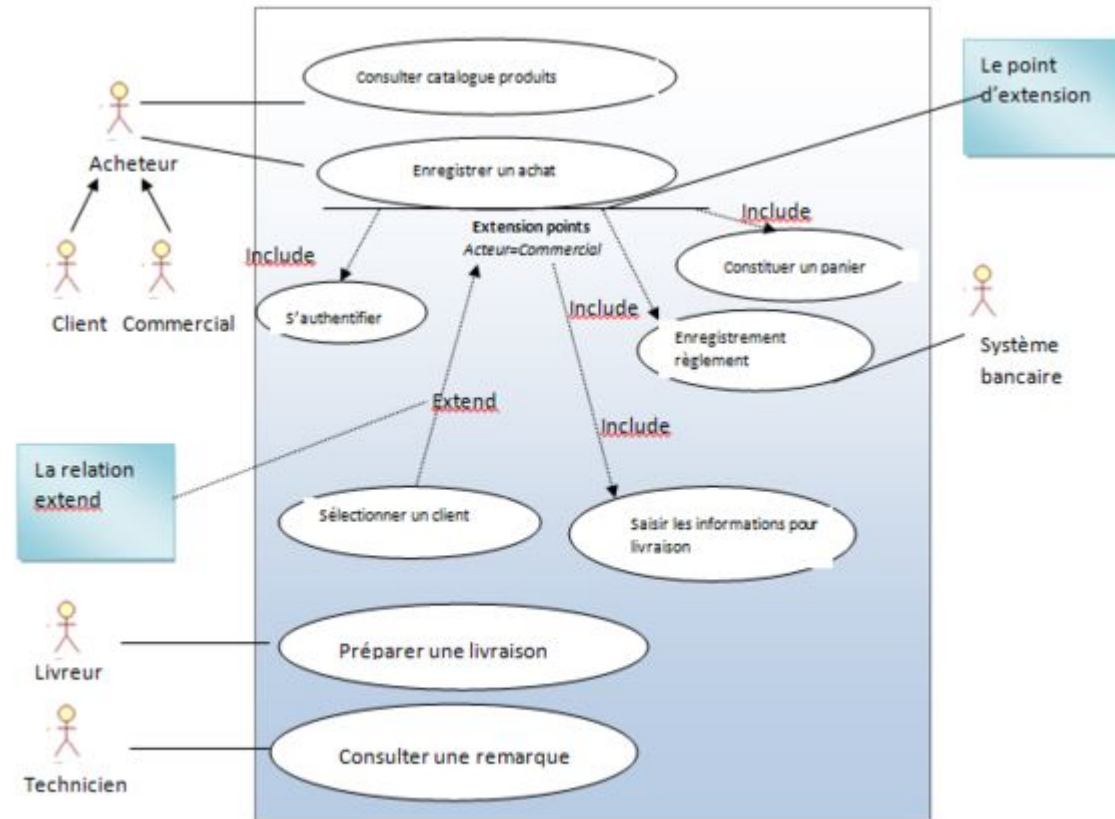
Diagramme des cas d'utilisation

Les cas d'utilisation principaux sont complétés par des cas d'utilisation internes, qui sont des précisions d'autres cas d'utilisation.

Les cas d'utilisation internes sont reliés au cas initial par des relations de type « **include** » ou « **extend** ».

☐ Une relation « **include** » permet d'indiquer qu'un cas d'utilisation a toujours besoin du cas d'utilisation lié.

☐ Une relation « **extend** » c'est une relation qui est soumise à une **condition**.

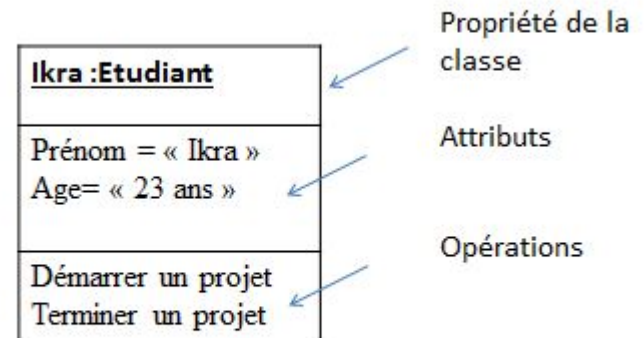


Introduction à l'UML

Diagramme des classes

Le diagramme de classe est un diagramme faisant partie des diagrammes structurels d'UML le plus utilisé du fait de sa notation syntaxique riche. Il représente la structure d'une application orientée objet en montrant les classes et les relations qui s'établissent entre elles.

Les classes sont schématisées par une boîte rectangulaire à trois sections, la première avec le nom de la classe et ses propriétés, la seconde avec les attributs et la troisième avec les opérations.



Représentation d'une classe

Introduction à l'UML

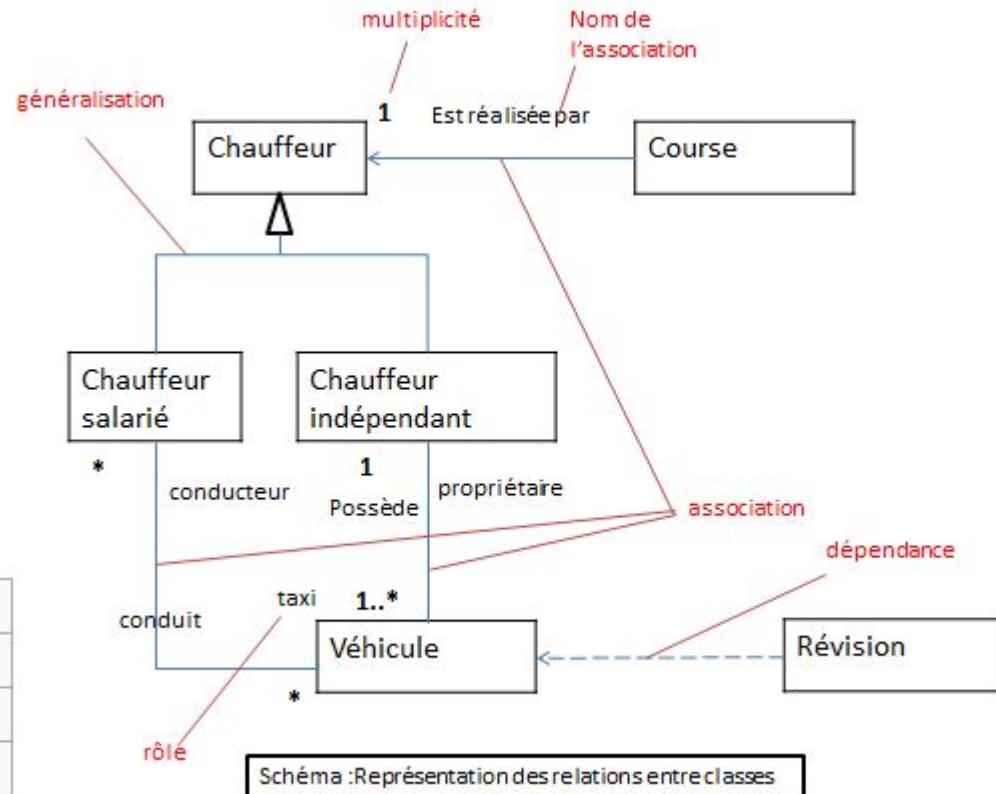
Diagramme des classes

❓ Il y a quatre types de relation entre classes :

- L'héritage ou généralisation
- La réalisation
- L'association
- La dépendance

❓ Association : Relation qui peut s'établir entre instances de classes.

Multiplicité	Signification
1	Exactement 1
5	Exactement 5
*	Plusieurs incluant la possibilité d'aucun
0..*	Idem
0..1	Au plus un, ce qui signifie également optionnellement
1..*	Au moins un
1..5	Entre un et cinq

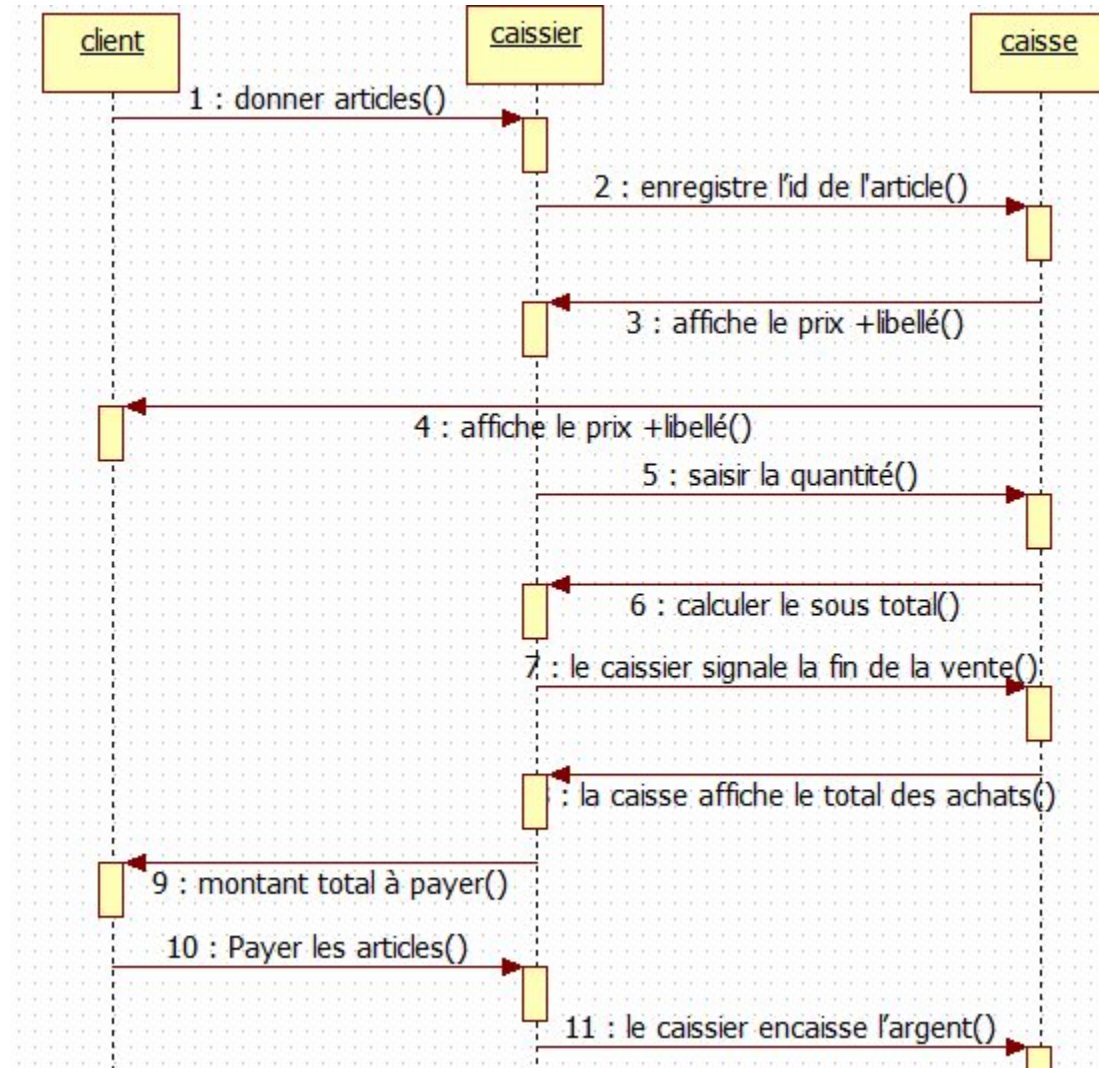


Introduction à l'UML

Diagramme de séquence

Les **diagrammes de séquences** sont la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique.

