

Java

Conception objet

Programmation orientée objet

- Une application est définie par des interactions entre objets
- Un objet modélise un concept, une entité du monde physique
 - `une Voiture`, `un Compte`, `une Facture`, ...
- Le modèle de tous les objets d'un type est la classe
 - les objets réels de type "voiture" (renault, citroën, ...) peuvent être modélisés par une classe `Voiture`

Programmation orientée objet

- Une classe est une structure informatique composée
 - de propriétés
 - ce sont les données de l'objet
 - couleur, marque, vitesse maximale, ...
 - de méthodes
 - accélérer, freiner, démarrer, ...

Programmation orientée objet

- Principe de l'encapsulation
 - les propriétés sont cachées (`private`)
 - seul l'objet auquel appartiennent les propriétés peut les modifier/lire directement
 - existence de méthodes spécifiques pour lire/modifier les propriétés depuis les autres classes
 - les setteurs/getteurs
 - `void setMarque(String marque)`
 - `String getMarque()`

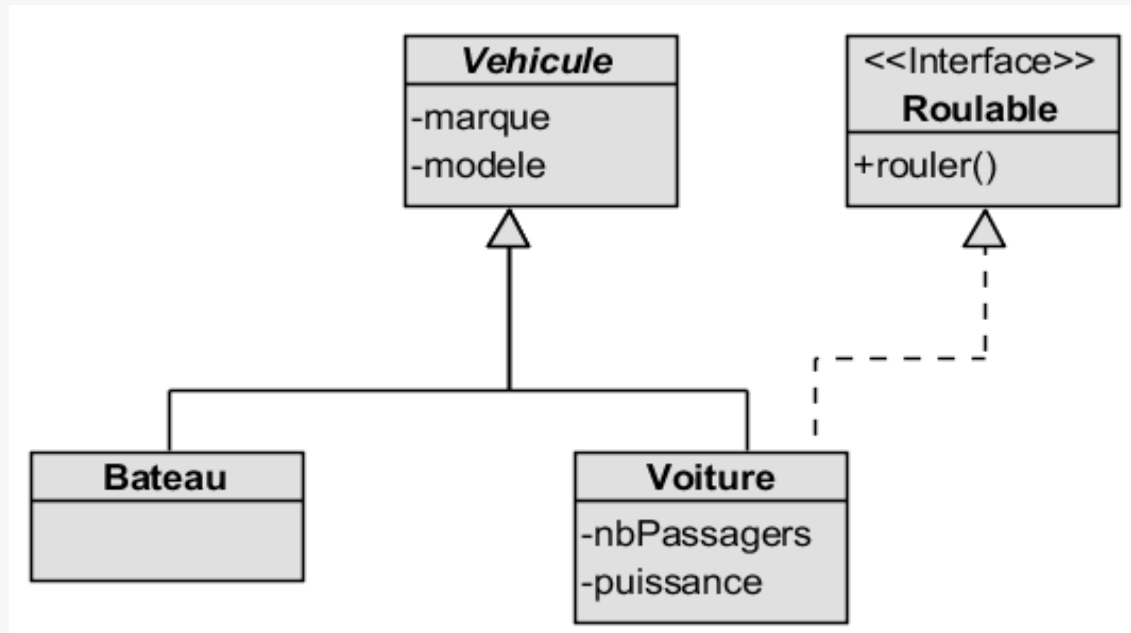
Programmation orientée objet

- Le typage
 - chaque objet est typé : `Voiture`, `Compte`, `Client`, ...
- Polymorphisme
 - un objet peut appartenir à plusieurs types différents
 - utilisation du sous-typage
 - les classes dérivées héritent de la structure des classes mères

Programmation orientée objet

- Signature d'une méthode
 - une méthode est caractérisée par
 - son type de retour
 - son nom
 - la liste des types de ses paramètres
 - la signature d'une méthode est constituée de
 - son nom
 - la liste des types de ses paramètres

Programmation orientée objet



Programmation orientée objet

- La redéfinition
 - une classe fille peut modifier le comportement d'une méthode héritée
- La définition
 - une classe fille implémente la comportement d'une méthode abstraite de sa classe mère
- La surcharge
 - une classe fille ajoute une nouvelle signature à une méthode

