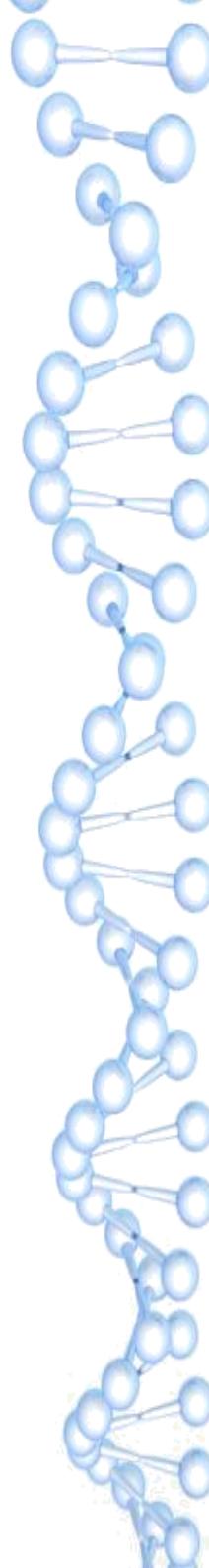
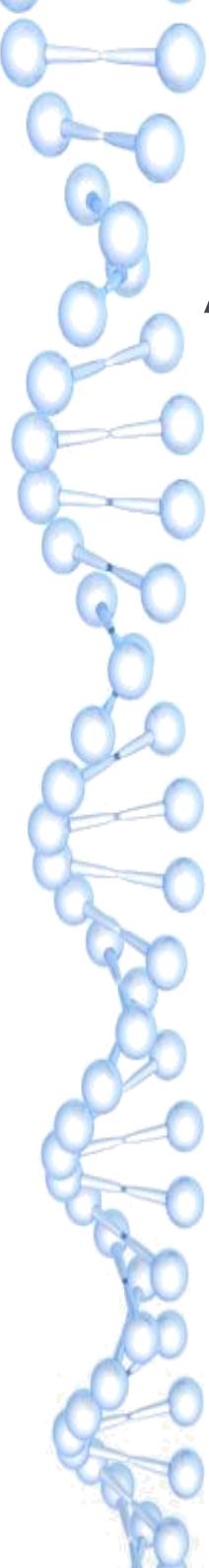


Formation Bonnes Pratiques Java



Objectifs

- Assimiler les bonnes pratiques du développement Java
- Découvrir les subtilités de Java et de sa plateforme
- Mieux comprendre des éléments clés JavaSE
- Acquérir les automatismes indispensables à la conception d'applications d'entreprises robustes



Agenda 1/2

Avant-Propos :

Présentations
Rappels Objet (5mn maxi)

Partie 2: Subtilité du langage

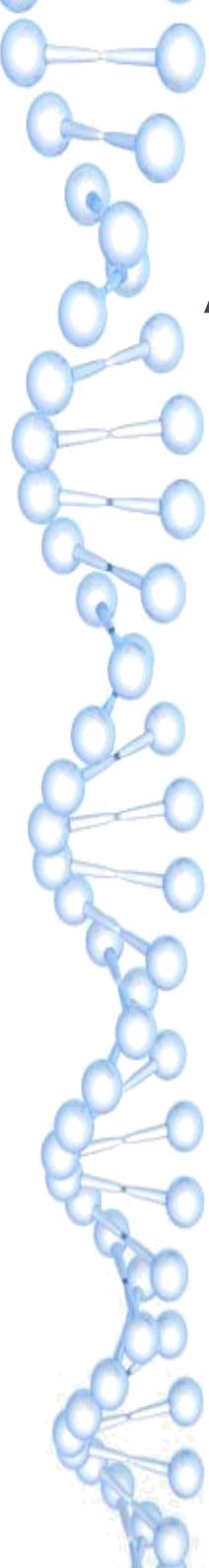
- Les mots clés (final, static, volatile, native...).
- Les niveaux de visibilité.
- Classloader
- Les Initializers
- throw, throws,

Partie 1 : Découplage, Abstraction,..

- Interface
- Classe Abstraite
- Classe Interne
- Classe Anonyme
- Lambda dans tout ça ?

Partie 3 : Gestion de la mémoire

- Le Garbage Collector.
- Détection et résolution d'une fuite mémoire



Agenda 2/2

Partie 4 : La classe **Object** et quelques interfaces de base

- Méthodes de la classe Object
- Principales interfaces proposées par le framework Comparable, Serializable, Clonable, Iterator

Partie 6 : Quelques **Design Patterns**

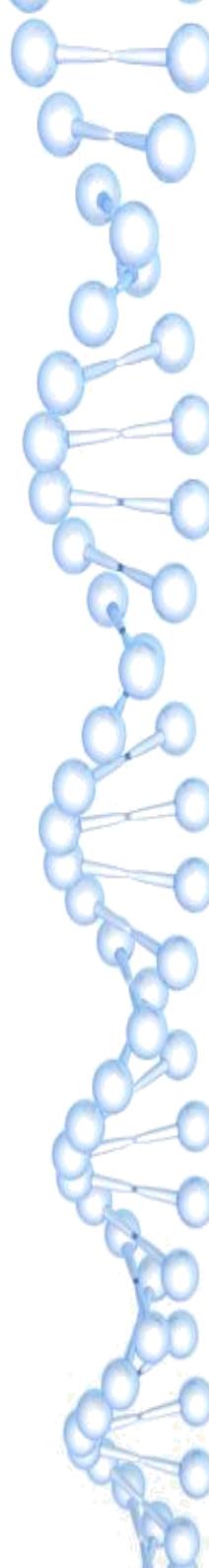
- Singleton
- Strategy
- DAO
- Factory
- Observer
- Builder

Partie 5 : **Generics** et **Collections**

- Generics
- L'interface Collection
- Les 'List', les 'Set' et les 'Map'

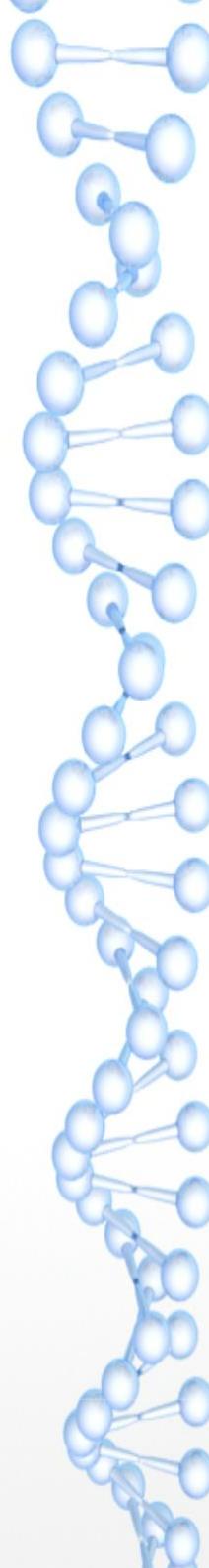
Partie 7 : **Conception d'Applications**

- Enjeux d'un développement d'entreprise
- Découpage en couches
- Des frameworks !!
- Hibernate
- Spring



Avant-Propos

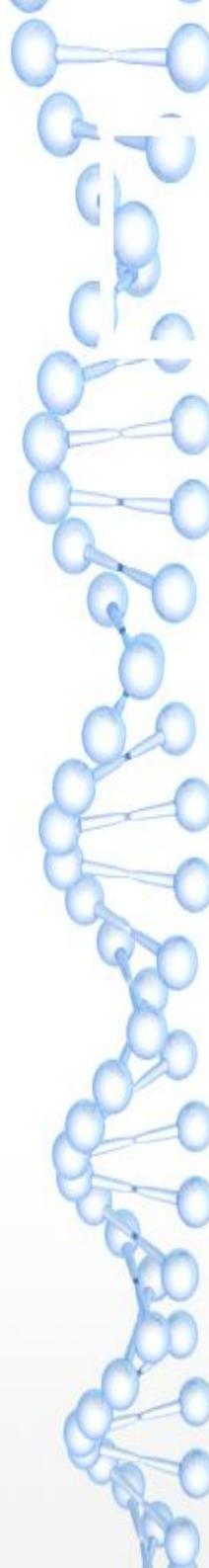
Rappel POO & Java



Mots clés Java

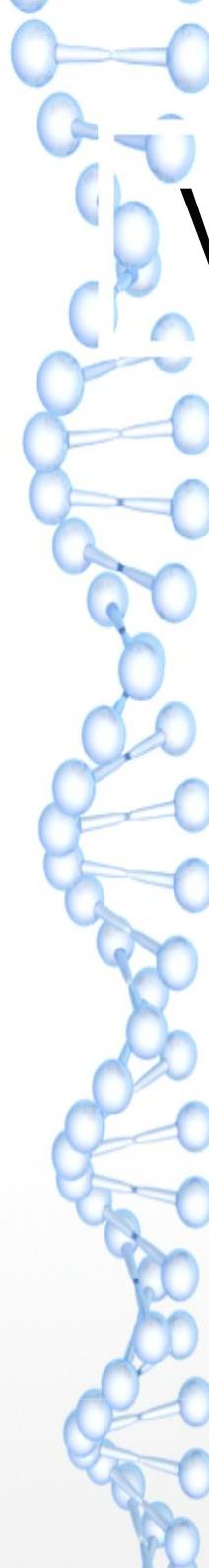
break finally byte long for
extends return while true static volatile catch
super double switch enum assert
abstract transient synchronized continue private try
final goto short char class void of if float protected import
implements throws implements throws
instanceof throw boolean else package
this interface false implements case
public new do

POO , ENCAPSULATION , POLYMORPHISME ,
HERITAGE , INTERFACE , DECOUPLAGE ,
ABSTRACTION , GENERALISATION , JVM ,
INSTANCEOF , EXCEPTION , PATTERNS,
ANTI-PATTERNS



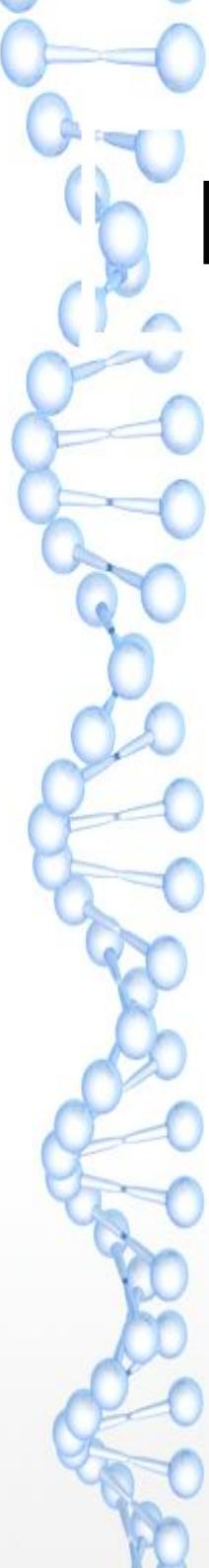
Encapsulation

- . Empaqueter ensemble données et fonctions avec une visibilité réduite depuis l'extérieur
- . La mise en œuvre de l'encapsulation est effectuée par les classes
- . Les attributs et méthodes sont masqués pour les objets qui n'ont pas besoin d'y accéder
- . Le masquage est défini dans une classe par une interface qui précise quels sont les membres (attributs ou méthodes) accessibles depuis l'extérieur

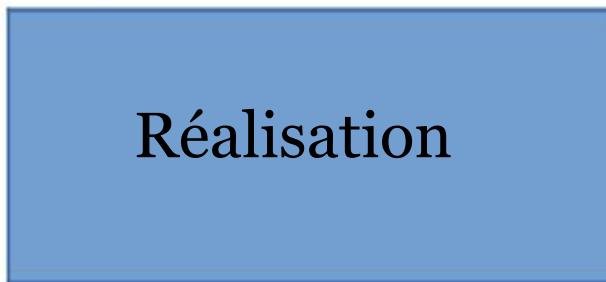
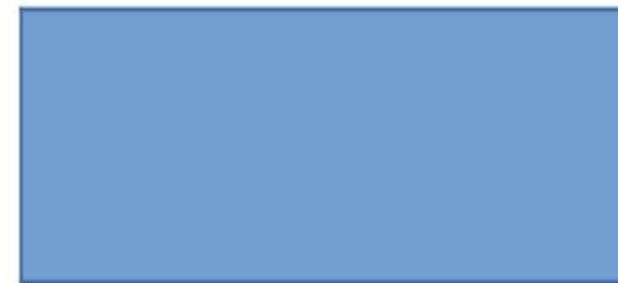
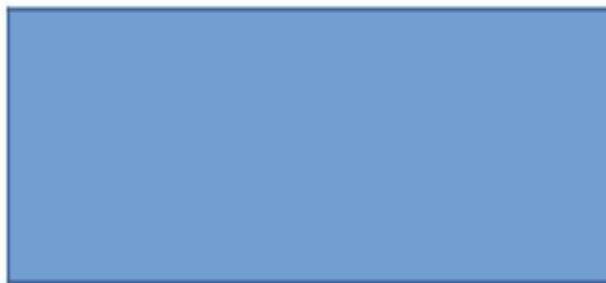


Visibilité / Accès

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
<i>no modifier*</i>	✓	✓	✗	✗
private	✓	✗	✗	✗

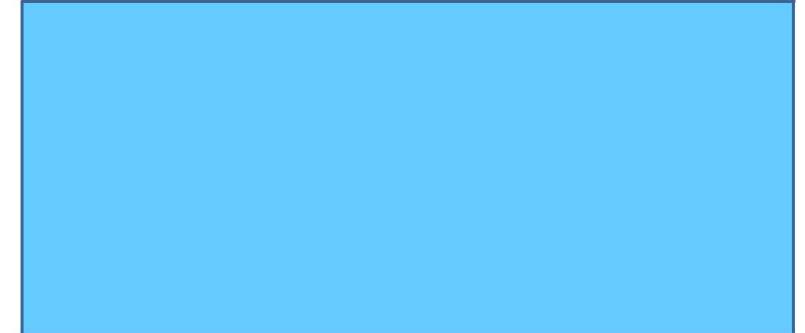
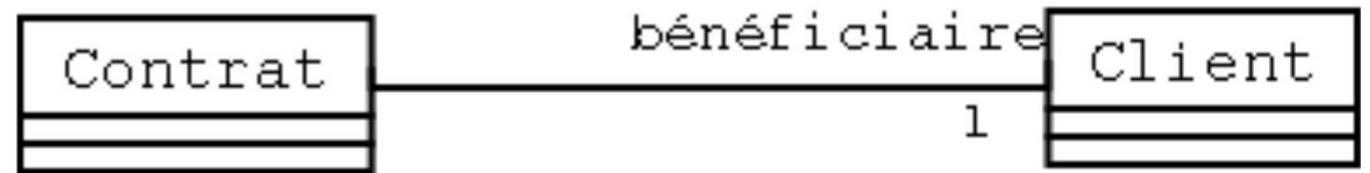


Relation de Classes

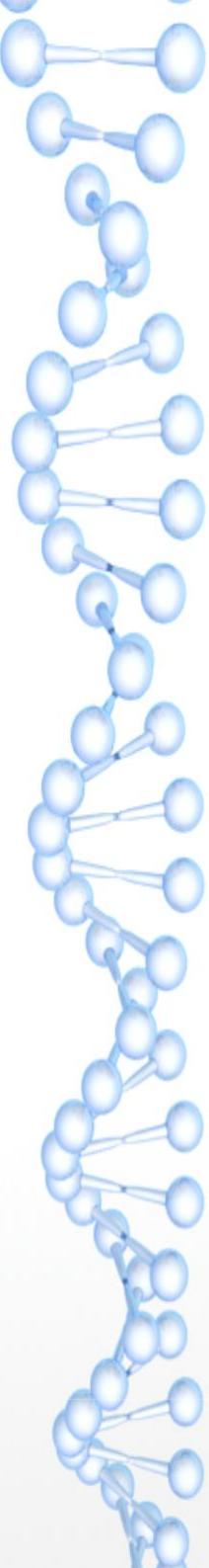


Trois structures et Une dépendance..

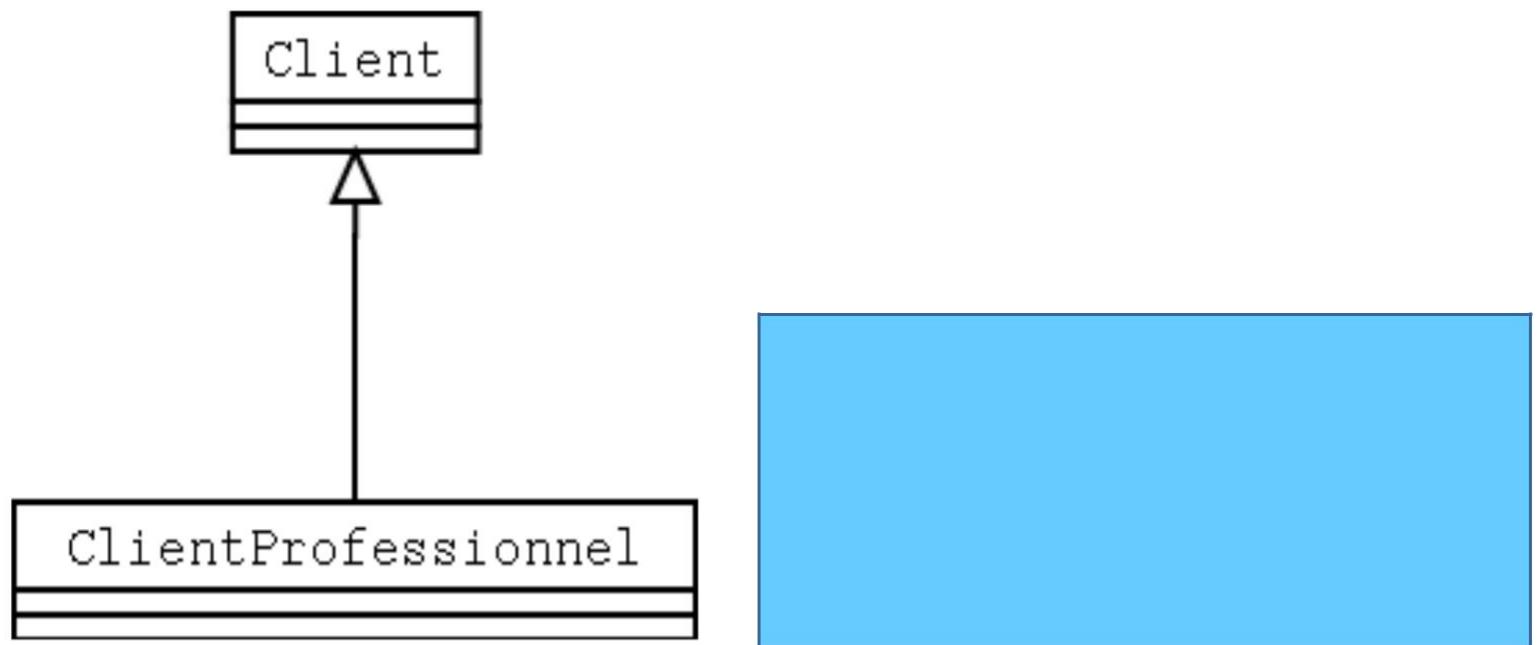
Association



Client utilise Contrat ; on peut ici réfléchir à une cardinalité différente (dans une liste par exemple)
C'est une relation structurelle par agrégation ou composition

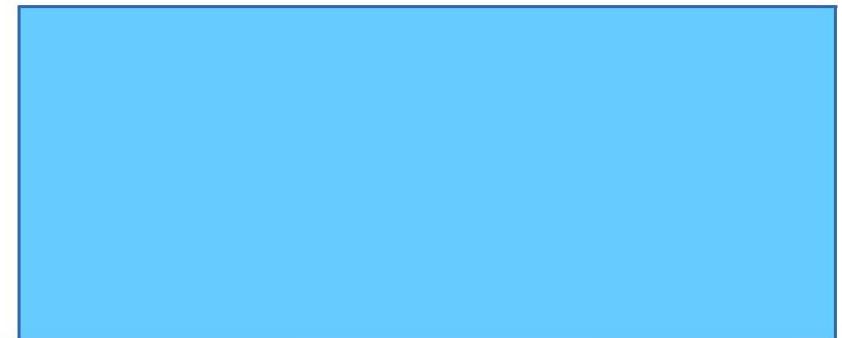
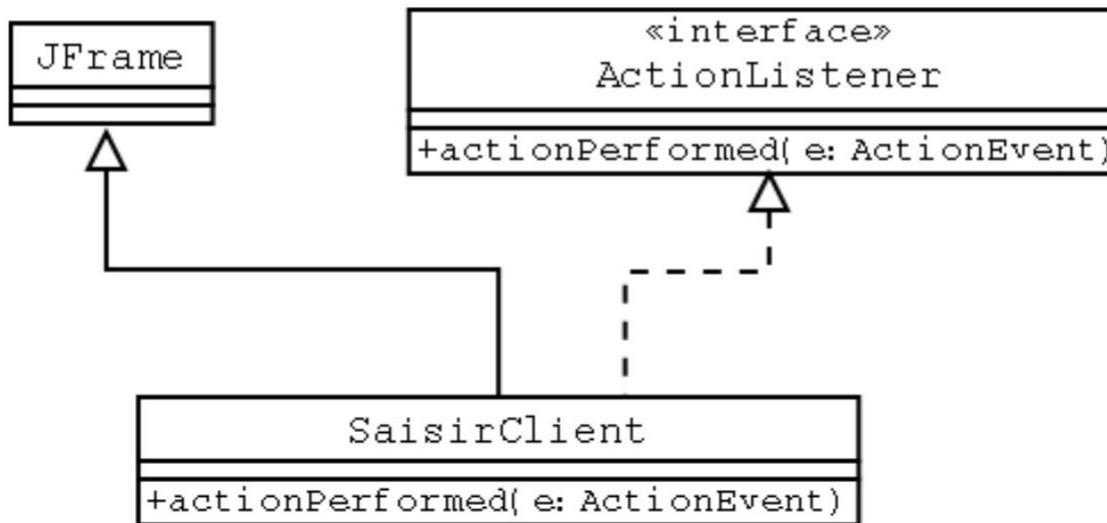


Spécialisation/Généralisation



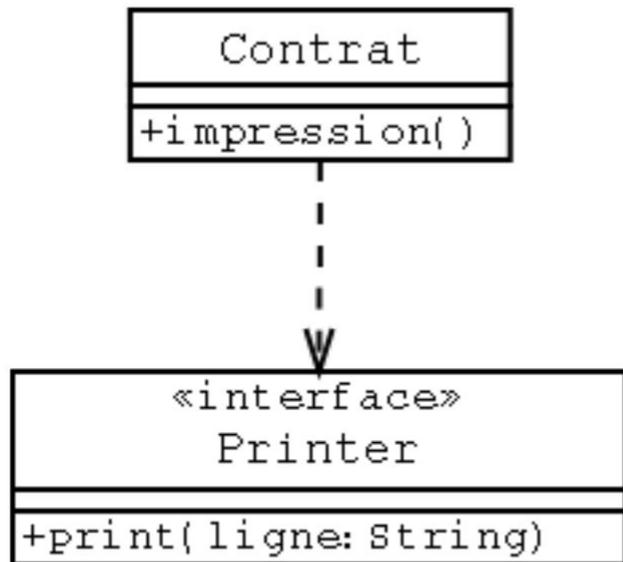
Lorsque ClientProfessionnel **spécialise** Client
On peut dire que Client **généralise** ClientProfessionnel
C'est une relation structurelle également

Réalisation



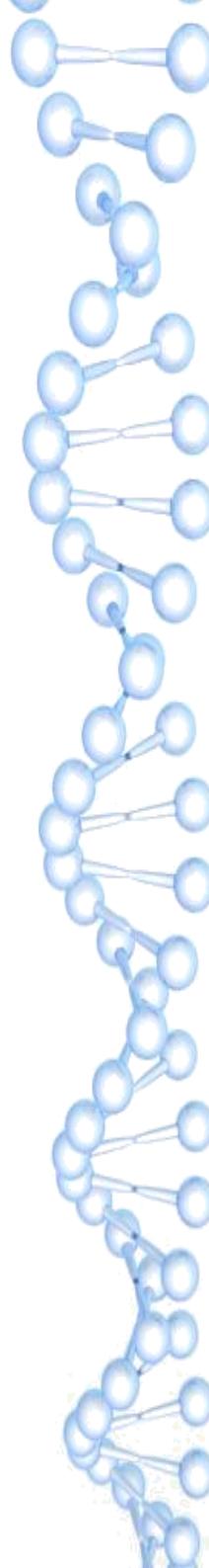
SaisirClient réalise l'interface ActionListener. Cela n'est pas sa seule mission puisqu'elle étend aussi JFrame (javax.swing)

Dépendance

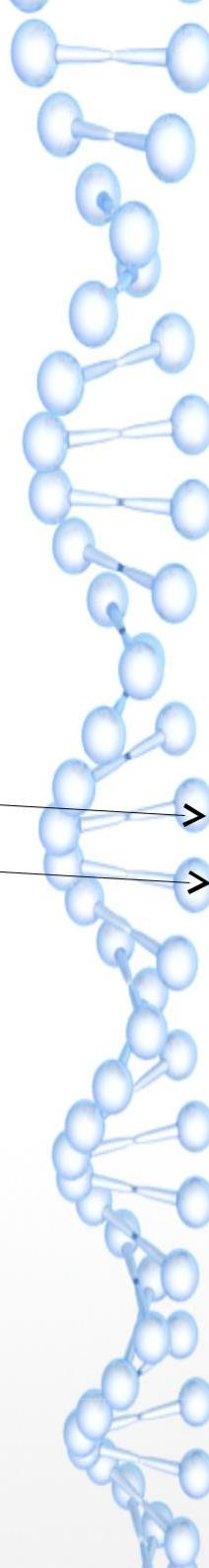


Contrat utilise Printer de **façon ponctuelle** par le biais de la méthode impression ; on n'est plus dans du structurel notamment grâce à l'interface.

Cela favorise un couplage faible, une plus grande flexibilité



Factoriser et Hiérarchiser



À méthode abstraite, classe abstraite

Mi-chemin entre Généralisation et Réalisation

La classe abstraite ‘factorise’ une partie du code

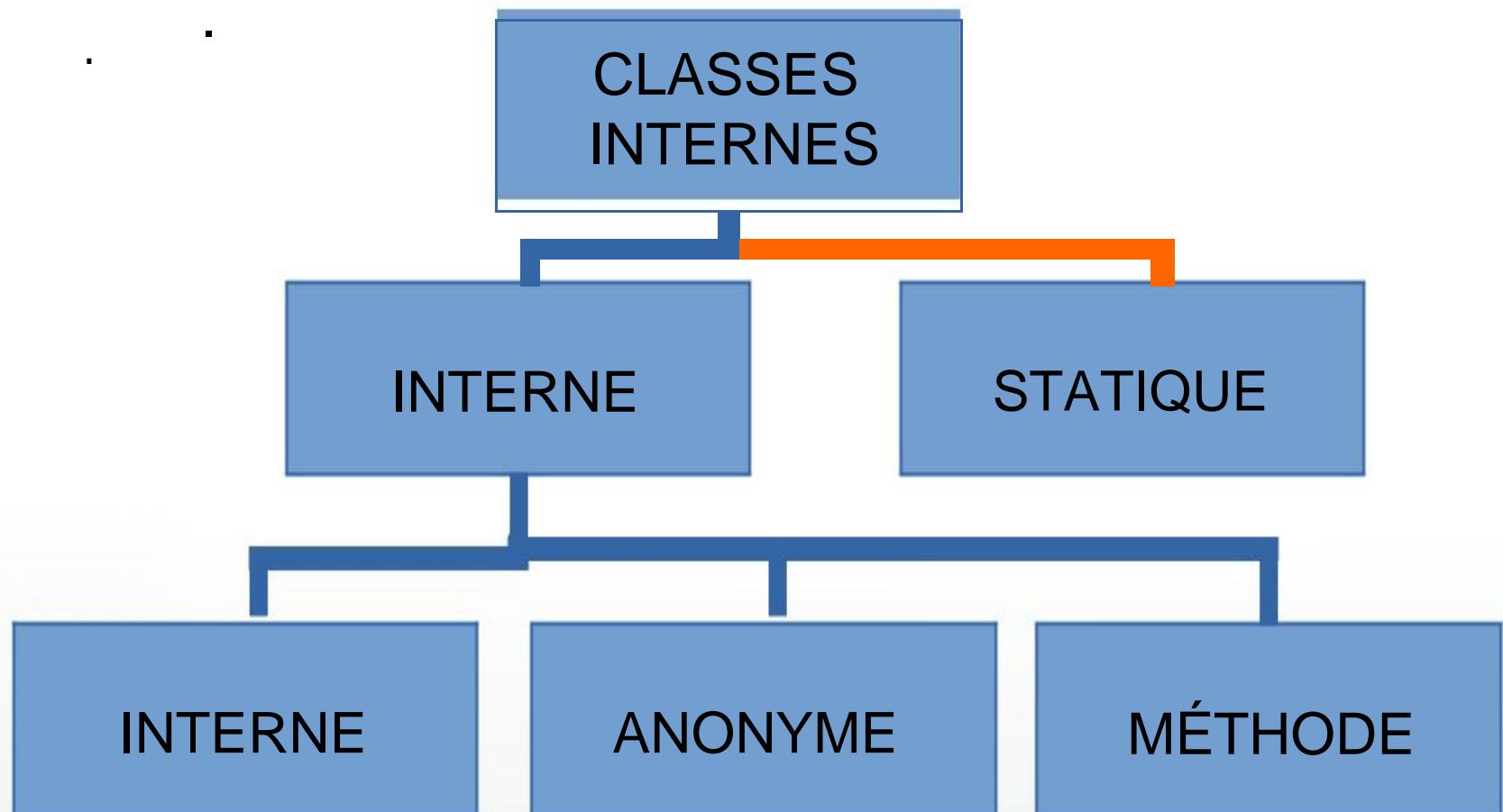
Une méthode suffit : La classe est abstraite

L'héritage se perpétue jusqu'à implémentation...

```
public abstract class Forme {  
    public abstract float perimetre(); //methode abstraite  
    public abstract float surface(); //methode abstraite  
  
    public double coefficientEtalement() {  
        double lePerimetre = perimetre();  
        return 16 * surface() / (lePerimetre * lePerimetre);  
    }  
}
```

Classes Internes

- Une classe interne n'est ni plus ni moins qu'une classe définie dans une classe avec quatre types différents.
- Le choix dépend de l'utilisation



Classe Interne (dite simple)

(inner class)

```
public class OuterClass {  
    private int index = 0;  
  
    class InnerClass {  
        public void printIndex() {  
            // InnerClass est membre de OuterClass  
            // donc accède aux attributs privés  
            System.out.println("Index: " + index);  
        }  
    }  
}
```

Deux classes fortement liées

Couplage renforcé, dialogue simplifié

Interne pas besoin de l'extérieur

Ex : Messagerie instantanée avec listeners

Classe Interne Anonyme

(anonymous inner classes)

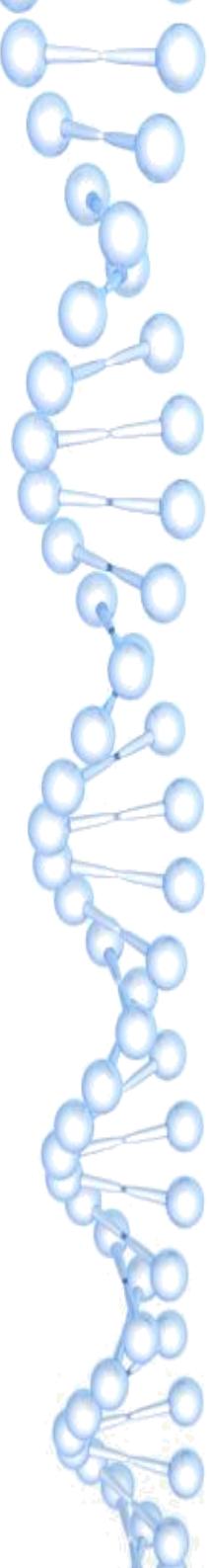
```
abstract class Bird {  
    abstract void fly();  
}  
public class AnonymousInnerClass {  
  
    public static void main(String... args) {  
  
        Bird bird = new Bird() {  
            @Override  
            void fly() { System.out.println("Flying...");  
        } }; // PointVirg obligé!  
        bird.fly(); //Résultat "Flying..."  
    }  
}
```

Code est prévu comme très spécifique

Utilisation comme variable, param. de méthode

Quasi de la programmation fonctionnelle

Ex : Swing actionPerformed, Google Guava API



Classe Interne à Méthode

(method-local inner class)

```
public class MethodLocallnnerClassExample {  
    private int x = 10;  
    void printFromInner(final int a) {  
        final int y = 10; // INTÉRESSANT !!  
  
        class MethodLocallnnerClass {  
            int w = 5;  
            public void print(int z) {  
                System.out.println("x + y - z + w + a = " + (x+y-z+w+a));  
            }  
        }  
        // OBLIGATOIRE Après la définition et DANS LA MÉTHODE  
        MethodLocallnnerClass innerClass = new MethodLocallnnerClass();  
        innerClass.print(5);  
    }  
}
```

} Code aussi prévu comme spécifique
Possible utilisation extérieure, et multi
instance Final : ici copie séparée qui survit à la
méthode **Ex** : Threads...

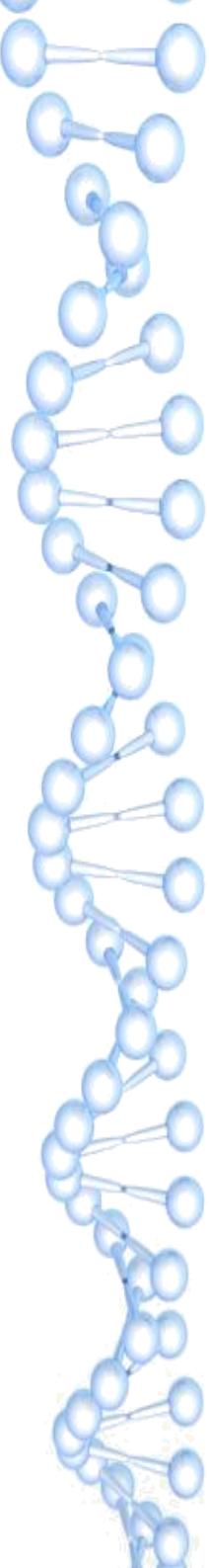
Classe statique embarquée

(static nested classes)

```
class HelperClass {  
    static class InnerHelper {  
        void print() {  
            doPrint();  
            int b = a;  
        }  
    }  
  
    static void doPrint() {  
        System.out.println("Printing from HelperClass");  
    }  
  
    static int a = 1;  
}  
  
public static void main(String... args) {  
    HelperClass.InnerHelper h1 =  
        new HelperClass.InnerHelper();  
    h1.print();  
}
```

Classe statique embarquée

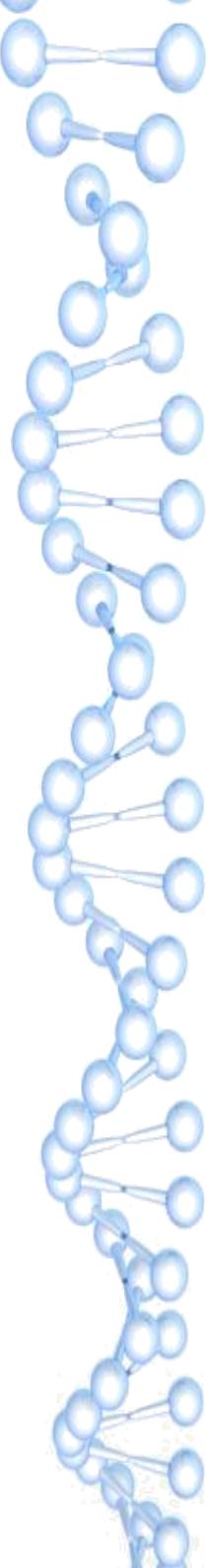
(static nested classes)



```
// tout de suite la suite
public class StaticInnerClassExample {
    static class AnotherHelper {
        void print() {
            System.out.println("Printing from AnotherHelper");
        }
    }
    public static void main(String... args) {
        /* Remark the way we create an instance of a
        static inner class
        is not the same as for inner classes */
        HelperClass.InnerHelper h1 = new
        HelperClass.InnerHelper();
        h1.print(); // will print: Printing from HelperClass
        AnotherHelper h2 = new AnotherHelper();
        h2.print(); // will print: Printing from AnotherHelper
    }
}
```

Accès aux membres statiques seulement
Économie de référence

Pas forcément besoin de la classe externe



Classes embarquées : Résumé

Classes internes statiques:

- Sont considérés comme "haut niveau".
- Ne nécessite pas une d'instance de la classe contenant pour sa construction.
- Cycle de vie propre.
- Nécessite une instance de la classe contenant pour se construction.
- **Contient automatiquement une référence implicite à l'instance contenant.**
Accède aux membres de la classe du conteneur sans la référence.
- Durée de vie est censé être plus que celle du Contenant

Type Enuméré

une classe qui s'ignore

Une énumération peut, comme une classe, comporter plusieurs membres, des méthodes, des constructeurs.

```
public enum Numbers {  
    ONE  ("un",      1),  
    TWO  ("deux",    2);  
  
    private String description;  
    private int value;  
  
    Numbers(String description, int value) {  
        this.description = description;  
        this.value = value;  
    }  
  
    public String getDescription() {  
        return this.description;  
    }  
  
    public int getValue() {  
        return this.value;  
    }  
}
```

Constantes
Enum :
valueOf()

Le compilateur
la transforme en

Class
+
Attributs final

Les Lambdas

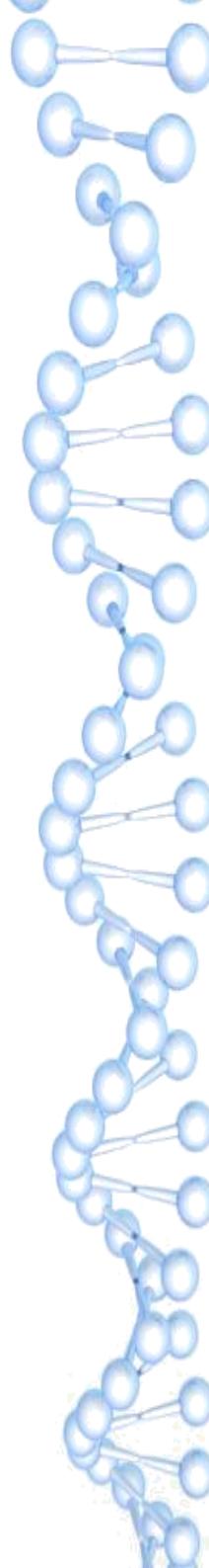


- **Nouveauté de Java 8:**
 - Introduction d'un aspect de la programmation fonctionnelle.
 - Des fonctions ponctuelles et simples (lambda) que l'on passe en paramètre.
 - Allégement et meilleure organisation du code.