Dans un souci de simplification, on représente l'acteur principal à gauche du diagramme, et les acteurs secondaires éventuels à droite du système. Le but étant de décrire comment se déroulent les actions entre les acteurs ou objets.

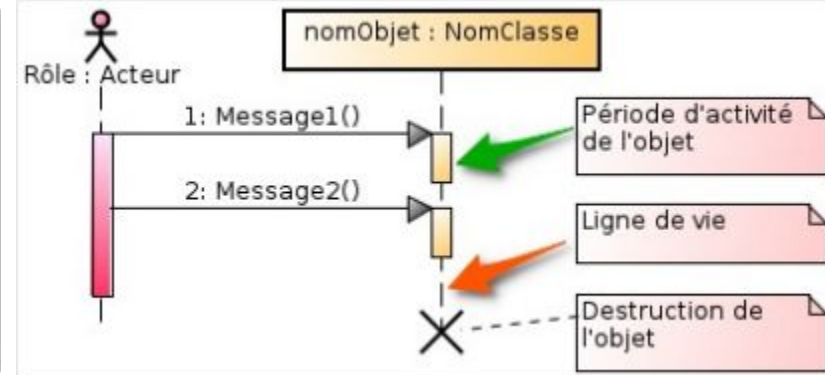
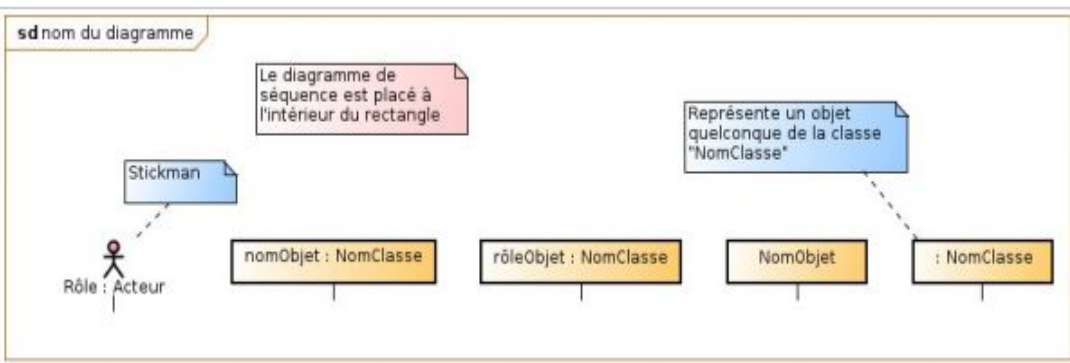


Introduction à l'UML

Diagramme de séquence

Méthodologie :

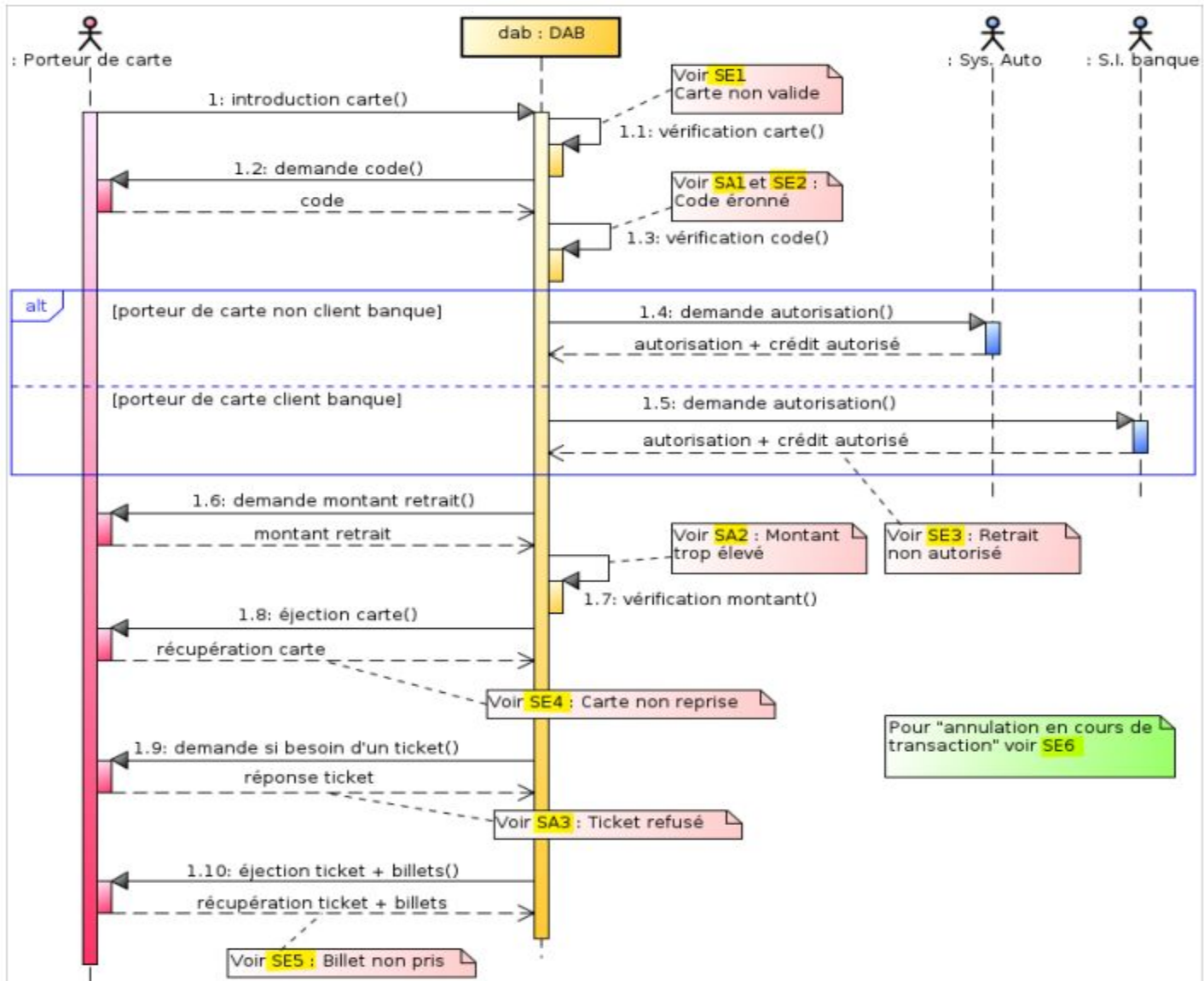
- ▣ **Délimitation du diagramme de séquence** : Le diagramme de séquence est placé dans un rectangle qui dispose d'une étiquette sd en haut à gauche (qui signifie sequence diagramm) suivi du nom du diagramme.
- ▣ **L'objet** : Un rectangle dans lequel figure le nom de l'objet. Le nom de l'objet est généralement souligné et peut prendre l'une des quatre formes suivantes :



- ▣ **La ligne de vie** : chaque objet est associé à une ligne de vie (en trait pointillés à la verticale de l'objet) qui peut être considéré comme un axe temporel (le temps s'écoule du haut vers le bas).
- ▣ **Les messages** : Un message définit une communication particulière entre des lignes de vie. Ainsi, un message est une communication d'un objet vers un autre objet.