Classe

- La classe structure les applications Java
- Elle est le support de l'encapsulation
- Syntaxe de déclaration d'une classe
 - `modificateur class Nom [extends ...] [implements ...]`
 - les modificateurs, l'extension et l'implémentation seront abordés plus loin
- Il n'y a pas de règle à suivre pour l'ordre de déclaration des méthodes et des propriétés dans la classe
 - en général : propriétés suivies des méthodes

Instanciación

- L'opérateur `new` instancie une classe
 - l'objet est créé sur le tas
 - une exception `OutOfMemoryError` est levée si l'allocation mémoire n'est pas possible
 - la référence de l'objet est récupérée dans la variable

```
Contact c1 = new Contact("M", "LAGAFFE", "Gaston");
```

- il n'y a pas d'opérateur `delete`
 - le garbage collector libère la mémoire

Instanciación

- `this` représente l'objet en cours
 - `this` n'est pas utilisable dans les méthodes statiques
- `super` représente l'instance de la classe mère
 - `super()` appelle le constructeur par défaut de la classe mère
 - `super.toto()` appelle la méthode `toto()` sur la classe mère

Classe `Object`

- Superclasse de tous les objets
 - même les tableaux
- La classe `Object` possède des méthodes qui seront héritées par tous les objets
 - en fonction des besoins, certaines méthodes devront être redéfinies

Tableau

- Le tableau Java est un objet
 - possède une propriété `length`
- exemples de création et de parcours de tableaux

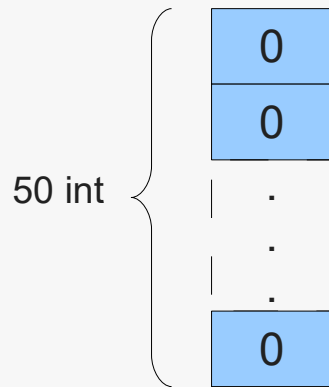
```
Contact c1 = new Contact("M", "LAGAFFE", "Gaston");
int[] ti1 = {1, 2, 3};
int[] ti2 = new int[50];
Contact[] tc1 = {new Contact(), c1, new Contact("M", "NAUDIN", "Fernand")};
Contact[] tc2 = new Contact[5];

for(int i=0 ; i< ti1.length ; i++)
    System.out.println(ti1[i]);
for(Contact c : tc1)
    System.out.println(c.getNom());
```

Tableau

- Initialisation des tableaux
 - à la création d'un tableau par `new`, chaque élément contient la valeur `0`, `false` ou `null`

```
int[] ti2 = new int[50];
```