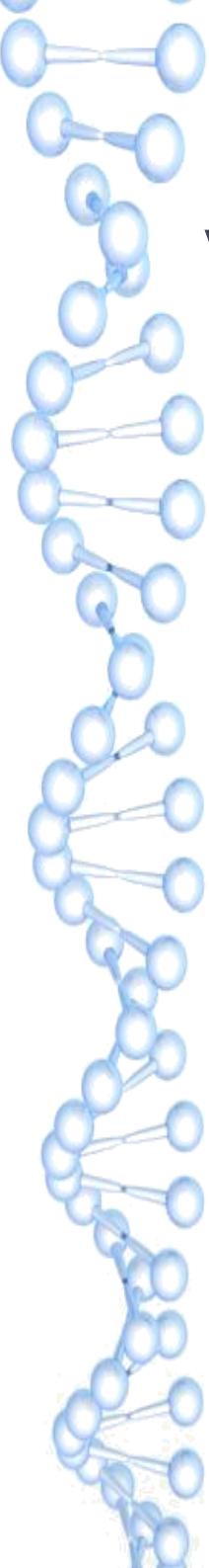
- **Formalisation:**(params) ->expression
OU (params) ->{ declarations;}

```
() -> 5      // aucun param, retourne 5 ;  
x -> 2 * x // prend un nbre, retourne son double  
(int x, int y) -> x + y // deux entiers, on additionne  
(String s) -> System.out.print(s)
```

À SUIVRE...



Subtilité du Langage



Visibilité de classe

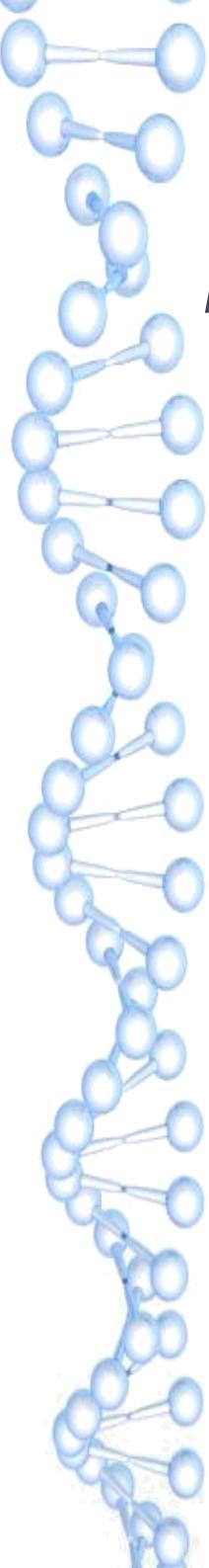
- **public**

La classe peut être utilisée par toutes les autres classes.

- **<aucun>**

La classe ne sera utilisable que par les autres classes faisant partie du même package. « *package protected* »

Une grande majorité des classes est définie comme public. La protection par package est une spécificité Java, pas présente en C++ par exemple – Utilisation limitée par architectes



Modificateurs de Classe

abstract

La classe est abstraite et ne peut donc pas être instanciée. **final**

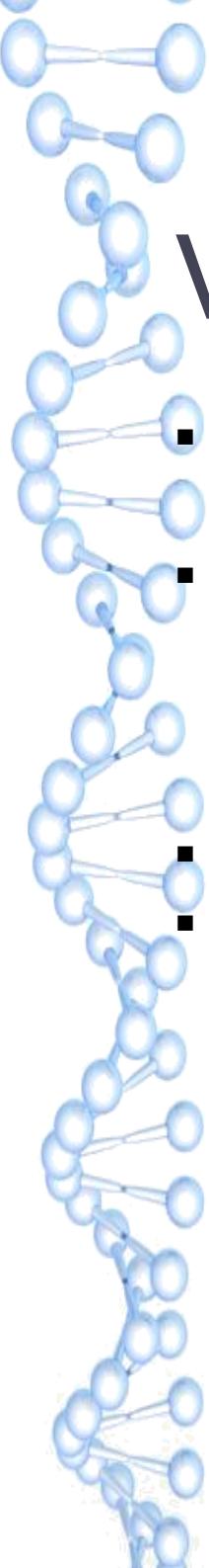
La classe ne peut pas être utilisée dans une relation d'héritage et peut uniquement être instanciée.

abstract et final étant contradictoires leur utilisation simultanée est interdite.

strictfp

les expressions float ou double sont explicitement FP-strict : IEEE 754 arithmetic

Tous les méthodes déclarées dans la classe, et tous les types déclarés dans la classe, sont implicitement strictfp
On s'assure par ce biais la portabilité des float sur toutes les plateformes Java ou autres.

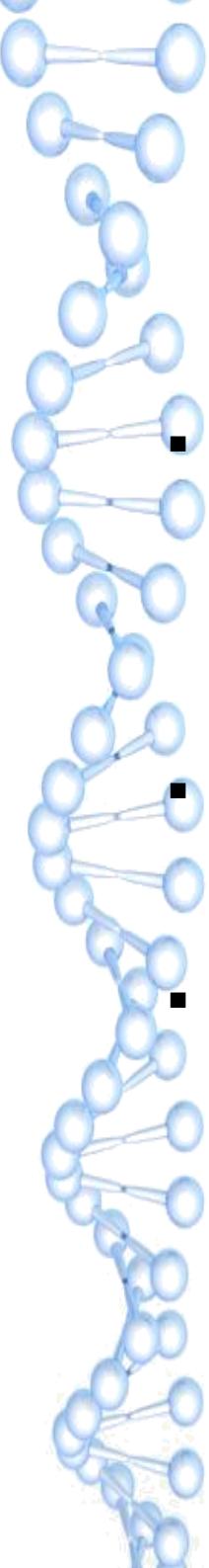


Visibilité des attributs

- **private** : la variable n'est accessible que dans la classe où elle est déclarée.
- **protected** : la variable est accessible dans la classe où elle est déclarée, dans les autres classes faisant partie du même package et dans les classes héritant de la classe où elle est déclarée.
- **public** : la variable est accessible à partir de n'importe où.
- **<aucun>** Si aucune information n'est fournie : la variable est accessible à partir de la classe où elle est déclarée et d'autres classes faisant partie du même package.

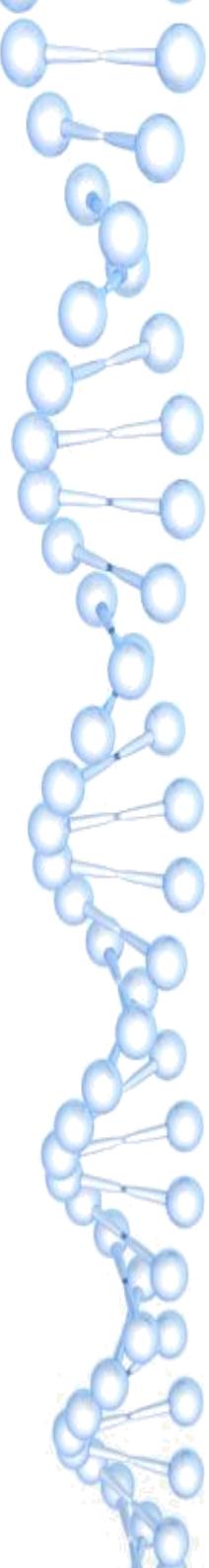
On parle de protection de paquetage / package protected

Usage : Limiter au maximum la visibilité des variables. L'idéal étant de toujours avoir des variables private ou protected, mais jamais public.



Modificateur des attributs

- Si un champ est déclaré static, il existe un seul exemplaire du champ, quel que soit le nombre d'instances (éventuellement nul) de la classe.
Usage : compteur, limitation d'accès
- Introduit une portée globale plus difficile à contrôler, débugger qu'une variable à portée plus restreinte / Sujet à controverse
- Un champ peut être déclaré final. Les deux variables de classe et d'instance (statiques et les champs non statiques) peuvent être déclarées final.
Une variable final doit être définitivement attribuée à la fin de l'exécution **tous les constructeurs**



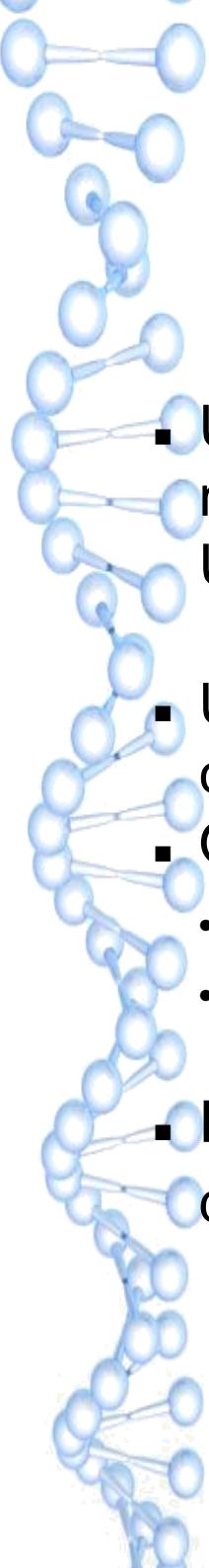
static : limite conceptuelle

```
MainLoop mainLoop1 = new MainLoop();
MainLoop mainLoop2 = new MainLoop();

new Thread(mainLoop1).start();
new Thread(mainLoop2).start();

mainLoop1.finished = true; // static variable also shuts down mainLoop2
```

Consensus : global variables are bad.

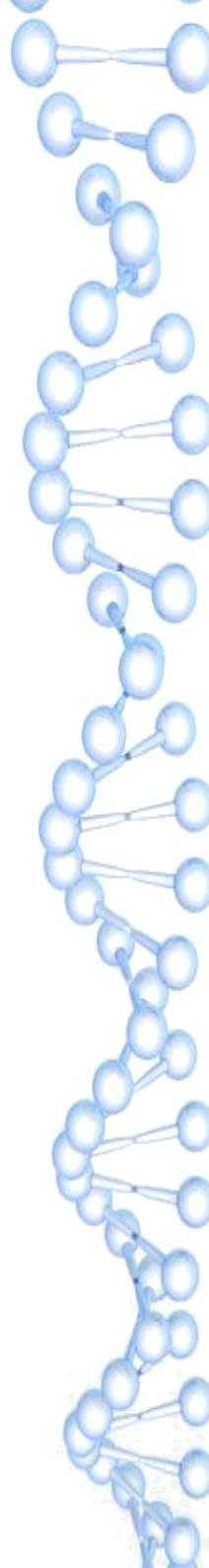


Modificateur des attributs

- Une variable dite transient permet d'indiquer qu'elle n'appartient pas à la partie persistante de l'objet
Usage : L'attribut est calculé par ailleurs , inutile de le stocker
- Une variable peut être déclarée volatile, dans ce cas, Java comprend qu'elle pourra être utilisée par différents threads
 - On assure ainsi une valeur cohérente de cette variable
 - Pas stockée au niveau du thread, mais globalement
 - L'accès est similaire à un synchronized block;
- Erreur de compilation si une variable final est également déclarée volatile

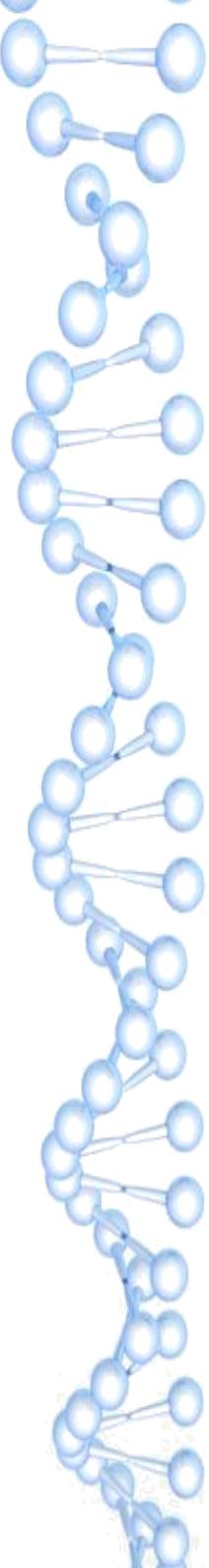
<http://rom.developpez.com/java-synchronisation/>

http://www.javamex.com/tutorials/synchronization_volatile.shtml



Mise en Application

Classes, modificateurs, etc..



Signature d'une méthode

[modificateurs]

type methode ([params])

[throws listeException] { ... }

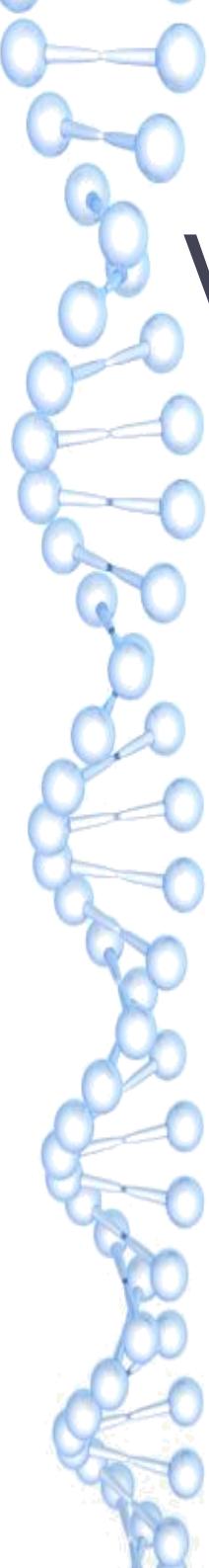
int add(int a, int b) ;

private void resetPassword();

void chargerTousLesScores() throws IOException

protected final valideFormulaire(Form f) {...}

public, protected, private, <aucun>, abstract, static, final, synchronized, native, strictfp



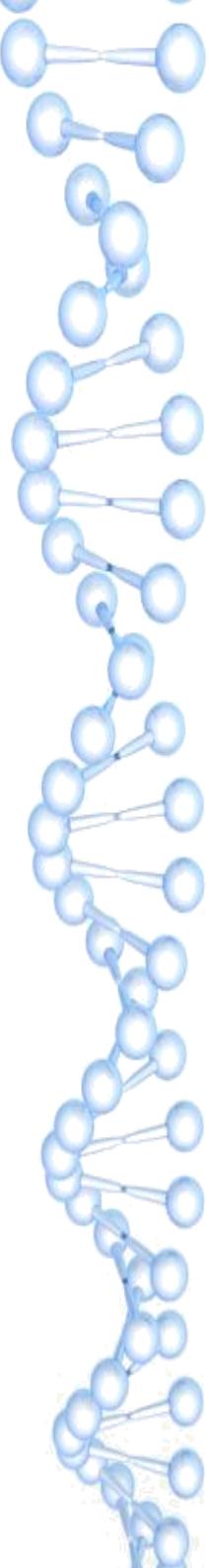
Visibilité des méthodes

private : indique que la méthode ne peut être utilisée que dans la classe où elle est définie

protected : indique que la méthode peut être utilisée dans la classe où elle est définie, dans les sous-classes de cette classe et dans les classes faisant partie du même package.

public : indique que la méthode peut être utilisée depuis n'importe quelle autre classe.

<aucun>: La visibilité sera limitée au package dans lequel la classe est définie « package protected ».



Modificateurs de méthodes

static : indique que la méthode est une méthode de classe.

Usage : Math.abs, etc. , main(String...) ;

abstract : indique que la méthode est abstraite et qu'elle ne contient pas de code. Effet domino : la classe doit être abstraite.

Usage : La classe ne « sait » pas faire : GraphicObject draw() ;

final : indique que la méthode ne peut pas être redéfinie dans une sous-classe.

Usage : On bloque un comportement : Chess nextToPlay() ;

native : indique que le code de la méthode se trouve dans un fichier externe écrit dans un autre langage (javadoc JNI).

synchronized : indique que la méthode ne peut être exécutée que par un seul thread à la fois

Variable static

Pour une variable locale

- Toujours déclarée et initialisée dans la méthode qui l'héberge.
- Visible uniquement par cette méthode.
- Sa valeur est mémorisée entre deux appels.

```
public void f() {  
    static int nbPassageDansF = 10 ; // init°, decl°  
}
```

Pour un membre de classe

- commun à toutes les instances de la classe (constantes, compteur d'instance...)
- Accessibles par les méthodes statiques ou non

```
class Personne {  
    public static int nbPersonnes = 0; // init°,  
    decl° public Personne() {  
        ++Personne.nbPersonnes; // utilisation  
    }  
}
```

Méthode static

Pour une **méthode commune**

- Si **this** est inutile → la méthode *devrait* être statique.
- Méthode statique → **this** interdit.
 - Elle peut accéder à des membres ou méthodes statiques.
 - Elle ne peut pas accéder aux autres membres et méthodes, qui dépendent de **this**.
- Pour signifier que la méthode est indépendante de **this**, elle ne s'applique pas à une instance, mais à la **classe**.

```
class Math {  
    public static double abs(double d) {  
        return d < 0 ? -d : d;  
    }  
}
```

Ne dépend
pas de **this**

```
class Personne {  
    //...  
    public static int getNbPersonnes() {  
        return Personne.nbPersonnes;  
    }  
}
```

Static & Non Static

- Une méthode / membre **statique** s'applique toujours à une **classe**
- Si la classe est omise, on sous-entend la classe courante

```
class Main {  
    public static void f() {}  
  
    public static void main(String [] args)  
    {      f();      }  
}
```

Implicitement
Main.f();

- Une méthode / membre **non statique** s'applique toujours à une **instance**
- Si l'instance est sous-entendue, l'instance utilisée est **this**

```
class Personne {  
    public String prenom;  
    public String getPrenom() { return prenom; }  
}
```

Implicitement
this.prenom

Import Static (JDK 1.5)

- Un attribut static est accessible par sa classe
- Vu par fastidieux par certaines équipes, qui contournent .
par une implémentation d'Interface : mauvaise pratique !!
- JDK introduit l'import static afin d'éviter cette 'Anti Pattern'

```
import static java.lang.Math.PI; // UNITE
import static java.lang.Math.*; // EN MASSE
// -----
double r = cos(PI * 2);
```

Attention, on perd de vue la classe origine

Un peu de final Objet

- **final** verrouille en lecture seule une **référence**
 - L'objet pointé par la référence reste modifiable
 - Contrairement au C++ qui utilise le mot clé **const**, on ne peut pas verrouiller en lecture seule l'objet référencé.

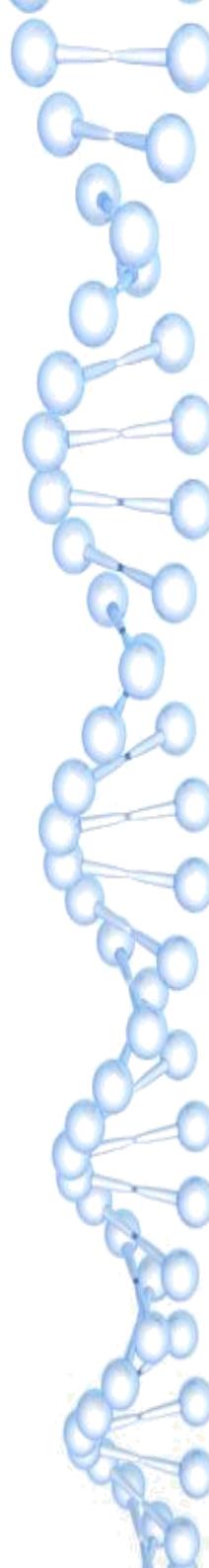
```
void maMethode(final Personne p) {  
    p.setPrenom("Sophie"); // ok la référence ne change pas p  
    = new Personne("Marc"); // pas ok, la référence change  
}
```

- On utilise généralement le mot clé **final** lorsqu'on manipule une constante.

```
class Planck {  
    public final static double PLANK = 6.62607004*Math.pow(10,-34);  
}
```

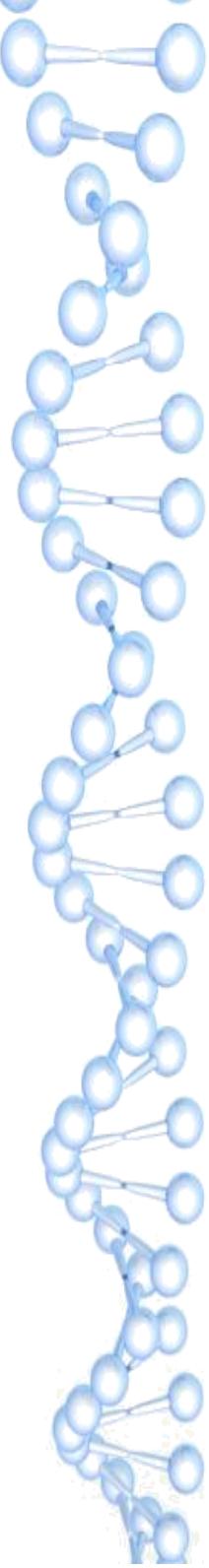
Rappel de l'utilisation

Modifier	Class	Class Variables	Methods	Method Variables
public	✓	✓	✓	
private		✓	✓	
protected		✓	✓	
<i>default</i>	✓	✓	✓	
final	✓	✓	✓	✓
abstract	✓		✓	
strictfp	✓		✓	
transient		✓		
synchronized			✓	
native			✓	
volatile		✓		
static	✓	✓	✓	



Mise en Application

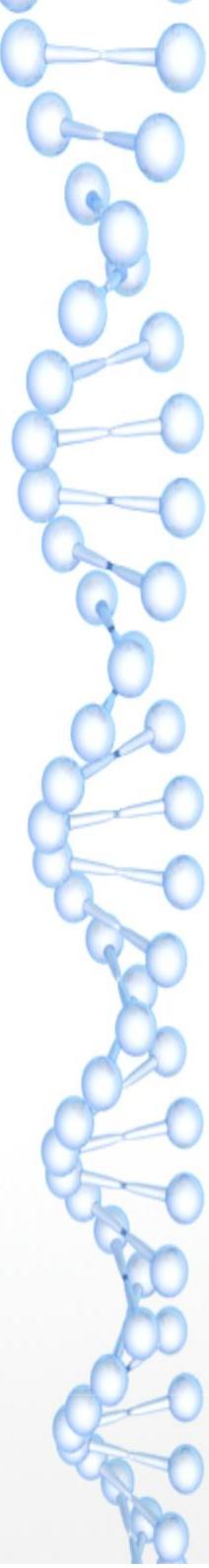
De la bonne organisation du code



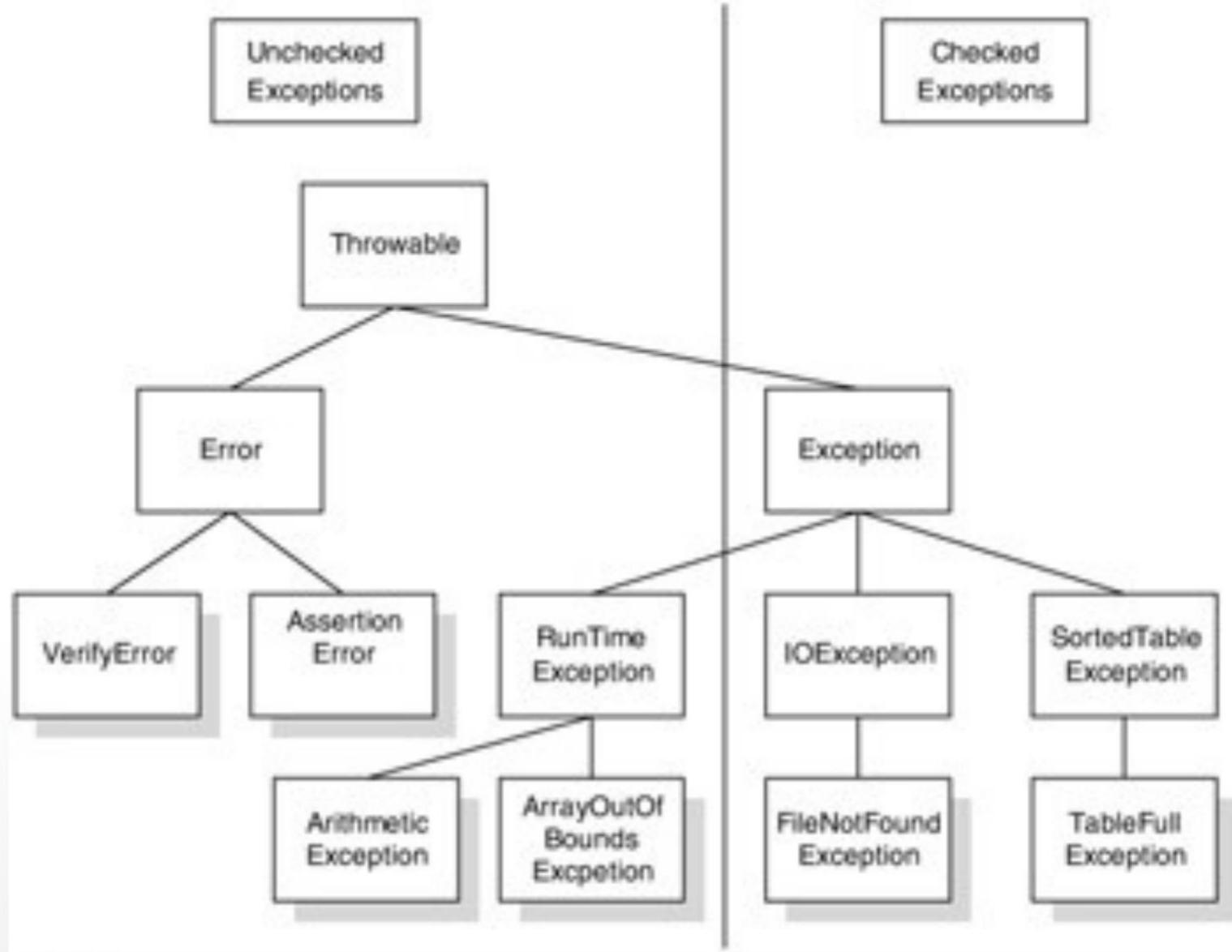
Du traitement des exceptions

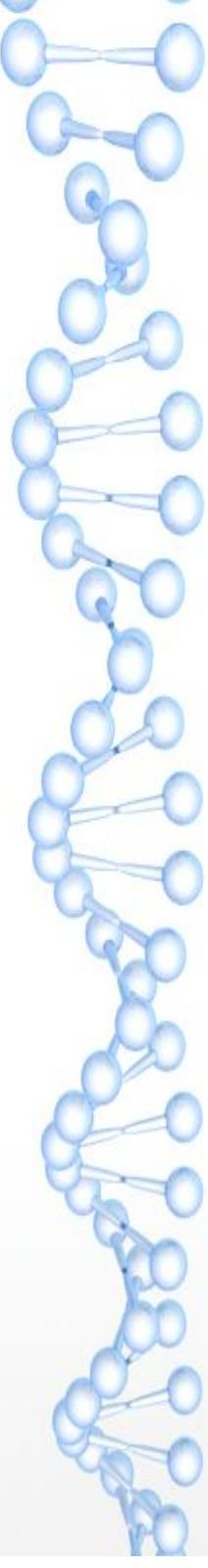
Vous connaissez probablement tous ...

```
try
{
    // quelques actions potentiellement risquées
}
catch(SomeException se)
{
    // que faire si une exception de ce type survient
}
catch(Exception e)
{
    // que faire si une exception d'un autre type survient
}
finally
{
    // toujours faire ceci, quelle que soit l'exception
}
```

Parlons ‘Throwable’



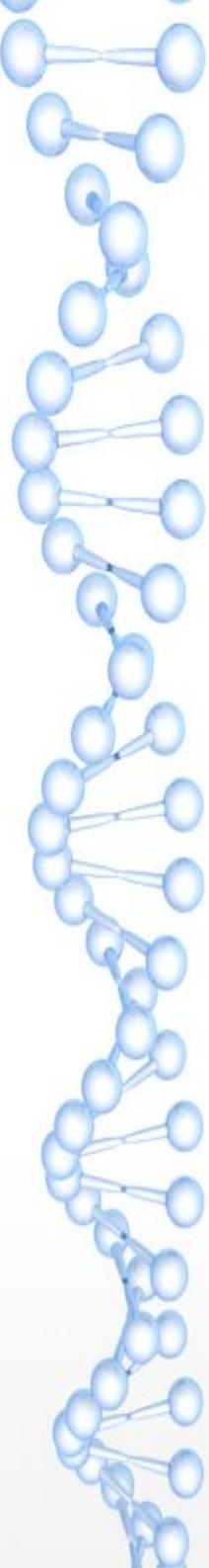


Traitement des exceptions

Interception par bloc *try-catch-finally*

Implémentation

```
public ExampleException() {  
    for(int i=0;i<3;i++) {  
        } test[i]=Math.log(i);  
    try {  
        for(int i=0;i<4;i++) {  
            } System.out.println("log("+i+") =  
"+test[i]); } catch(ArrayIndexOutOfBoundsException ae) {  
    } System.out.println(<< Arrivé à la fin du tableau >>);  
  
    } System.out.println(<< Continuer le constructeur >>);
```

Transmettre les exceptions

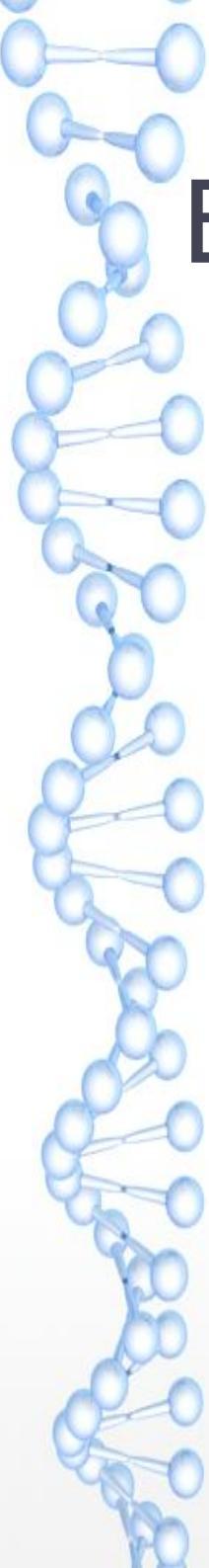
Lancement avec *throws* et *throw*

- Si la méthode n'est pas censée la traiter elle-même, elle doit en « lancer » cette instance à son appelant
- Il faut déclarer que la méthode peut lancer cette/ces exception(s)
- Ajouter la clause *throws* à la signature de la méthode

```
public void writeList() {  
    FileWriter fw = new FileWriter("OutFile.txt")  
    PrintWriter out = new PrintWriter(fw);  
    for (int i = 0; i < vector.size(); i++)  
        out.println("Valeur = " + vector.elementAt(i));  
}
```

- Ici deux exceptions peuvent être lancées
- Peut lancer une *ArrayIndexOutOfBoundsException*

```
public void writeList() throws  
    IOException,  
    ArrayIndexOutOfBoundsException {
```

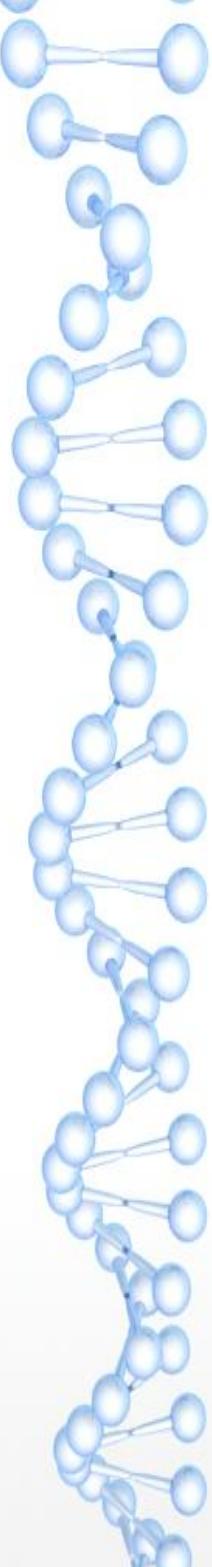



Enchainement d'exceptions

```
public void connectMe(String serverName)
    throws ServerTimedOutException {
    boolean success;
    int portToConnect = 80;
    success = open(serverName, portToConnect);
    if (!success) {
        throw new ServerTimedOutException("Trop long", 80);
    }
}
```

```
public void connectMe(String serverName) {
    boolean success;

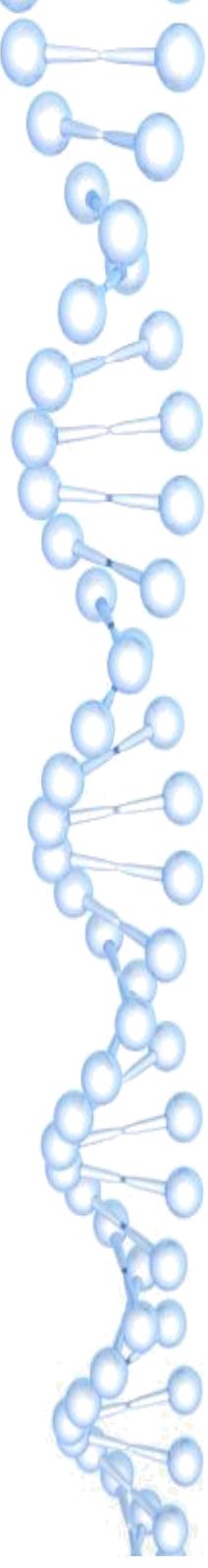
    int portToConnect = 80;
    success = open(serverName,
    portToConnect); if (!success) {
        try{ throw new ServerTimedOutException("Trop long ", 80);
        } catch(ServerTimedOutException stoe){ }
    }
}
```



Vos propres exceptions

- Une méthode signée *throws* peut lancer des Exceptions avec *throw*
- On peut vouloir créer ses propres classes d'Exceptions
 - Pour couvrir un nouveau domaine d'erreurs
 - Afin d'affiner une erreur existante
- Une exception dite CHECKED doit finalement être attrapée dans le code

```
class MyException extends Exception
{ MyException(String msg) {
    System.out.println("MyException lancee, msg =" + msg);
}
void someMethod(boolean flag) throws MyException {
    if(!flag) throw new MyException(<<someMethod>>);
    ...
}
```

Les six commandements

Throwable et Error tu n'attraperas pas

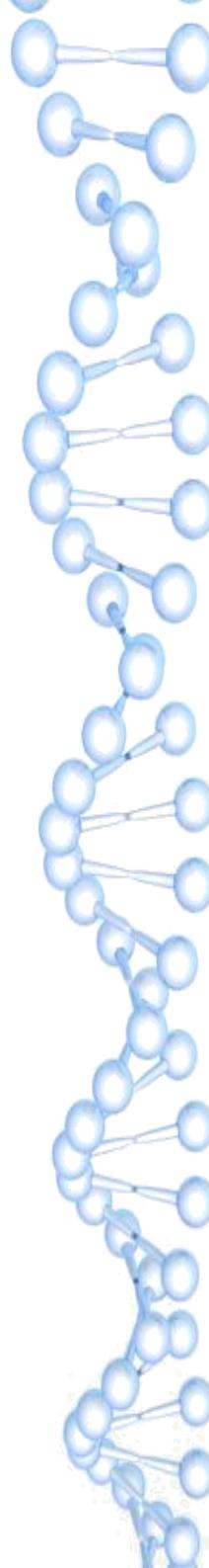
Throwable.printStackTrace(...) tu n'utiliseras pas en production

Les exceptions Generic , Error, RuntimeException,
Throwable and Exception tu ne lanceras pas

Le receveur de l'erreur , l'exception d'origine il préservera

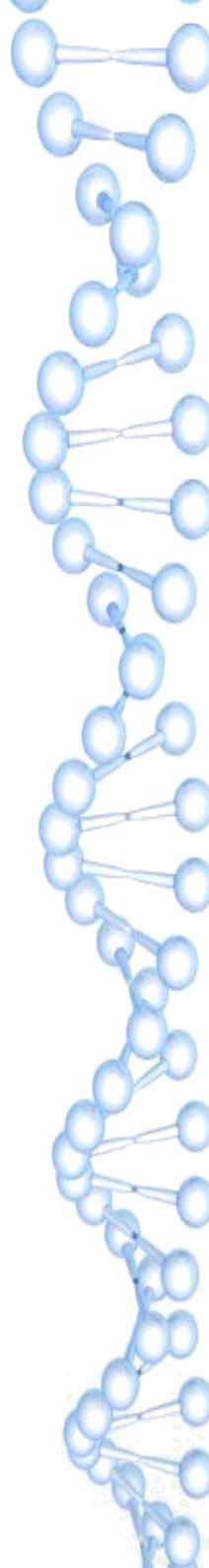
Pour logger, vers System.out ou System.err tu n'enverras pas