Introduction à l'UML

Diagramme de séquence

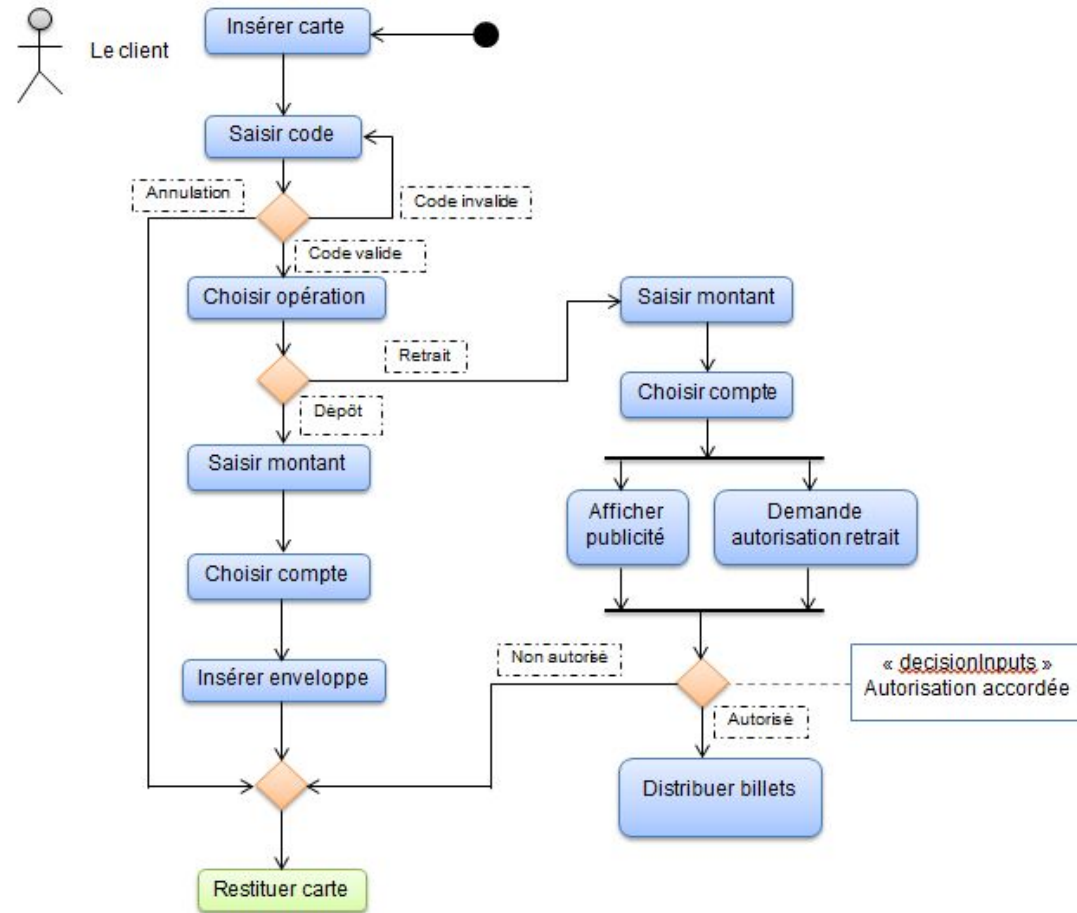


Introduction à l'UML

Diagramme d'activité

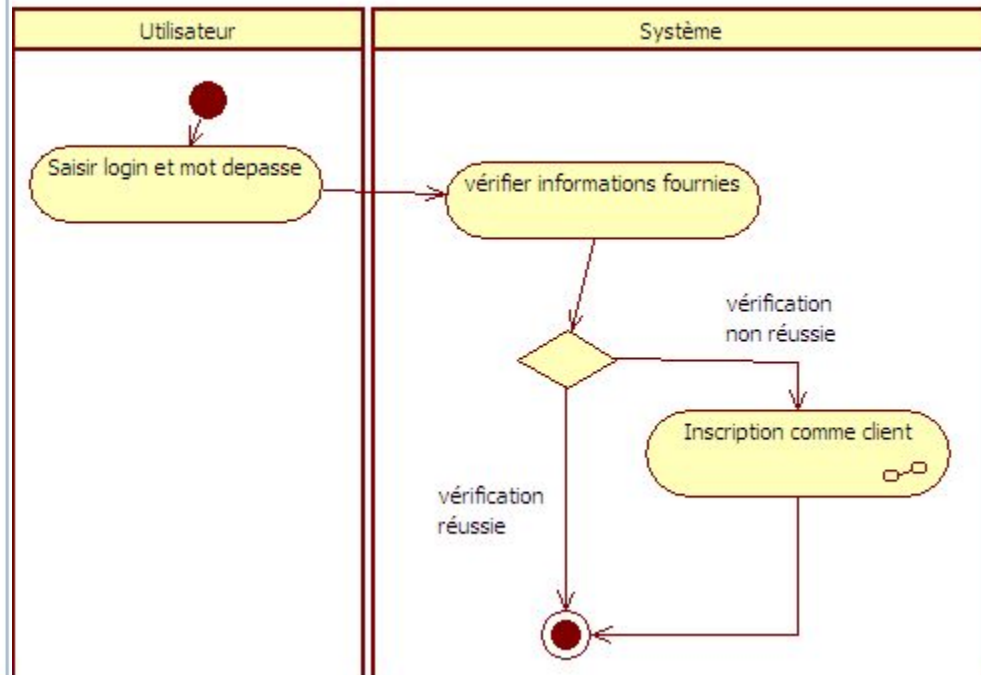
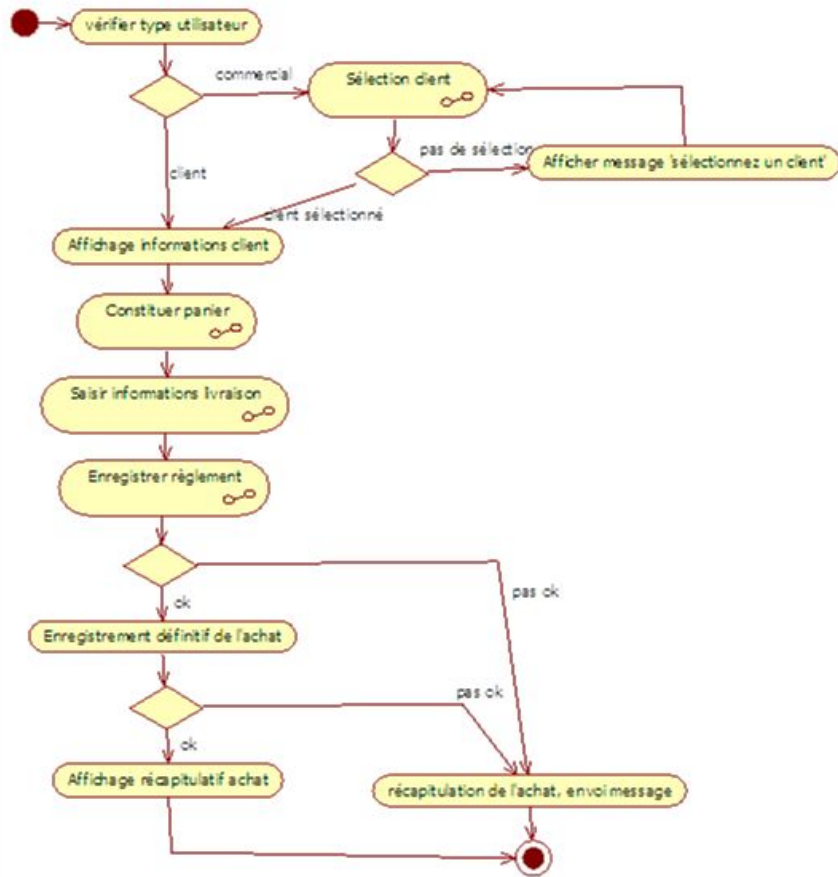
Les **diagrammes d'activités** permettent de déterminer des traitements *a priori* séquentiels.

Les diagrammes d'activités sont adaptés à la modélisation du cheminement de flots de contrôle (toutes les instructions, branches, chemins...) et de flots de données (toutes les définitions de variable, les utilisations...). Ils représentent graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation.



Introduction à l'UML

Diagramme d'activité





ORACLE®



POO en Java : Partie II

1- UML

2- Classe & Objet

3- Héritage

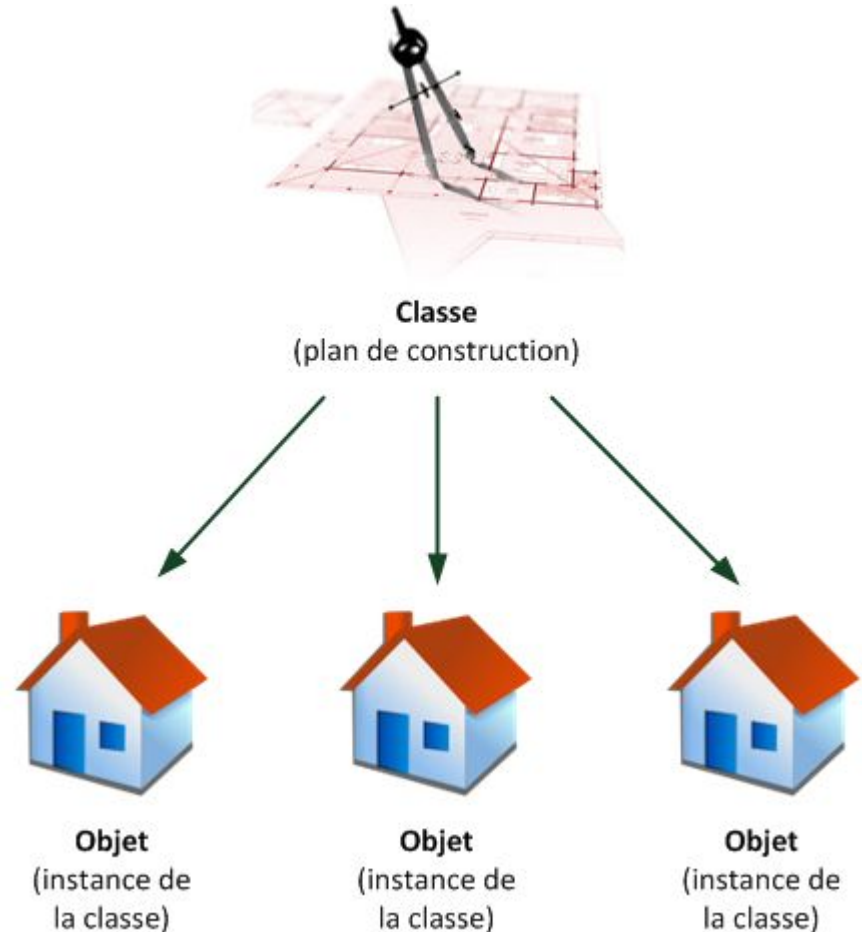
4- Exceptions

5-Collection

Classe & Objet

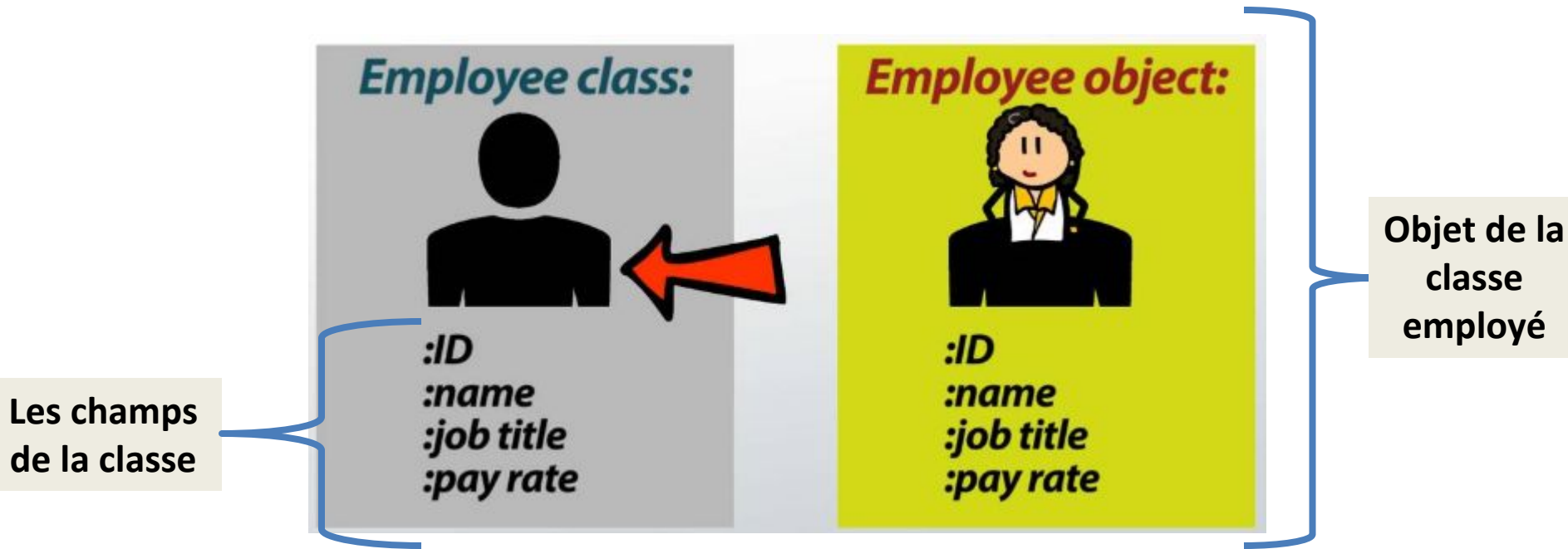
Définitions

- ❓ Une **classe** permet de regrouper dans une même entité, les données et les fonctions membres dites : méthodes, permettant de manipuler ces données.
- ❓ La **classe** est une évolution de la notion de structure
- ❓ Un **objet** est un exemplaire d'une **classe** : on parle de *l'instance d'une classe*.



Classe & Objet

Instantiation d'un objet ? Création d'un exemplaire de la classe



SYNTAXE

```
<nomclasse> obj = new nomclasse();
```

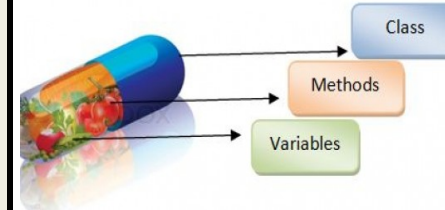
Classe & Objet

Encapsulation

❓ **L'encapsulation** est un mécanisme du POO qui interdit l'accéder à certaines données depuis l'extérieur de la classe ❓ à partir d'une autre classe/méthode du même package ou d'un autre package.

❓ Un utilisateur de la classe sera **obligé** d'utiliser certaines fonctions membres de la classe (celles qui sont **publiques**).

❓ Vue de l'extérieur, la **classe** apparaît comme **une boîte noire**, qui a un certain comportement à laquelle on ne peut y accéder que par **les méthodes publiques**.