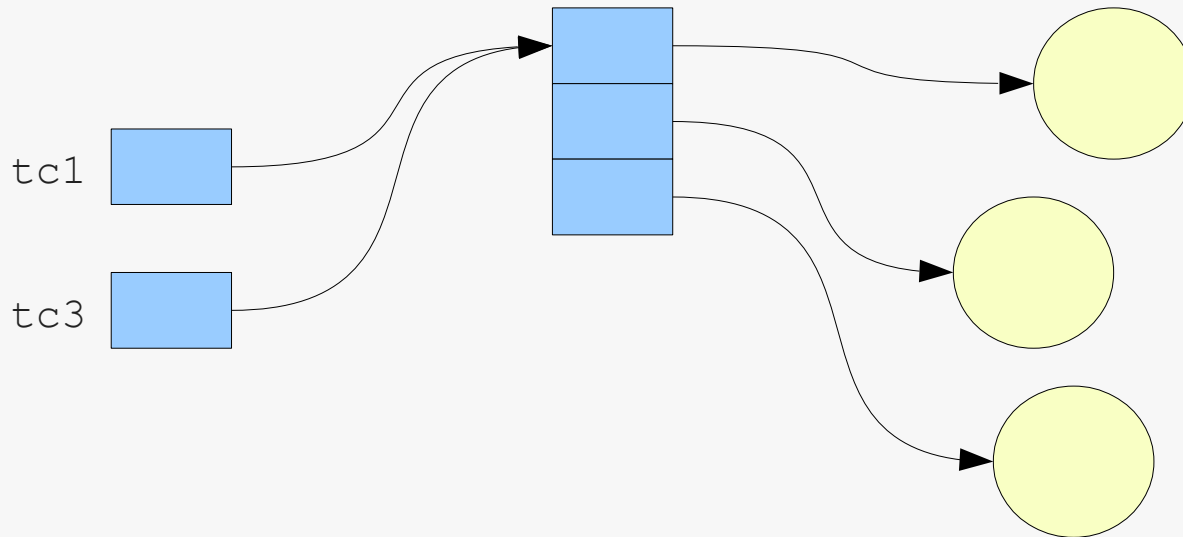
```
Contact[] tc2 = new Contact[5];
```



Tableau

- La copie des variables de types tableaux provoque la copie des adresses

```
Contact[] tc1 = {new Contact(), c1, new Contact("M", "NAUDIN", "Fernand")};  
Contact[] tc3 = tc1;
```



Quelques ajouts syntaxiques du JDK 1.7

- `String` dans le `switch`

```
String day = "Monday";  
switch (day){  
    case "Monday":  
        break;  
    ...  
}
```

- Littéraux entiers

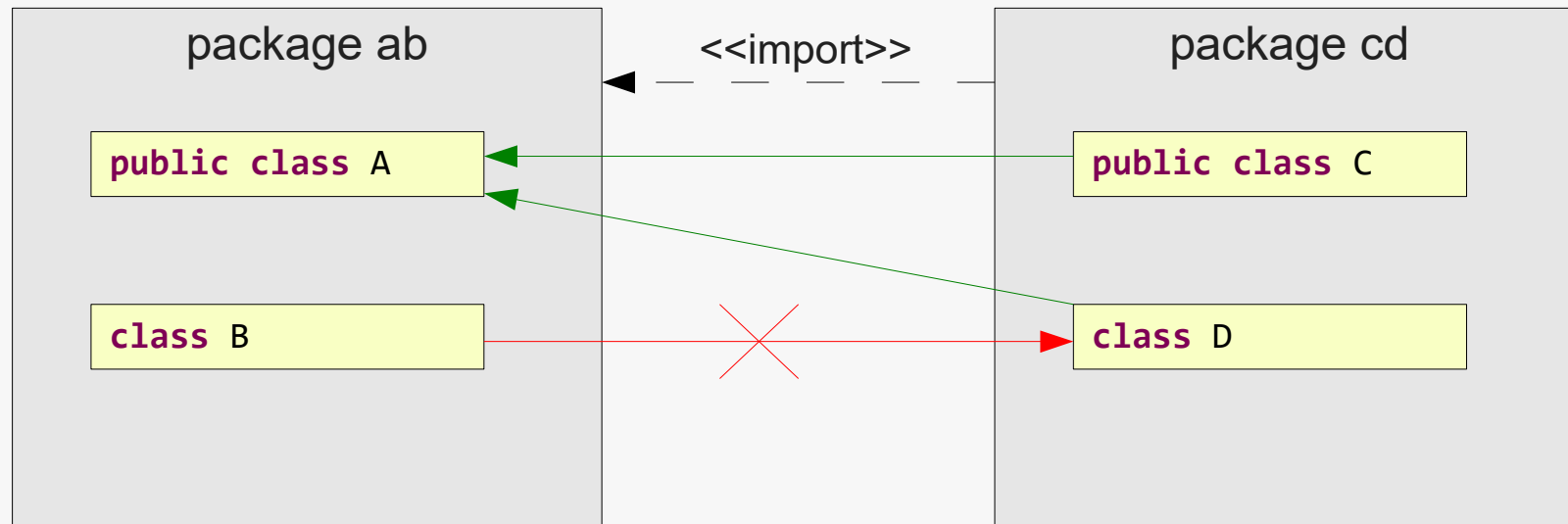
- ajout du format binaire
- utilisation de l'underscore comme séparateur

```
byte octet = 0b01011001;  
int mask = 0x01_FF_AE_01;  
long value = 1_000_000;
```


Déclaration des classes

- Une classe est déclarée dans un package
- La déclaration de la classe est accompagnée de modificateurs
 - `public` : la classe est utilisable dans tous les autres packages
 - si `public` n'est pas précisé, la classe est visible uniquement dans son package
 - `final` : la classe ne peut pas être dérivée
 - `abstract` : la classe ne peut pas être instanciée
 - `strictfp` : utilisation du mode IEEE pour les calculs réels