Des exceptions, un style de programmation tu n'en feras pas

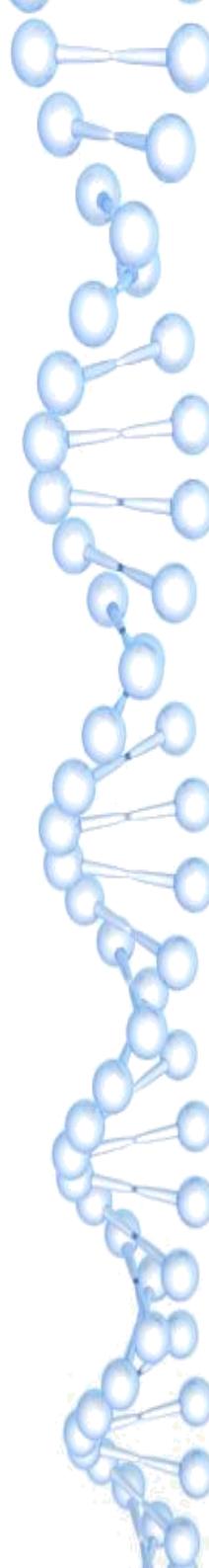


Mise en Application

Utilisation des Exceptions avec Héritages



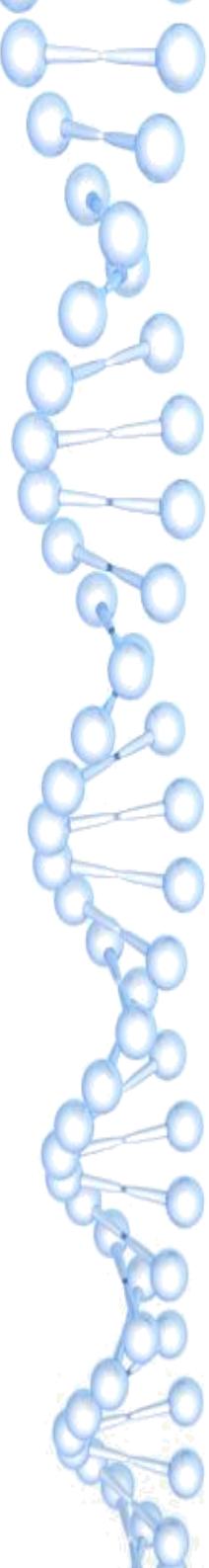
Classloader



Java comme Plateforme

Java Virtual Machine (1 / 2)

- Définit les spécifications hardware de la plateforme
- Lit le bytecode compilé (indépendant de la plateforme)
- Software
 - Desktop, JavaME, Serveurs Web, IBM AIX
 - Navigateurs Web
- Hardware
 - PicoJava, FemtoJava, Komodo, ajile Systems

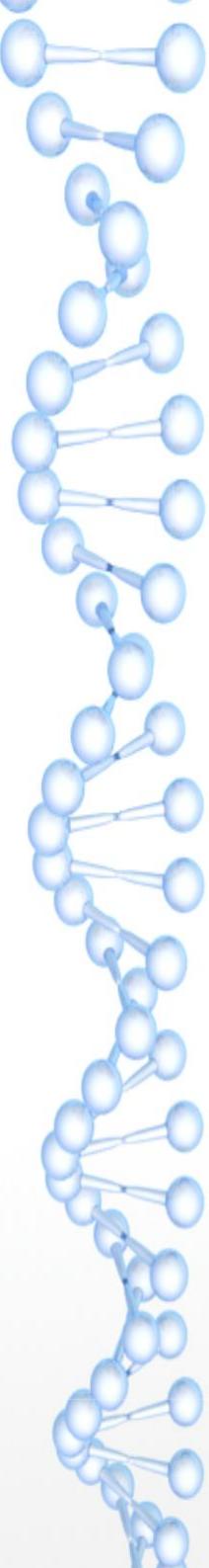


Java comme Plateforme

Java Virtual Machine (2/2)

La JVM définit :

- Les instructions de la CPU
- Les différents registres
- Le format des fichiers .class
- Le « Stack »
- Le tas (« Heap ») des objets « garbage-collectés »
- L'espace mémoire en général



Java comme Plateforme

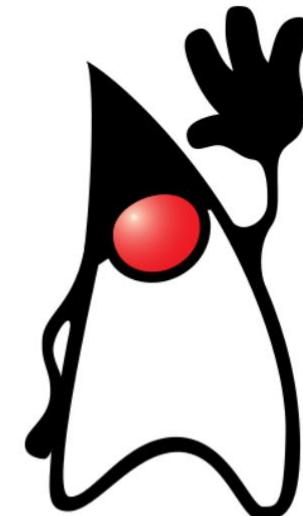
Java Runtime Environment

Trois tâches principales :

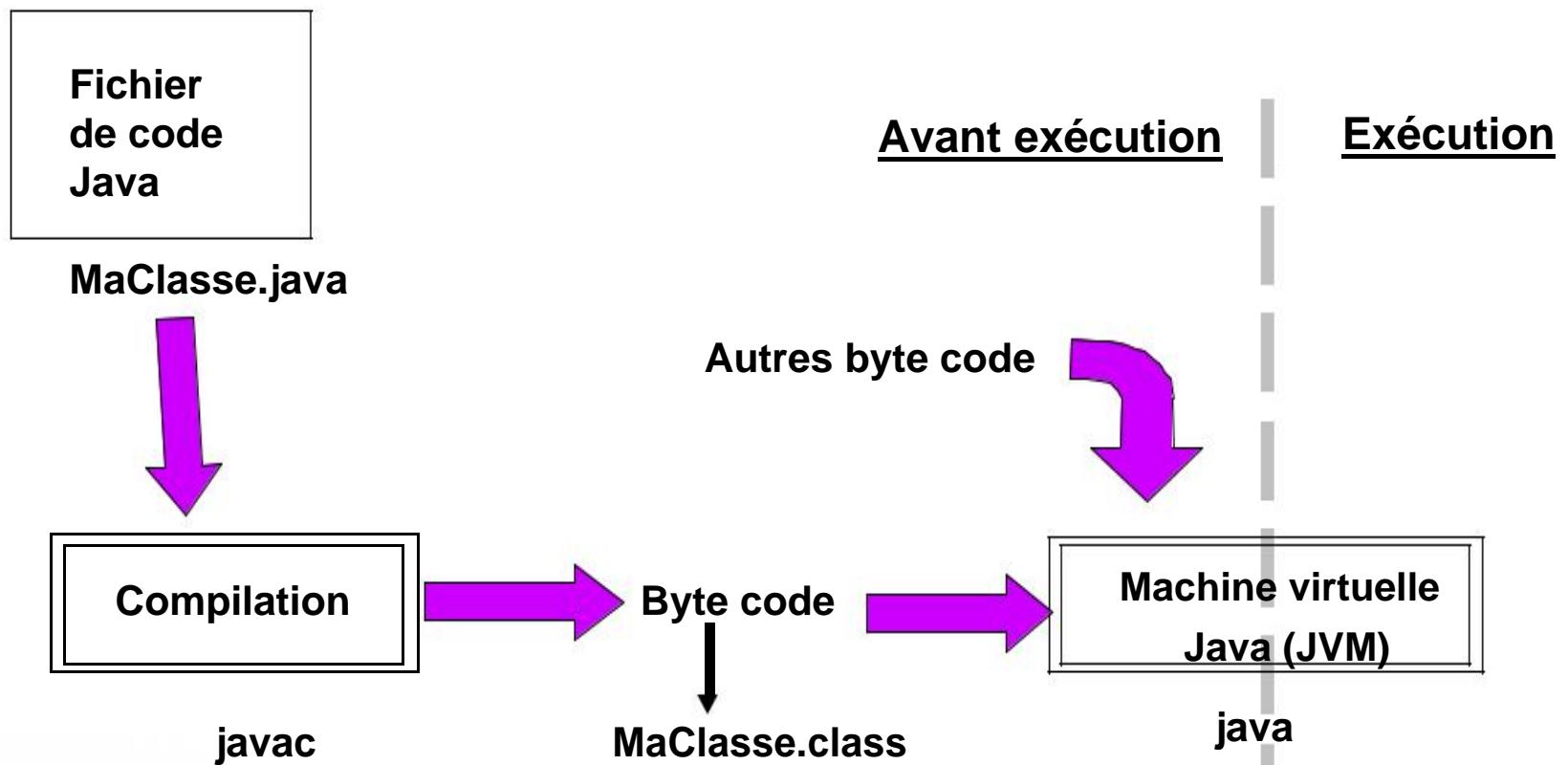
1. Charger le code (class loader)
2. Vérifier le code (bytecode verifier)
3. Exécuter le code (runtime interpreter)

D'autres Threads s'exécutent :

- Garbage Collector
- Compilateur Just In Time (aka JIT)



Java : Compilé & Interprété



Constructeurs = Initialisation

Une classe comporte en général un ou plusieurs **constructeurs**.

- Un constructeur alloue l'instance en mémoire et l'initialise une instance
- Un constructeur prend ou non des paramètres
 - Pas de paramètres : "**constructeur par défaut**"
 - Paramètres ayant pour type la classe : "**constructeur de copie**" ... ou d'autres paramètres.
 - Le constructeur est une quasi-méthode : il fait partie du design de l'objet. Il n'a cependant pas de type de retour et porte le nom de la classe.

Personne
- String m_sPrenom - String m_sNom
+ Personne(m_sPrenom,m_sNom) + afficher()

Constructeurs Java

new invoque un constructeur.

this désigne l'instance courante

super désigne l'instance parente.

```
public class Personne {  
    private String prenom;  
    private String nom;  
  
    public Personne(String prenom, String  
                    nom) { super();  
        this.prenom = prenom;  
        this.nom = nom;  
        //...  
    }  
}
```

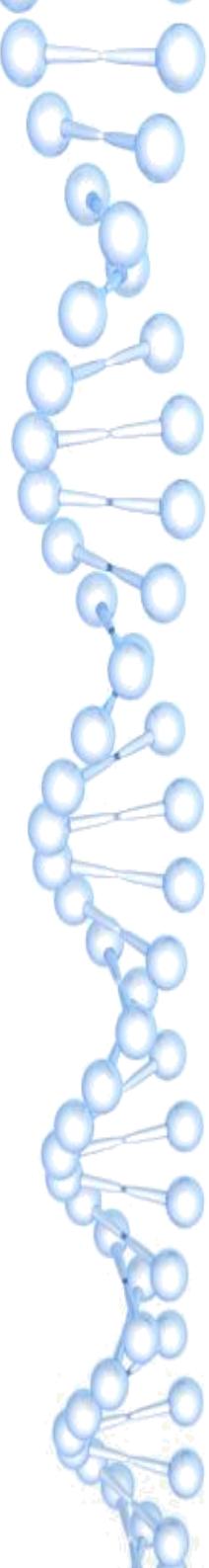
```
Personne prof = new Personne("Julien", "Dupuis");
```

Constructeurs Java

À l'instanciation, il faut :

- de l'espace mémoire alloué à l'objet
 - ainsi que de l'espace pour chaque variable d'instance (classe et super classes)
- Quand :
- Explicite : new
 - Implicite : String littoral , String concaténation, Boxing

```
public static void main(String args[]){
    String s = new String("MARCO");
    s = s + "POLO";
    Integer i = 5;
}
```

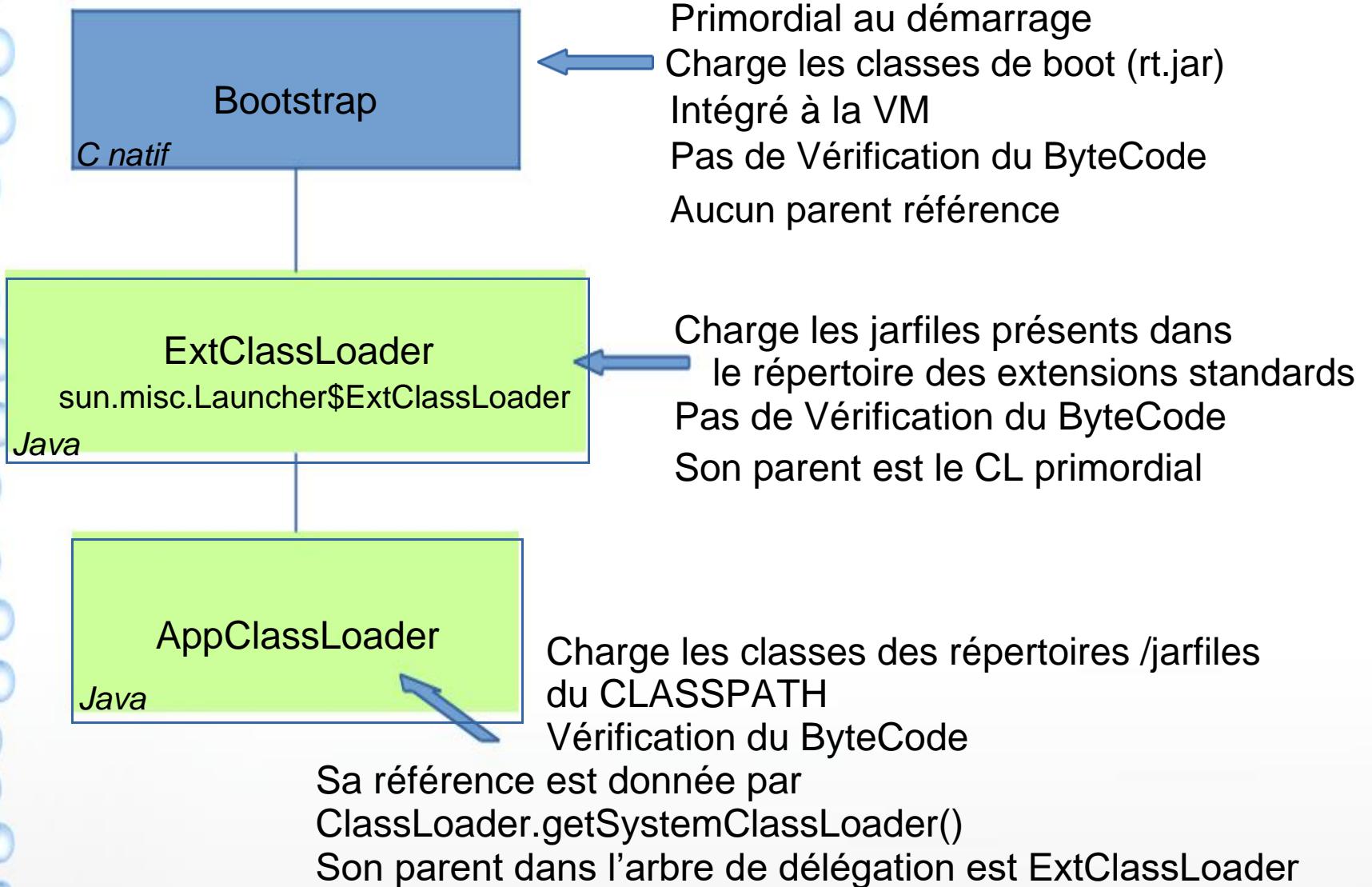


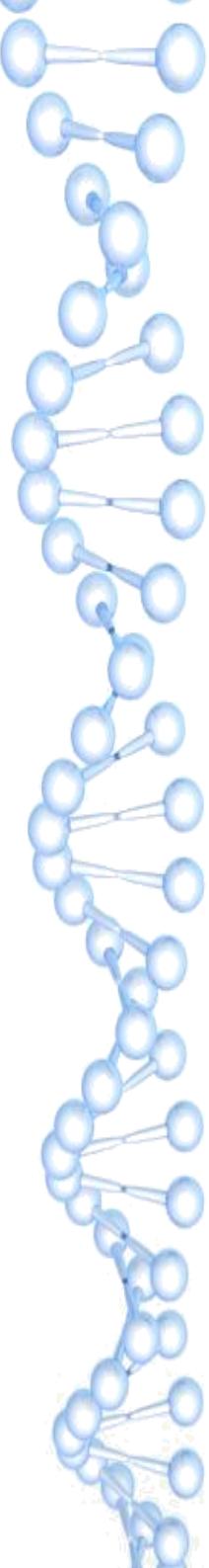
Chargement d'une Classe

Avant qu'une référence à l'objet créé soit retournée, le constructeur utilisé est procédé comme suit:

- 1. Assigne les arguments aux paramètres du constructeur**
2. Si le constructeur commence par un appel à un autre constructeur de la même classe (**this(...)**)
=> On reprend à l'étape 1 avec ce constructeur
3. N'utilise pas **this(..)**. Sauf si c'est la classe Object, il y aura un appel à **super(..)**
=> On continue à l'étape 1 avec ce constructeur de la classe parente
4. Exécution des **initialiser d'instance et de variables d'instance** de la classe.
 1. Assigne les valeurs des initialiseurs de variables aux variables (de gauche à droite)
- 5. Exécute le reste du constructeur**

CL : Arbre de Délégation

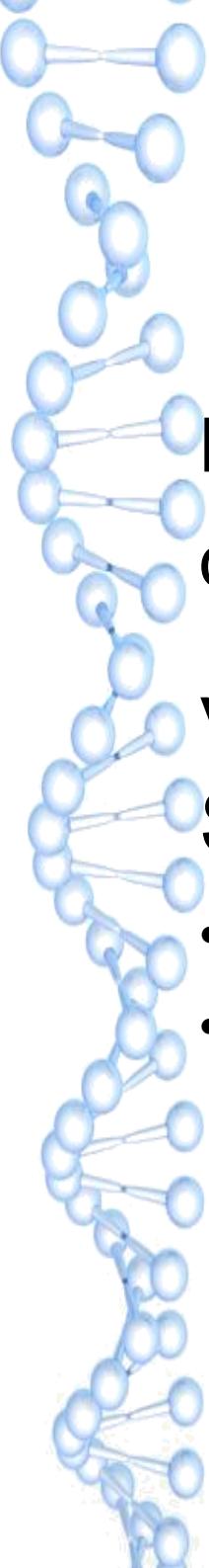




Mise en Application

Ecriture d'un Custom ClassLoader

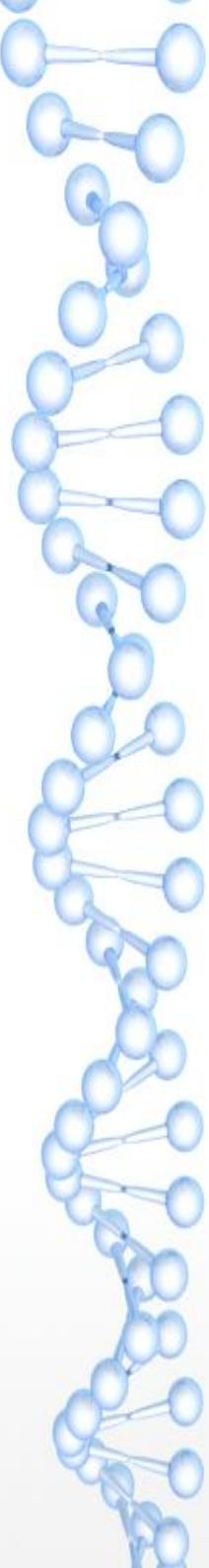
<http://www.journaldev.com/349/java-classloader>



Initializers

Les objets sont initialisables par d'autres moyens que par les seuls constructeurs ; on parle de :

- Variable initialization : Init de variable
- Static Initialization : Bloc static
- Instance initialization : Bloc d'instance
- Alt. : Private static method : Méthode privée



Modèles d'Initializer

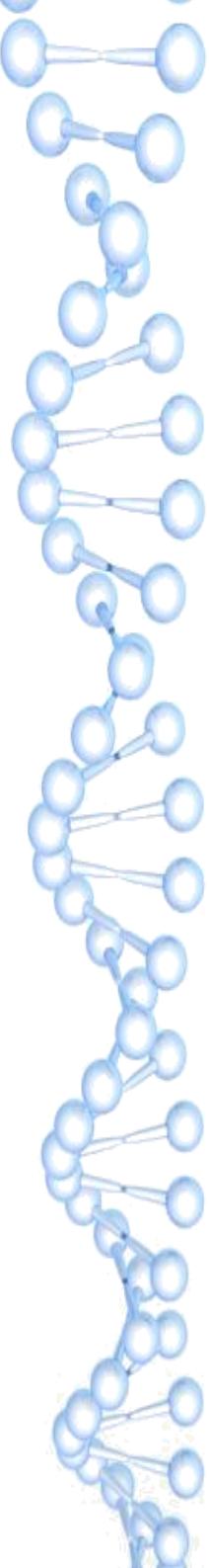
```
1 class CoffeeCup {  
    private int innerCoffee;  
  
    public CoffeeCup() {  
        innerCoffee = 355;  
    }  
}
```

```
2 class CoffeeCup {  
    private int innerCoffee=355;  
}
```

```
3 class CoffeeCup {  
    private int  
    innerCoffee; {  
        innerCoffee = 355;  
    }  
}
```

```
4 public class CoffeShop {  
    public final static List<String> JA;  
  
    static {  
        List<String> list = new ArrayList<String>();  
        list.add("Deca");  
        list.add("Arabicca");  
        list.add("Moka");  
        JA = Collections.unmodifiableList(list);  
    }  
}
```

```
5 public class CoffeShop {  
    public final static List<String> JA =  
    initList(); private static initList() {  
        .....  
        return aList ;  
    }  
}
```



Modèles d'Initializer

1) Par constructeur – Pas un initializer.

Utilisation du constructeur est correcte, mais si on gère plusieurs constructeurs, peut s'avérer lourde.

2) Initializer de variable

Appelé avant les constructeurs donc commun à tous

3) Initializer d'Instance

Dans un bloc pour quelques lignes simples

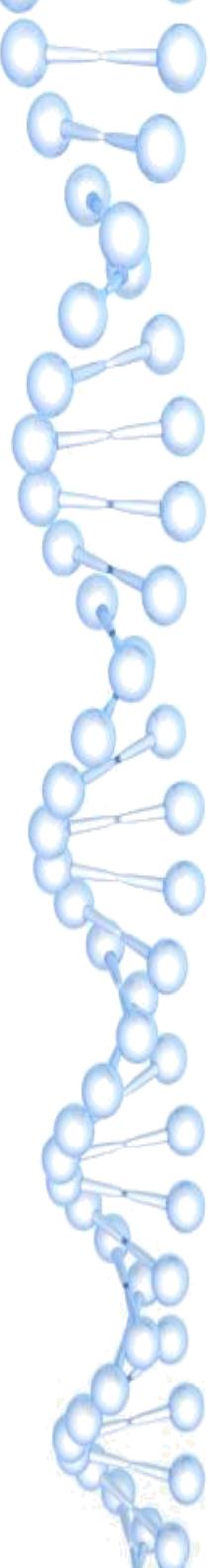
4) Initializer static

Accès à des variables statiques (pas un constructeur)

Efficace

5) Méthode statique privée

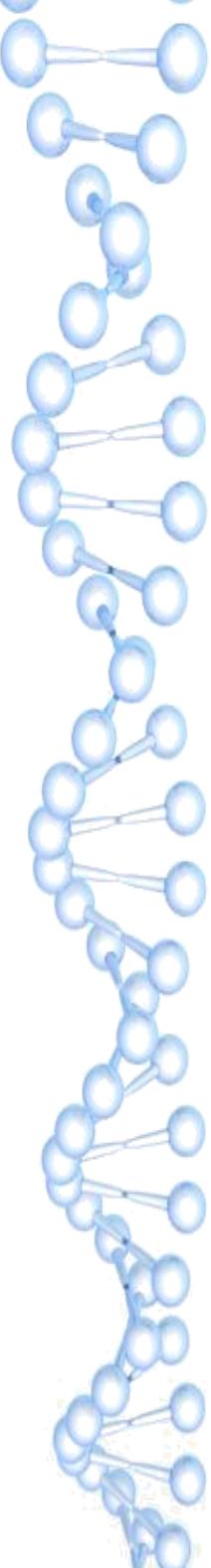
Pour des codes plus complexes, un code plus propre Contourne le problème de *forward reference*



Sans initializer = Inefficace

```
import java.util.*;  
  
public class Person {  
    private final Date birthDate;  
  
    // ici les autres données etc.  
    // NE PAS FAIRE CA!!! C'EST MAL  
    public boolean isBabyBoomer() {  
        // Objets couteux et inutiles  
        Calendar gmtCal =  
  
        Calendar.getInstance(TimeZone.getTimeZone("GMT"));  
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0,  
0);  
        Date boomStart = gmtCal.getTime();  
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0,  
0);  
        Date boomEnd = gmtCal.getTime();  
        return birthDate.compareTo(boomStart) >= 0  
        &&  
                birthDate.compareTo( boomEnd ) < 0;  
    }  
}
```

}



Usage d'un static initializer

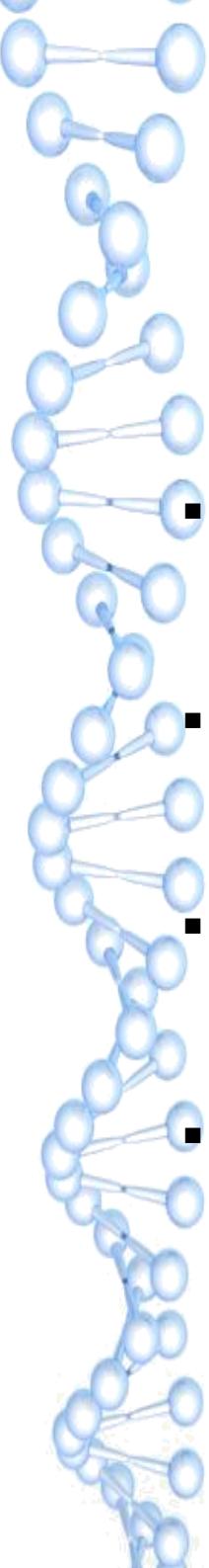
```
class Person {  
  
    private final Date birthDate;  
    private static final Date BOOM_START; private  
    static final Date BOOM_END;  
  
    static {  
        Calendar gmtCal =  
  
        Calendar.getInstance(TimeZone.getTimeZone("GMT"));  
        gmtCal.set(1946, Calendar.JANUARY, 1, 0,  
        0, 0);  
        BOOM_START = gmtCal.getTime();  
        gmtCal.set(1965, Calendar.JANUARY, 1, 0,  
        0, 0);  
        BOOM_END = gmtCal.getTime();  
    }  
  
    public boolean isBabyBoomer() {  
        return birthDate.compareTo(BOOM_START) >=  
        0 &&
```

birthDate.compareTo(***BOOM_END***) <

Destructeur ?

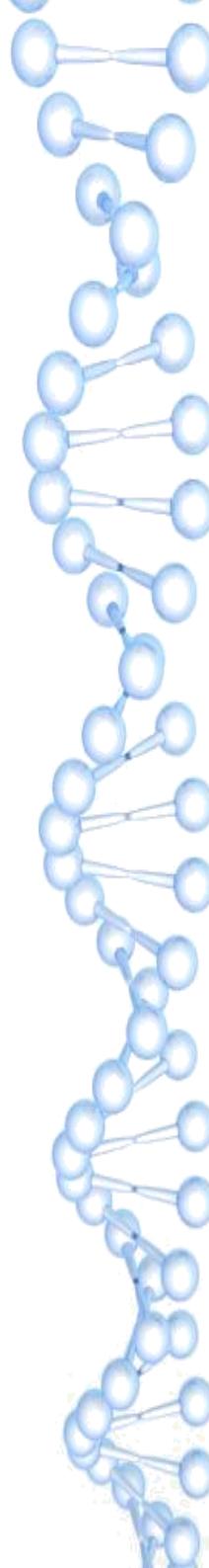
- Des langages permettent d'implémenter un **destructeur** qui est invoqué quand l'on veut détruire l'objet qui sert surtout à 'désallouer'
- Peut aussi servir à décrémenter un compteur d'instance, libérer une ressource, tracer un fonctionnement (log).
- Le **garbage collector** (ramasse-miettes) gère la mémoire :
Les destructeurs n'existent pas en java..
-
- Si du code doit être déclenché au moment où l'objet est libéré, on l'implémente dans la méthode `finalize`.

```
protected void finalize() {  
    System.out.println("détruit !");  
    super.finalize();  
}
```



Usage de finalize()

- la méthode finalize d'une classe doit toujours appeler la méthode finalize de sa super classe
- Si des exceptions peuvent être levées, il faut les rattraper et mettre super.finalize() dans un bloc **finally**.
- Appelé par le GC lorsque celui-ci le détermine (objet orphelin) donc asynchronisme avec la disparition de l'objet.
- Ne JAMAIS baser la suite du code sur inexécution immédiate de finalize()
-



Mémoire(s) & Nettoyage de la Mémoire

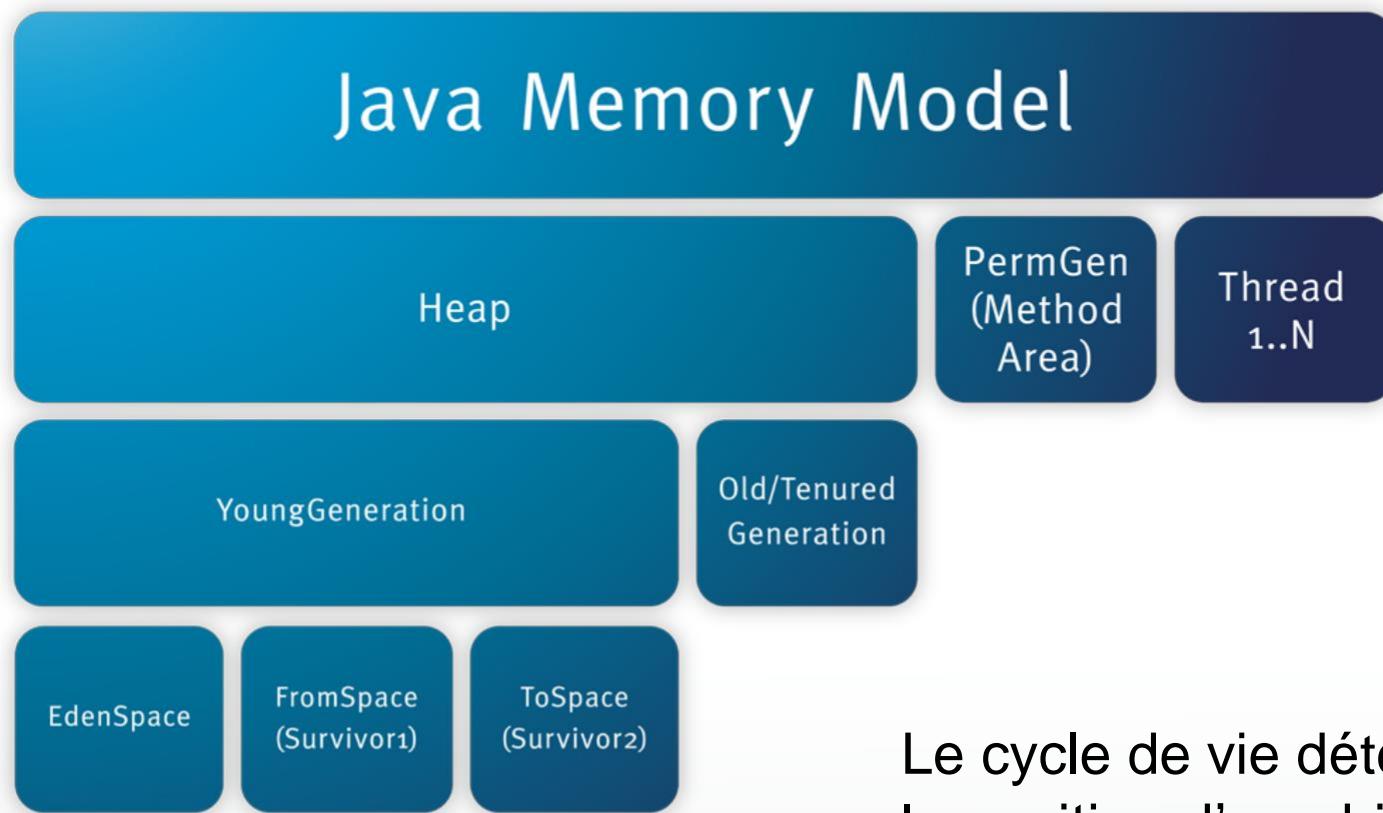
JVM et sa mémoire

- En résumé les primitives, objets et données de classe sont stockés dans 3 zones distinctes:
 - **HEAP space** : un ‘tas’ qui stocke les données objets
 - **Method area** : code des classes
 - **Native area** : le référentiel code et objet (la colle!!)

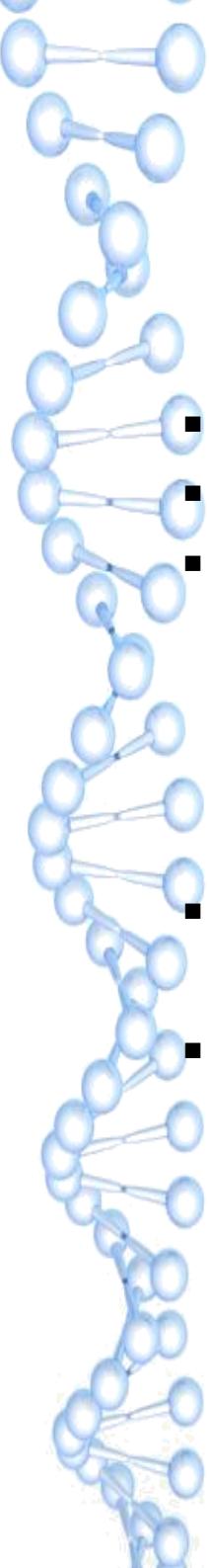
Heap Space					Method Area		Native Area				
Young Generation				Old Generation		Permanent Generation		Code Cache			
Virtual	From Survivor 0	To Survivor 1	Eden	Tenured	Virtual	Runtime Constant Pool	Thread 1..N		Compile	Native	Virtual
Virtual	From Survivor 0	To Survivor 1	Eden	Tenured	Virtual	Field & Method Data	Virtual	PC	Stack	Native Stack	Virtual
Virtual	From Survivor 0	To Survivor 1	Eden	Tenured	Virtual	Code	Virtual	PC	Stack	Native Stack	Virtual

HEAP : Les données

Puisque les informations objets sont stockées ici, regardons de plus près.



Le cycle de vie détermine la position d'un objet



HEAP

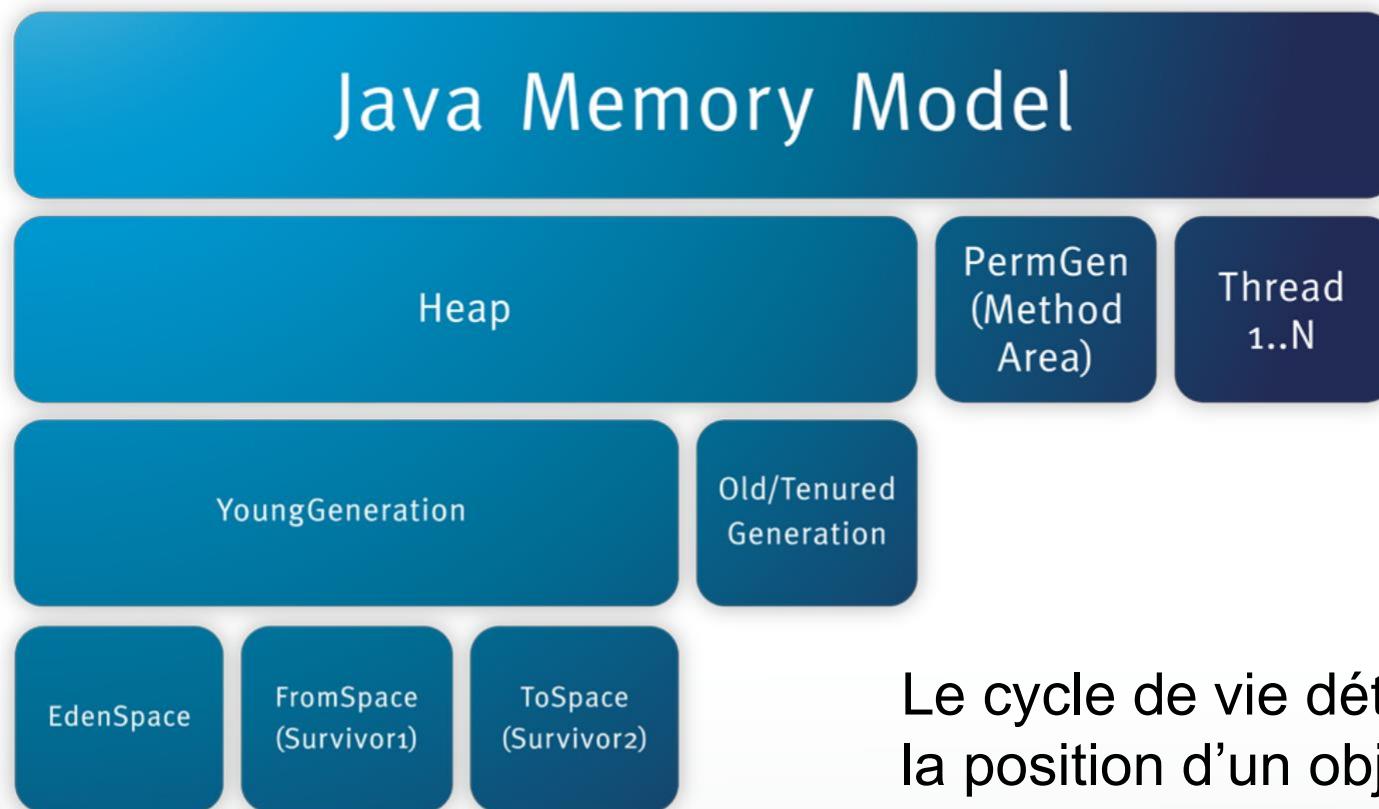
- Eden: Les données objets qui viennent juste d'être créées
- Survivor : Ces objets ont survécu au Garbage Collector d'Eden
- Tenured Generation: Ceux qui existent depuis déjà pas mal de temps en Survivor (les veterans)

Pas HEAP

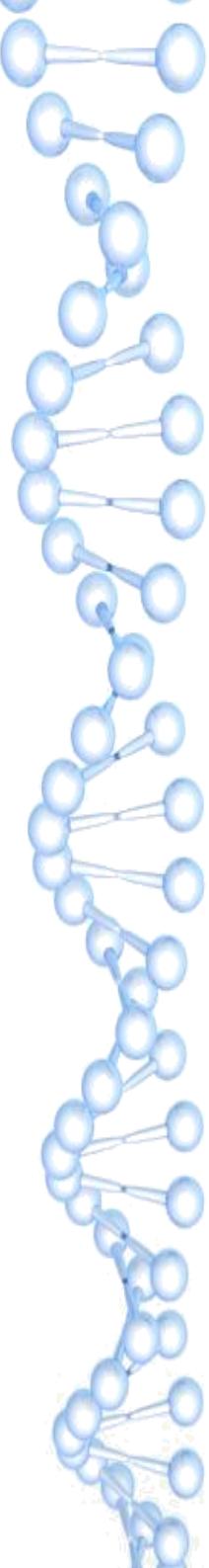
- Permanent Generation (PermGen): Les classes, les méthodes. Il y a des zones en lecture seule, d'autres lecture-écriture
- Code Cache : L'accélérateur HotSpot y stocke des caches, des données issues de code natif

HEAP : Les données

Puisque les informations objets sont stockées ici, regardons de plus près.