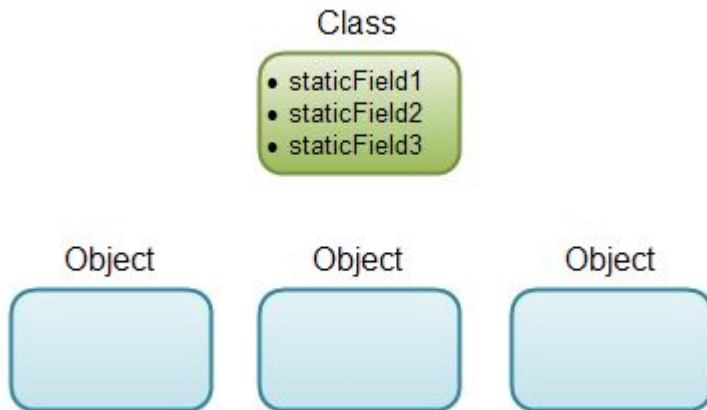
Classe & Objet

Implémentation de l'encapsulation

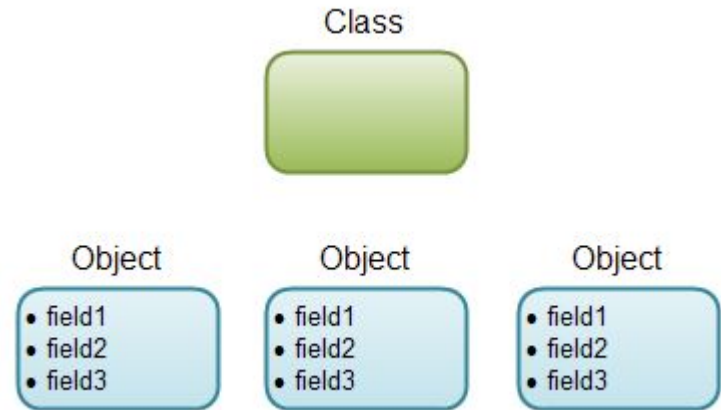
- ❓ L'implémentation de l'encapsulation se fait en déclarant les champs d'une classe avec l'un des mots réservés ci-dessous qui définissent la portée :
- **public**, il sera accessible depuis n'importe quelle classe.
 - **private**, il sera uniquement accessible, depuis les fonctions qui sont membres de la classe.
 - **protected**, il aura les mêmes restrictions que s'il était déclaré *private*, mais il sera en revanche accessible par les classes filles.
 - **« par default »**, si la portée n'est pas défini, le membre est accessible depuis la même classe et par toutes autres classes du même package.
- En Java, l'encapsulation est implémenté en déclarant les variables en « **private** » et on développe des méthodes public dites **« accesseurs » (getter)** et **« modificateurs » (setter)**.
- Dans le cas d'un champ déclaré statique « **static** » (**static type nomchamps;**), il n'est pas obligatoire d'instancier un objet pour y accéder, on accède juste avec le nom de la classe :
- <nomclasse>.<nomchamps> = valeur;**

Classe & Objet

Implémentation de l'encapsulation (membre donnée)



*Une classe avec des champs
déclarés en « **static** »*



*Une classe avec des champs
déclarés en **non-static***

Classe & Objet

Implémentation de l'encapsulation (membre méthode)

- ❑ **une méthode** est une **fonction** regroupant un ensemble d'instruction et appartenant à une classe.
- ❑ Elle possède un ***nom***, des ***paramètres***, une ***valeur de retour*** et une ***visibilité***.
- ❑ Java possède un nombre important de ***classe et méthode prédéfini (plus de 6000 classe en 2017)***.
- ❑ Après la **définition** de la méthode, on procède à son **utilisation (appel)**.

SYNTAXE

```
modifier returnType nameOfMethod (Parameter List)
{
    // method body
}
```

Classe & Objet

getter & setter

❓ **Accesseurs (getter):** ce sont des fonctions membres qui ne modifient pas l'état de l'objet, et permettent de récupérer les valeurs de ses propriétés.

❓ **Modificateurs (setter):** ce sont des fonctions membres qui peuvent modifier l'état de l'objet (ses propriétés).

Exemple

```
public class Test {  
  
    public String brand = null;  
    public String model = null;  
    public String color = null;  
  
    public void setColor(String newColor) {  
        this.color = newColor;  
    }  
  
    public String getColor() {  
        return color;  
    }  
  
    public static void main(String[] args) {  
        Test veh = new Test();  
        veh.setColor("Red");  
        System.out.println(veh.getColor());  
    }  
}
```



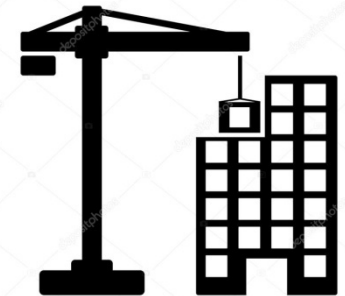
Red

Classe & Objet

Constructeur & Destructeur

❓ Un constructeur est une fonction membre qui sera appelée au moment de la création , qui porte le même nom que la classe et il ne retourne pas de valeur, et elle permet surtout de créer une nouvelle instance d'une classe (création d'un objet)

- Le **constructeur par défaut**: c'est le constructeur sans argument. Si aucun constructeur n'est présent explicitement dans la classe, il est généré par le compilateur.
- Le **constructeur avec argument** : c'est un constructeur dont on initialise les champs de l'objet par les arguments passés au constructeur.
- Le **constructeur de copie (clonage)**: c'est un constructeur dont le type de l'argument est celui de la classe. Ce type de constructeur sert à créer un objet à l'image d'un autre objet.



Classe & Objet

Constructeur & Destructeur

SYNTAXE

POUR LA CLASSE RECTANGLE

Le constructeur est une méthode membre de la classe, qui porte le même nom de la classe, avec ou sans argument.

Constructeur par défaut (sans argument)

```
rectangle obj1 = new  
rectangle();
```

Constructeur avec argument

```
rectangle obj2 = new  
rectangle(arg...);
```

Constructeur par copie

```
rectangle obj3 = new  
rectangle(obj1);
```

Classe & Objet

Constructeur & Destructeur

EXEMPLE : soit la classe rectangle, qui possède les propriétés suivantes : hauteur et largeur (des entiers, public)

Constructeur par défaut (sans argument)

```
public rectangle ()  
{  
    Hauteur = 0;  
    Largeur = 0;  
}
```

Constructeur avec argument

```
public rectangle (int h, int l)  
{  
    Hauteur = h;  
    Largeur = l;  
}
```

Constructeur par copie

```
public rectangle (rectangle A)  
{  
    Hauteur = A.Hauteur;  
    Largeur = A.Largeur;  
}
```

Classe & Objet

Constructeur & Destructeur

```
public class Puppy {  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
        System.out.println("Passed Name is : " + name );  
    }  
  
    public static void main(String []args) {  
        // Following statement would create an object myPuppy  
        Puppy myPuppy = new Puppy( "tommy" );  
    }  
}
```



Passed Name is :tommy

```
Public class MyClass {  
    Int num;  
    MyClass() {  
        num = 100;  
    }  
}
```



100 100

```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();  
        System.out.println(t1.num + " " + t2.num);  
    }  
}
```

Classe & Objet

Constructeur & Destructeur

Le destructeur est automatique en java, grâce au mécanisme du ramasse-miettes

(GARBAGE COLLECTOR)

- ❑ *En Java, on n'implémente pas de méthode pour la destruction comme en C++.*
- ❑ Lorsqu'il n'existe plus aucune référence sur un objet (l'objet n'est plus utilisé) ❑ l'emplacement correspondant est candidat au ramasse-miettes.

Classe & Objet

Constructeur & Destructeur

Exemple définition de la classe Point

```
public class Point
{ public Point (int abs, int ord) // constructeur
  { x = abs ;
    y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ;
    y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
  }
  private int x ; // abscisse
  private int y ; // ordonnee
}
```

Définition d'une classe Point munie d'un constructeur

Classe & Objet

Constructeur & Destructeur

Exemple d'utilisation de la classe Point

```
public class TstPnt3
{ public static void main (String args[])
  { Point a ;
    a = new Point(3, 5) ;
    a.affiche() ;
    a.deplace(2, 0) ;
    a.affiche() ;
    Point b = new Point(6, 8) ;
    b.affiche() ;
  }
}

class Point
{ public Point (int abs, int ord) // constructeur
  { x = abs ;
    y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ;
    y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;
  }
  private int x ; // abscisse
  private int y ; // ordonnee
}
```

```
Je suis un point de coordonnees 3 5
Je suis un point de coordonnees 5 5
Je suis un point de coordonnees 6 8
```

Classe & Objet

Méthode classique

Une **méthode** est un **bloc de traitement** (une fonction) membre d'une classe, identifié (la signature) avec un nom, des arguments, une valeur de retour et visibilité (*public/privé/protected*). comme il peut aussi être statique ou pas. Une méthode quand il n'est pas statique ou de classe, elle est appelé une méthode classique (d'instance)

SYNTAXE GENERALE

```
<VISIBILITÉ> <TYPE DE RETOUR> NOM (TYPE ARG....)  
    {  
    .....  
    }
```

EXEMPLE :

Le constructeur est une méthode particulière, les geter et seter sont des méthodes.....

Classe & Objet

Méthode statique (méthode de classe)

- Une méthode statique est une méthode partager entre les différents objets de la classe.

SYNTAXE

```
<VISIBILITÉ> <STATIC> <TYPE DE RETOUR> NOM (TYPE ARG....)
{
.....
}
```

- Il n'est pas nécessaire de créer un objet pour appeler la méthode statique :
On peut appeler une méthode statique soit :

(1) Avec son NOM

Ou

(2) avec NOMCLASS.METHODE(...)

- Une méthode statique peut afficher, comparer, tester...les données d'une classe.
Par contre elle ne peut agir (ou manipuler) que les membres données déclarés statique.

- EXEMPLE : la méthode main, il est statique vu que c'est traitement partagé par les différents objets.

Classe & Objet

EXEMPLE

Méthode classique | Méthode statique

```
public class Test
{
    public Test()
    {
        MaMethode(50);
    }
    public void MaMethode(int variable)
    {
        System.out.println("Le nombre que vous
avez passé en paramètre vaut : " + variable);
    }
}
```

```
public class Test
{
    public static String chaine = "bonjour";
    public Test()
    {
        MaMethodeStatique();
    }
    public static void MaMethodeStatique()
    {
        System.out.println("Appel de la
méthode statique : " + chaine);
    }
}
```

Classe & Objet

Arguments de méthode

Les arguments de la méthode, sont :

❓ **Des arguments muets** : ceux utilisés lors de la définition de la méthode (dans sa signature)

❑ **Des arguments effectifs** : ceux utilisés lors de l'appel de la méthode (à l'intérieur de main par exemple).

DEFINITION DE LA METHODE

```
<VISIBILITÉ> <TYPE DE RETOUR> NOM (TYPE ARGMUETS....)
{
.....
}
```

APPEL DE LA METHODE :

```
Obj1.Methode classique(ARGEFFECTIF...) ou
NomClass.Methodestatique(ARGEFFECTIF...)
```

Classe & Objet

Transmission des arguments de méthode

Par Valeur

- Passage par valeur : la méthode reçoit sur son argument muet, une copie de la valeur de l'argument effectif.
- Une méthode ne peut pas modifier la valeur d'un argument effectif de **type primitif**. En effet, la méthode reçoit une copie de la valeur de l'argument effectif et toute modification effectuées sur cette copie n'a aucune incidence sur la valeur de l'argument effectif.

Par Référence

- ***Passage par référence*** : la méthode reçoit sur son argument muet, une référence à l'objet passé en argument effectif.
- Une méthode peut modifier l'objet référencé par un argument effectif de **type classe**. En effet, la méthode reçoit une copie de la référence à un objet, puisque l'argument contient une référence à un objet. La méthode peut donc modifier l'objet concerné.