Visibilité des classes



Déclaration des classes

```
package org.antislashn.formation;
```

package de la classe

```
import java.util.GregorianCalendar;
```

déclaration des classes importées

```
public class Contact {  
    private String civilite;  
    private String nom;  
    private String prenom;  
    private GregorianCalendar dateNaissance;  
  
    public Contact() {}  
  
    public Contact(String civilite, String nom, String prenom) {  
        this.civilite = civilite;  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    public String getCivilite() {  
        return civilite;  
    }  
    public void setCivilite(String civilite) {  
        this.civilite = civilite;  
    }  
    ...  
}
```

Import

- L'utilisation d'une classe qui n'appartient pas au même package s'effectue par son nom complètement qualifié
- L'import permet de simplifier l'écriture
 - classe par classe

```
import java.util.GregorianCalendar;
```

- par package – pas d'import des sous-packages

```
import java.sql.*;
```

Import

- Cas des classes de mêmes noms mais de packages différents
 - le nom de la classe doit être complètement qualifié

```
private java.sql.Date dateSql;  
private java.util.Date date;
```

- Import statique – Ajout du JDK 1.5

```
import static java.lang.Math.PI ;  
...  
double r = PI ;
```

- l'import en masse est aussi possible

```
import static java.lang.Math.*;
```

Propriétés de classe

- Déclaration des propriétés

`[modificateur] type nom [= valeur] ;`

- modificateurs d'accès

- `public` : visible en dehors de la classe et du package
- `protected` : visible dans le package et les classes dérivées
- `private` : visible uniquement dans la classe
- aucun modificateur : visibilité de package

Propriétés de classe

- Autres modificateurs
 - `static` : propriété de classe
 - la propriété est partagée entre toutes les instances d'une même classe
 - `final` : propriété constante
 - la propriété peut-être initialisée lors de sa déclaration ou dans un constructeur
 - une propriété `final static` est utilisée comme constante partagée
 - `volatile` : force la lecture et écriture en mémoire
 - désactive les optimisations d'utilisation des registres

Propriétés de classe

- Autres modificateur
 - `transient`: interdit la sérialisation d'une propriété
 - la propriété est partagée entre toutes les instances d'une même classe

Propriétés de classe

- Exemple de déclaration de constantes
 - un `Enum` pourrait remplacer cette déclaration

```
public interface LecteurMedia {  
    public static final int STOP = 1;  
    public static final int START = 2;  
    public static final int FORWARD = 3;  
    public static final int BACKWARD = 4;  
}
```

Méthodes de classe

- Déclaration d'une méthode

```
[modificateur] type_retour nom(paramètres) ;
```

- Modificateurs

- modificateurs d'accès : `public`, `private`, `protected`, ou par défaut
 - comme les propriétés
- `static` : méthode de classe
 - peut être utilisée directement sur le nom de la classe
 - ne peut utiliser que des membres statiques
 - ne peut pas utiliser `this`
- `abstract` : méthode ne possédant pas de corps
 - la classe doit être marquée `abstract`

Méthodes de classe

- Autres modificateurs
 - `final` : interdit la redéfinition de la méthode
 - `strictfp` : utilisation du mode IEEE pour les calculs réels
 - `native` : méthode dont l'implémentation est codée en langage C/C++
 - `synchronize` : gestion des accès concurrents

Paramètres de méthode

- La syntaxe de passage des paramètres est classique

`[modificateur] type nom`

- `modificateur`
 - `final` : interdit la modification de la valeur
- Pas de valeur par défaut
- Nombre variable d'arguments

Paramètres de méthode

- Déclaration d'une méthode avec un nombre variable d'arguments

```
public int add(int...is){  
    int result = 0;  
    for(int i : is)  
        result += i;  
    return result;  
}
```

déclaration du nombre
variable de paramètres

- appel de la méthode

```
int[] integers = {1,2,3,4};  
  
calcul.add(1,2,3,4);  
calcul.add(integers);
```