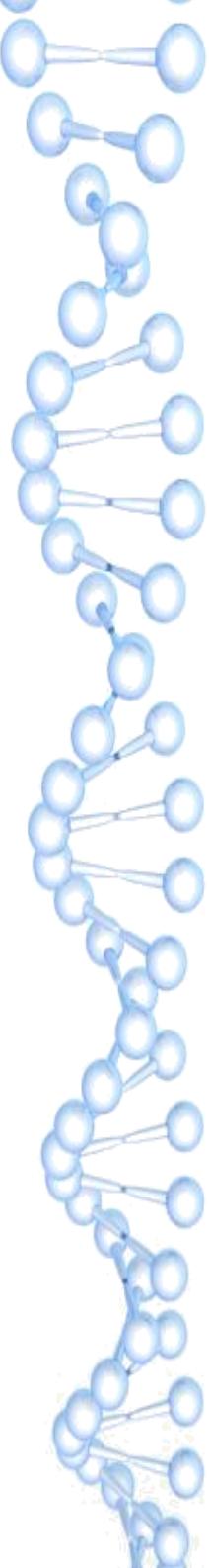


Le cycle de vie détermine la position d'un objet choisié par le Garbage Collector (ou plutôt les GCs)



Garbage Collector

- Détermine si un objet n'est plus utilisé dans l'application
- Le supprime de la mémoire.
- Un objet peut être définitivement supprimé lorsque l'application n'a plus aucun moyen d'y accéder.
 - Par exemple à la sortie d'une fonction, lorsqu'une variable locale est utilisée pour référencer l'objet.
 - Affectation de la valeur null à une variable.
- Si un objet est stocké dans une collection ou un tableau, la collection ou tableau, conserve une référence vers l'objet.



Garbage Collector

Quand ?

La machine virtuelle surveille les ressources mémoires disponibles et provoque l'entrée en action du Garbage Collector lorsque celles-ci ont atteint un seuil limite (défaut à 85 %).

System.gc(), Runtime.gc() :

→ Seulement des suggestions qui demandent à la JVM de lancer le Garbage Collector

Quand une référence est placée à null et que de ce fait l'objet référencé devient 'orphelin' , il est éligible pour être collecté au prochain cycle

Fuite mémoire, mais où ?

Avec cette architecture en tête on peut mieux comprendre les exceptions et les messages telles que :

Exception in thread “main”: java.lang.OutOfMemoryError: Java heap space

Raison: Un objet ne peut être alloué faute de place dans le Heap Space.

Exception in thread “main”: java.lang.OutOfMemoryError: PermGen space

Raison: Une classe ou des méthodes ne peuvent être chargées dans le PermGen. Cette application nécessite sûrement beaucoup d'espace mémoire ou des librairies tiers nombreuses

.

Exception in thread “main”: java.lang.OutOfMemoryError: Requested array size exceeds VM limit

Raison: Une Array est remplie qui dépasse la taille du Heap.

Fuite mémoire, mais où ?

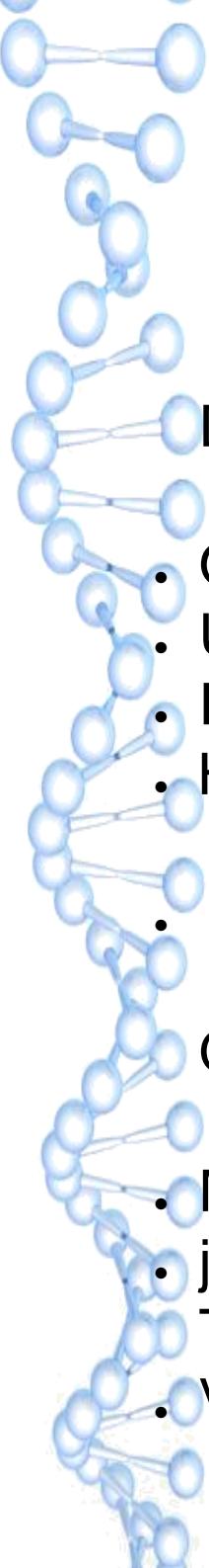
Exception in thread “main”: java.lang.OutOfMemoryError: request bytes for . Out of swap space?

Raison :Cela se produit lors d'une demande d'allocation échoue ou atteint les limites. Le module natif est indiqué

Exception in thread “main”: java.lang.OutOfMemoryError: (Native method)

Raison: Cette erreur provient d'un appel natif plutôt que de la JVM elle-même.

On va pouvoir se lancer dans la correction de la fuite mémoire ; quelques conseils et un bon outil peuvent s'avérer utiles :



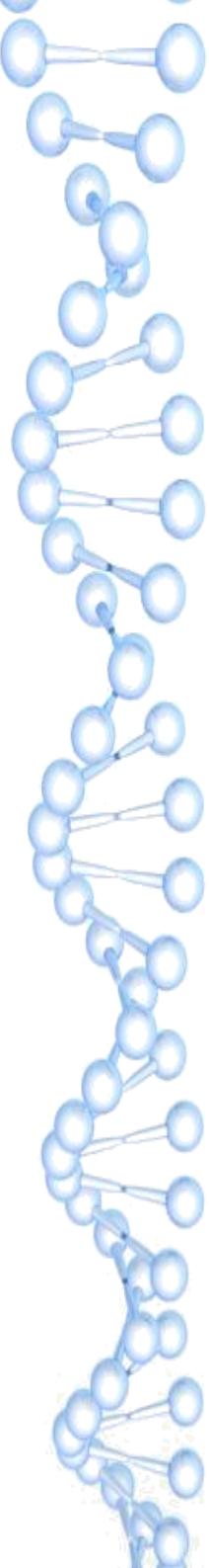
Stopper la fuite

Des pistes de recherches :

- Oubli de fermer une ressource (File, Stream, DB)
- Une référence à un objet n'est pas relachée
- File/Text/Connexion buffers non fermés.
- Hash maps conserve les références if equals() et
hashcode() ne sont pas implémentés (pour la clé)
- Inner classes utilisées à outrance

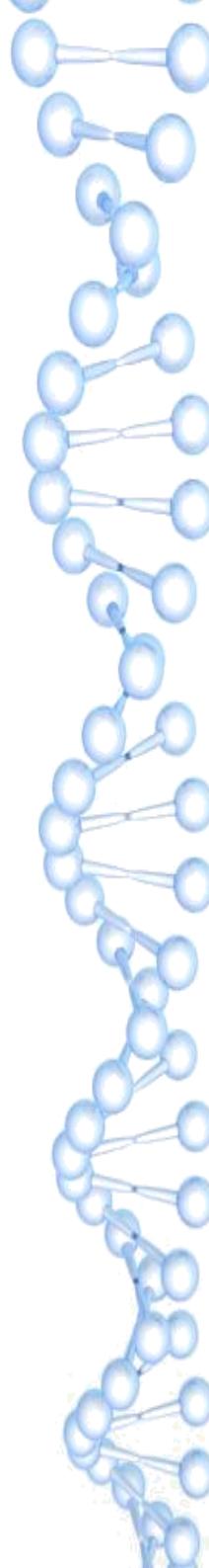
Quelques outils :

- Memory Analyzer (MAT)
- jvisualvm - Java Virtual Machine Monitoring,
Troubleshooting, and Profiling Tool
- VisualVM

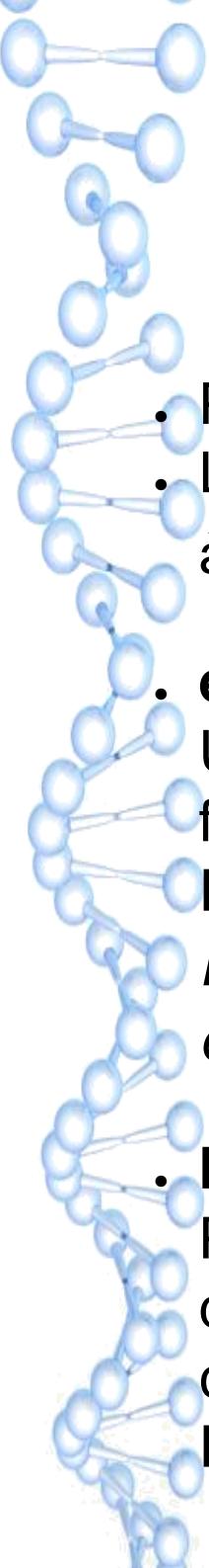


Mise en Application

Recherche de fuite mémoire



Remarkable Object



Object

- Présente dans la hiérarchie de toute classe d'une application.
- Les méthodes définies dans la classe Object sont disponibles à n'importe quelle classe. Regardons de plus près...
- **equals**
 - Utilisé pour comparer deux instances de classe de façon sémantique.
Par défaut : effectue une comparaison des références
Deux objets identiques sont égaux, mais deux objets égaux ne sont pas forcément identiques.
- **hashCode**
 - Permet d'obtenir une valeur hachée (un entier sur 32 bits) qui servira dans certaines parties du JDK , notamment des collections.
Par default : adresse mémoire de l'Object (int native)

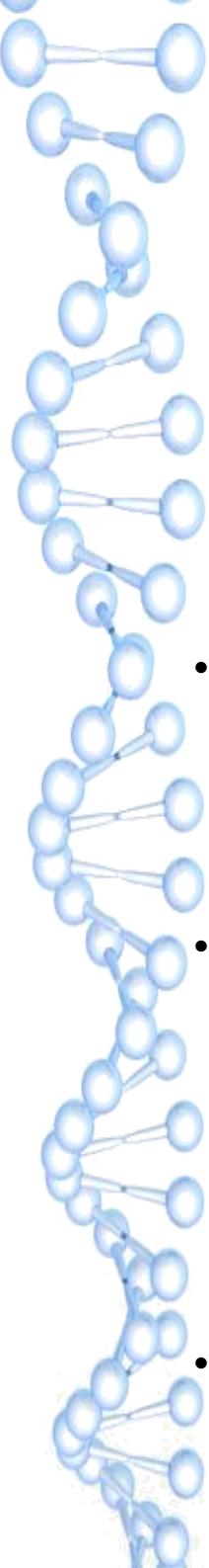
Object, equals & hashCode

equals() et **hashCode()** sont fortement liés, leur redéfinition doit TOUJOURS répondre à ces 5 règles !

- Symétrie : si `a.equals(b)` alors `b.equals(a)`
- Réflexivité : `a.equals(a)` vrai (supposé `a!=null`)
- Transitivité : `a.equals(b)`, `b.equals(c)` alors `a.equals(c)`
- Consistance : si `a.equals(b)`, `a.hashCode() == b.hashCode()`
- Pour toute `a!=null` : `a.equals(null)` est false

Usage de hashCode()

- Dans les collections de type Hashxxx (`HashTable`, `HashMap`, `HashSet`) pour éviter les clusters
- Pour améliorer les performances , JVM utilise par exemple dans des clauses ‘switch’



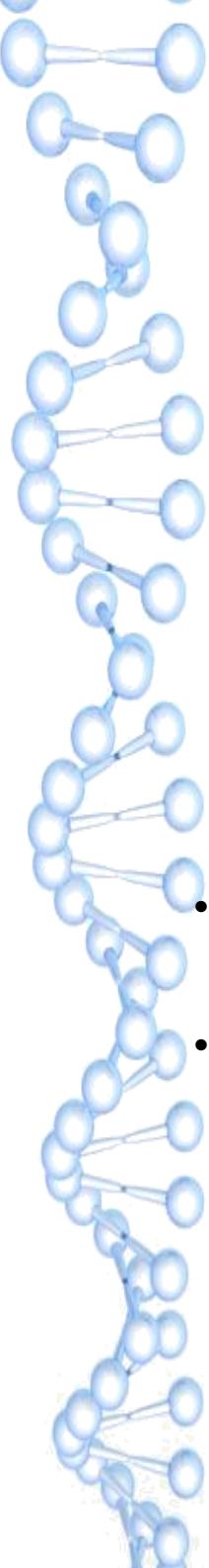
Bâtir un Clone

clone()

Crée et retourne une copie en dupliquant les propriétés.

On peut redéfinir `clone()` :

- Copie de surface (shallow copy) : toutes les valeurs des propriétés de type primitif sont copiées dans leurs propriétés correspondantes.
 - propriétés de type objet, **leurs références** sont copiées.
- copie profonde (deep copy) : toutes les valeurs des propriétés de type primitif sont copiées dans leurs propriétés correspondantes.
 - Propriétés de type objet, copie de l'objet qui est associée à la propriété de la copie. Cette copie respecte l'encapsulation.
- ATTENTION : la copie profonde est non triviale
(pensez référence circulaire).



Bâtir un Clone

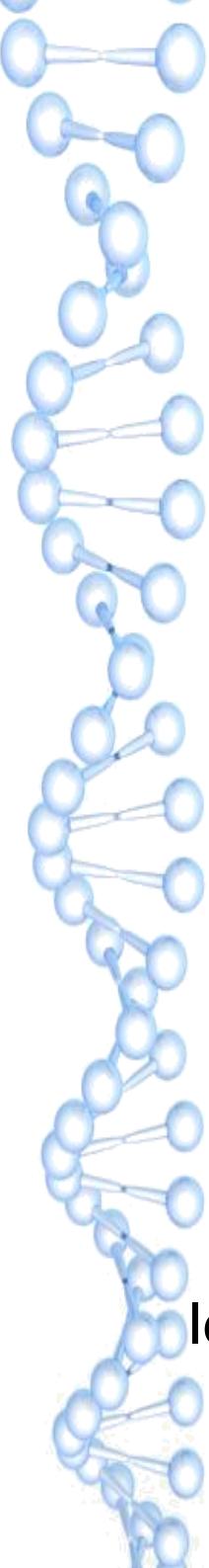
protected void **finalize()** ← vu

précédemment **getClass()**

Renvoie la classe de l'instance ; souvent utile pour
equals() ou en préparation d'un (**cast**) d'objet

À NOTER

- Cette méthode `getClass()` ne permet une comparaison que pour des objets de même type
- L'opérateur `instanceof` permet une comparaison entre instances d'une classe ou de ses classes filles



Object & threads 85

String `toString()`

Un string qui représente l'objet : très utile notamment en debug ou pour sérialiser rapidement un objet

void `notify()`

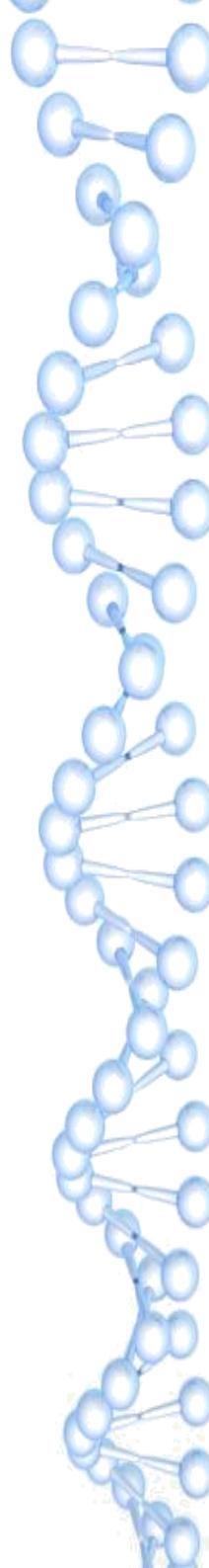
Reveille de tous les threads en attente du signal de cet objet.

void `notifyAll()`
Reveille de tous les threads en attente du signal de cet
objet.

wait()
Mettre en pause un thread jusqu'à ce qu'un autre
thread appelle `notify()` ou `notifyAll()`.

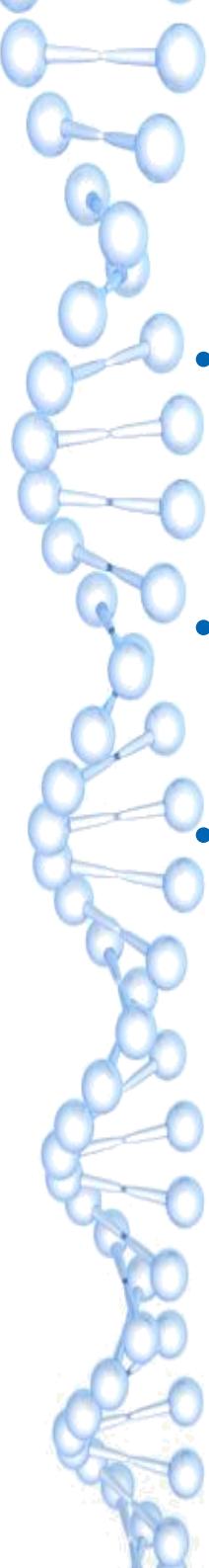
wait(long timeout)
Idem, mais avec un timeout en
secondes

wait(long timeout, int nanos)
Idem, mais avec un timeout en secondes et en millisecondes.



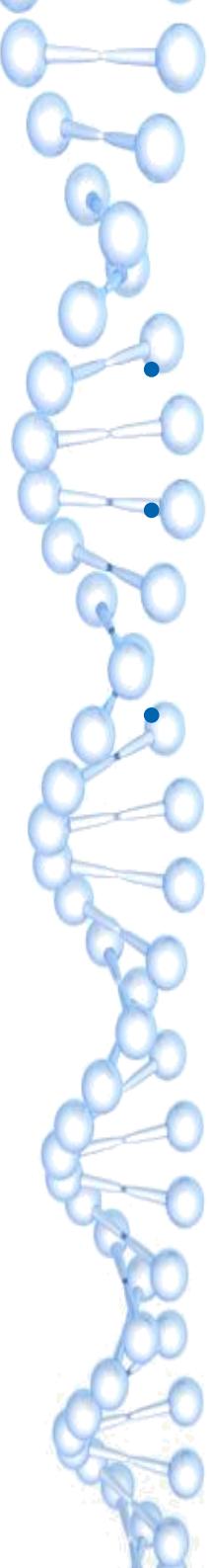
Mise en Application

Redéfinition, Utilisation des méthodes Object



La sérialisation d'Objets

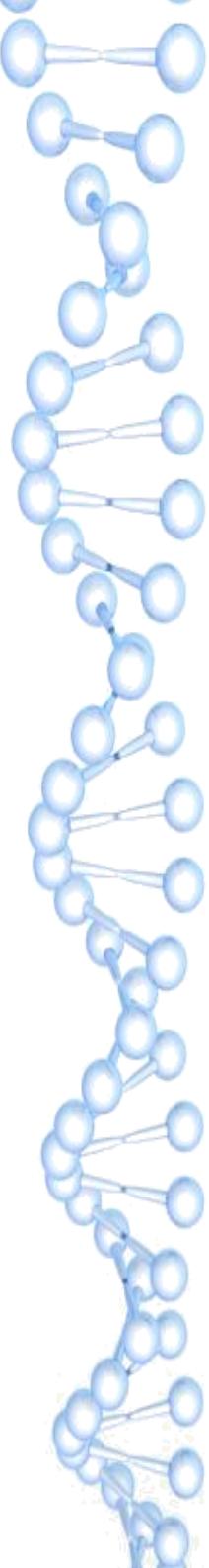
- Lorsque l'on souhaite sauvegarder l'état d'un objet, ou bien véhiculer un objet sur le réseau, il faut **transformer** cet objet en **une suite d'octets** et être en mesure d'effectuer l'opération inverse
- Java propose un mécanisme sophistiqué de transformation d'objets en un flux d'octets, que l'on nomme sérialisation, la désérialisation étant l'opération inverse
- Ce mécanisme est exploité notamment afin de rendre des objets persistants, ainsi que dans le RMI (Remote Method Invocation) pour transmettre un objet à une autre machine sur le réseau



Méthode de sérialisation

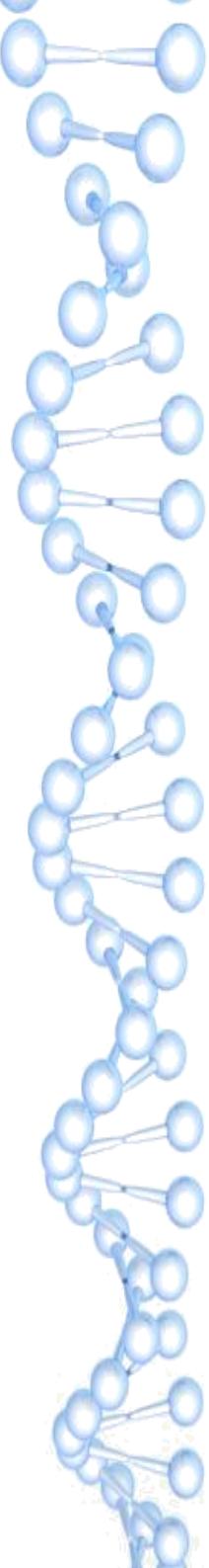
- D'abord obtenir une instance d'***ObjectOutputStream*** en précisant le flux destinataire
- Il faut ensuite invoquer la méthode ***writeObject*** de ***ObjectOutputStream*** en lui passant l'objet à sérialiser en argument
- La méthode ***writeObject*** ne s'applique qu'à des objets qui implémentent l'interface ***Serializable***, laquelle ne comporte aucune méthode!

```
class Personne implements Serializable {  
    . . .  
    //...  
    . . .  
}
```



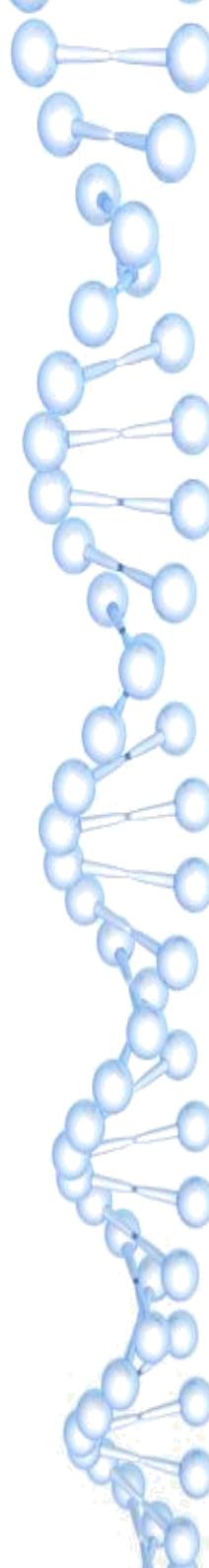
Exemple de sérialisation

```
class PersSerializer {  
    public static void main(String[] args) {  
        Personne martin=  
            new Personne("MARTIN", "Pierre", 40);  
        Personne durand=  
            new Personne("DURAND", "Martine", 25); try  
        {  
            FileOutputStream ficser=new  
                FileOutputStream("personne.ser");  
            ObjectOutputStream ser=new  
                ObjectOutputStream(ficser);  
            .  
            .  
            .  
            ser.writeObject(martin);  
            ser.writeObject(durand);  
        } catch( Exception e) { .... }  
    }  
}
```

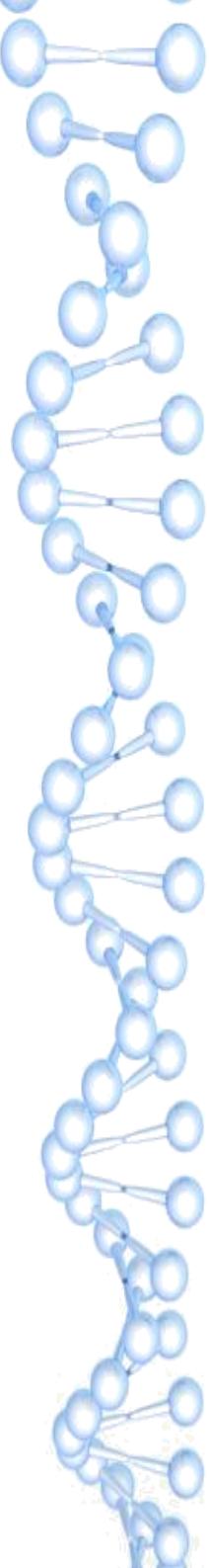


La sérialisation d'objets

- La relecture d'objets sérialisés demande simplement d'invoquer la méthode ***readObject*** pour une instance de ***ObjectInputStream*** qui représente le flux duquel proviennent les objets sérialisés
- La méthode ***readObject*** retourne un objet de type ***Object*** qu'il faut donc transtyper dans le type adéquat avant utilisation



Interfaces Remarquables

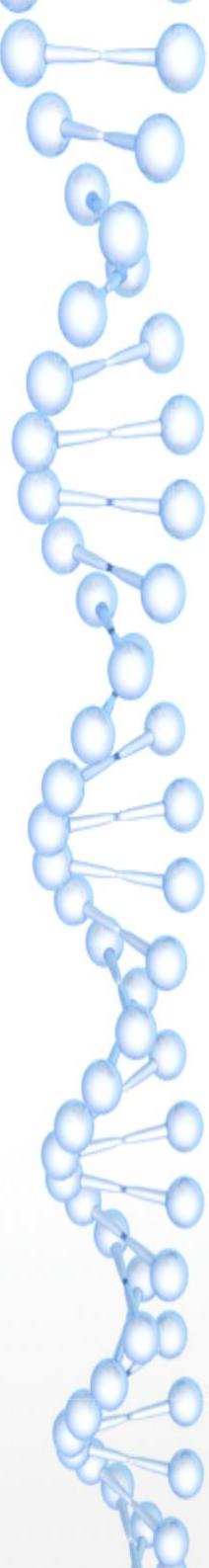


L'interface Comparable

L'interface Comparable, nécessite l'implémentation de la méthode compareTo (qui doit retourner 1, -1 ou 0):

```
interface Comparable  
    int compareTo(Object o);  
}
```

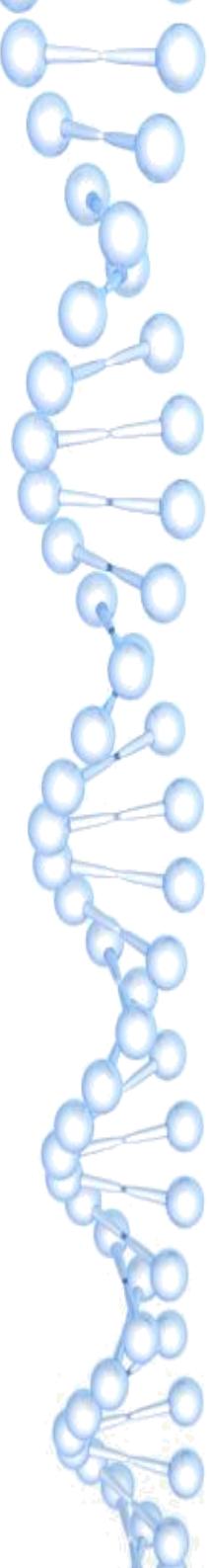
Largement utilisée dans les fonctions de tri et de recherche, elle permet d'externaliser la logique.



L'interface Comparable

```
public abstract class Surface implements Comparable<Surface>{
    public abstract double getArea();

    @Override
    public int compareTo(Surface o) {
        if (getArea()>o.getArea()){
            return 1;
        }else if (getArea() < o.getArea()){
            return -1;
        }
        return 0;
    }
}
```

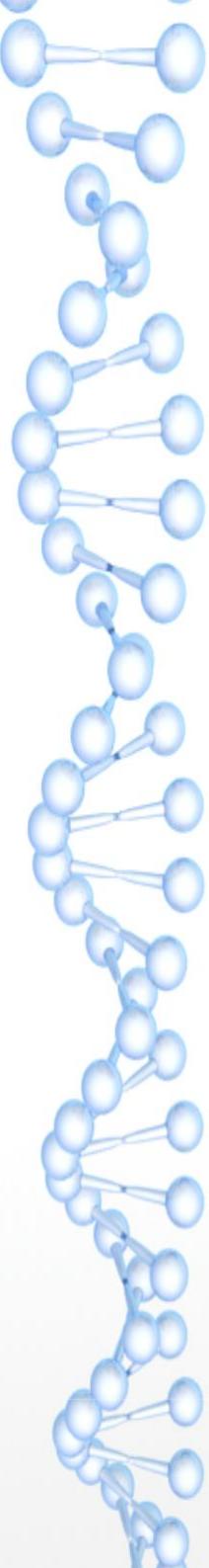


L'interface Cloneable

Une classe qui implémente l'interface Cloneable, indique à Object.clone() qu'il est légal pour cet objet de faire une copie attribut par attribut pour les instances de cette classe.

Cela permet de faire une ‘shallow copy’ une ‘deep copy’ est possible en redéfinissant la méthode clone().

Attention aux références cycliques lors
d'un clonage ?



L'interface Iterator

```
import java.util.Iterator;

public class ChannelIterator implements Iterator<Channel> {
    private List<Channel> channels;
    private int position;
    private ChannelTypeEnum type;

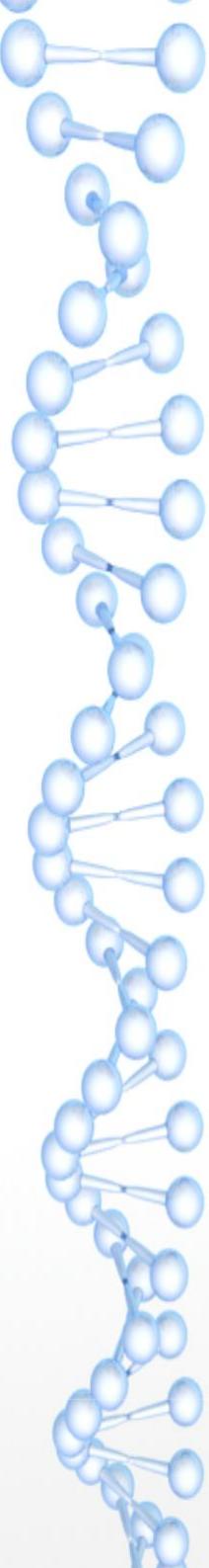
    public ChannelIterator(ChannelTypeEnum type, List<Channel> channels) {
        super();
        this.channels = channels;
        this.type = type;
    }

    @Override
    public boolean hasNext() {
        while (position < channels.size()) {
            Channel c = channels.get(position);

            if (c.getTYPE().equals(type)
                || type.equals(ChannelTypeEnum.GENERAL)) {
                return true;
            } else
                position++;
        }
        return false;
    }

    @Override
    public Channel next() {
        Channel c = channels.get(position);
        position++;
        return c;
    }
}
```

- ## Méthodes
- `hasNext()`
 - `next()`
 - `remove()`

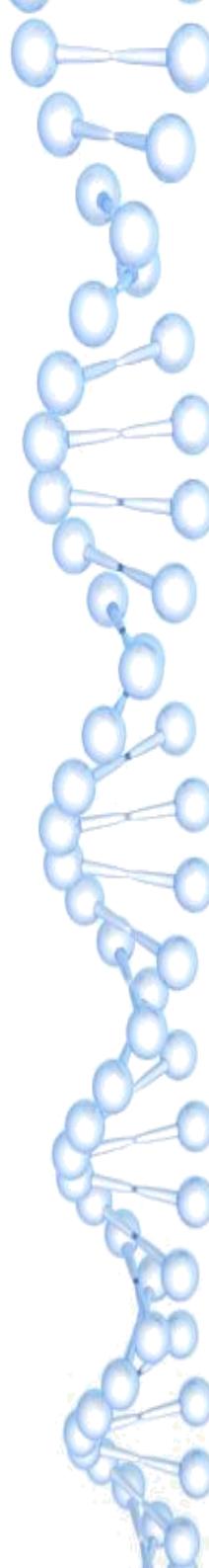


L'interface Iterator

```
private static void testChannels() {  
    ChannelCollection channels = generateChannels();  
    Iterator<Channel> iterator = channels.iterator(ChannelTypeEnum.NEWS);  
    while (iterator.hasNext()) {  
        Channel channel = iterator.next();  
        System.out.println("Iterator java "+channel.toString());  
    }  
}
```

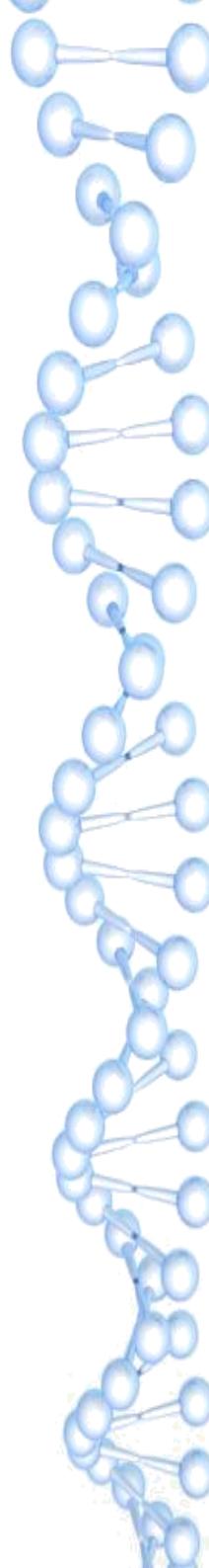
Encore une fois on externalise la logique , on rend interchangeable la manière de se déplacer dans une collection :

Alphabétique, Numerique, Aléatoire, selon un ou plusieurs critères.
On peut même réfléchir à un Iterator multi critères combinés.

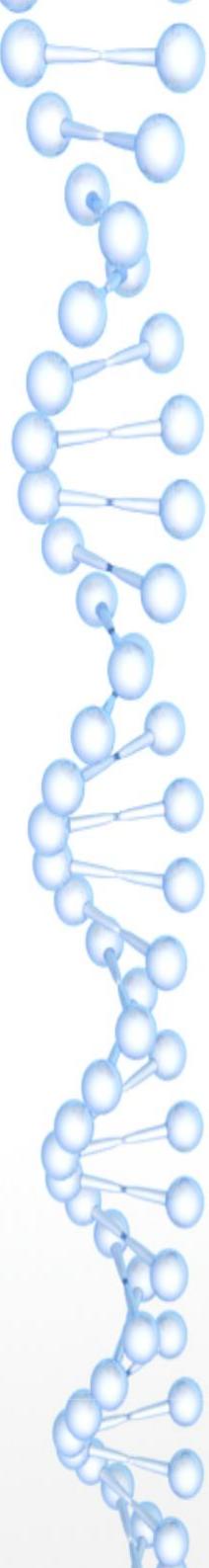


Applications

Après avoir étudié les Collections



Generics et Collections



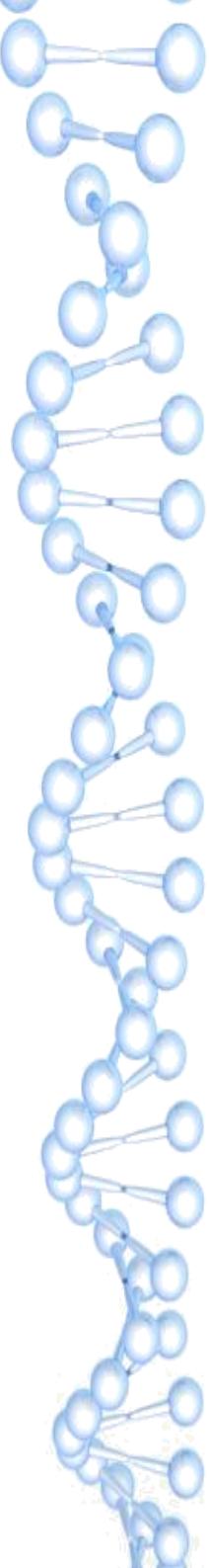
Du pseudo-générique

Besoin de casting

Certaines erreurs détectées uniquement

l'exécution très limitée dans les possibilités

```
public class Paire {  
    Object premier ;  
    Object second;  
    public Paire (Object a, Object b){  
        premier = a; second = b;  
    }  
    public Object getPremier(){  
        return premier;  
    }  
    public Object getSecond(){  
        return second;  
    }  
}  
  
Paire p = new Paire ("abc", "xyz");  
String x = (String)p.getPremier(); // le casting est obligatoire  
Double y = (Double)p.getSecond(); // Il faut attendre l'exécution pour avoir  
// une levée d'exception (ClassCastException)
```

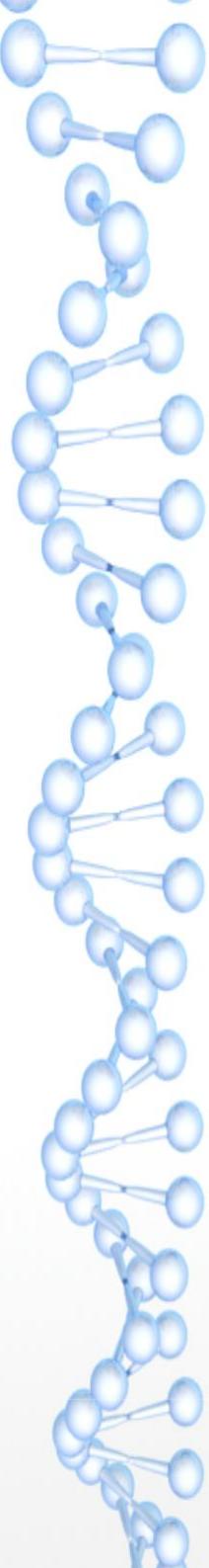


Généricité, c'est quoi ?