Classe & Objet

Transmission des arguments de méthode

Par Valeur

```
class Chien {
    int taille;
    String nom;

    void aboyer(int nbFois) {
        while (nbFois > 0) {
            System.out.println("Ouaf");
            nbFois = nbFois - 1;
        }
    }
}

class TestChien {

    public static void main(String[] args){
        Chien fido= new Chien();
        fido.aboyer(3); // demande à fido d aboyer 3 fois
    }
}
```

Par Référence

```
class Point
{ public Point(int abs, int ord)
  { x = abs ; y = ord ; }
  public boolean coincide (Point pt)
  { return ((pt.x == x) && (pt.y == y)) ;
  }
  private int x, y ;
}

public class Coincide
{ public static void main (String args[])
  { Point a = new Point (1, 3) ;
    Point b = new Point (2, 5) ;
    Point c = new Point (1,3) ;
    System.out.println ("a et b : " + a.coincide(b) + " " + b.coincide(a) ;
    System.out.println ("a et c : " + a.coincide(c) + " " + c.coincide(a) ;
  }
}

a et b : false false
a et c : true true
```

NB : lorsqu'une méthode d'une classe A reçoit en argument un objet de classe B, différente de A, elle n'a pas accès aux champs ou méthodes **privées** de cet objet.

Classe & Objet

Surcharge des méthodes

- ❓ La surcharge ou surdéfinition d'une méthode, lorsque la même méthode ou symbole, possède plusieurs significations différentes.
- ❑ En Java, la surdéfinition s'applique aux méthodes d'une classe, y compris aux méthodes statiques.
- ❑ Plusieurs méthodes peuvent porter le même **nom**, le compilateur reconnaît celle à prendre en compte, avec le nombre et le type des arguments.
- ❑ **En Java, on ne peut pas surcharger une méthode en changeant juste le type de sa valeur de retour.**

```
class Voiture {  
    public void start() {  
        System.out.println("démarrez la voiture");  
    }  
  
    public void start(int num) {  
        for(int i=0; i < num; i++)  
            System.out.println("démarrez la voiture"+i);  
    }  
}
```

Le même nom de méthode
et paramètre différent

int division(int, int); // division entière
float division(int, int); // division réelle



Classe & Objet

Surcharge des méthodes

```
public class Surcharge
{ public static void main(String args[ ])
  { int i=5,j=6;
    carre(i);
    carre(j);
    carre(i,j);
  }
  static void carre(int n)
  { System.out.println(n*n);
  }
  static void carre(int m,int n)
  { System.out.println(n*n+"\t"+m*m);
  }
}
```

```
class DisplayOverloading
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(char c, int num)
    {
        System.out.println(c + " "+num);
    }
}
class Sample
{
    public static void main(String args[])
    {
        DisplayOverloading obj = new DisplayOverloading();
        obj.disp('a');
        obj.disp('a',10);
    }
}
```

Classe & Objet

Objet « this » ou autoréférence

Le mot-clé « *this* » désigne, dans une classe, l'instance courante de la classe elle-même.

EXEMPLE

```
class Account{  
    int a;  
    int b;  
    public void setData(int a , int b){  
        this.a=a;  
        this.b=b;  
    }  
    public static void main(string args[]){  
        Account obj = new Account();  
    }  
}
```

```
    public void setData(int c , int d){  
        this.a=c;  
        this.b=d;  
    }  
    public static void main(string args[]){  
        Account object1 = new Account();  
        object1.setData(2,3);  
        Account object2 = new Account();  
        object2.setData(4,3);  
    }  
}
```

Classe & Objet

Objet « this » ou autoréférence

```
// Java code for using this() to
// invoke current class constructor
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        this(10, 20);
        System.out.println("Inside default constructor \n");
    }

    //Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
        System.out.println("Inside parameterized constructor");
    }

    public static void main(String[] args)
    {
        Test object = new Test();
    }
}
```

```
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    //Method that returns current class instance
    Test get()
    {
        return this;
    }

    //Displaying value of variables a and b
    void display()
    {
        System.out.println("a = " + a + " b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.get().display();
    }
}
```

Travaux dirigés





ORACLE®



POO en Java : Partie II

1- UML

2- Classe & Objet

3- Héritage

4- Exceptions

5-Collection

Héritage

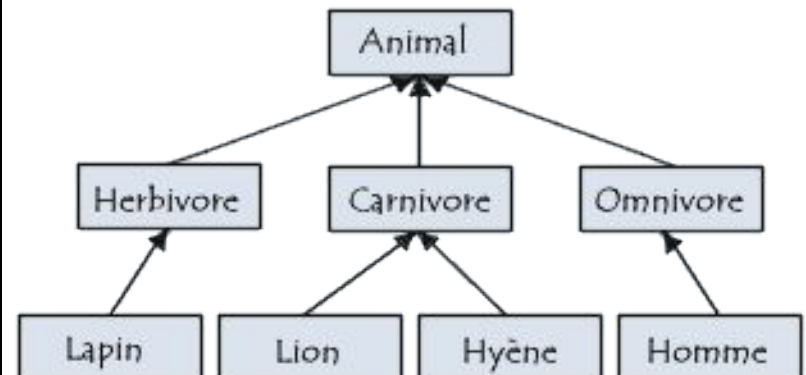
Définitions

❓ Une classe peut hériter d'une autre classe ce qui amélioration de la productivité : réutilisation du code, ajout des nouveaux champs/méthodes aux champs/méthodes déjà existant(e)s...

□ La classe qui hérite d'une classe de base, est appelé : **Classe fille ou dérivée ou sous-classe.**

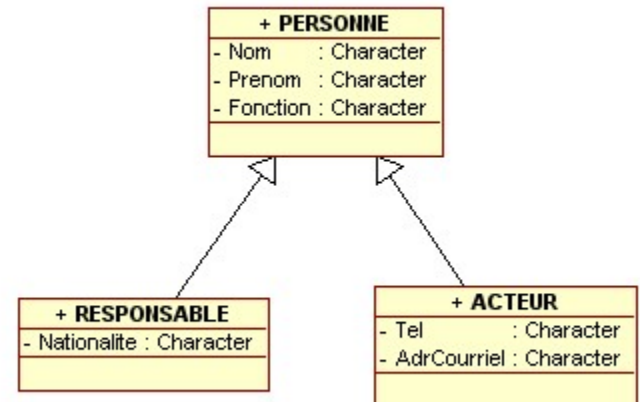
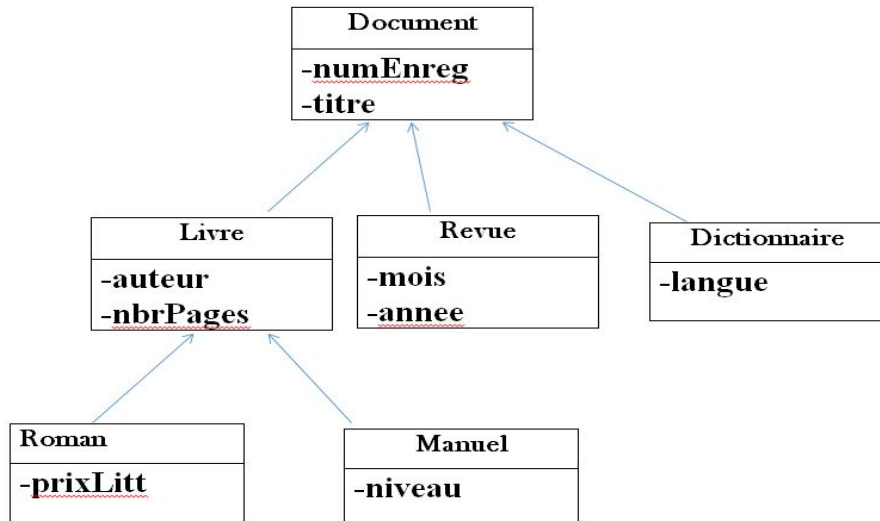
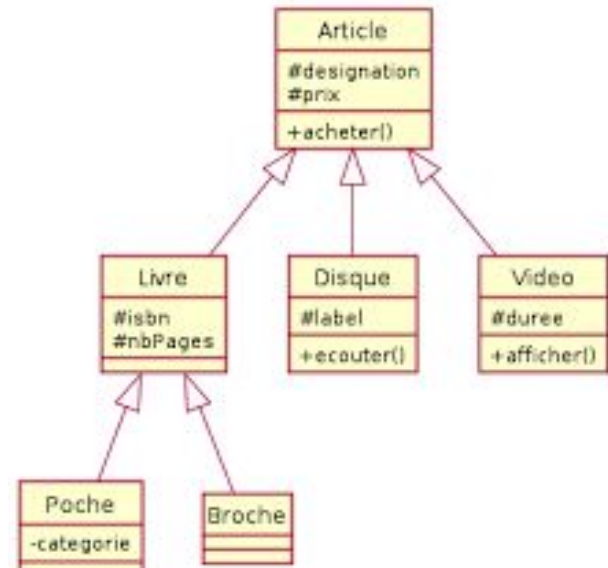
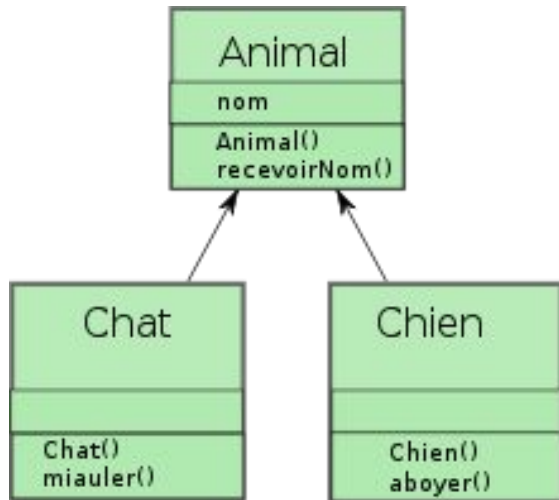
□ La classe de base est appelée, **classe mère ou super-classe**

- Classe de base A
- Classe dérivée B
 - Classe dérivée D
 - Classe dérivée E
- Classe dérivée C
 - Classe dérivée E



Héritage

Exemples



Héritage

MISE EN ŒUVRE EN JAVA

- ❓ Le but étant **d'étendre** (*extends*) les fonctionnalités de la classe de base, en créant des classe dérivées qui héritent les mêmes fonctionnalités de la classe mère, en y ajoutant d'autres.
- ❓ **En java** pour indiquer qu'une **classe fille** (dérivée) **hérite** d'une classe mère (de base), il suffit d'ajouter le mot clés : ***extends*** <classe de base>, lors de la création de la classe dérivée.

SYNTAXE

```
Class <classe dérivée> extends <classe de base>
```

```
{
```

```
....
```

```
}
```

Héritage

Exemples

```
// classe de base
class Point
{ public void initialise (int abs, int ord)
  { x = abs ; y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}

// classe derivee de Point
class Pointcol extends Point
{ public void colore (byte couleur)
  { this.couleur = couleur ;
  }
  private byte couleur ;
}

// classe utilisant Pointcol
public class TstPcoll
{ public static void main (String args[])
  { Pointcol pc = new Pointcol() ;
    pc.affiche() ;
    pc.initialise (3, 5) ;
    pc.colore ((byte)3) ;
    pc.affiche() ;
```

```
    pc.deplace (2, -1) ;
    pc.affiche() ;
    Point p = new Point() ; p.initialise (6, 9) ;
    p.affiche() ;
  }
}

Je suis en 0 0
Je suis en 3 5
Je suis en 5 4
Je suis en 6 9
```

Héritage

DROITS D'ACCES

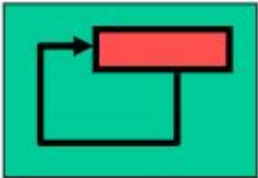
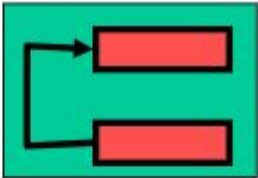
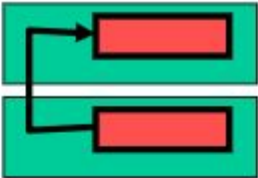








- un membre **public** est accessible à toutes les classes
- un membre **«rien» (friendly)** est accessible à toutes les classes du même paquetage,
- un membre **private** n'est accessible qu'aux fonctions membre de la classe.