Les constructeurs

- Les constructeurs répondent à la même logique qu'avec le langage C++
 - le constructeur ne renvoie rien
 - même pas `void`
 - le constructeur a le même nom que la classe
 - l'appel d'un constructeur de la classe mère est effectué par `super()`
 - avec paramètres pour appeler un constructeur autre que le constructeur par défaut
 - doit être la première instruction du constructeur de la classe fille

La destruction d'objet

- Pas de destructeur en Java
 - les objets sont libérés de la mémoire par le garbage collector
- méthode `finalize()`
 - appelée juste avant la destruction d'un objet par le garbage collector
 - un objet n'est réellement détruit que si la mémoire manque
 - l'appel de `finalize()` n'arrivera peut-être qu'en fin d'application
 - peut être forcé par un appel à `System.gc()`

Héritage

- En POO l'héritage permet de spécialiser une classe mère
 - la classe fille hérite de l'ensemble des membres de la classe mère
- Java ne supporte que le mono-héritage
 - contrairement à C++
- `extends` permet de spécialiser une classe mère

```
public class A {  
    ...  
}  
  
class B extends A{  
    ...  
}
```

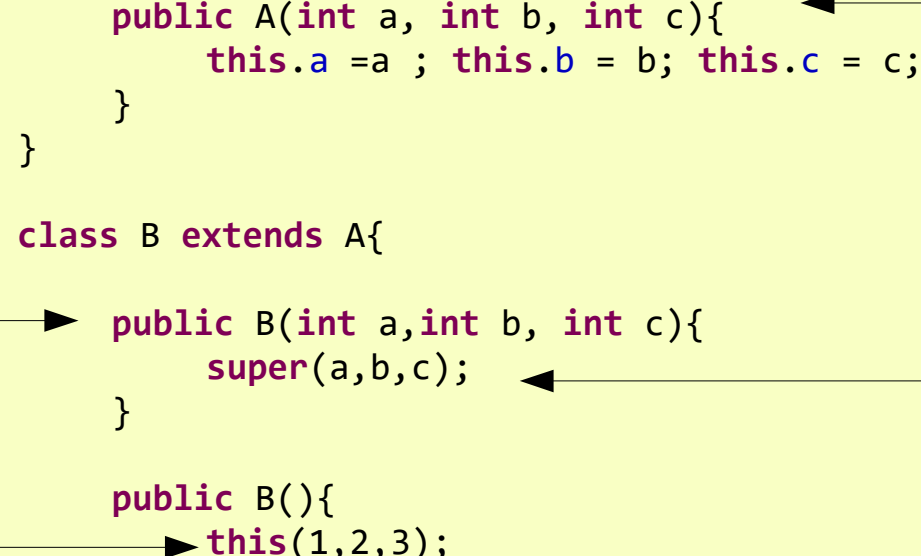

Appel des constructeurs

- Si aucun constructeur n'est codé dans une classe le compilateur fournit un code
- Si un constructeur est codé, le constructeur par défaut n'est plus fourni par le compilateur
- Pour construire une instance de classe le constructeur de la classe mère est appelé
 - le constructeur par défaut si rien n'est précisé
 - ou le constructeur précisé par `super (. . .)`

Appel des constructeurs

- Un constructeur d'une classe peut invoquer un constructeur de cette même classe par `this(...)`

```
public class A {  
    private int a,b,c;  
  
    public A(int a, int b, int c){  
        this.a =a ; this.b = b; this.c = c;  
    }  
}  
  
class B extends A{  
    public B(int a,int b, int c){  
        super(a,b,c);  
    }  
  
    public B(){  
        this(1,2,3);  
    }  
}
```



Empêcher l'héritage

- Le modificateur `final`
 - au niveau de la classe : permet d'interdire la spécialisation de la classe

```
final class D{}
```

- au niveau d'une méthode : permet d'interdire sa redéfinition au niveau de la classe fille
 - pas sa surcharge

```
public final void foo(){} 
```

Transtypage

- Permet de convertir un type de base en un autre

```
double x = 3.14;  
int i = (int) x;
```

- Le transtypage des instances d'objets est parfois nécessaire
 - framework renvoyant un objet
 - méthode de recherche d'instance

```
Voyage v = (Voyage) session.load(Voyage.class, id);
```