La généricité permet de rendre un programme plus **stable** en **détectant certaines erreurs dès la compilation.**

La généricité est très utilisée avec les collections, mais peut être aussi utilisée dans n'importe quelle classe.

➤ Nous allons dans un premier temps parler de généricité sans collection en retravaillant la classe Paire



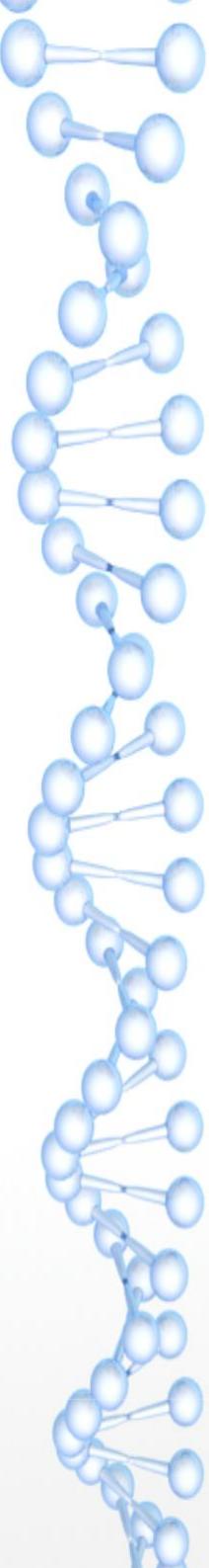
Généricité, c'est quoi ?

On limite l'utilisation du casting (pas intuitif)

Erreurs détectées à la compilation

Code peut fonctionner avec toute classe

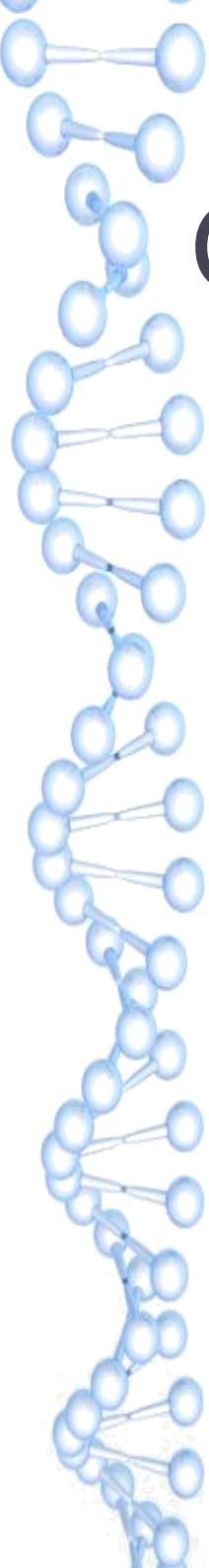
```
public class Paire<T> {  
    T premier ;  
    T second;  
    public Paire (T a, T b){  
        premier = a; second = b;  
    }  
    public T getPremier(){  
        return premier;  
    }  
    public T getSecond(){  
        return second;  
    }  
}  
  
Paire<String> p = new Paire<String> ("abc", "xyz");  
String x = p.getPremier(); // pas de cast  
Double y = p.getSecond(); // erreur de compilation (type mismatch)
```



Et les méthodes aussi

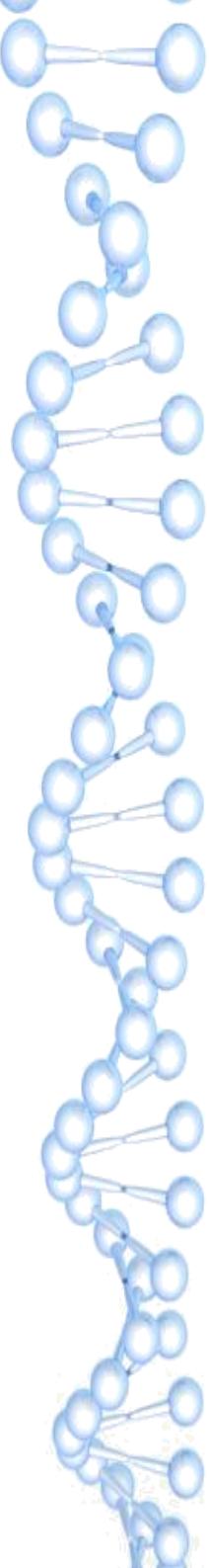
Pouvant manipuler des classes génériques, il apparait souhaitable de créer des méthodes génériques

```
public class X {  
    public void affiche(Paire<?> p){  
        System.out.println(p.v1 + " " + p.v2);  
    }  
  
    public static void main(String [] a){  
        Paire<String> ps = new Paire<String>("un", "deux");  
        Paire<Integer> pi = new Paire<Integer>(1, 2);  
        X x = new X();  
        x.affiche(ps);  
        x.affiche(pi);  
    }  
}
```



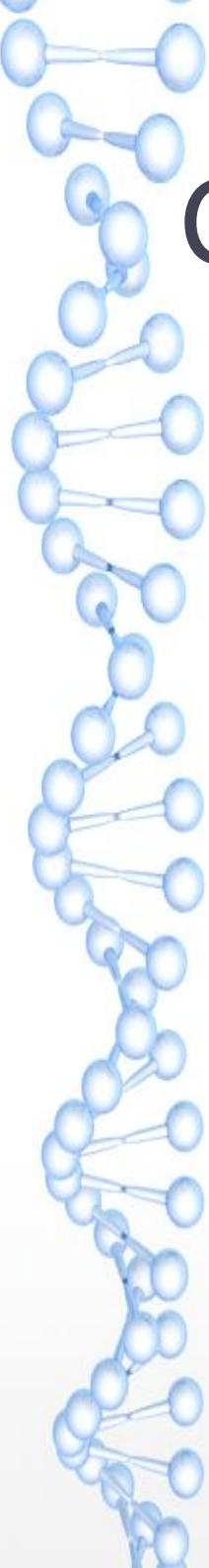
Classes génériques

- Une **classe générique** est une classe **paramétrée** par une ou plusieurs autres classes
 - Syntaxe générique : MaClasse<T1, T2, T3, ...>
 - Une liste d'entiers : List<Integer>
 - Un ensemble de mots : Set<String>
 - Un annuaire : Map<Personne, NumeroTelephone>
 -
- Évite de coder ListeEntier, ListeString
 - Les méthodes d'une classe génériques doivent **s'affranchir du(des) type(s) passé(s)** en paramètre...
 - En général une classe générique sert de container.
 - **Attention :** Liste<Fille> n'hérite pas de Liste<Mère>



Le choix du générique

- Si un algorithme ne dépend pas directement du type du contenu, on peut le coder pour importe quel contenu :
 - Chercher un élément dans une liste générique
 - Trouver un plus court chemin dans un graphe générique
 - Trier des éléments sur la base de compareTo()
- Les classes génériques permettent de **factoriser** le code
 - Gain de temps
 - Maintenance plus aisée
 - Design plus constraint donc plus rigoureux / sûr
 - On n'est jamais sûr de trouver l'objet que l'on attend dans une Collection.



Contraintes

Pour la classe générique, il n'y a pas d'héritage induit par les classes qu'elle utilise

```
public class Super { ... }

public class Sous extends Super { ... }

Paire<Super> pSup = new Paire<Super>(new Super())
; Paire<Sous> pSous = new Paire<Sous>(new Sous())
; Psup = pSous // INTERDIT
```

Ce sont bien deux classes différentes qui sont compilées.

Contraintes

Une méthode peut manipuler un ou plusieurs paramètres / variables génériques.
On peut contraindre la nature de ces types génériques.

```
// list1 doit contenir des instances de Voiture (ou
// des instances de classes qui héritent de Voiture
// telle que Monospace, Cabriolet...)
List<Voiture> list1;

// list2 doit contenir des instances de Voiture ou
// d'une classe fille de Voiture
// (Cabriolet, Monospace...)
List<? extends Voiture> list2;

// list3 doit contenir des instances de classe
// mère // de Voiture (Véhicule...)
List<? super Voiture> list3;

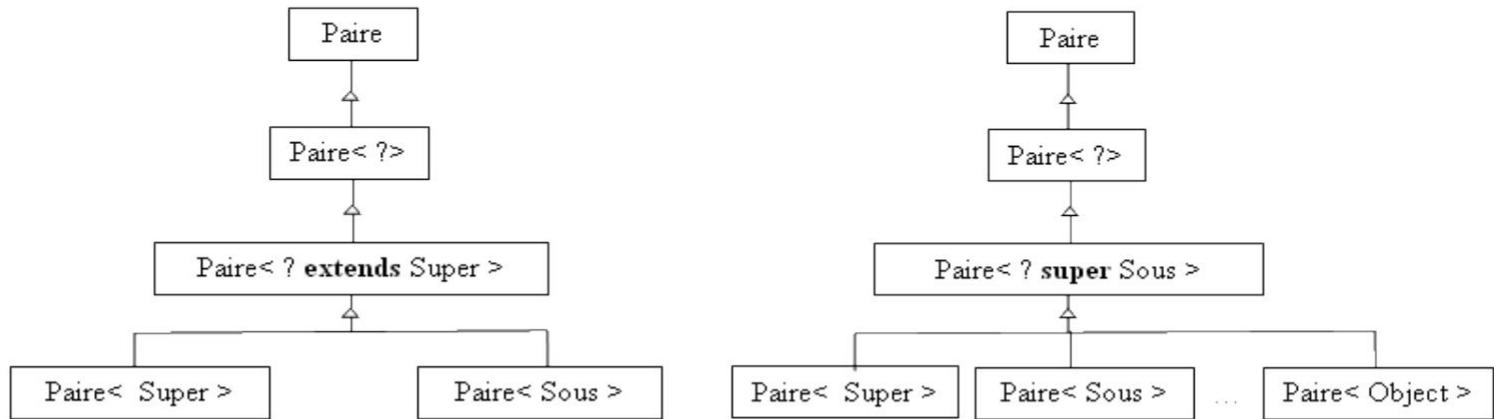
// list4 peut contenir n'importe
// quoi List<?> list4;
```

Attention : utiliser '?' verrouille le paramètre en lecture seule.

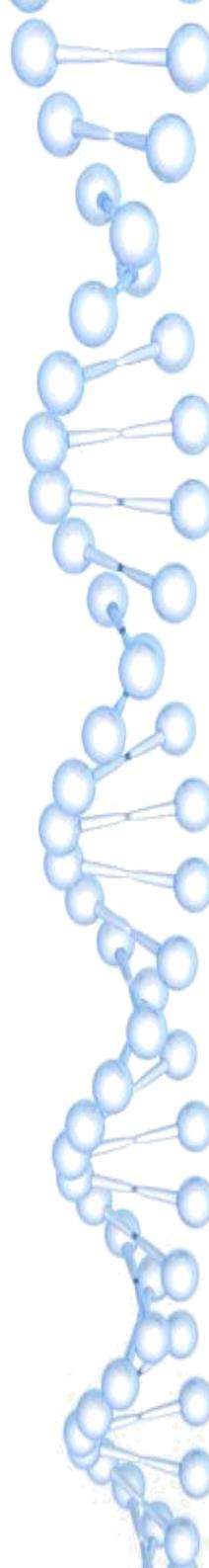
Contraintes

Les jokers peuvent être limités vers le haut (`extends`) ou vers le bas (`super`).

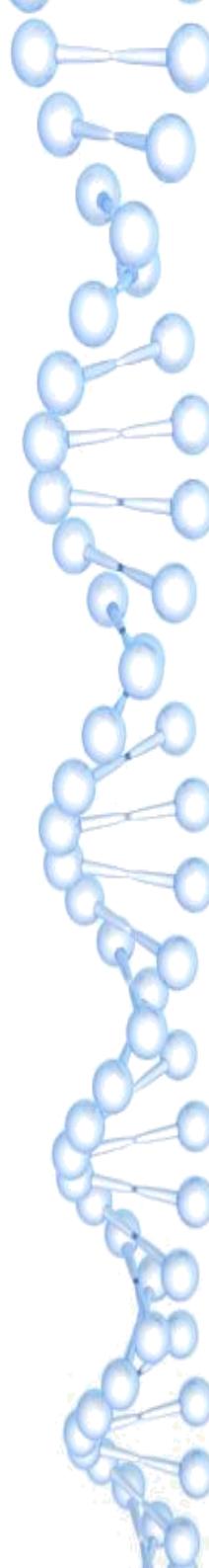
On a les relations d'héritage suivantes :



- La relation d'héritage entre `Paire` et `Paire<?>` est là pour assurer la compatibilité avec les codes existants avant la version 5 de Java.

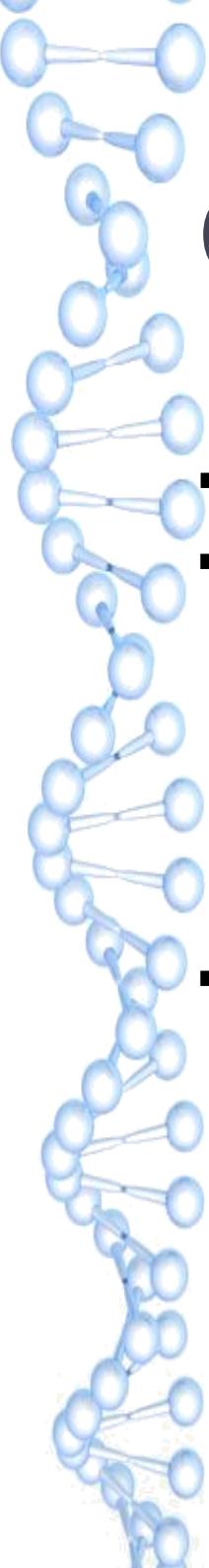


Mise en Application des Génériques



Collections

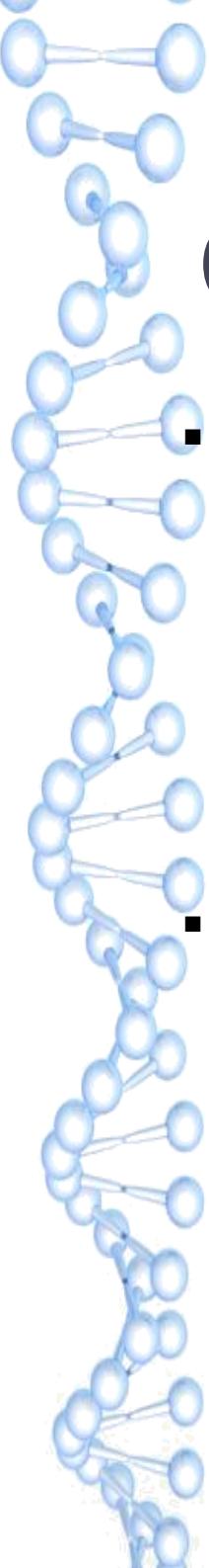
https://en.wikibooks.org/wiki/Java_Programming/Collections



Containers de données

Java fournit des containers autres que les tableaux

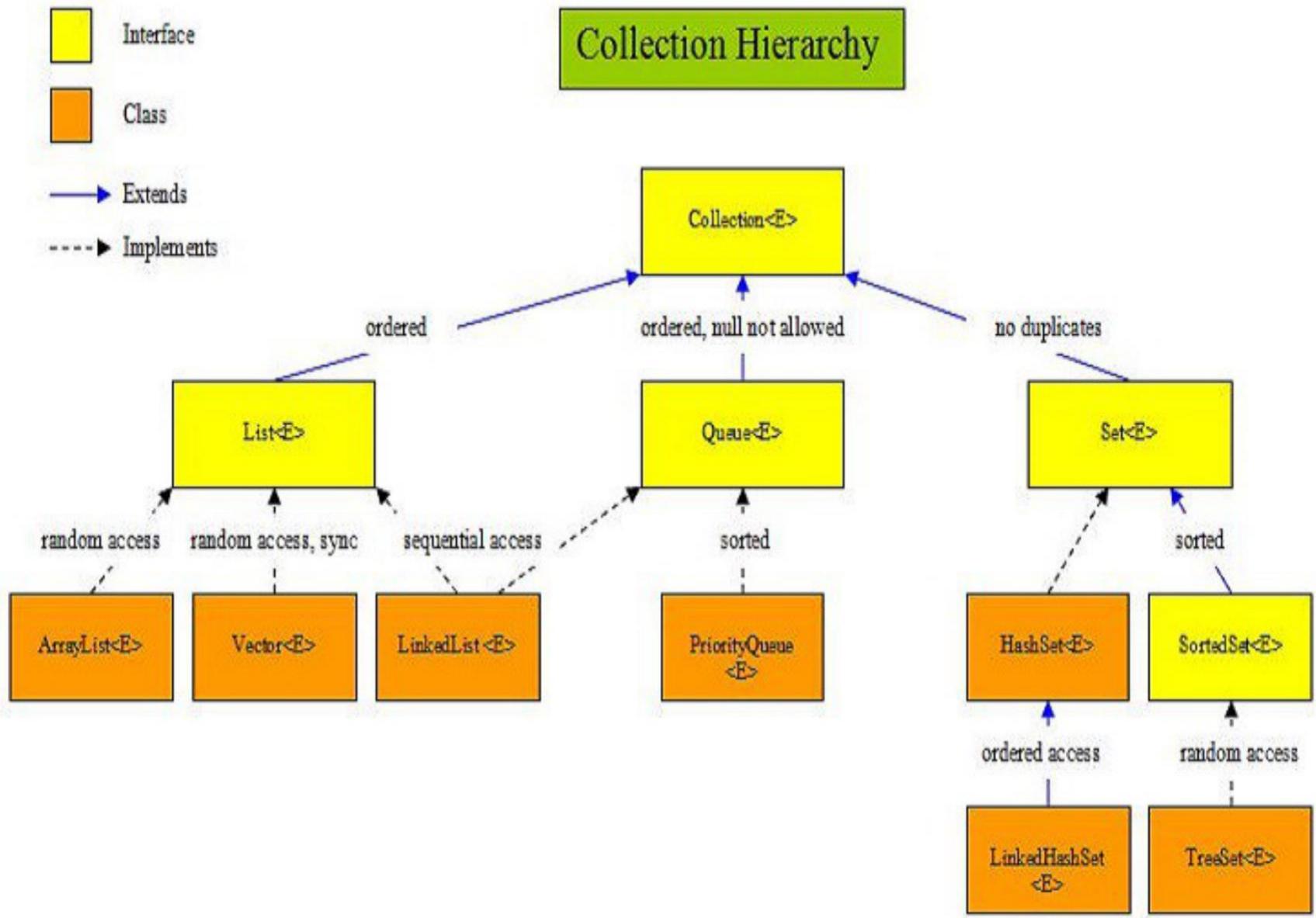
- Principaux containers
 - LinkedList : référence sur le maillon suivant (**null** sinon)
 - ArrayList : liste dans un tableau
 - HashSet : ensemble d'éléments uniques
 - HashTable : fait partie des Maps (associe une clé à une valeur)
- Avantages / inconvénients :
 - Taille dynamique
 - Possibilité d'itérer dessus : avec un index ; un objet Iterator ; un objet Enumeration (selon le container).
 - Pas de contrainte sur les éléments dans le container... comment faire ?

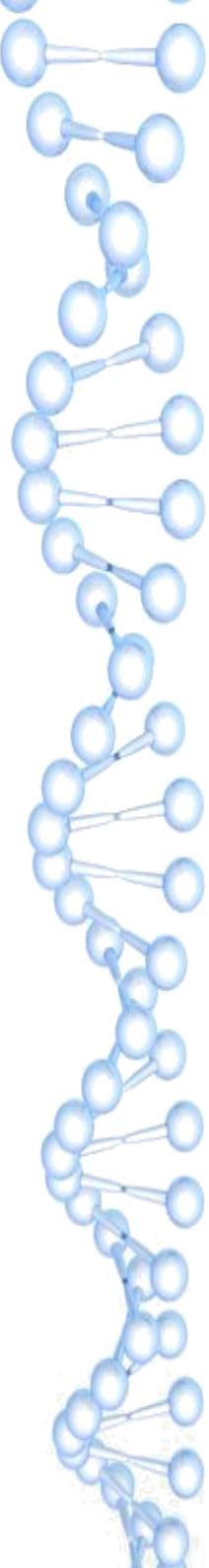


Containers génériques

- Java fournit depuis le JDK1.5 des containers génériques :
 - Interface (type manipulé dans un prototype) :
 - List<T>, Set<T>, Map<K,V>
 - Permet de s'affranchir de l'implémentation.
 - Implémentation (type instancié) : HashSet<T>, ArrayList<T>,
 - ...
 - ...
 - Avantages :
 - Algorithmes génériques (contains, ...)
 - Les **itérations** sont plus simples à mettre en place (nous les verrons bientôt) :
 - Avant JDK 5 : Iterator + Enumeration
 - Depuis JDK 5 : à l'aide de l'opérateur :

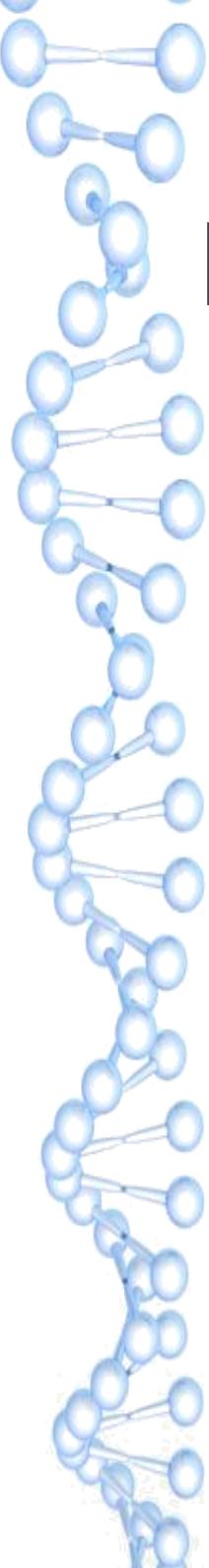
Hiérarchie des données





Interface Collection

- `java.util Interface Collection<E>`
- `boolean add(E e)`
- `boolean addAll(Collection<? extends E> c)`
- `void clear()`
- `boolean contains(Object o)`
- `boolean containsAll(Collection<?> c)`
- `boolean isEmpty()`
- `Iterator<E> iterator()`



List

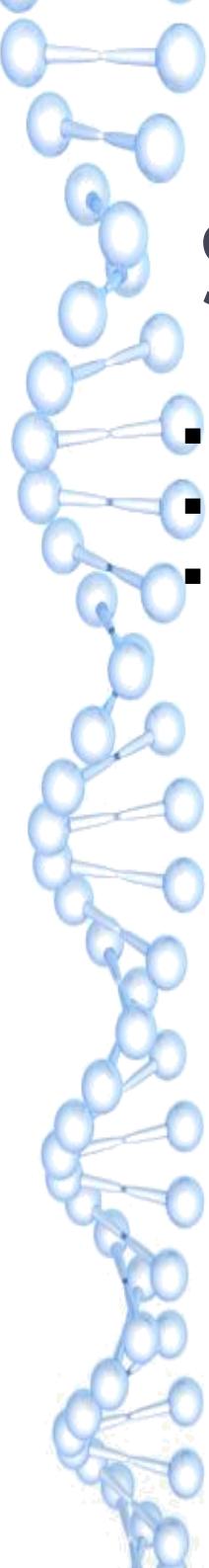
Interface List<E>

- **ArrayList**

Méthodes add et addAll. Par défaut, ces deux méthodes ajoutent les éléments à la suite de ceux déjà présents.

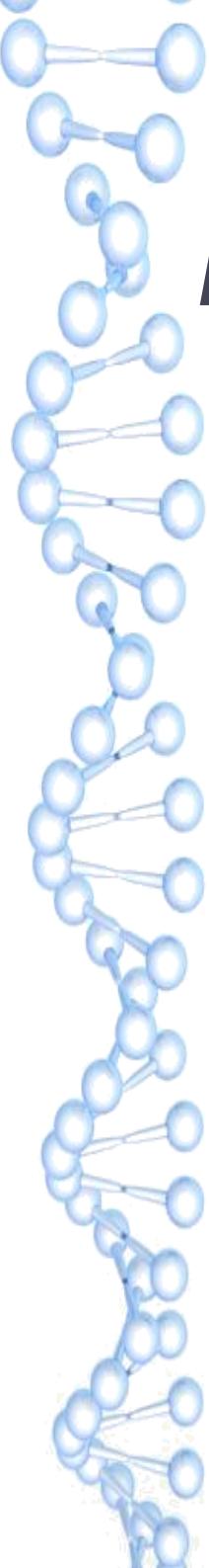
- **LinkedList**

- très complète : implémente les interfaces **Iterable**, **Collection**, **Deque**, **List** et **Queue**.
- Adaptée pour des éléments dans le même ordre que celui dans lequel ils ont été stockés ou dans l'ordre inverse. (FIFO) (LIFO).
- Ajout avec les méthodes addFirst ou addLast
- Pour obtenir un élément
- En conservant l'élément peekFirst ou peekLast.
- En supprimant de la liste pollFirst ou pollLast.



Set

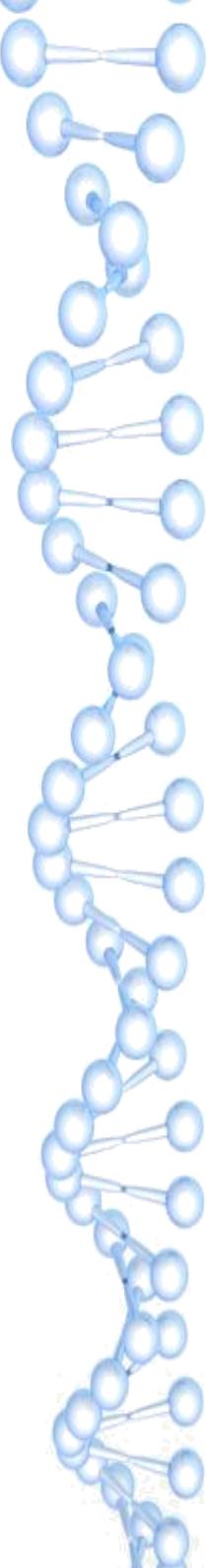
- **HashSet**
- Implémentation de l'interface Set.
- Un HashSet n'accepte pas le stockage de deux éléments identiques.
 - Lorsqu'un élément est ajouté à un HashSet (add), celle-ci vérifie s'il n'y a pas déjà un élément identique en comparant le hashCode de l'élément avec ceux des éléments déjà présents dans la liste.
 - La méthode add retourne true si l'ajout est effectué.
 - Il faut redéfinir le hashCode de toutes les classes dont les instances sont stockées dans un HashSet.



Map

HashMap et HashTable

- Utilise fortement hashCode et equals
- void clear()
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- Set<Map.Entry<K,V>> entrySet()
- V get(Object key)
- boolean isEmpty()
- Returns true if this map contains no key-value mappings.
- Set<K> keySet()
- V put(K key, V value)
- V remove(Object key)
- Collection<V> values()



Iterator

- Un Iterator peut être utilisé pour parcourir un Set ou une List
- **ListIterator** ne peut itérer que sur une List.
- Iterator traverse la collection en avançant alors que ListIterator traverse une List dans les deux directions.
- ListIterator implemente Iterator. Il permet en plus ajout, remplacement, récupérer les index des éléments précédent et suivant

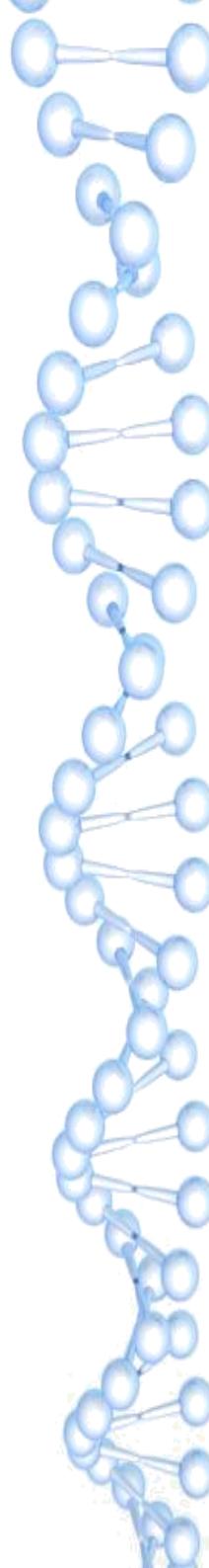
Itérations par Stream

Pour itérer sur un container on peut utiliser la classe Iterator
Modifier le contenu d'un container peut invalider ses Iterators

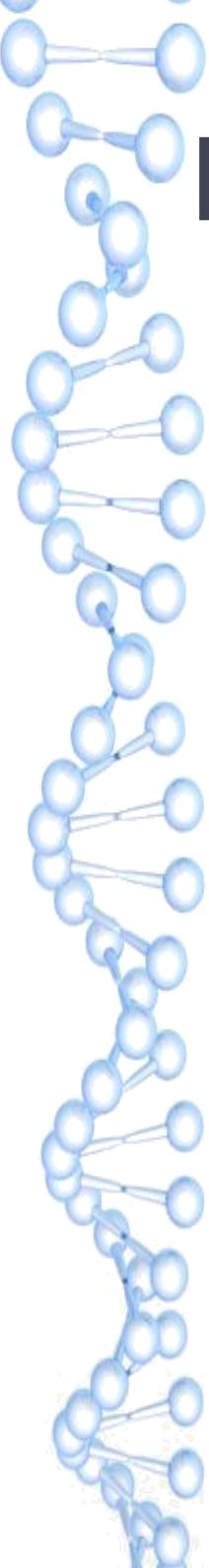
```
List<String> listePrenoms = new  
ArrayList<String>(); listePrenoms.add("Amandine");  
listePrenoms.add("Alizée");  
for(Iterator<String> it = listePrenoms.iterator();  
    it.hasNext(); ) {  
    String prenom = (String)it.next();  
    System.out.println(prenom);  
}
```

Pour les maps on peut reproduire cette approche via la méthode entrySet().

Pour les HashMap on peut itérer grâce à une Enumération (voir méthode éléments())



Mise en Application des Collections



La conception objet

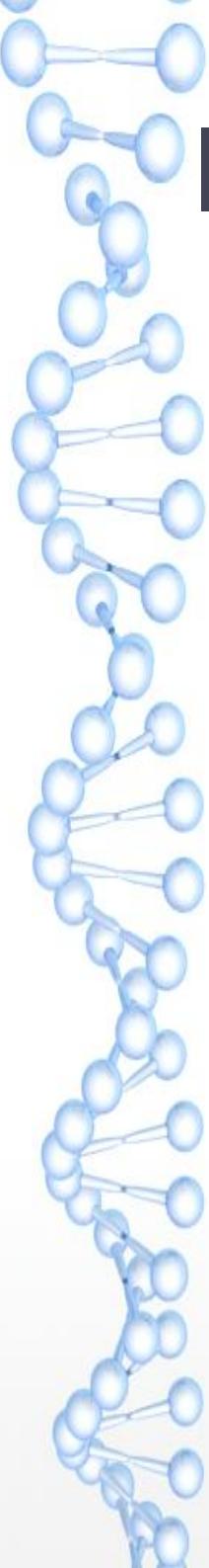
Les **concepts objets** ne sont pas simples à apprécier. Il est plus facile de raisonner de manière linéaire et concrète, que de manière abstraite et avec des notions d'héritage ou de polymorphisme.

Cela nécessite **un certain recul et une capacité d'abstraction** qui n'est pas totalement intuitive et plus difficilement accessible.

La **POO offre des mécanismes uniques de représentation** de notre environnement mixant données et comportement, déclinant des notions génériques en de multiples plus spécialisés.

Outre cette capacité à représenter les choses, la conception orientée objet offre également **des mécanismes de tolérance au changement** très puissants.

Cette tolérance est un des **facteurs clés** du développement logiciel, la maintenance demandant souvent des efforts croissants au fil du temps et de l'évolution du logiciel.



Intolérance aux changements

Fragilité

Tendance d'un logiciel à casser en plusieurs endroits à chaque modification, répercussions sur des modules n'ayant aucune relation.

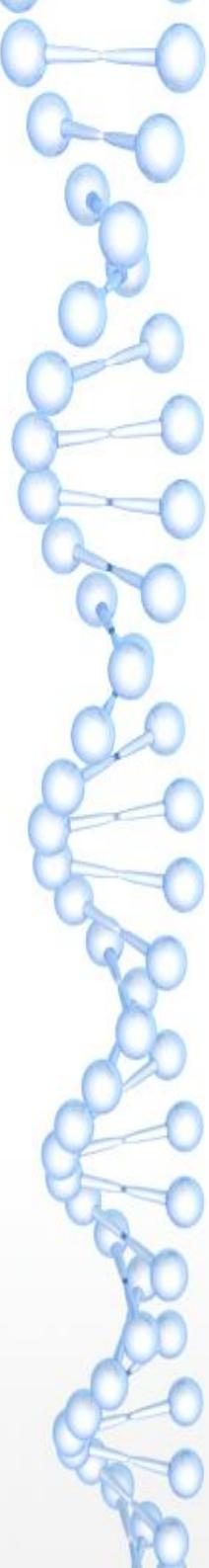
Rigidité Chaque changement cause une cascade de modifications dans les modules dépendants.

Immobilisme Incapacité du logiciel à pouvoir être réutilisé par d'autres projets ou par des parties de lui-même. Efforts et risques trop importants.

Viscosité Il est plus facile de faire un contournement plutôt que de respecter la conception qui a été pensée.

Opacité Grand manque de lisibilité et de simplicité du code. La situation la plus courante est un code qui n'exprime pas sa fonction première.





Principes Agile

Cinq principes fondamentaux de conception, répondant à cette problématique d'évolutivité, sous l'acronyme SOLID (R. Martin)

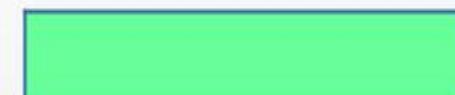
Single Responsibility Principle

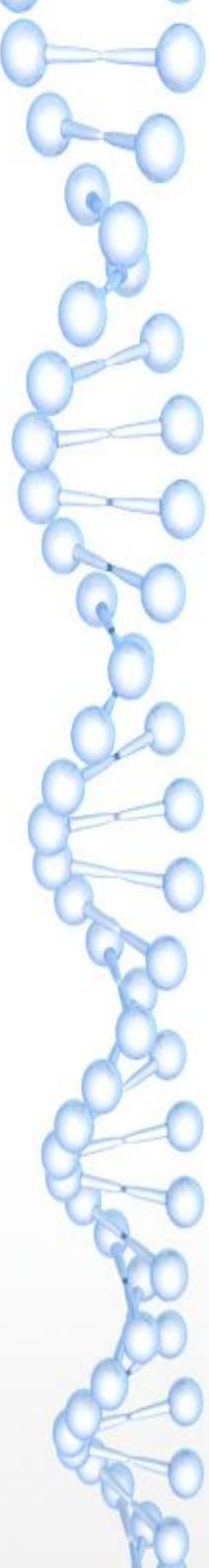
Une classe ne doit avoir qu'une responsabilité (une chose à faire). Si une classe en possède plusieurs, les modifications apportées à une responsabilité impacteront l'autre, augmentant la rigidité et la fragilité du code.

Open Closed Principle

Une classe, une méthode, un module doit pouvoir être étendu, supporter différentes implémentations (Open for extension) sans pour cela devoir être modifié (closed for modification).

Non-respect par exemple : instanciations conditionnelles dans un constructeur. Une spécialisation amènera une modification de la condition, ce qui est contre ce principe.





Principes Agile

Cinq principes fondamentaux de conception, répondant à cette problématique d'évolutivité, sous l'acronyme SOLID (R. Martin)

Liskov Substitution Principle

Un utilisateur d'une classe de base doit pouvoir continuer de fonctionner correctement si une classe dérivée de la classe principale lui est fournie à la place.

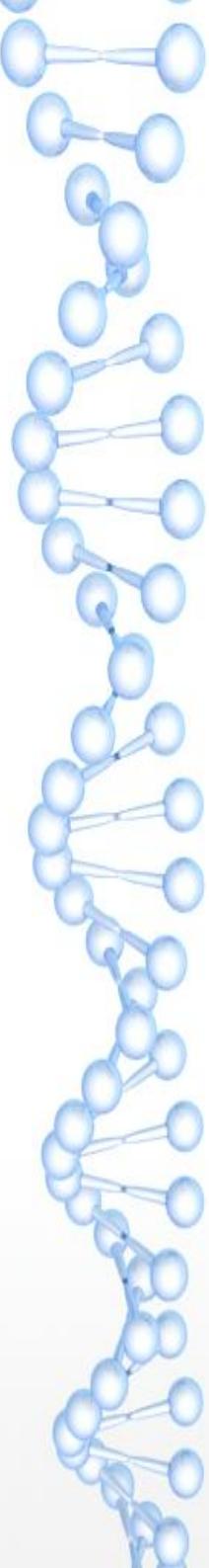
Interface Segregation Principle

Le but de ce principe est d'utiliser les interfaces pour définir des contrats, de ne pas se contenter de l'abstraction seule. Attention cependant à ne pas multiplier inutilement les interfaces.