Dans le cas de l'héritage : Si les membres de la classe de base sont :

- **public ou « rien »** : les membres de la classe dérivée auront accès à ces membres.
- **private** : les membres de la classe dérivée n'auront pas accès
- **protected**. Un membre (méthode ou champs) de la classe de base déclaré **protected** est accessible à ses classes dérivées ainsi qu'aux classes du même paquetage.

Héritage

DROITS D'ACCES

			
public			
protected			
private			
<i>friendly</i> (par défaut)			

Héritage

Exemples

```
class Point
{ public void initialise (int abs, int ord)
  { x = abs ; y = ord ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}

class Pointcol extends Point
{ public void colore (byte couleur)
  { this.couleur = couleur ;
  }
  public void affichec ()
  { affiche() ;
    System.out.println (" et ma couleur est : " + couleur) ;
  }
  public void initialisec (int x, int y, byte couleur)
  { initialise (x, y) ;
    this.couleur = couleur ;
  }
  private byte couleur ;
}
```

```
public class TstPcol2
{ public static void main (String args[])
  { Pointcol pc1 = new Pointcol() ;
    pc1.initialise (3, 5) ;
    pc1.colore ((byte)3) ;
    pc1.affiche() ; // attention, ici affiche
    pc1.affichec() ; // et ici affichec
    Pointcol pc2 = new Pointcol() ;
    pc2.initialisec(5, 8, (byte)2) ;
    pc2.affichec() ;
    pc2.deplace (1, -3) ;
    pc2.affichec() ;
  }
}
```

```
Je suis en 3 5
Je suis en 3 5
    et ma couleur est : 3
Je suis en 5 8
    et ma couleur est : 2
Je suis en 6 5
    et ma couleur est : 2
```

CONSTRUCTEURS

- Un constructeur est une méthode permettant d'instancier un objet
- La classe dérivée **DOIT** prendre en charge la construction de l'objet de la classe de base
- *Exemple : pour construire un avion, il faut d'abord construire un véhicule*
- Donc, le constructeur de la classe de base est d'abord appelé **AVANT** le constructeur de la classe dérivée. C'est la première instruction du constructeur de la classe dérivée, et on utilise le mot clés **super** pour le désigné

Héritage

Exemples

```
class A {  
  
    int i;  
  
    // Constructeur de la classe A  
    public A(int x) {  
        i = x;  
    }  
}  
  
// Héritage  
class B extends A {  
    double z;  
  
    // Constructeur de la classe B  
    public B(int f, double w) {  
  
        // Appel explicite du constructeur de A,  
        // et en 1ère ligne.  
        super(f);  
  
        z = w;  
    }  
}
```

REMARQUE

Si on utilise juste le mot clés : **super** **()** « sans argument » au niveau du constructeur de la classe dérivé ? il y aura une erreur de compilation, vu que la classe de base ne dispose pas d'un constructeur sans argument.

Héritage

Exemples

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }

  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}

class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;           // obligatoirement comme première
    this.couleur = couleur ;
  }
  public void affichec ()
  { affiche() ;
    System.out.println (" et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}
```

```
public class TstPool3
{ public static void main (String args[])
  { Pointcol pc1 = new Pointcol(3, 5, (byte)3) ;
    pc1.affiche() ; // attention, ici affiche
    pc1.affichec() ; // et ici affichec

    Pointcol pc2 = new Pointcol(5, 8, (byte)2) ;
    pc2.affichec() ;
    pc2.deplace (1, -3) ;
    pc2.affichec() ;
  }
}
```

```
Je suis en 3 5
Je suis en 3 5
    et ma couleur est : 3
Je suis en 5 8
    et ma couleur est : 2
Je suis en 6 5
    et ma couleur est : 2
```

Héritage

CONSTRUCTEURS

- ❑ **Cas 1** : La classe de base et la classe dérivée ont au moins un constructeur public, c'est le cas général. De ce fait, le constructeur de la classe dérivée **doit appeler** le constructeur de la classe de base disponible.
- ❑ **Cas 2** : La classe de base n'a aucun constructeur. La classe dérivée **peut ne pas appeler** explicitement le constructeur de la classe de base. Si elle le fait, elle ne peut appeler que le constructeur par défaut, vu que c'est le seul qui est disponible dans ce cas dans la classe de base.
- ❑ **Cas 3**: La classe dérivée ne possède aucun constructeur. Dans ce cas, la classe de base **doit avoir** un constructeur public sans argument.

Héritage

PHASE D'INITIALISATION

Les étapes par la quelle l'objet passe lors de l'instanciation. Soit que l'objet est de type classe de base ou de la classe dérivée.

Un objet de la classe de base A	Un objet b de la classe dérivée B
allocation mémoire pour un objet du type A	allocation mémoire pour un objet du type B (donc A+B)
initialisation par défaut des champs	initialisation par défaut des champs (A+B)
initialisation explicite des champs	initialisation explicite des champs hérités de A
exécution des instructions du constructeur de A	exécution des instructions du constructeur de A
	initialisation explicite des champs hérités de B
	exécution des instructions du constructeur de B

Héritage

Redéfinition & surdéfinition

Définition : La redéfinition ou la surdéfinition permet à la classe dérivée de conserver le même nom de méthode héritée de la classe de base, tout en modifiant son traitement.

L'intérêt : permettre de créer de nouvelle fonctionnalité pour la classe dérivée, à la base des fonctionnalités existantes chez la classe mère.

□ **Redéfinition** : quand on conserve la même signature (nom + paramètres + type de valeur de retour).

□ **Surdéfinition** : quand on conserve le même nom, mais la signature change (les paramètres et/ou le type de valeur de retour).

□ Il est impossible de redéfinir/surdéfinir les méthodes statiques

Héritage

Redéfinition & surdéfinition

Contraintes sur la redéfinition

a- signature identique :

```
class A {  
    public void unefonction(int x) { //etc.}  
}  
class B extends A {  
    public void unefonction(int z) { //etc.}  
}
```

The diagram illustrates the constraint of identical signatures. It shows two classes: `class A` and `class B extends A`. In `class A`, there is a method `public void unefonction(int x) { //etc.}`. In `class B`, there is a method `public void unefonction(int z) { //etc.}`. Arrows point from the method signature in `class A` to the corresponding method signature in `class B`, indicating that the signature must be identical for inheritance.

b- droits d'accès : la redéfinition ne doit pas diminuer les droits d'accès à une méthode.

```
class A {  
    private void unefonction(int x) { //etc.}  
}  
class B extends A {  
    public void unefonction(int z) { //etc.}  
}
```

The diagram illustrates the constraint of access rights. It shows two classes: `class A` and `class B extends A`. In `class A`, there is a method `private void unefonction(int x) { //etc.}`. In `class B`, there is a method `public void unefonction(int z) { //etc.}`. The `public` keyword is crossed out with a large 'X', indicating that the access rights must not be decreased when redefining a method.

Héritage

Exemple de surdéfinition

```
class A
{ public void f (int n) { ..... }
  .....
}
class B extends A
{ public void f (float x) { ..... }
  .....
}
A a ; B b ;
int n ; float x ;
.....
a.f(n) ; // appelle f (int) de A
a.f(x) ; // erreur de compilation : une seule méthode acceptable (f(int) de A
        // et on ne peut pas convertir x de float en int
b.f(n) ; // appelle f (int) de A
b.f(x) ; // appelle f(float) de B
```


Héritage

Exemple de redéfinition

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }

  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}

class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;           // obligatoirement comme première instruction
    this.couleur = couleur ;
  }
  public void affiche ()
  { super.affiche() ;
    System.out.println ("  et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}
```

```
public class TstPcol4
{ public static void main (String args[])
  { Pointcol pc = new Pointcol(3, 5, (byte)3) ;
    pc.affiche() ; // ici, il s'agit de affiche de Pointcol
    pc.deplace (1, -3) ;
    pc.affiche() ;
  }
}

Je suis en 3 5
  et ma couleur est : 3
Je suis en 4 2
  et ma couleur est : 3
```

Héritage

DUPLICATION DES CHAMPS

Il est possible de définir des champs dans la classe dérivée, portant le même nom des champs de la classe mère.
C'est ce qu'on appelle la duplication des champs (pas une redéfinition).

EXEMPLE

```
class A
{ public int n= 4 ; }
class B extends A
{ public float n = 4.5f ; }
public class DupChamp
{ public static void main(String[] args)
  { A a = new A() ; B b = new B() ;
    System.out.println ("a.n = " + a.n) ;
    System.out.println ("b.n = " + b.n) ;
  }
}
```

```
a.n = 4
b.n = 4.5
```

REMARQUE : Pour faire appel au champs de la classe mère on peut utiliser **le mot clés : super ou en instanciant un objet de type classe de mère.**

Exemple :

super.n = 3 ? désigne le champs n de la classe de base

Héritage

POLYMORPHISME

□ Polymorphisme 'تعدد الأشكال' , consiste à fournir une interface unique (message) à des entités (objets) pouvant avoir différents types (retour).

□ C'est la faculté des objets différents à réagir différemment en réponse au même message.

EXEMPLE

□ Marcher sur la queue d'un chat => il miaule

□ Marcher sur la queue d'un chien => il aboie

- Animal
- Chat
- Chien



Héritage

Exemple 1 polymorphisme

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void deplace (int dx, int dy)
  { x += dx ; y += dy ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ; // obligatoirement comme première instruction
    this.couleur = couleur ;
  }
  public void affiche ()
  { super.affiche() ;
    System.out.println (" et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}

public class Poly
{ public static void main (String args[])
  { Point p = new Point (3, 5) ;
    p.affiche() ; // appelle affiche de Point
    Pointcol pc = new Pointcol (4, 8, (byte)2) ;
    p = pc ; // p de type Point, reference un objet de type Pointcol
    p.affiche() ; // on appelle affiche de Pointcol
    p = new Point (5, 7) ; // p reference a nouveau un objet de type Point
    p.affiche() ; // on appelle affiche de Point
  }
}
```

```
Je suis en 3 5
Je suis en 4 8
    et ma couleur est : 2
Je suis en 5 7
```

Héritage

Exemple 2 polymorphisme

```
class Point
{ public Point (int x, int y)
  { this.x = x ; this.y = y ;
  }
  public void affiche ()
  { System.out.println ("Je suis en " + x + " " + y) ;
  }
  private int x, y ;
}

class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;           // obligatoirement comme première instruction
    this.couleur = couleur ;
  }

  public void affiche ()
  { super.affiche() ;
    System.out.println (" et ma couleur est : " + couleur) ;
  }
  private byte couleur ;
}

public class TabHeter
{ public static void main (String args[])
  { Point [] tabPts = new Point [4] ;
    tabPts [0] = new Point (0, 2) ;
    tabPts [1] = new Pointcol (1, 5, (byte)3) ;
    tabPts [2] = new Pointcol (2, 8, (byte)9) ;
    tabPts [3] = new Point (1, 2) ;
    for (int i=0 ; i< tabPts.length ; i++) tabPts[i].affiche() ;
  }
}