- version de Java ne supportant pas les types génériques (< JDK 1.5)

Transtypage

- Le transtypage peut aussi être utilisé pour passer d'une classe mère vers une classe fille
 - devrait être exceptionnel
 - casse l'abstraction
 - peut être sécurisé par `instanceof`

```
if(o instanceof B){  
    ...  
}
```

Classe abstraite

- Plus les classes sont hautes dans la hiérarchie des classes plus elles sont abstraites
 - mise en place du vocabulaire
 - modificateur `abstract` au niveau de la classe
- Une méthode abstraite n'a pas de corps de méthode
 - déclarée abstraite
 - une classe possédant au moins une méthode abstraite doit être déclarée abstraite

Classe abstraite

- La classe abstraite ne peut pas être instanciée
 - elle peut posséder des constructeurs
- Si une classe dérivée d'une classe abstraite ne définit pas les méthodes abstraites de sa classe mère elle doit être déclarée abstraite

Interface

- Java ne permet pas l'héritage multiple
- L'interface permet de contourner cette limite
 - pas de code
 - uniquement la déclaration des signatures de méthode
- Une classe peut implémenter plusieurs interfaces
 - le mot clé `extends` est réservé à l'héritage d'une classe et l'héritage entre interfaces
 - le mot clé `implements` est réservé à l'héritage des interfaces

Interface

- Une interface ne peut pas être instanciée
 - mais une variable peut être du type de l'interface

```
public interface Roulable {  
    public void rouler();  
}
```

```
public class Ballon implements Roulable {  
    @Override  
    public void rouler() {  
        System.out.println(">>>> "+this.getClass().getSimpleName()+" roule");  
    }  
}
```

```
public static void main(String[] args) {  
    Roulable r1 = new Ballon();  
    r1.rouler();  
}
```

Interface

- Les méthodes d'une interface sont automatiquement `public abstract`
- Les propriétés d'une interface sont automatiquement `public final`
- Si une interface comporte beaucoup de méthodes il peut être intéressant de mettre à disposition une classe abstraite d'adaptation
 - implémente les méthodes de l'interface sans comportement
 - cf. `java.awt.event.MouseAdapter`

Documentation

- La documentation des classes est indispensable
- Le JDK dispose d'une documentation complète
 - *<http://docs.oracle.com/javase/6/docs/api/>*

The screenshot displays the Java Platform Standard Ed. 6 API documentation. On the left, a sidebar lists packages and classes. A box labeled 'packages' points to the 'Packages' section, and a box labeled 'classes du package sélectionné' points to the list of classes. The main content area shows the 'Class String' documentation, with a box labeled 'documentation de la classe' pointing to the class name. The documentation includes the class hierarchy, implemented interfaces, and a code example.

packages

Java™ Platform
Standard Ed. 6

[All Classes](#)

Packages

- [java.applet](#)
- [java.awt](#)
- [java.awt.color](#)
- [java.awt.datatransfer](#)
- [java.awt.dnd](#)

classes du package sélectionné

[StatementFactory](#)

[Statement](#)

[Statement](#)

[StatementEvent](#)

[StatementEventListener](#)

[StAXResult](#)

[StAXSource](#)

[Streamable](#)

[StreamableValue](#)

[StreamCorruptedException](#)

[StreamFilter](#)

[StreamHandler](#)

[StreamPrintService](#)

[StreamPrintServiceFactory](#)

[StreamReaderDelegate](#)

Overview **Package** **Class** **Use** **Tree** **Deprecated** **Index** **Help**

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO F](#)

DETAIL: [FIELD](#) |

java.lang

Class String

[java.lang.Object](#)

↳ [java.lang.String](#)

All Implemented Interfaces:

[Serializable](#), [Comparable<String>](#), [CharSequence](#)

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The String class represents character strings. All string literals in Java programs, suc

Strings are constant; their values cannot be changed after they are created. String bu

they can be shared. For example:

```
String str = "abc";
```

documentation de la classe

Documentation

- Création d'une documentation
 - utiliser les commentaires de documentation
 - `/** ... */`
 - créer la documentation avec
 - javadoc : en ligne de commande
 - Eclipse : **Project** → **Generate Javadoc...**