Dependency Inversion Principle

Les modules de haut niveau ne doivent pas dépendre de modules de plus bas niveau...

Car des modifications effectuées dans les modules « bas niveau » ne doivent pas avoir des répercussions vers le haut sous peine de rendre le tout couplé et inutilisable



Des principes de bons sens

Au-delà des principes agiles, on peut s'appuyer sur des éléments plus généraux de conception.

G.I.G.O. Garbage In → Garbage Out

Si vous ne contrôlez pas les entrées de votre système (fichiers, clavier, DB, streams), n'espérez pas produire des résultats de qualité

K.I.S.S. Keep It Simple Stupid.

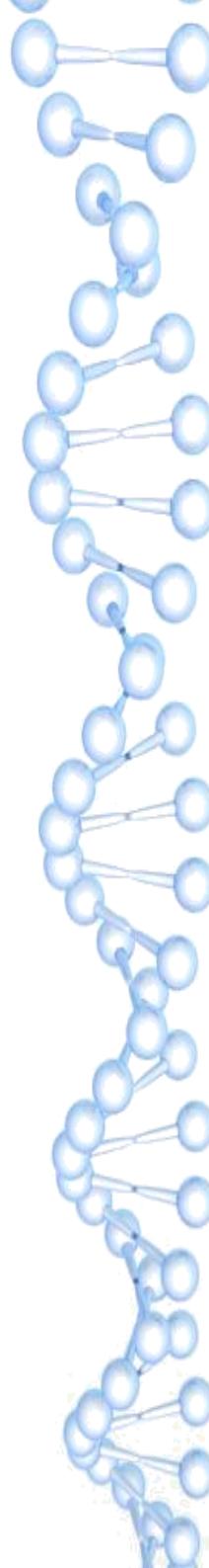
Dans la mesure du possible, la conception doit rester simple, une classe ne peut supporter à elle seule toutes les alternatives.

D.R.Y Don't repeat yourself

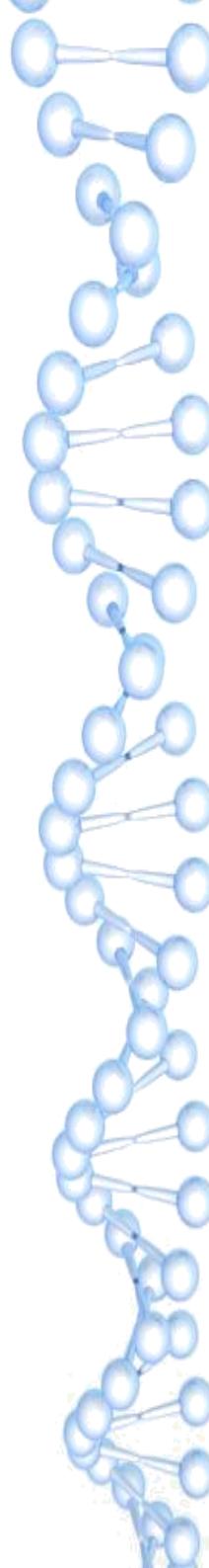
Éviter la duplication de code ; en préservant à un seul endroit et en factorisant les comportements, sans aller vers l'excès, vous amènera à une meilleure maintenabilité

Y.A.G.N.I. You Ain't Gonna Need It,

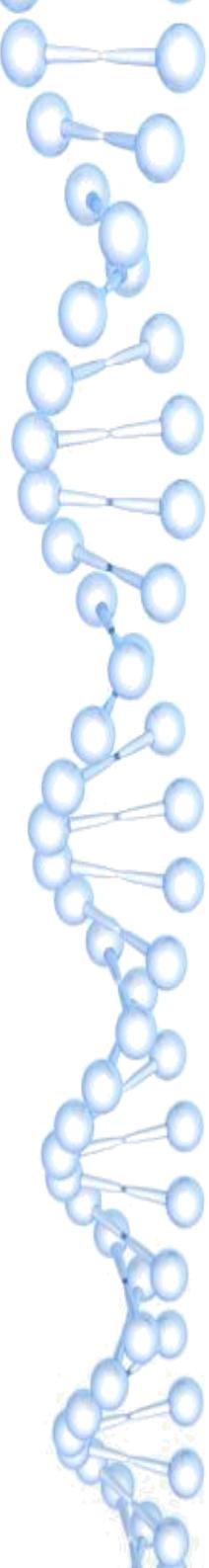
Vous n'en aurez pas besoin !! Mettez toujours en œuvre les choses quand vous en avez effectivement besoin, pas lorsque vous prévoyez simplement que vous en aurez besoin.



Application Conception OO Solutions aux Problèmes



Design Patterns

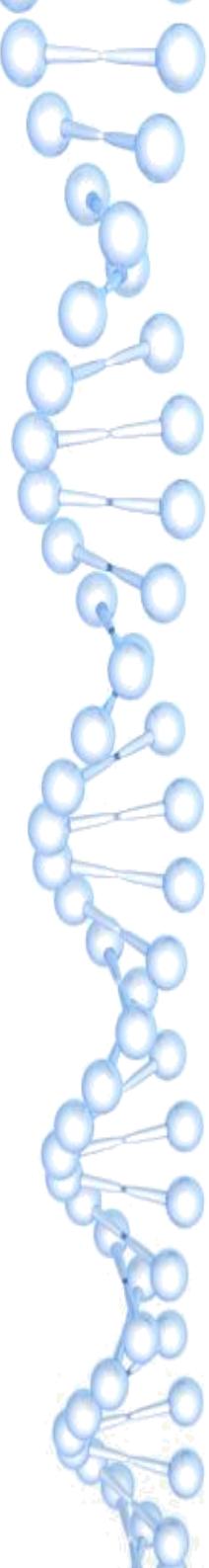


Intérêt

Problèmes récurrents sous le terme Anti Patterns Erreurs courantes
Couplage fort
Effet spaghetti
Manque de modularité
Manque d'interface
Copier-coller

Solutions : Modèles de Conception (aka Design Patterns) Un design pattern constitue une base de savoir-faire pour résoudre un problème récurrent dans un domaine particulier.

L'expression de ce savoir-faire :
permet d'identifier le problème à résoudre
propose une solution possible et correcte pour y répondre
offre les moyens d'adapter cette solution



Intérêt

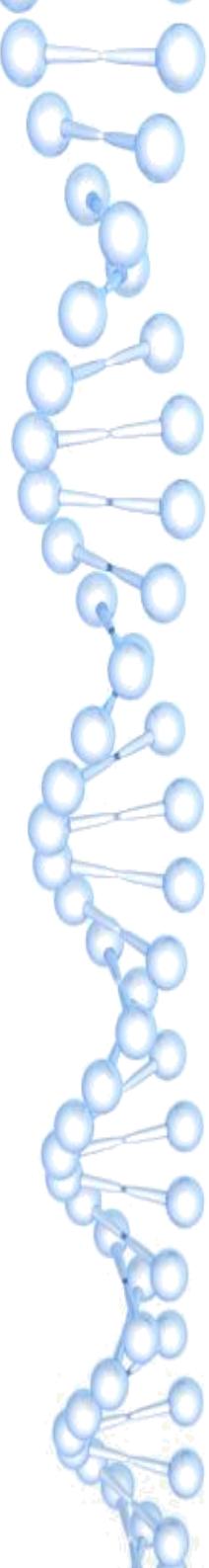
Un patron décrit à la fois un problème qui se produit très fréquemment dans votre environnement

Il amène une proposition de solution à ce problème de telle façon que vous puissiez utiliser cette solution des milliers de fois sans jamais l'adapter deux fois de la même manière.

Décrire des types de solutions récurrentes à des problèmes communs dans des types de situations

23 modèles de conception du «Gang of Four»

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides



Principe

Un nom

Utile pour l'identifier, communiquer, documenter, faire de l'abstraction.

Un problème

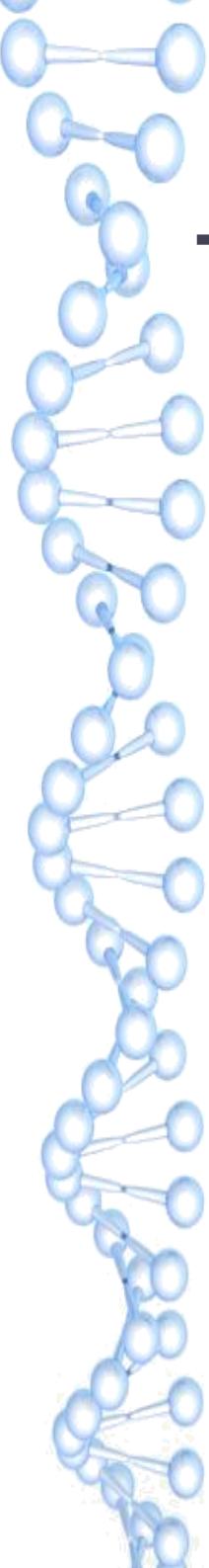
Expose le sujet à traiter, la situation et conditions pour qu'il puisse être appliqué.

Une solution

Décrit les éléments qui constituent la conception, leurs relations et coopérations.

Conséquences

Implications de la mise en œuvre de la solution (avantages / inconvénients) en termes de mémoire, vitesse exécution, portabilité, extensibilité, etc.



Trois Types de Patrons

Création instantiation et configuration des objets

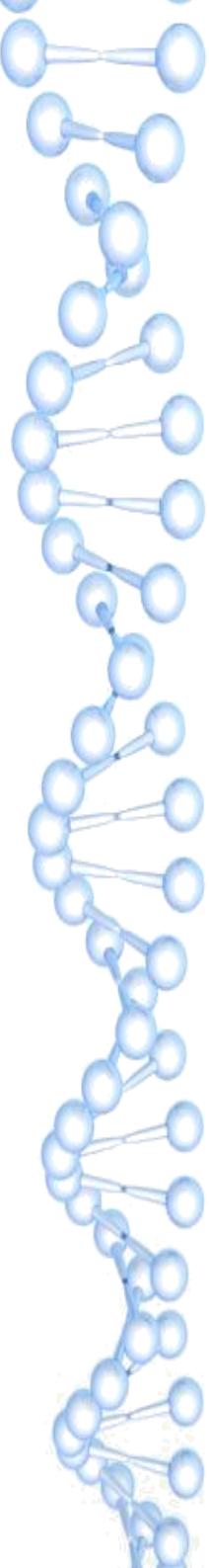
- Expriment le principe du processus d'instanciation. Pour rendre les systèmes indépendants de la façon dont les objets sont créés.

Structure : organisation des classes

- Étudie la manière de composer classes et objets pour réaliser des structures plus importantes. Utiles notamment pour permettre à des bibliothèques de classes développées séparément de coopérer.

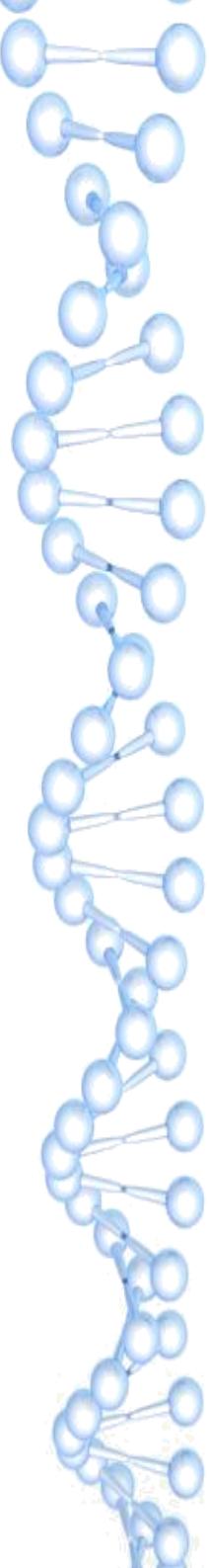
Comportement : interactions entre les objets

- Traitent des comportements/algorithmes et de l'affectation des responsabilités entre objets. Gestion des interactions dynamiques entre des classes et des objets.



Patterns (Création)

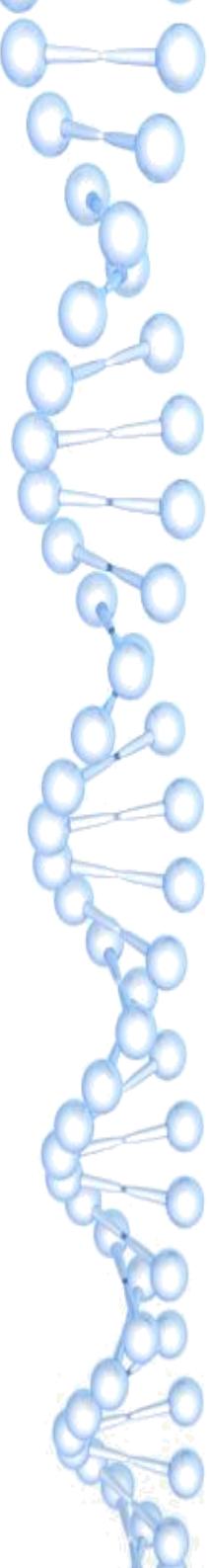
- Fabrique abstraite (Abstract Factory)
 - Interface pour la création de familles d'objets sans spécifier de classe concrète.
- Fabrication (Factory Method)
 - définit une interface pour la création d'objets en laissant le choix à des sous-classes les choix des classes à instancier.
- Singleton
 - assure l'unicité de l'instance d'une classe.



Patterns (Création)

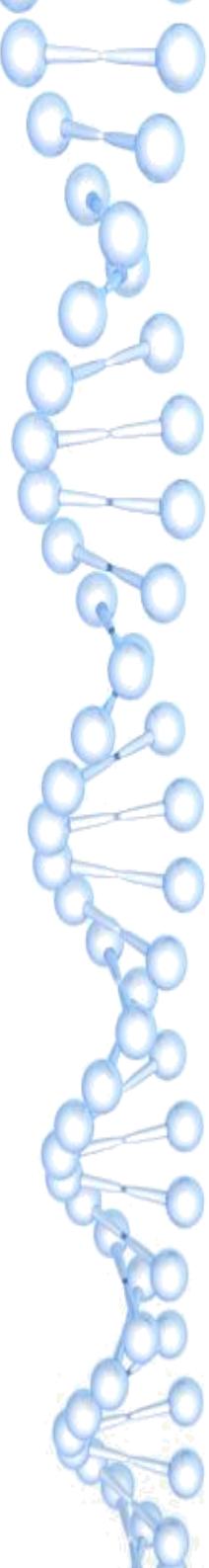
- Monteur (Builder)
Séparation de la construction d'objets complexes des parties qui les composent afin qu'un même processus de construction puisse créer différentes représentations.

- Prototype
Spécification des types d'objets à créer à partir d'un objet prototype et crée les nouveaux objets en copiant le prototype.



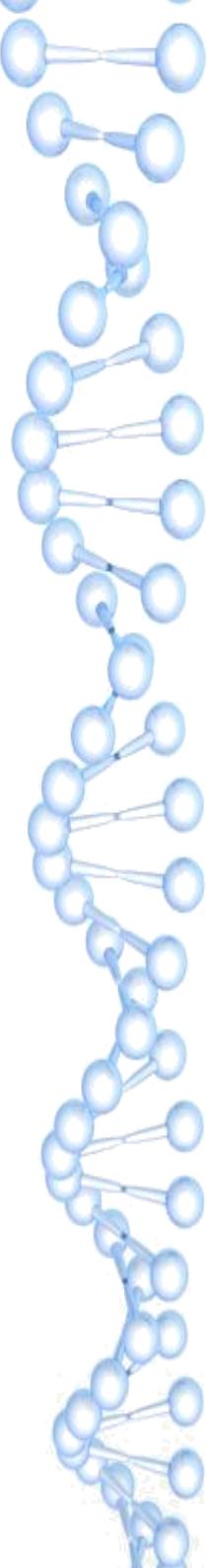
Patterns (Structure)

- Adaptateur (Adapter)
Adapte/traduit l'interface d'une classe en une autre interface qui correspond aux attentes du client.
- Pont (Bridge)
Découple une abstraction de son implémentation afin que les deux puissent être modifiés indépendamment l'un de l'autre.
- Composite
Structure pour la construction d'agrégations récursives.



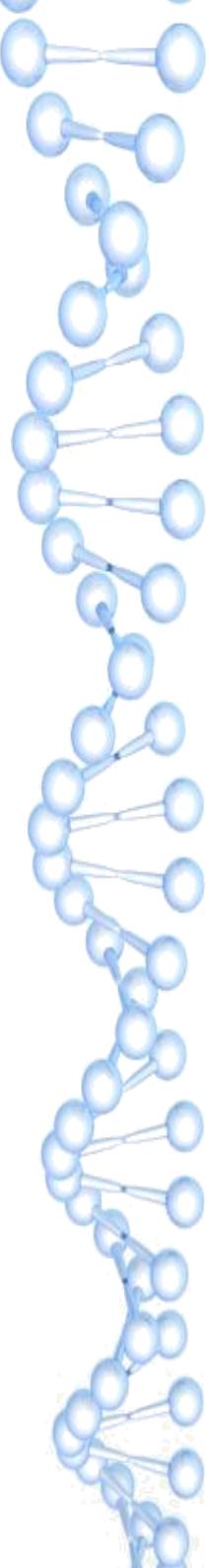
Patterns (Structure)

- Décorateur (Decorator)
Attache dynamiquement des responsabilités supplémentaires à un objet ; alternative à l'héritage.
- Facade
- fournit une interface unifiée à l'ensemble des interfaces d'un sous-système ; simplifie l'utilisation du sous-système.
- Procuration (Proxy)
- fournit un mandataire pour contrôler l'accès à un objet



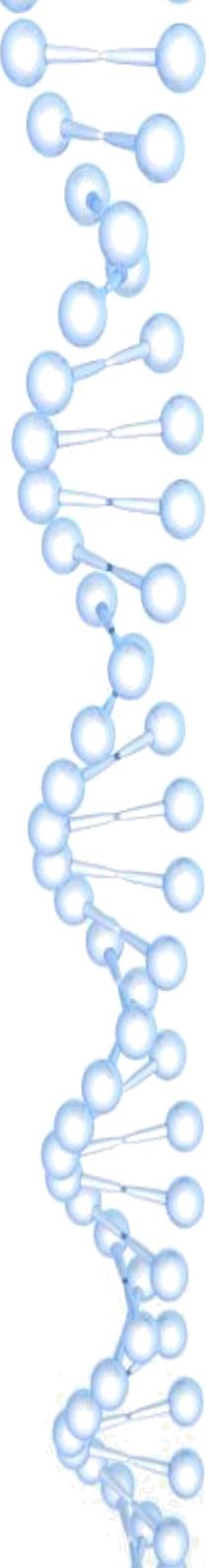
Patterns (Comportement)

- Chaîne de responsabilités (Chain of Responsibility)
Délègue des requêtes à des responsables de services ; évite couplage émetteur-récepteur.
- Commande (Command)
Encapsule une requête par un objet ; permet notamment la réversion des opérations (Undo).
- Poids mouche (Flyweight)
utilise une technique de partage quand de nombreux objets de fine granularité sont à mettre en œuvre.



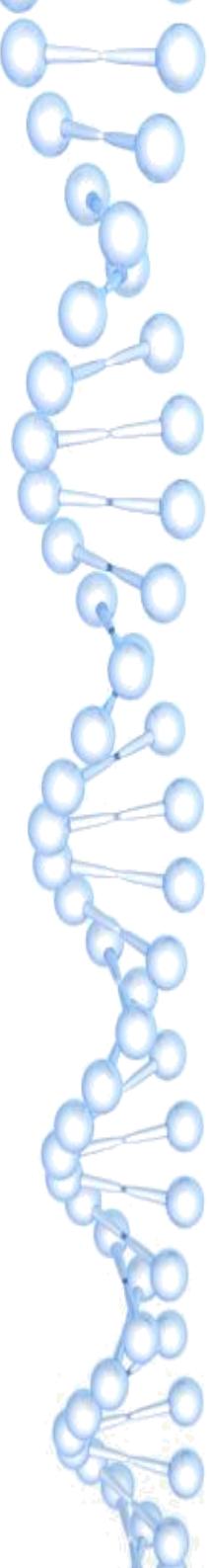
Patterns (Comportement)

- Interprète (Interpreter)
Étant donné un langage, définit une représentation de sa grammaire.
- Itérateur (Iterator)
permet le parcours séquentiel des collections.
- Médiateur (Mediator)
encapsule les modalités d'interaction d'un ensemble d'objets ; favorise le couplage faible ; schéma en étoile.



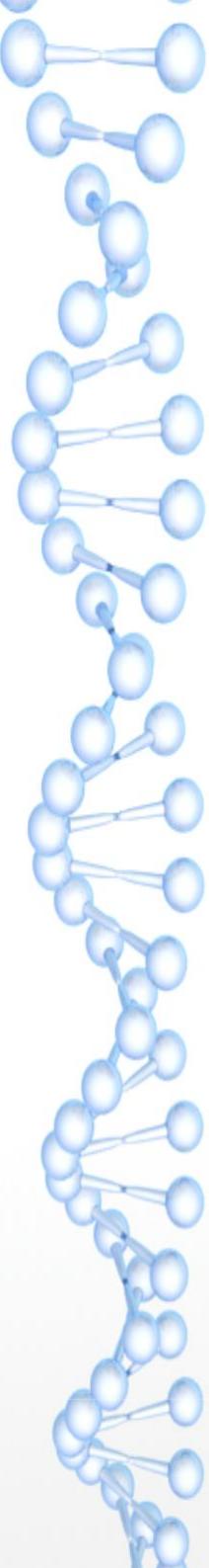
Patterns (Comportement)

- Memento (Memento)
capture et restauration de l'état d'un objet.
- Observateur (Observer)
Mise à jour automatique des objets dépendants d'un objet.
- Etat (State)
Permettre à un objet de modifier son comportement lorsque son état interne change, comme s'il changeait de classe.

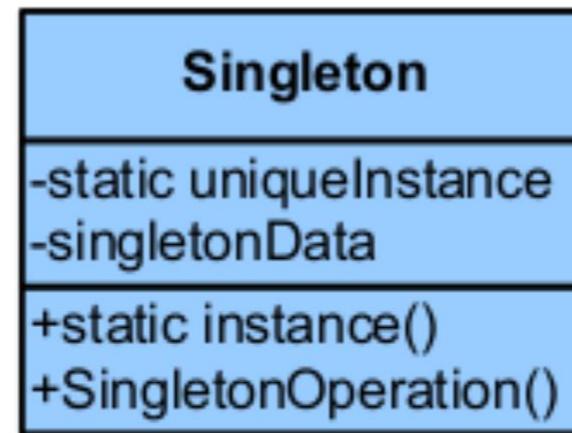


Patterns (Comportement)

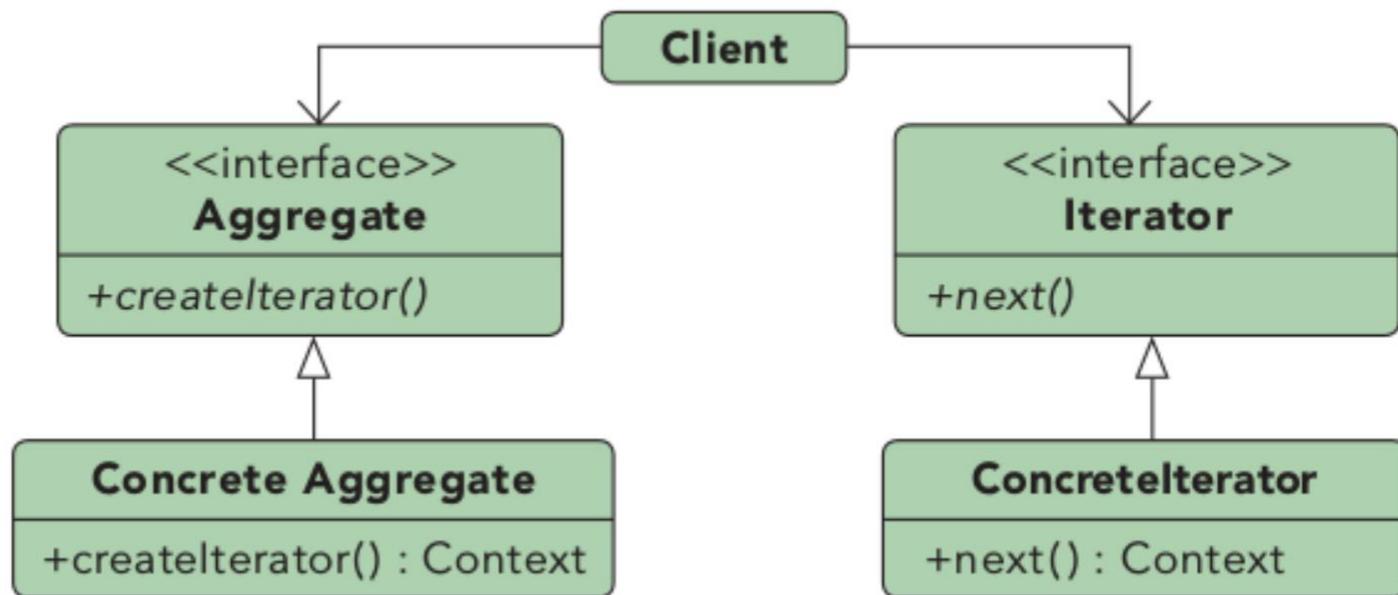
- Stratégie (Strategy)
Abstraction pour sélectionner un algorithme parmi plusieurs disponibles.
- Patron de méthode (Template method)
Dans une opération, définit le squelette d'un algorithme dont certaines étapes sont fournies par des sous-classes.
- Visiteur (Visitor)
- Représente une opération à effectuer sur les éléments d'une structure d'objet ; permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère



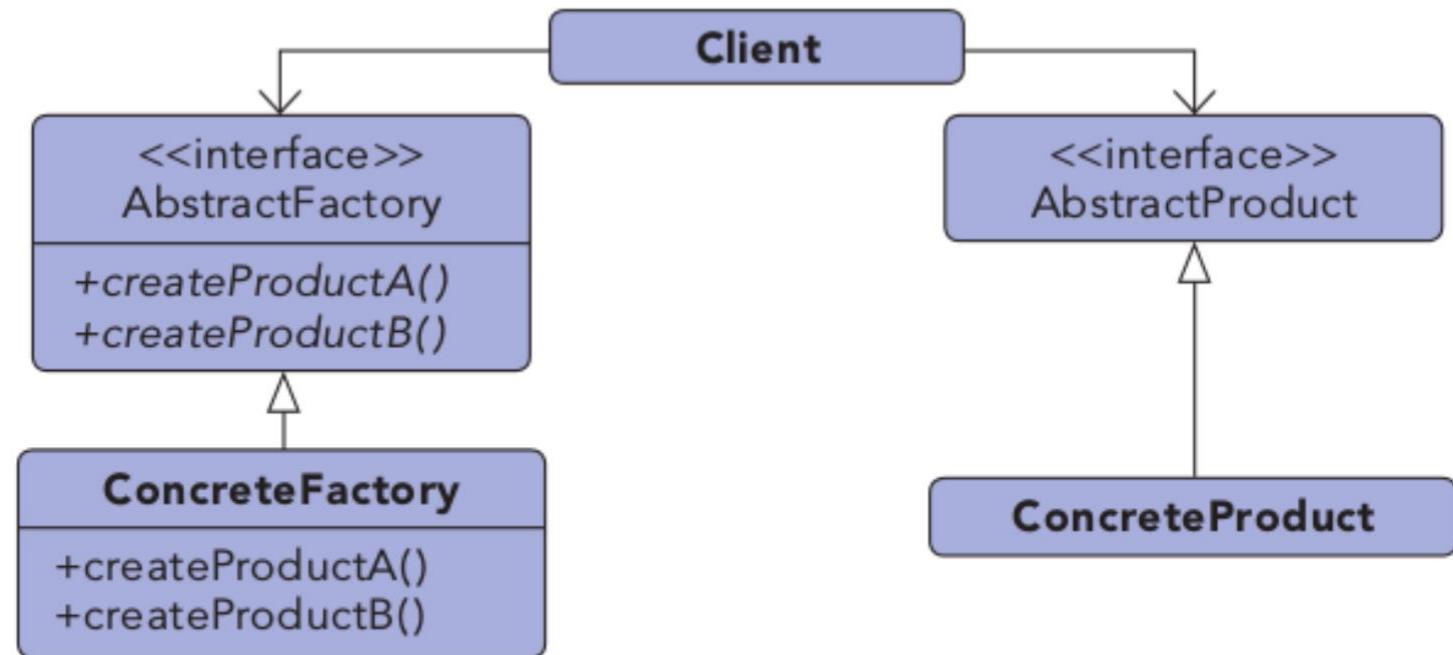
Singleton



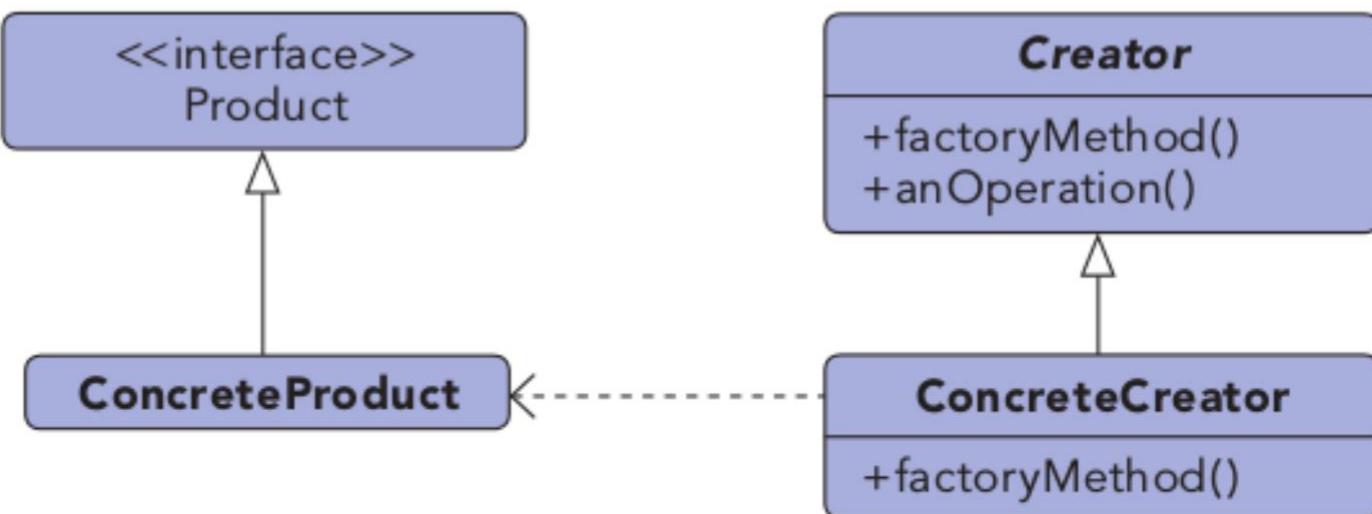
Iterator

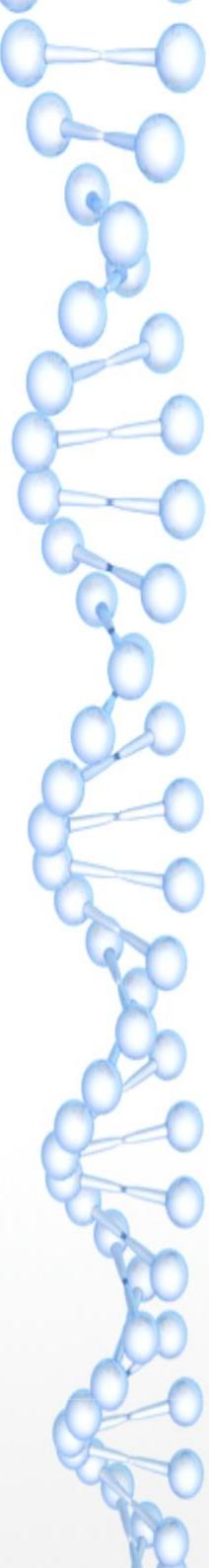


Abstract Factory

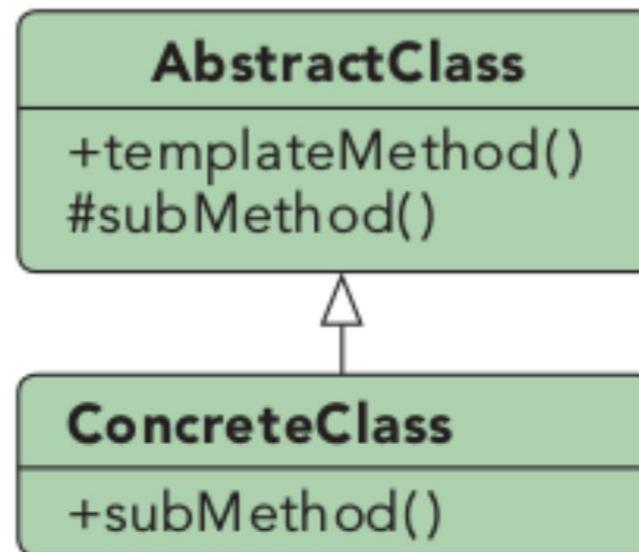


Factory

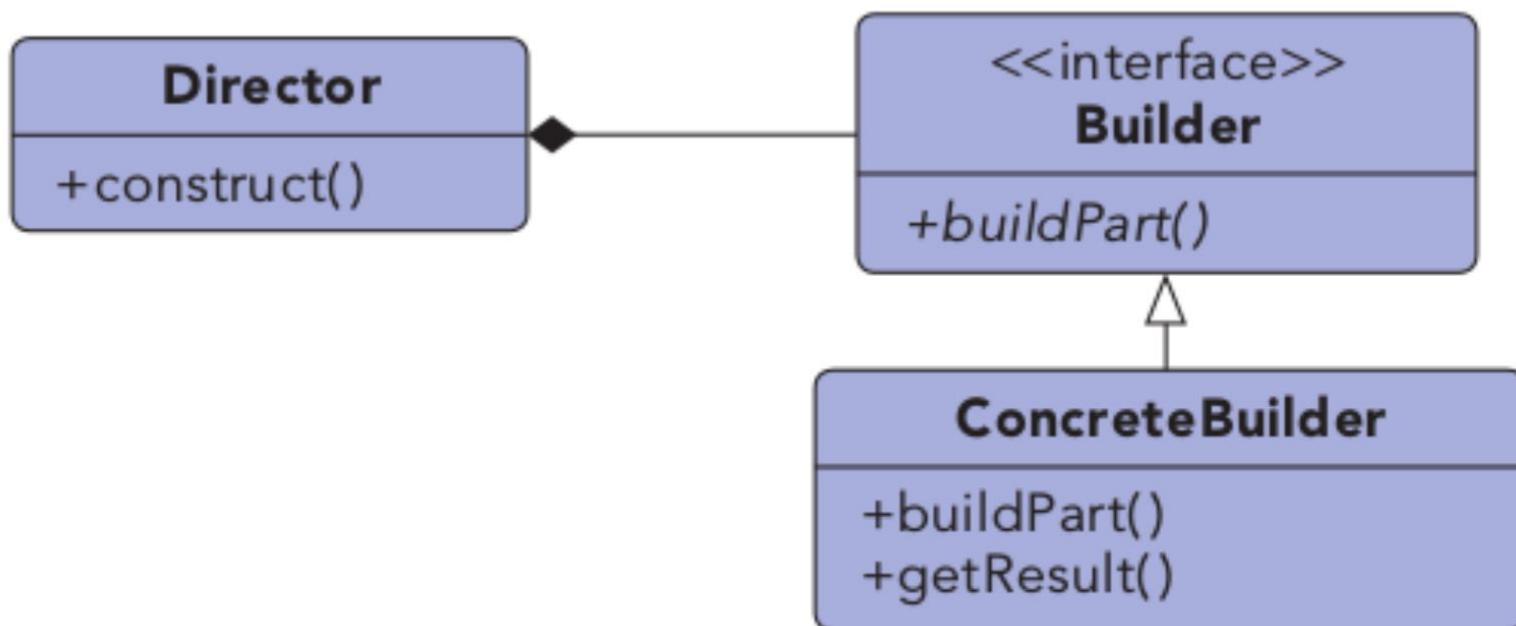




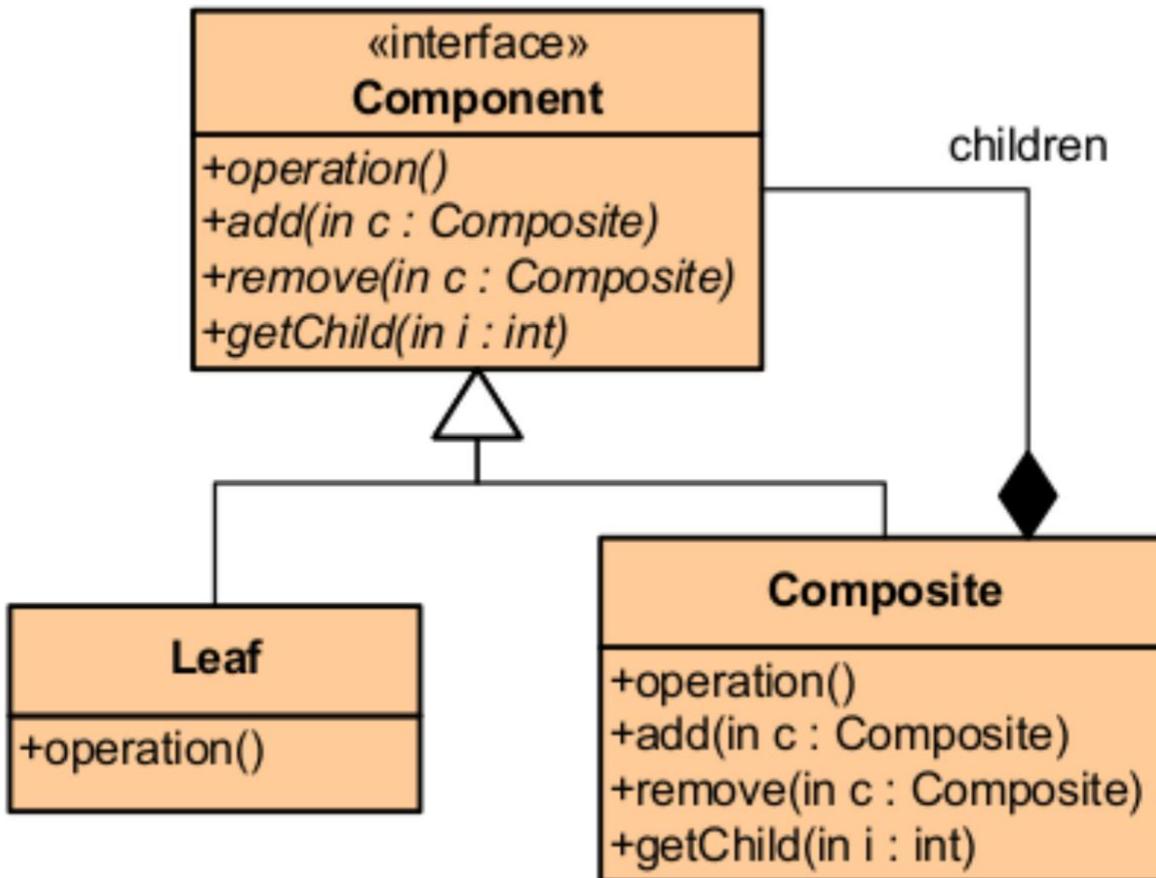
Template



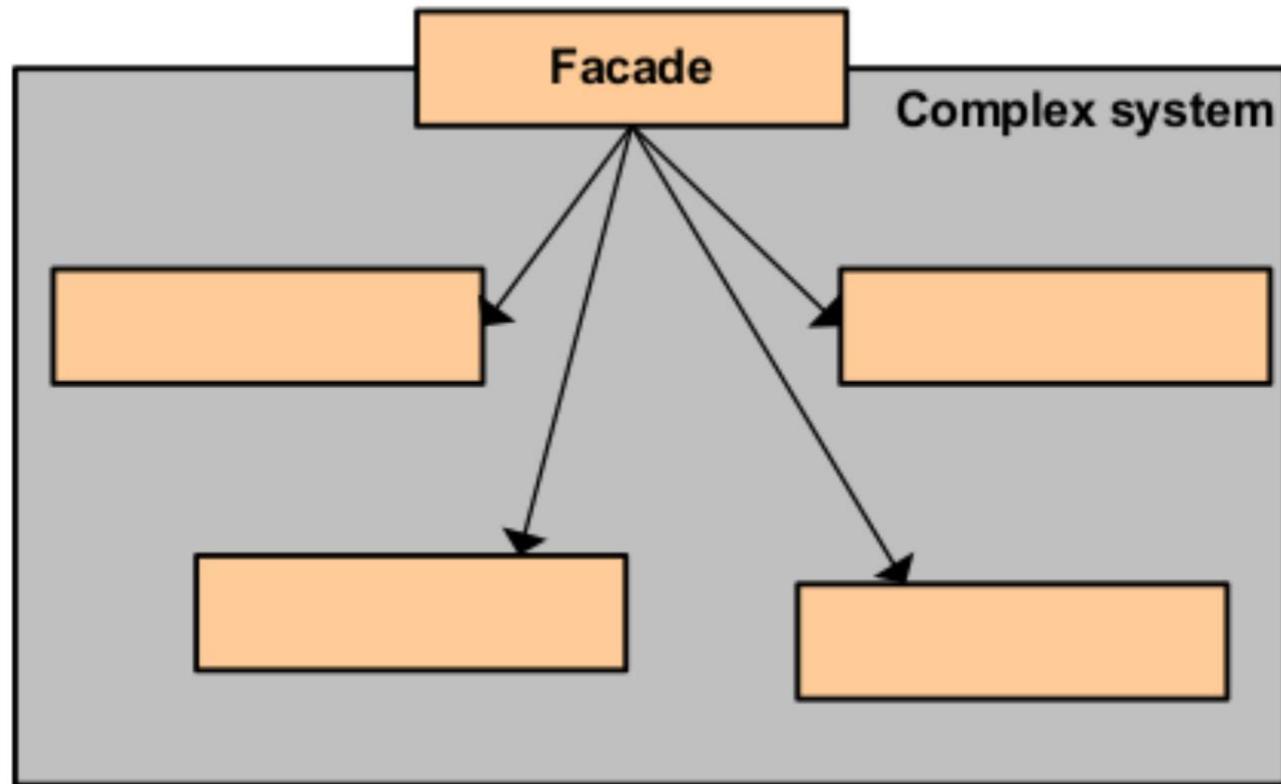
Builder



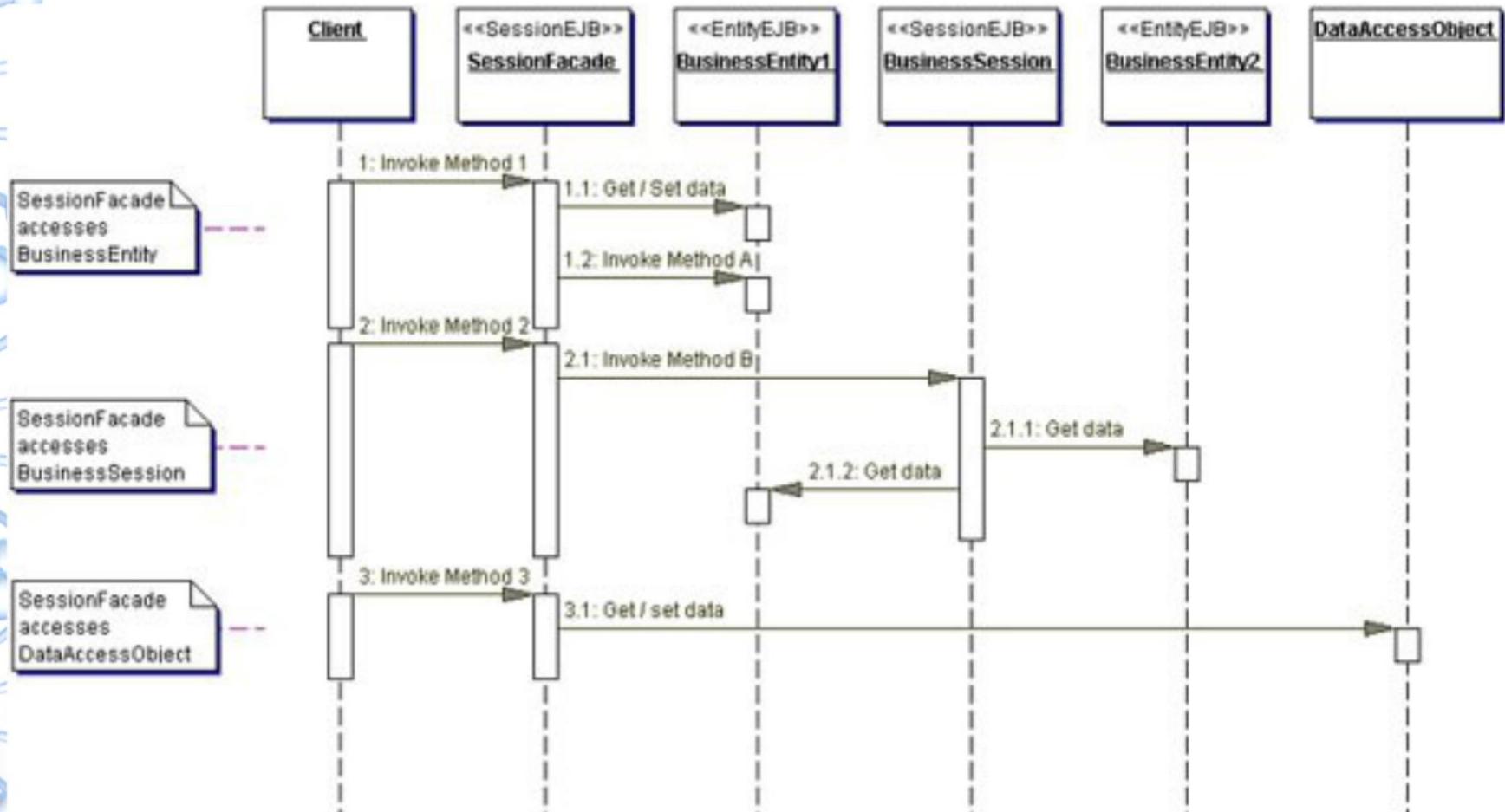
Composite



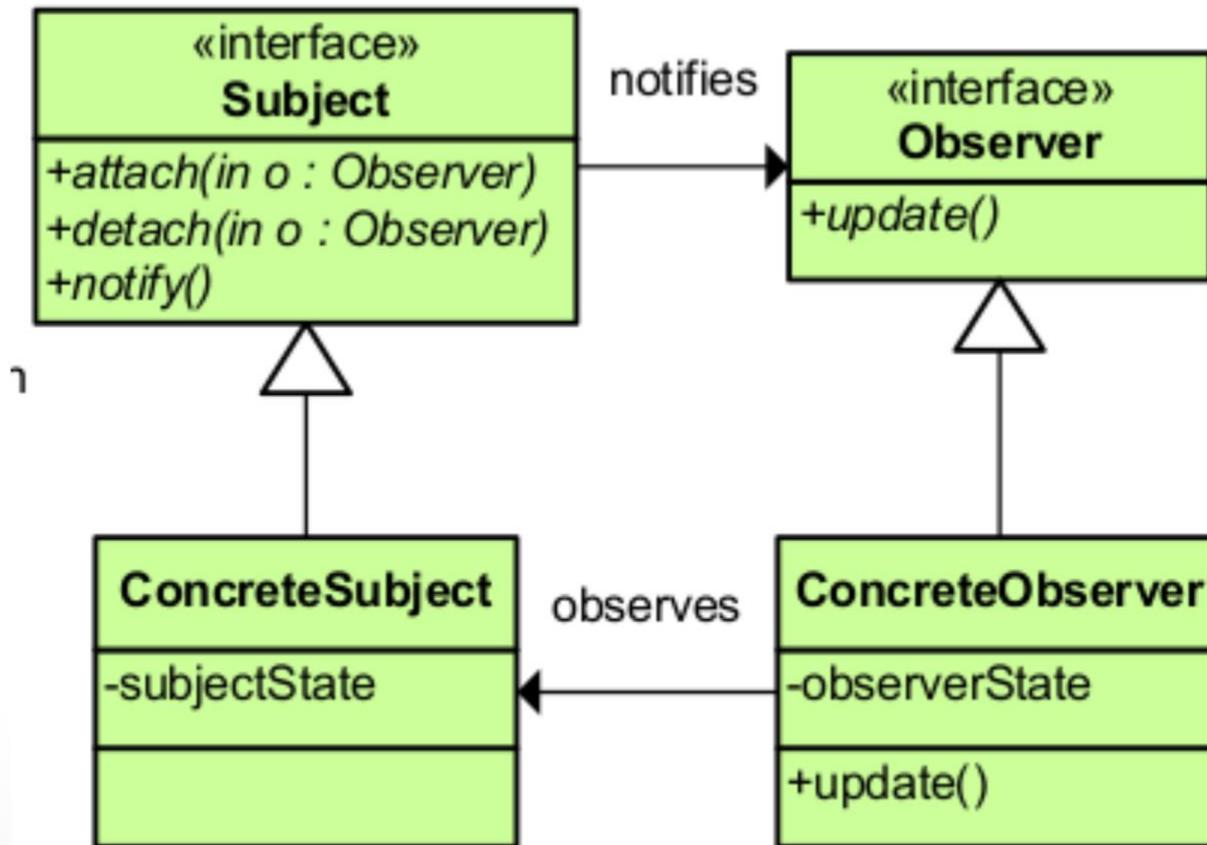
Façade



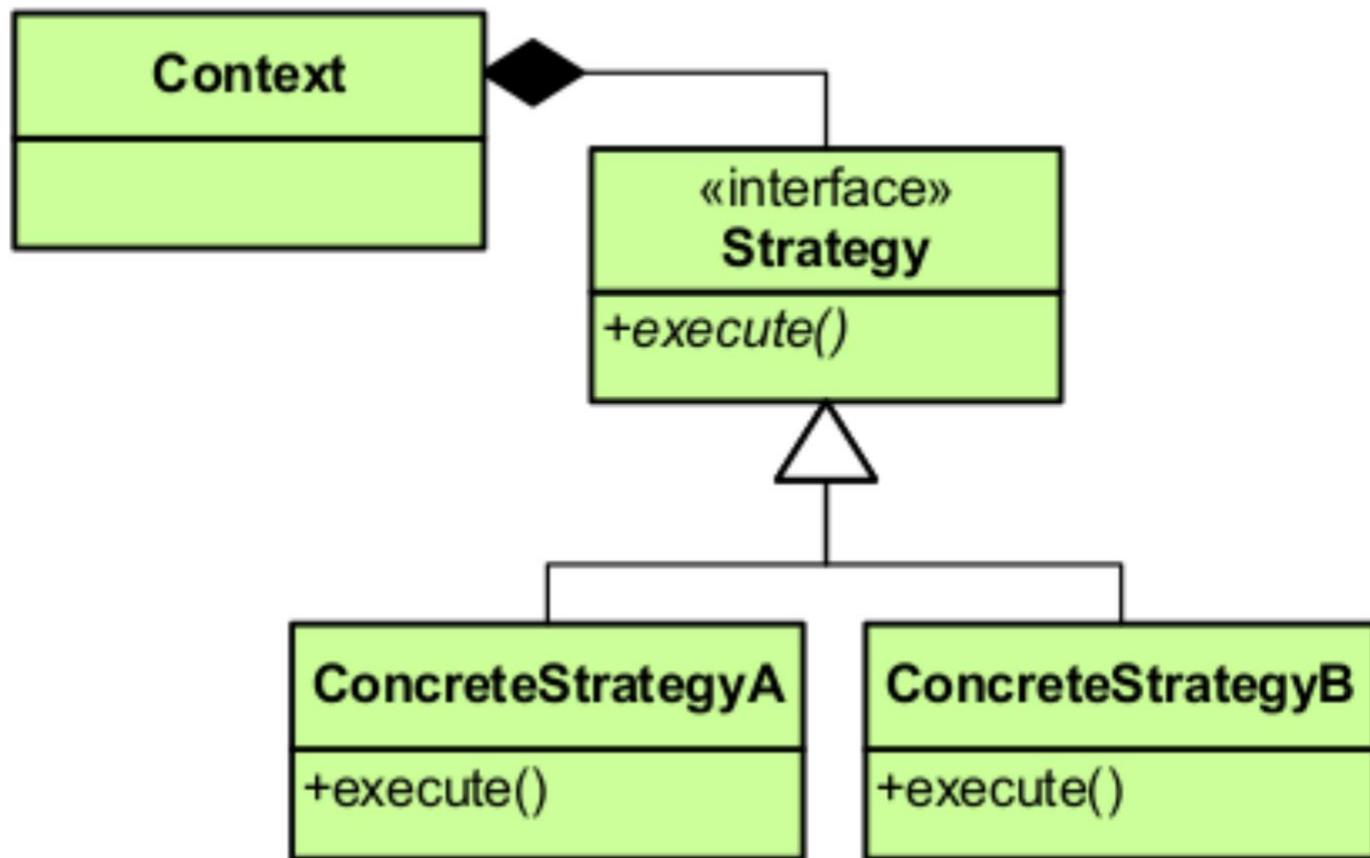
Session d'une Façade



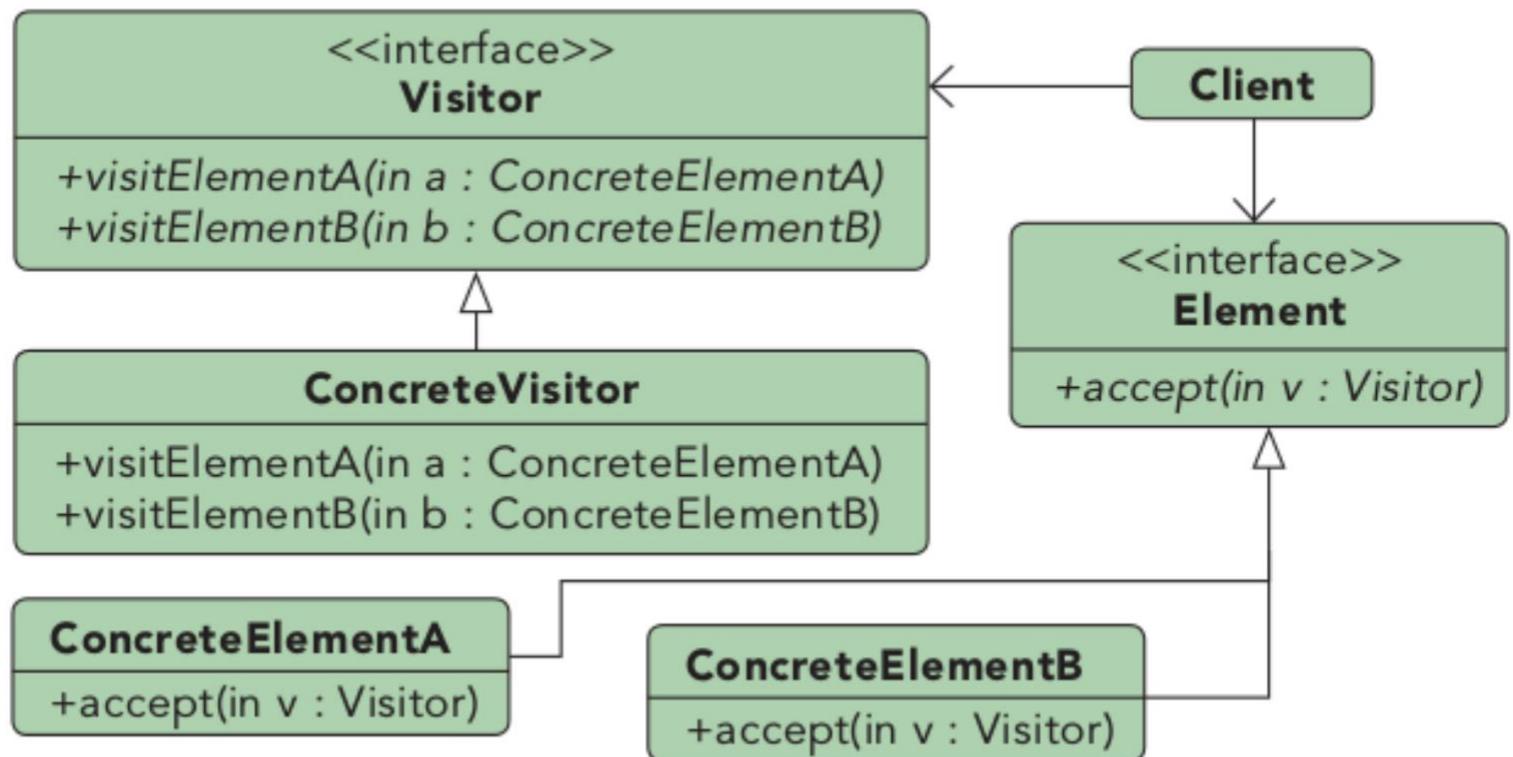
Observer



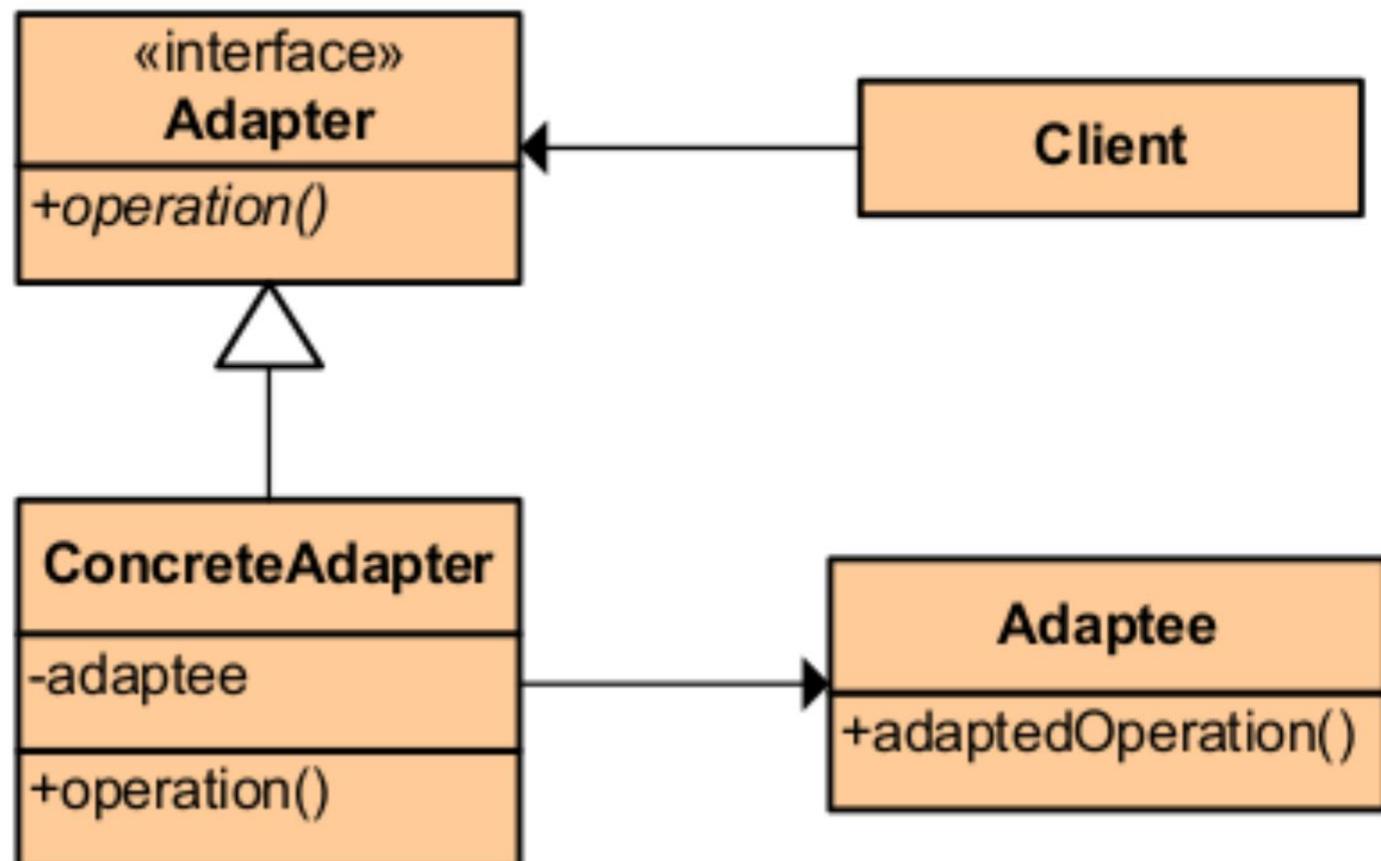
Strategy

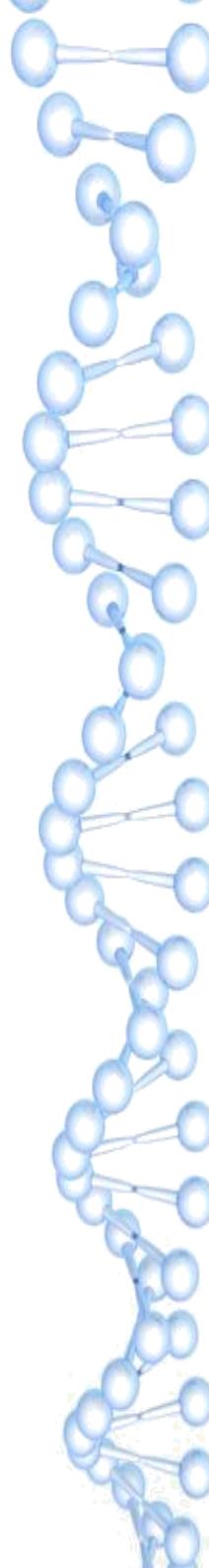


Visitor



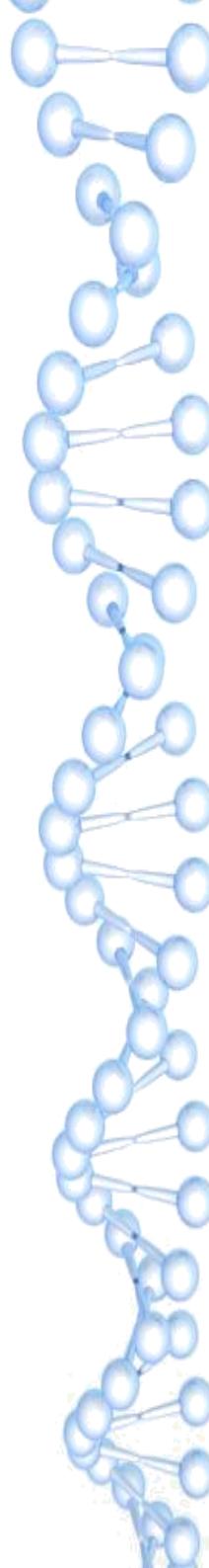
Adapter





Application

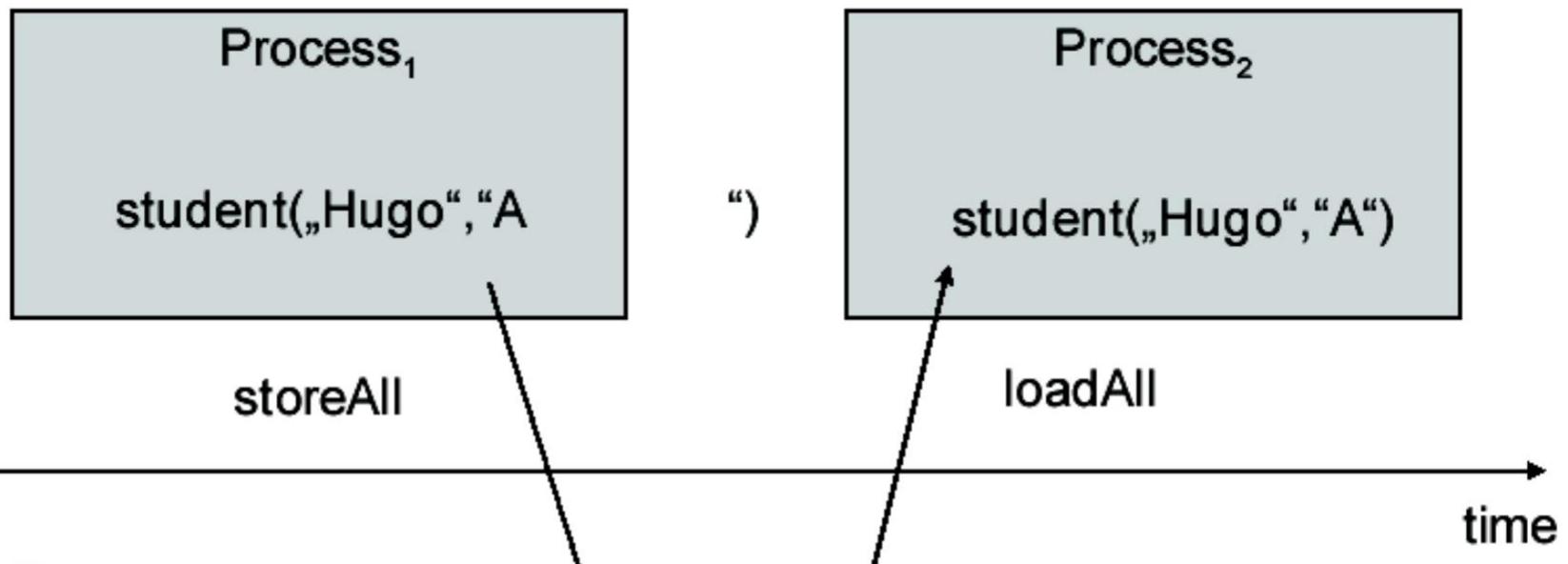
Modèles de Conception en Java

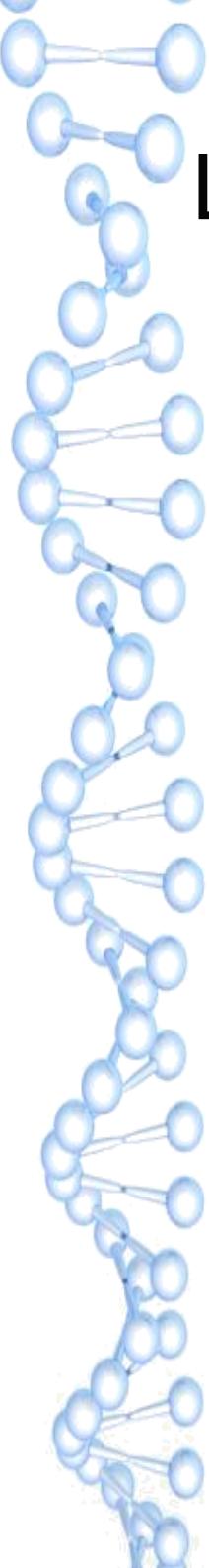


Persistante de Données

Qu'est-ce que la persistance ?

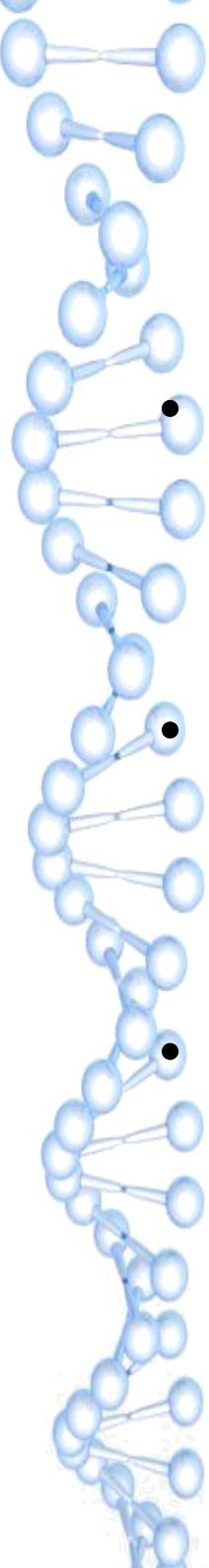
- La capacité qu'un objet à survivre même si la session ou le programme se termine.





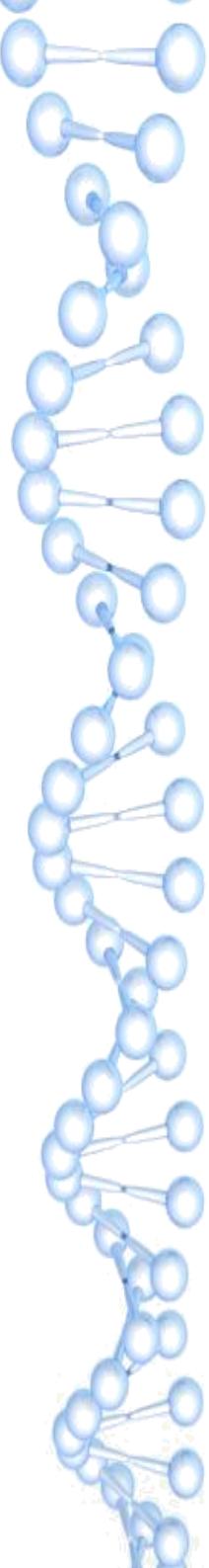
Les bases de données relationnelles

- Prédominances
 - Meilleure performances
 - Fiables
 - Largement supportées
- Nécessite un pont entre le monde objet et le monde relationnel
- SQL intégré dans le code objet
 - Object-Relational Mapping (ORM)



Solutions de persistance

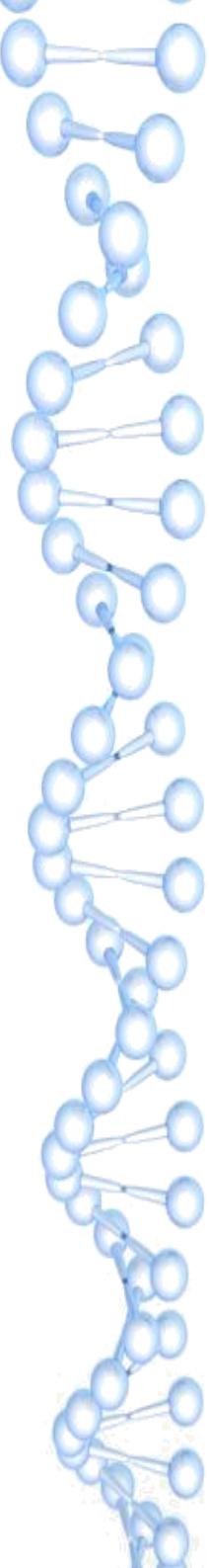
- **JDBC**
 - DAO pattern, String SQL dans code Java
 - Forte dépendance du code avec le schéma de BD,
Difficulté de maintenance
- **EJB (Entity Beans)**
 - Licence EJB container, Descripteur de déploiement propriétaire
 - Complexité, Performance et support pauvres
- **JPA**



Mapping ORM direct ?

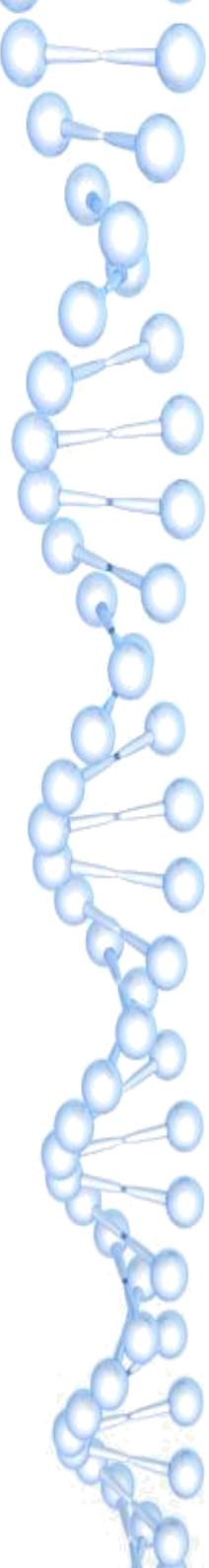
L'application interagit avec la base de données par des appels de fonction

```
String sql = "select name from items where id = 1";  
  
Connection c = DriverManager.getConnection( url );  
Statement stmt = c.createStatement();  
ResultSet rs = stmt.executeQuery( sql );  
  
if( rs.next() ) System.out.println( rs.getString("name") );
```



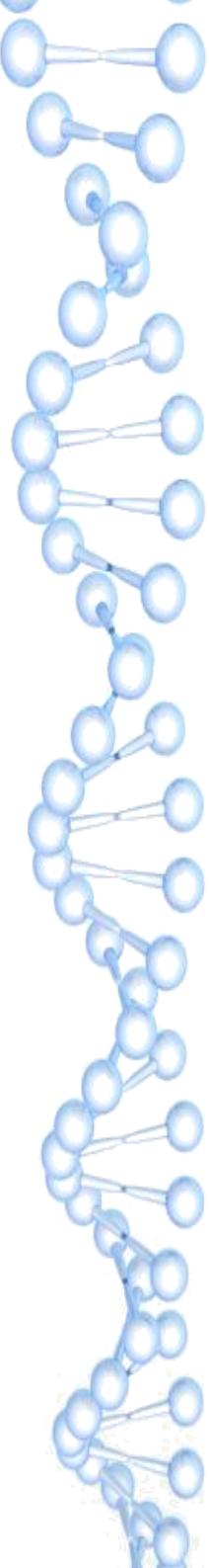
ORM, Un exemple

```
public class Employee {  
    Integer id;  
    String name;  
    Employee supervisor;  
}
```



ORM, Un exemple

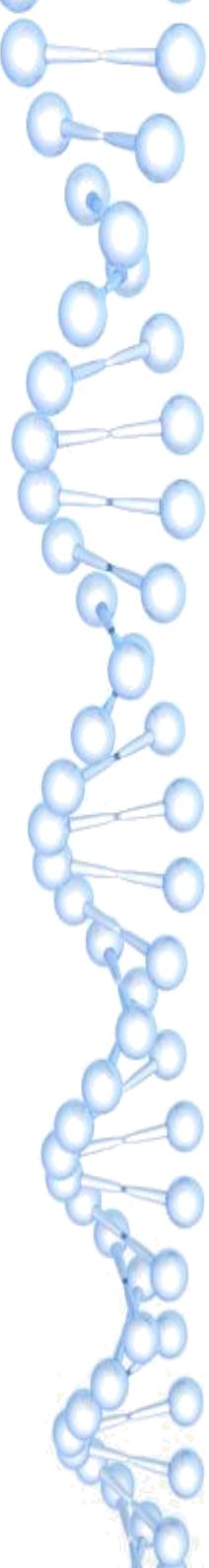
```
create table employees (  
    id      integer primary key,  
    name    varchar(255),  
    supervisor integer references employees(id)  
)
```



ORM, Un exemple

Comment construire l'objet basé sur les données de la base ?