Je suis en 0 2
Je suis en 1 5
  et ma couleur est : 3
Je suis en 2 8
  et ma couleur est : 9
Je suis en 1 2
```

Héritage

Exemple 3 polymorphisme

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

  public void affiche ()
  { identifie() ;
    System.out.println (" Mes coordonnees sont : " + x + " " + y) ;
  }
  public void identifie ()
  { System.out.println ("Je suis un point ") ;
  }
  private int x, y ;
}
class Pointcol extends Point
{ public Pointcol (int x, int y, byte couleur)
  { super (x, y) ;
    this.couleur = couleur ;
  }
  public void identifie ()
  { System.out.println ("Je suis un point colore de couleur " + couleur) ;
  }
  private byte couleur ;
}
public class TabHet2
{ public static void main (String args[])
  { Point [] tabPts = new Point [4] ;
    tabPts [0] = new Point (0, 2) ;
    tabPts [1] = new Pointcol (1, 5, (byte)3) ;
    tabPts [2] = new Pointcol (2, 8, (byte)9) ;
    tabPts [3] = new Point (1, 2) ;
    for (int i=0 ; i< tabPts.length ; i++)
      tabPts[i].affiche() ;
  }
}
```

```
Je suis un point
  Mes coordonnees sont : 0 2
Je suis un point colore de couleur 3
  Mes coordonnees sont : 1 5
Je suis un point colore de couleur 9
  Mes coordonnees sont : 2 8
Je suis un point
  Mes coordonnees sont : 1 2
```

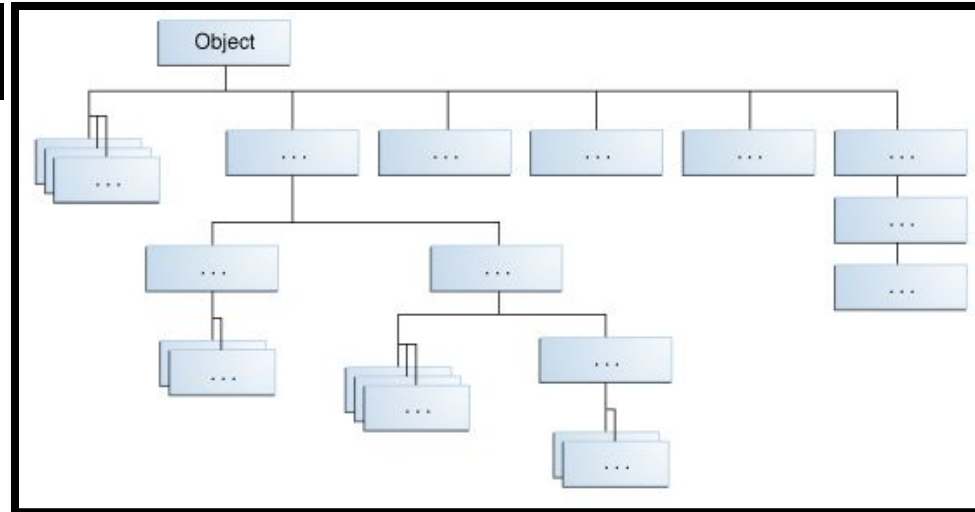
Héritage

SUPER-CLASSE OBJECT

- En Java, toutes les classes dérivent de la super classe « Object ».
- On peut définir un objet de type « Object », qui en fera référence à un objet de type une classe simple ? faire appel via l'objet « Object » à toutes les méthodes de la classe dérivée

EXEMPLE

```
Point p = new Point (...);  
Object o ;  
.....  
o = p ;  
o.deplace() ;           // erreur de compilation  
((Point)o).deplace() ;  // OK en compilation (attention aux parenthèses)
```



Héritage

SUPER-CLASSE OBJECT

La classe « Object », dispose des méthodes (sous-classe) : toString et equals.

EXEMPLE

```
class Point
{ public Point(int abs, int ord)
  { x = abs ; y = ord ;
  }
  private int x, y ;
}

public class ToString1
{ public static void main (String args[])
  { Point a = new Point (1, 2) ;
    Point b = new Point (5, 6) ;
    System.out.println ("a = " + a.toString()) ;
    System.out.println ("b = " + b.toString()) ;
  }
}

a = Point@fcl7aedf
b = Point@fclbaedf
```

> le nom de la classe concernée,
> l'adresse de l'objet en hexadécimal (précédée de @).

Héritage

CHAMPS, MÉTHODES ET CLASSES FINALE

En Java: on peut définir un champ, méthode et classe final, avec le mot clés « ***final*** »

☐ On définit un champs constant, avec le mot clés : FINAL, précédant le type du champs

☐ Une méthode finale n'est pas redéfini

☐ Une classe finale n'est pas dérivée

SYNTAXE

```
public final class MaClass {  
    public final double PI = 3.14159; // Impossible de modifier  
    public final double[] tailles = {50.2, 60.8};  
  
    public void uneMethode() {  
        tailles[0] = 99; // tailles est une référence  
    }  
}
```

CHAMPS FINALE

METHODE FINALE

```
public class MaClass {  
    public final void uneMethode() {  
        ...  
    }  
}
```

```
public final class ClasseFinal {  
    ...  
}
```

CLASSE FINALE

Héritage

Exemple « *final* »

```
class Base {
    int a;
    public Base(int i) {
        a = i;
    }
    public Base() {
        a = 1000;
    }
    final public void Show() {
        System.out.println(a);
    }
}
class D extends Base {
    public void Show() {
        System.out.println(super.a);
    }
}
class Abstract {
    public static void main(String argv[]) {
        new Base(50);
    }
}

#javac Abstract.java
Abstract.java:14: Final methods can't be overridden. Method void Show() is final in class Base.
    public void Show() {
                ^
```

Héritage

CLASSES ABSTRAITES

Une classe abstraite est une classe modèle qui ne permet pas d'instancier des objets.

❓ L'intérêt d'une classe abstraite, qu'il est juste utilisé pour l'héritage

❓ Une classe abstraite contient des méthodes et champs comme toute classe simple.

❑ Une classe abstraite peut contenir **une méthode abstraite** (on définit juste le prototype, ce type de méthode est public). Toute classe contenant *une méthode abstraite* est ***une classe abstraite***.

SYNTAXE

```
abstract class A
{
....
}
```

```
class A
{ .... abstract public type
methode(type ...);
.....}
```

Héritage

Exemple

```
abstract class Affichable
{ abstract public void affiche() ;
}

class Entier extends Affichable
{ public Entier (int n)
  { valeur = n ;
  }
  public void affiche()
  { System.out.println ("Je suis un entier de valeur " + valeur) ;
  }
  private int valeur ;
}

class Flottant extends Affichable
{ public Flottant (float x)
  { valeur = x ;
  }
  public void affiche()
  { System.out.println ("Je suis un flottant de valeur " + valeur) ;
  }
  private float valeur ;
}

public class Tabhet3
{ public static void main (String[] args)
  { Affichable [] tab ;
    tab = new Affichable [3] ;
    tab [0] = new Entier (25) ;
    tab [1] = new Flottant (1.25f) ; ;
    tab [2] = new Entier (42) ;
    int i ;
    for (i=0 ; i<3 ; i++)
      tab[i].affiche() ;
  }
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

Héritage

Quelques règles

1. Dès qu'une classe comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot clé *abstract* devant sa déclaration (ce qui reste quand même vivement conseillé). Ceci est correct :

```
class A
{ public abstract void f() ;    // OK
  .....
}
```

Malgré tout, *A* est considérée comme abstraite et une expression telle que *new A(...)* sera rejetée.

2. Une méthode abstraite doit obligatoirement être déclarée *public*, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.
3. Dans l'en-tête d'une méthode déclarée abstraite, les noms d'arguments muets doivent figurer (bien qu'ils ne servent à rien) :

```
abstract class A
{ public abstract void g(int) ; // erreur : nom d'argument (fictif) obligatoire
}
```

4. Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base (elle peut même n'en redéfinir aucune). Dans ce cas, elle reste simplement abstraite (bien que ce ne soit pas obligatoire, il est conseillé de mentionner *abstract* dans sa déclaration) :

```
abstract class A
{ public abstract void f1() ;
  public abstract void f2 (char c) ;
  .....
}
```

Source :

CLAUDE DELANNOY

Programmer
en
Java

Héritage

INTERFACE

Une interface est une classe abstraite n'implémentant aucune méthode, ni champs. Uniquement des constantes et méthodes abstraites.

☐ L'intérêt d'une interface, qu'elle oblige la redéfinition des méthodes abstraites par les classes dérivées.

☐ Une classe peut implémenter (au lieu d'hériter) une ou plusieurs interfaces

SYNTAXE

```
public/<droitpaquetage> interface I
{
    ....
    int methode (...);
}
```

Héritage

Exemple

```
interface Affichable
{ void affiche() ;
}
class Entier implements Affichable
{ public Entier (int n)
  { valeur = n ;
  }
  public void affiche()
  { System.out.println ("Je suis un entier de valeur " + valeur) ;
  }
  private int valeur ;
}
class Flottant implements Affichable
{ public Flottant (float x)
  { valeur = x ;
  }
  public void affiche()
  { System.out.println ("Je suis un flottant de valeur " + valeur) ;
  }
  private float valeur ;
}
public class Tabhet4
{ public static void main (String[] args)
  { Affichable [] tab ;
    tab = new Affichable [3] ;
    tab [0] = new Entier (25) ;
    tab [1] = new Flottant (1.25f) ; ;
    tab [2] = new Entier (42) ;
    int i ;
    for (i=0 ; i<3 ; i++)
      tab[i].affiche() ;
  }
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

RQ : une classe peut implémenter plusieurs interfaces :
Class A implements I, II....



ORACLE®



POO en Java : Partie II

- 1- UML
- 2- Classe & Objet
- 3- Héritage
- 4- Exceptions**
- 5-Collection

Exception

DEFINITION

❓ Une **exception** est :

- Une dérogation à une règle générale.
- Un événement informatique, si non gérée, provoquerai une anomalie et/ou un arrêt de l'exécution **normale** du programme.
- Une exception devrait être gérer



EXEMPLE

- Erreur de manipulation des pointeurs
- La division par zéro
- Echec de lecture/ecriture
- Echec de Ouverture/lecture/ecriture des fichiers
- Dépassement de la taille maximum d'un tableau

....

Exception

GESTION DES EXCEPTIONS

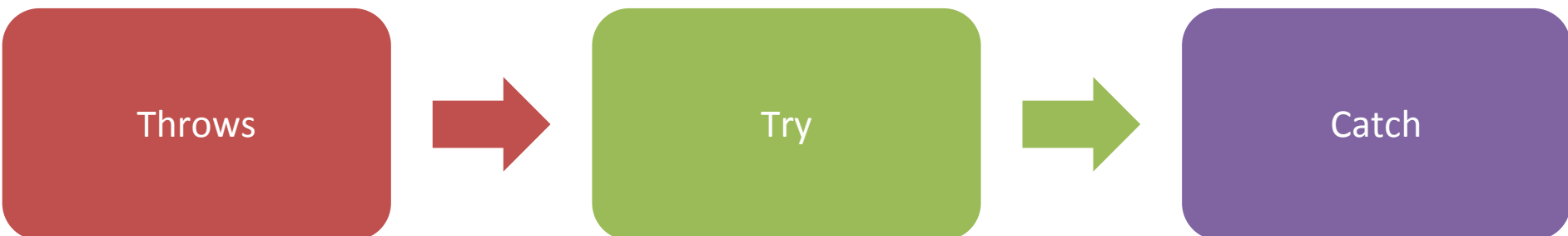
- Pour le bon fonctionnement du programme, les exceptions devraient être gérées ? Système de Gestion des Exceptions
- Exception est un événement susceptible de se produire lors d'un traitement, c'est-à-dire lors d'une méthode.
- On doit d'abord prévoir les méthodes qui peuvent **générer** une exception, puis **en cas** de production de l'événement, on devrait se **brancher** sur le traitement prévu de l'exception.



Exception

GESTION DES EXCEPTIONS

- En Java, une exception est une classe
- En Java, la gestion des exceptions se fait par les mots clés et bloc :
 - « throws » : pour identifier les méthodes susceptible de générer une exception et aussi les cas pour générer une exception
 - « try » : qui permet d'entourer le bloc de traitement utilisant une des méthodes identifiées comme génératrice d'exception.
 - « catch » : qui permet d'être à l'écoute pour qu'en cas de production de l'exception, on exécute le traitement prévu



Exception

Exemple

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0) ) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}

class ErrConst extends Exception
{ }
```

1

2

3

1 ? Identification de la méthode susceptible de provoquer une exception

Le mot clés « throws » est positionné à l'entête de la méthode et il renvoie vers un objet de type « Exception ».

2 ? Le cas où l'exception est généré, en envoyant un objet de type « Exception » avec le mot clés « throw »

3 ? La classe dérivée de l'Exception

Exception

Exemple

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
class ErrConst extends Exception
{ }
public class Except1
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println ("Erreur construction ") ;
      System.exit (-1) ;
    }
  }
}
```

coordonnees : 1 4
Erreur construction

Exception

GESTION DES EXCEPTIONS

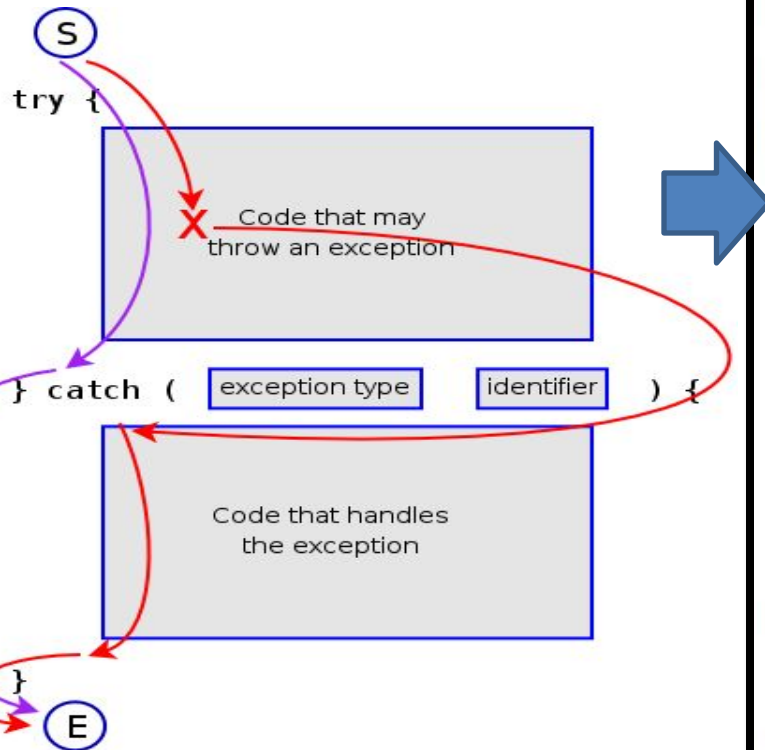
Throws



Try



Catch



```
class Point
{ public Point(int x, int y) throws ErrConst
{ if ( (x<0) || (y<0)) throw new ErrConst() ;
  this.x = x ; this.y = y ;
}
public void affiche()
{ System.out.println ("coordonnees : " + x + " " + y) ;
}
private int x, y ;
}

class ErrConst extends Exception
{ }

public class Except1
{ public static void main (String args[])
{ try
{ Point a = new Point (1, 4) ;
  a.affiche() ;
  a = new Point (-3, 5) ;
  a.affiche() ;
}
catch (ErrConst e)
{ System.out.println ("Erreur construction ") ;
  System.exit (-1) ;
}
}
}

coordonnees : 1 4
Erreur construction
```

Exception

GESTION DE PLUSIEURS EXCEPTIONS

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst() ;
    this.x = x ; this.y = y ;
  }

  public void deplace (int dx, int dy) throws ErrDepl
  { if ( ((x+dx)<0) || ((y+dy)<0)) throw new ErrDepl() ;
    x += dx ; y += dy ;
  }

  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }

  private int x, y ;
}

class ErrConst extends Exception
{ }

class ErrDepl extends Exception
{ }
```

```
public class Except2
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a.deplace (-3, 5) ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println ("Erreur construction ") ;
      System.exit (-1) ;
    }
    catch (ErrDepl e)
    { System.out.println ("Erreur deplacement ") ;
      System.exit (-1) ;
    }
  }
}
```

```
coordonnees : 1 4
Erreur deplacement
```


Exception

TRANSMISSION D'INFORMATION AU GESTIONNAIRE D'EXCEPTION

On peut transmettre une information au gestionnaire d'exception :

- par le biais de l'objet fourni dans l'instruction *throw*,
- par l'intermédiaire du constructeur de l'objet exception.

```
class ErrConst extends Exception
{ ErrConst (int abs, int ord)
  { this.abs = abs ; this.ord = ord ;
  }
  public int abs, ord ;
}

public class Exinfo1
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println ("Erreur construction Point") ;
      System.out.println (" coordonnees souhaitees : " + e.abs + " " + e.ord) ;
      System.exit (-1) ;
    }
  }
}
```

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0)) throw new ErrConst(x, y) ;
    this.x = x ; this.y = y ;
  }
  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}
```

```
coordonnees : 1 4
Erreur construction Point
  coordonnees souhaitees : -3 5
```


Exception

TRANSMISSION D'INFORMATION AU GESTIONNAIRE D'EXCEPTION

On peut transmettre une information au gestionnaire d'exception :

- par le biais de l'objet fourni dans l'instruction *throw*,
- par l'intermédiaire du constructeur de l'objet exception.

```
class Point
{ public Point(int x, int y) throws ErrConst
  { if ( (x<0) || (y<0))
    { throw new ErrConst("Erreur construction avec coordonnees " + x + " " + y) ;
      this.x = x ; this.y = y ;
    }
  }

  public void affiche()
  { System.out.println ("coordonnees : " + x + " " + y) ;
  }
  private int x, y ;
}

class ErrConst extends Exception
{ ErrConst (String mes)
  { super(mes) ;
  }
}
```

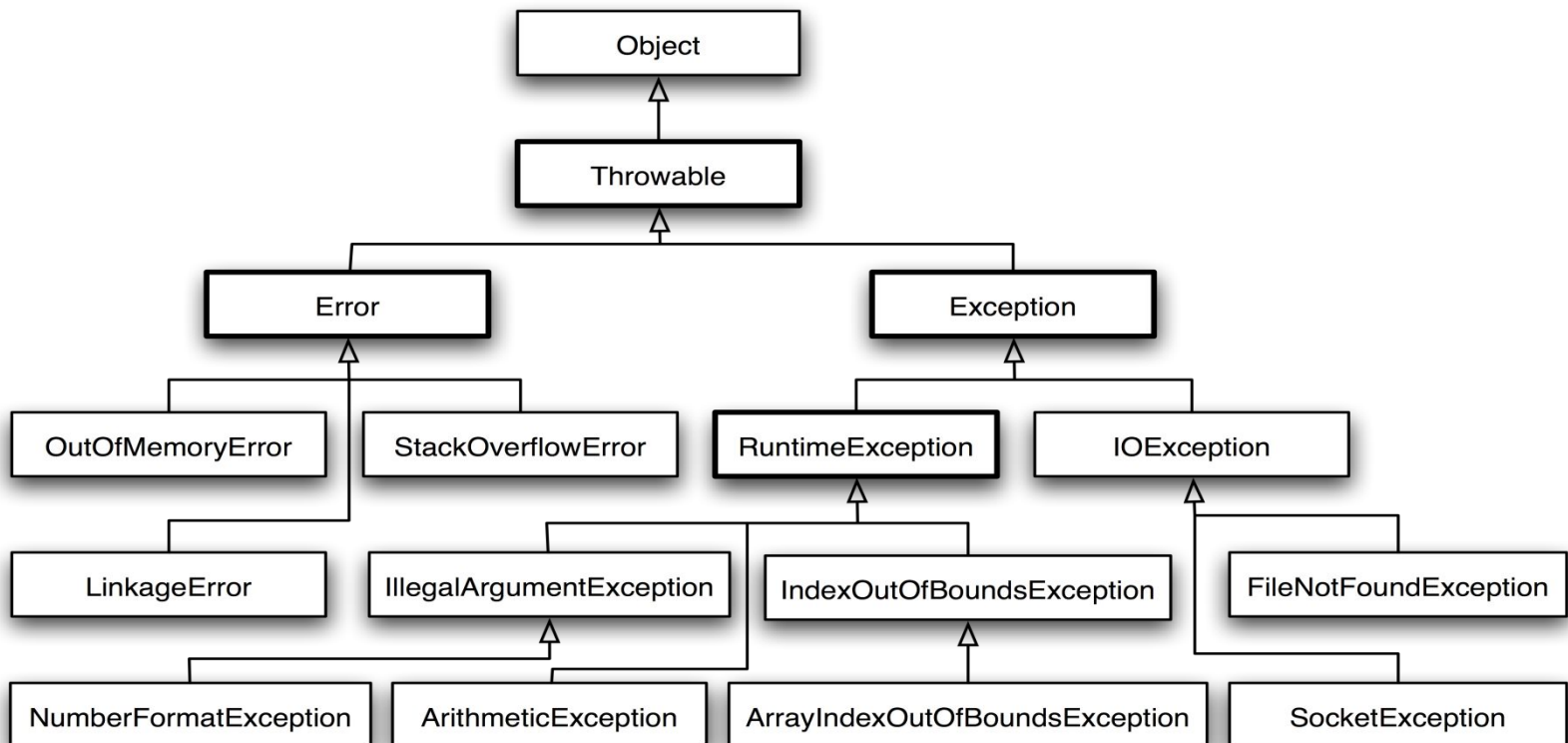
```
coordonnees : 1 4
Erreur construction avec coordonnees -3 5
```

```
public class Exinfo2
{ public static void main (String args[])
  { try
    { Point a = new Point (1, 4) ;
      a.affiche() ;
      a = new Point (-3, 5) ;
      a.affiche() ;
    }
    catch (ErrConst e)
    { System.out.println (e.getMessage()) ;
      System.exit (-1) ;
    }
  }
}
```

Exception

EXCEPTIONS STANDARDS

En Java, on dispose déjà d'une classe pour la gestion des exceptions, dont dérive d'autres classes et méthodes



Exception

Exemple

```
public class ExcStd1
{ public static void main (String args[])
{ try
{ int t[] ;
  System.out.print ("taille voulue : ") ;
  int n = Clavier.lireInt() ;
  t = new int[n] ;
  System.out.print ("indice : ") ;
  int i = Clavier.lireInt() ; t[i] = 12 ;
  System.out.println ("*** fin normale") ;
}

catch (NegativeArraySizeException e)
{ System.out.println ("Exception taille tableau negative : "
                    + e.getMessage() ) ;
}
catch (ArrayIndexOutOfBoundsException e)
{ System.out.println ("Exception indice tableau : " + e.getMessage() ) ;
}
}
}
```

```
taille voulue : -2
Exception taille tableau negative :

taille voulue : 10
indice : 15
Exception indice tableau : 15
```

Exception

Exemple

```
public class MultipleCatchBlock1 {  
  
    public static void main(String[] args) {  
  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
        }  
        catch(ArithmeticException e)  
        {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Exception

Exemple