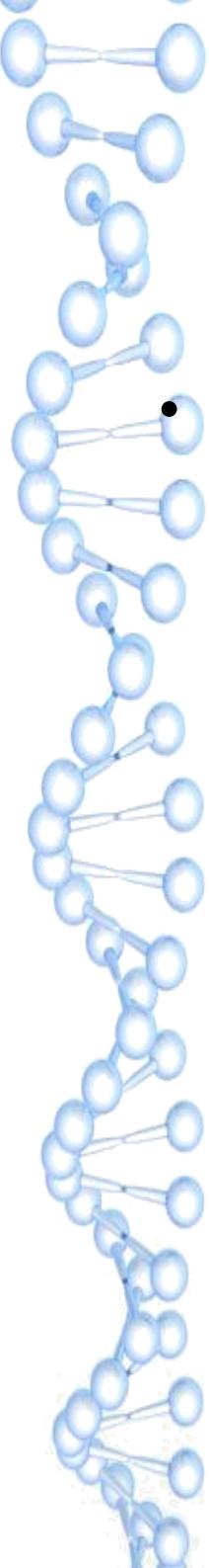
```
public class Employee {  
    Integer id;  
    String name;  
    Employee supervisor;  
  
    public Employee( Integer id )  
    {  
        // access database to get name and supervisor  
        ...  
    }  
}
```



ORM, Un exemple

- Les requêtes SQL sont codées en dur

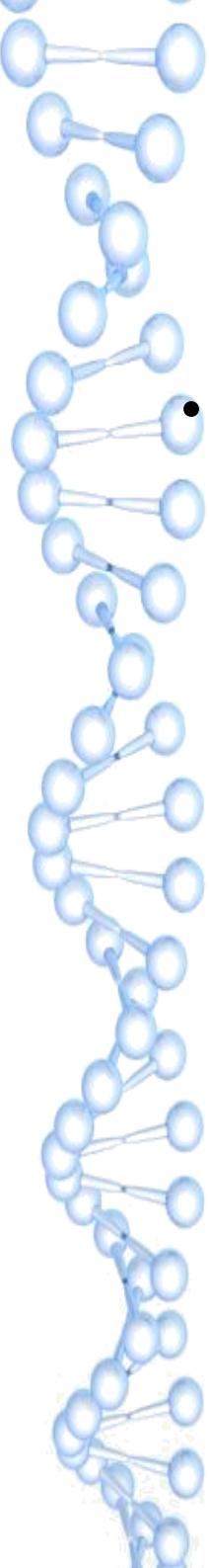
```
public Employee( Integer id ) {  
    ...  
    PreparedStatement p;  
    p = connection.prepareStatement(  
        "select * from employees where id = ?"  
    );  
    ...  
}
```



ORM, Un exemple

- Un travail fastidieux de traduction des résultats en objets ...

```
public Employee( Integer id ) {  
    ...  
    ResultSet rs = p.executeQuery();  
    if( rs.next() )  
    {  
        name = rs.getString("name");  
        ...  
    }  
}
```

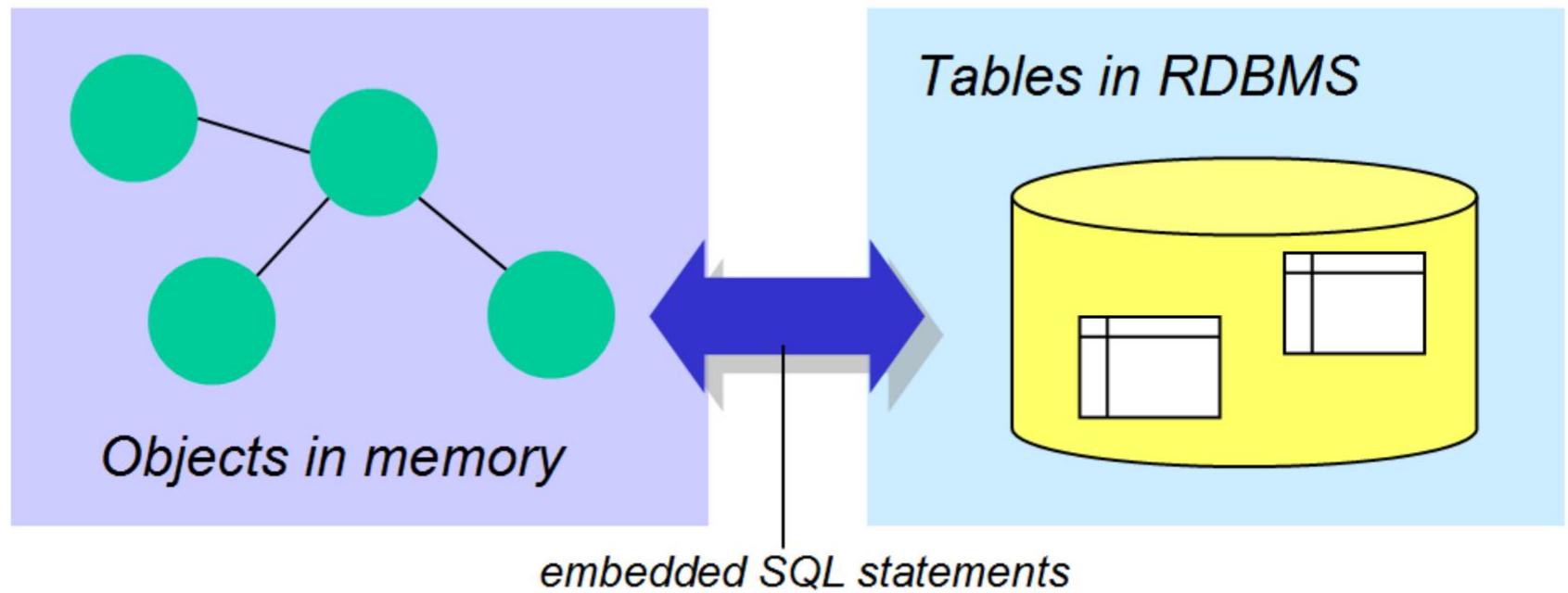


ORM, Un exemple

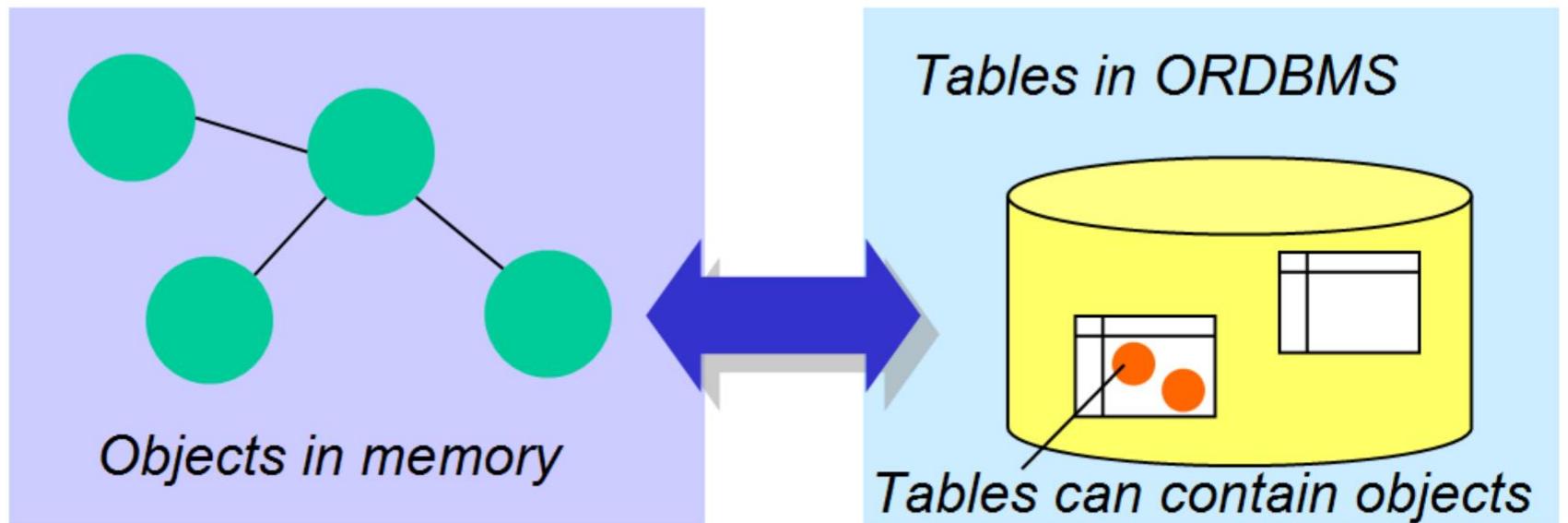
- ... qui doit contourner les limitations des RDBMS

```
public Employee( Integer id ) {  
    ...  
    ResultSet rs = p.executeQuery();  
    if( rs.next() )  
    {  
        ...  
        supervisor = ??  
    }  
}
```

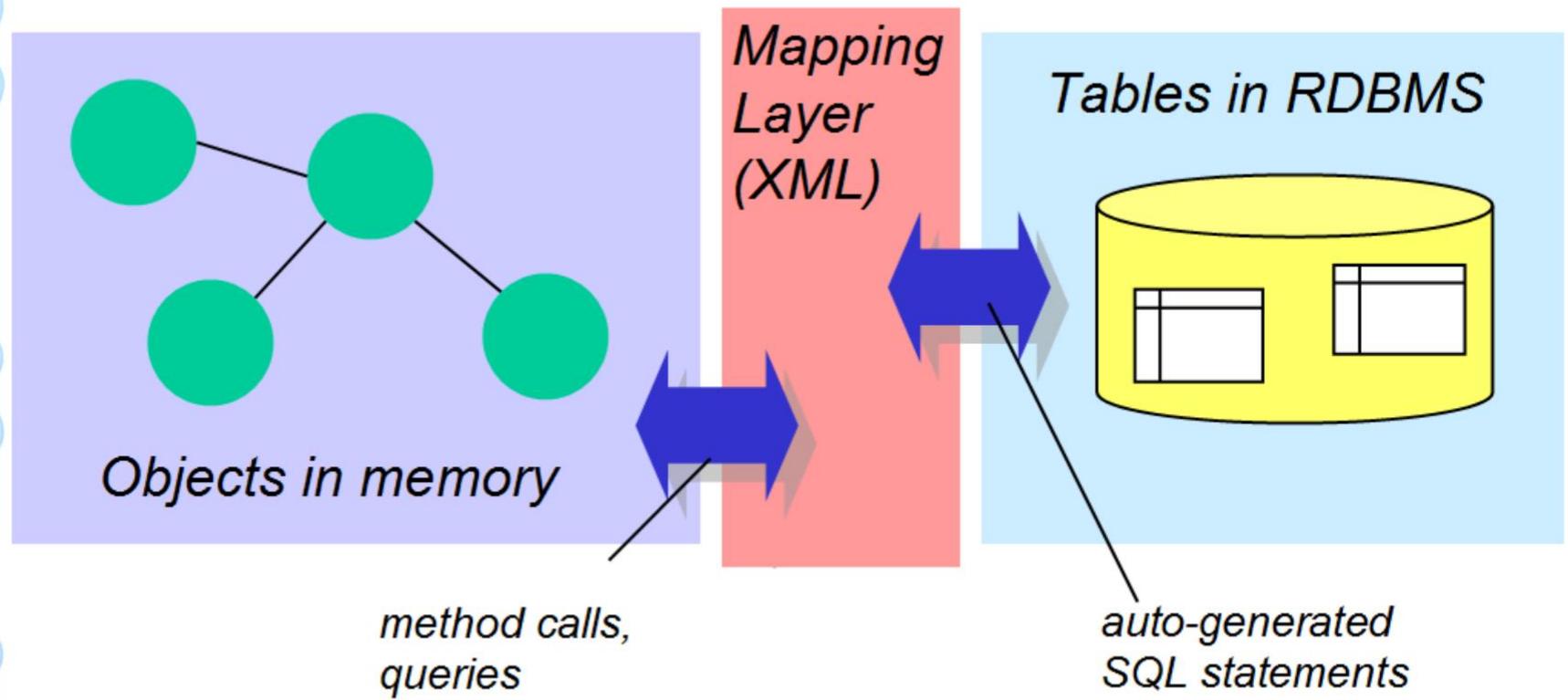
Le Mapping Objet Relationnel



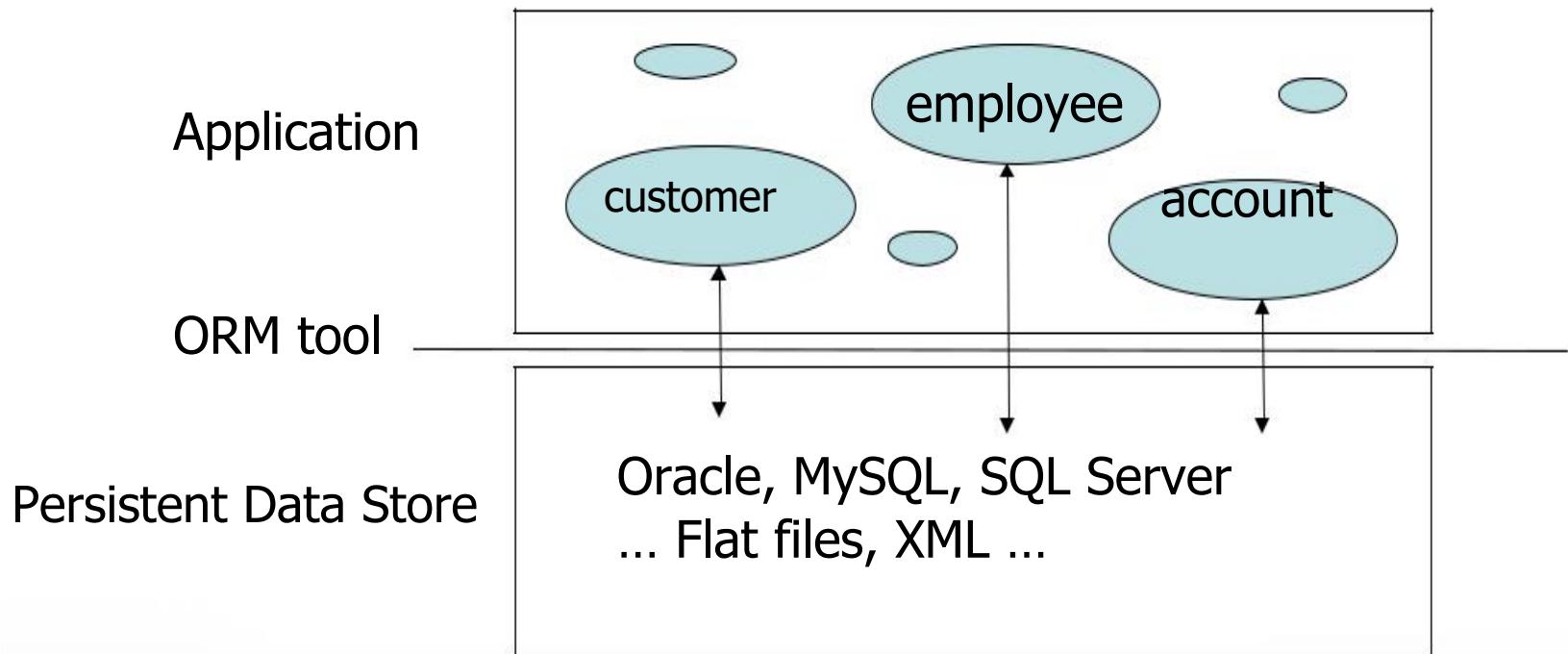
Stockage direct d'objets en bases

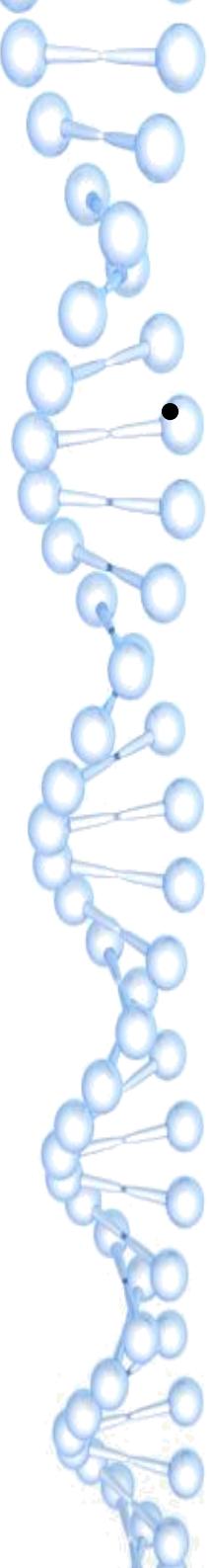


Couche de traduction : Mapping



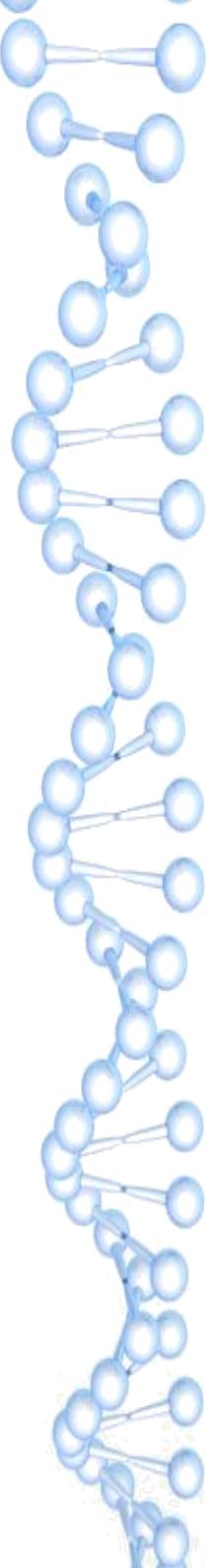
Le besoin ORM





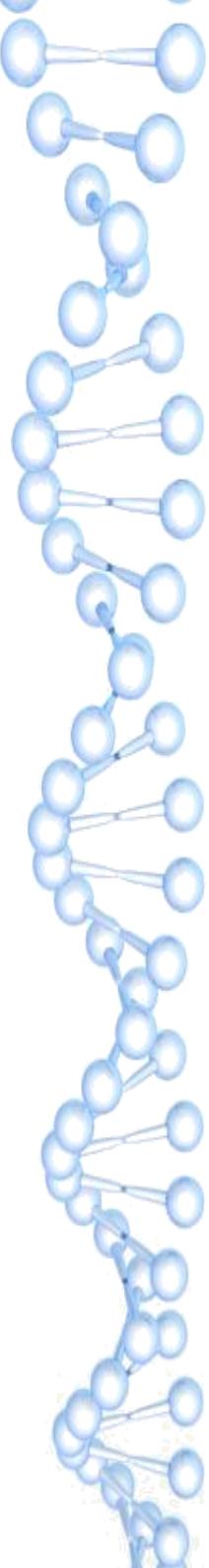
Exigences d'un ORM ?

- Une solution ORM répond aux exigences suivantes :
 - Gestion de la persistance offrant une API CRUD.
 - API de requêtage
 - Métadonnées de mapping
 - autre : transaction, lazy loading, caching...



Rappel CRUD

Create
Retrieve
Update
Delete



Mapping Objet Relationnel

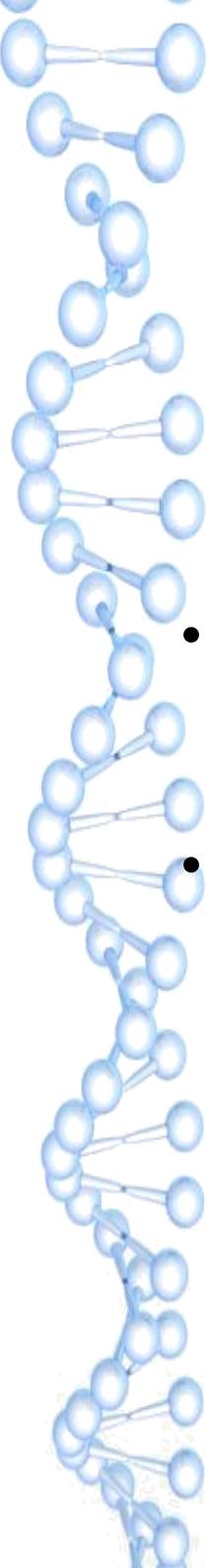
Les grandes questions ?

Quel design pour les classes de persistance ?

Comment définir les **métadonnées**?

Comment mapper les **hiérarchies** d'héritage ?

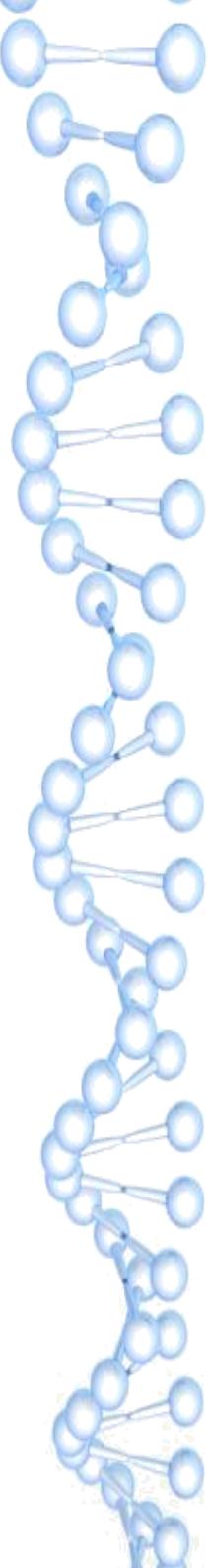
Comment résoudre la question des identités d'objets.



Mapping Objet Relationnel

Les grandes questions ?

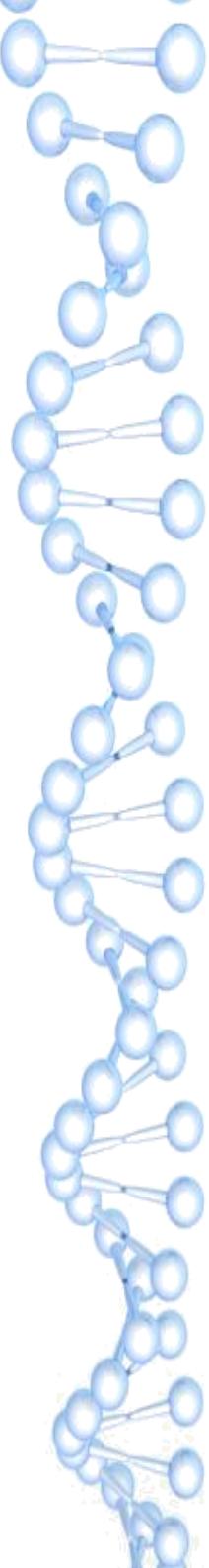
- Comment gérer l'interaction de la logique métier avec la logique de la couche de persistance ?
- Comment gérer le **CYCLE DE VIE** des entités ?



Mapping Objet Relationnel

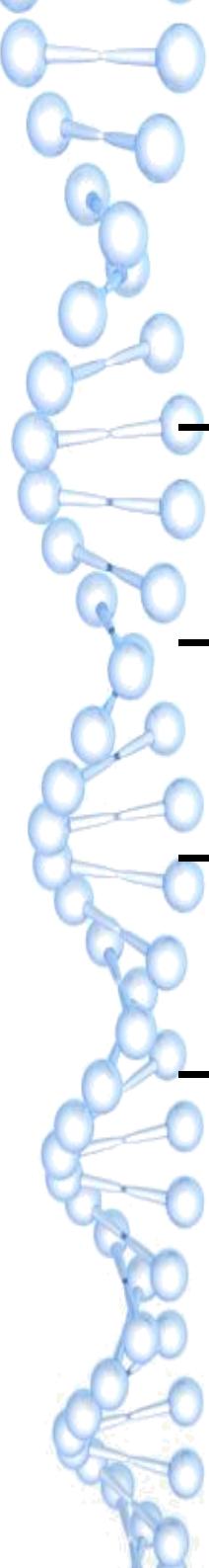
Les grandes questions ?

- Comment mettre en oeuvre le **tri, la recherche et l'agrégation?**
- Comment rendre les applications **performantes** ?



Solution : Framework ORM

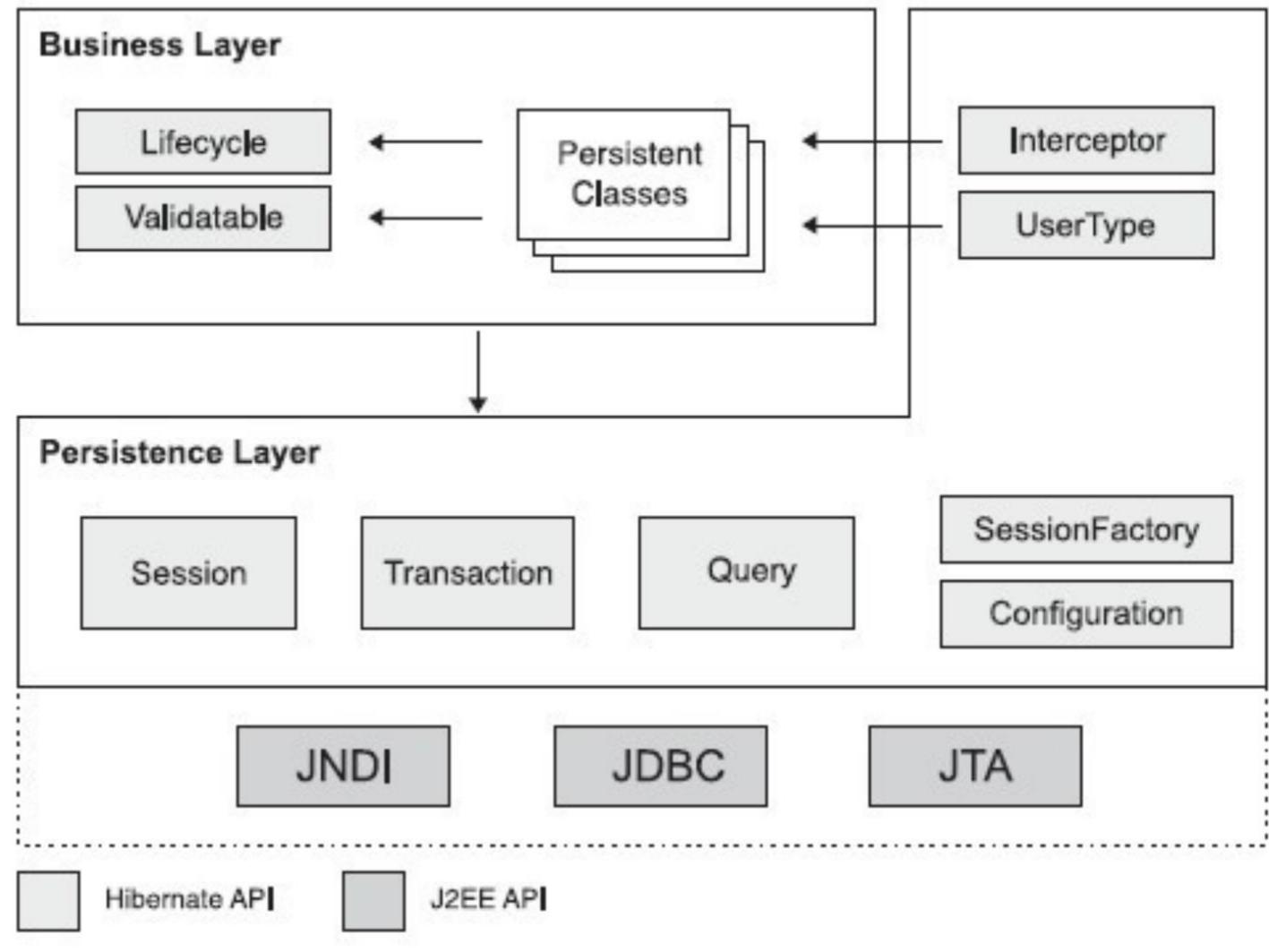
- Prend en charge la complexité
- Réutilisable
- Éprouvé
- Évolutif



L'intérêt des Frameworks

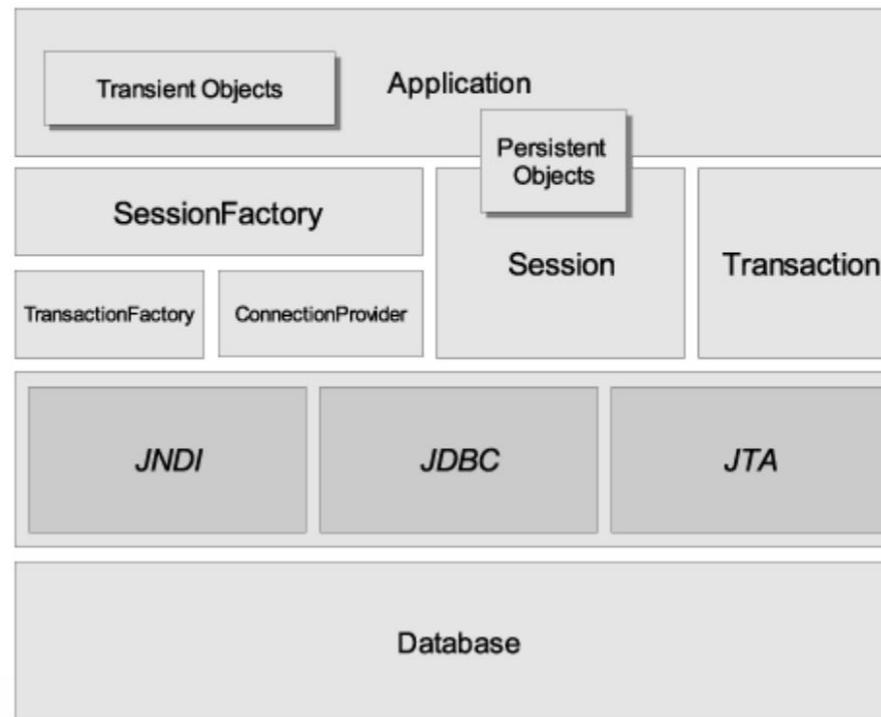
- Le programmeur se concentre sur les objets – pas de SQL dans le code
- Les objets persistants sur mise à jour de manière transparente
- Le framework gère le mapping des objets vers le RDBS
- Le mapping des objets vers la base relationnelle est décrit dans des fichiers XML

Hibernate - Architecture

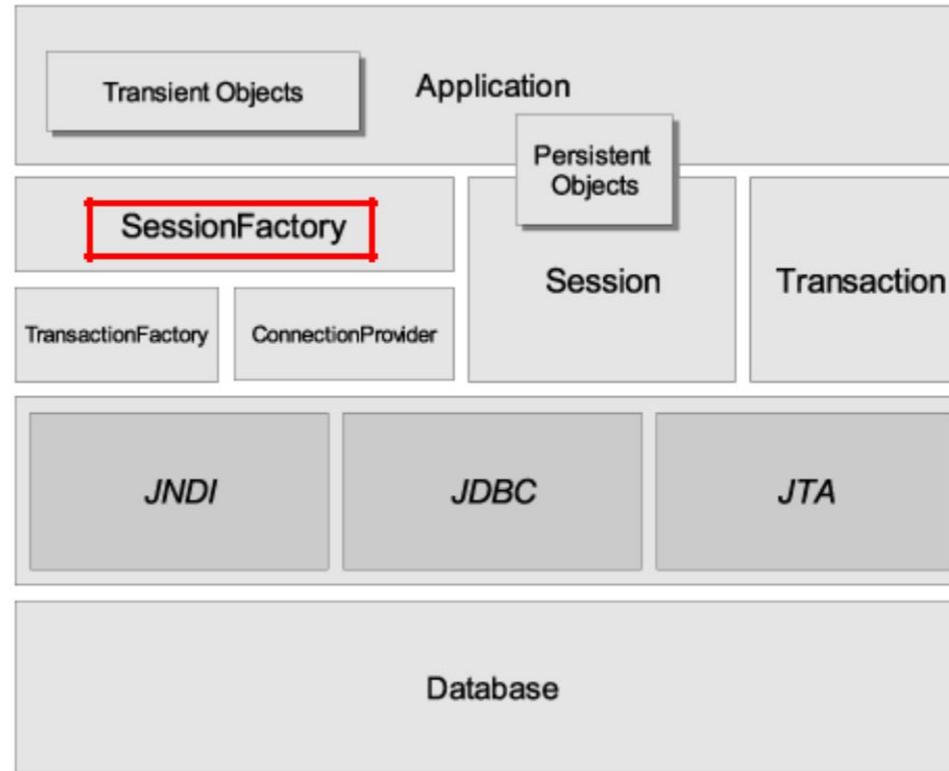


Hibernate – Architecture

Dans une architecture 3-tiers

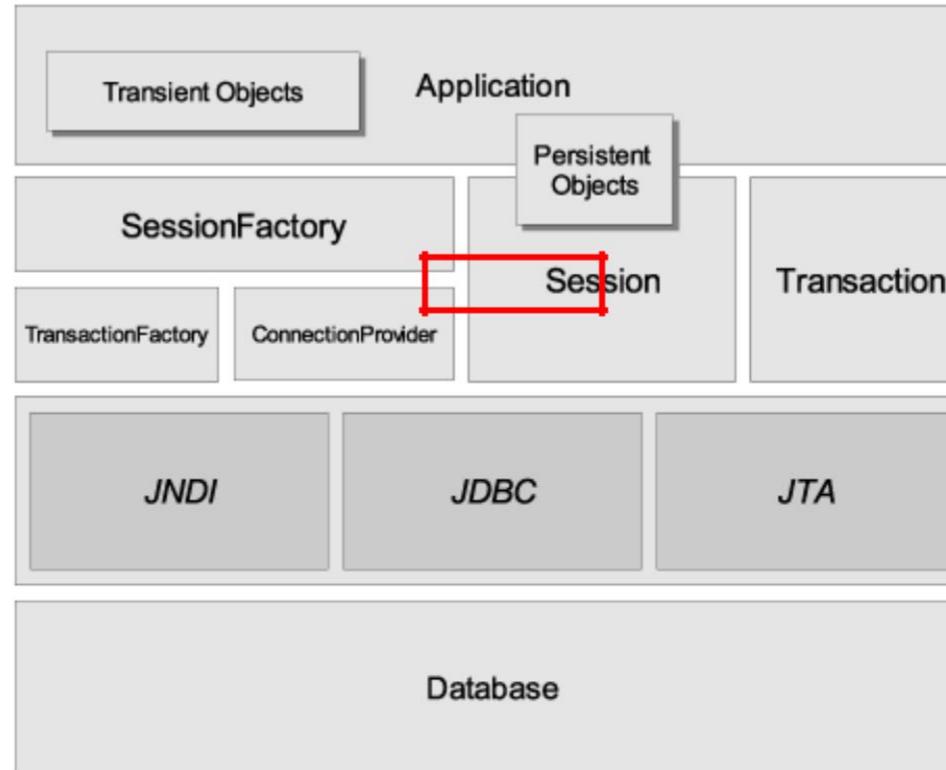


Hibernate – Aperçu



SessionFactory
crée la session (threadsafe)

Hibernate – Aperçu



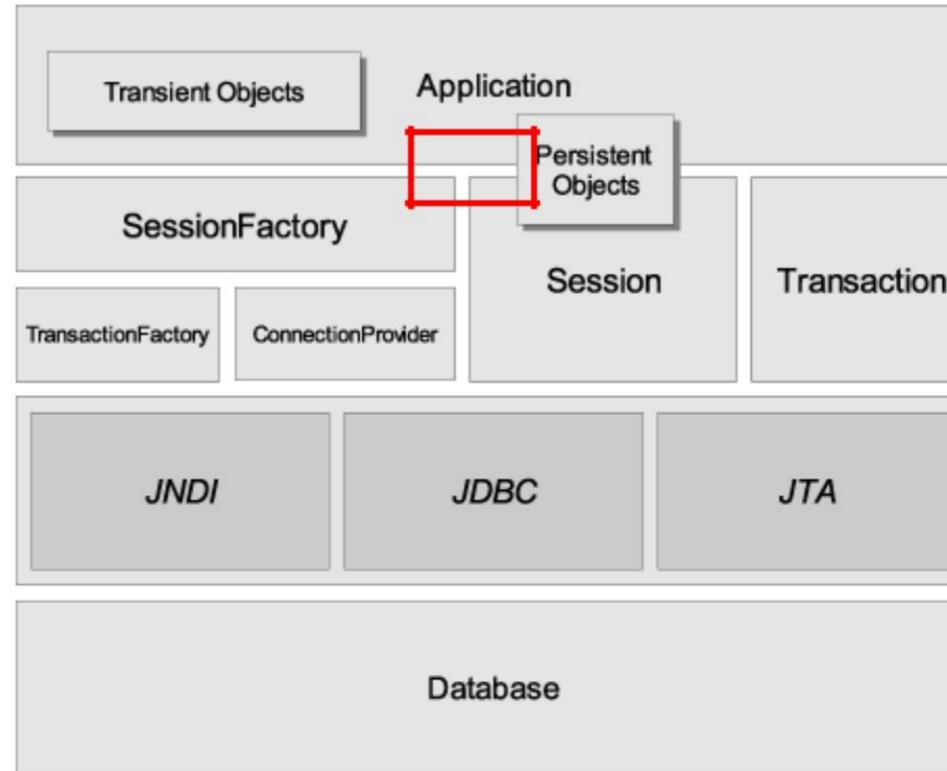
Session

Représente une conversation entre l'application et la source de données.

Utilise une connexion JDBC.

S'utilise comme cache de premier niveau pour naviguer dans les résultats

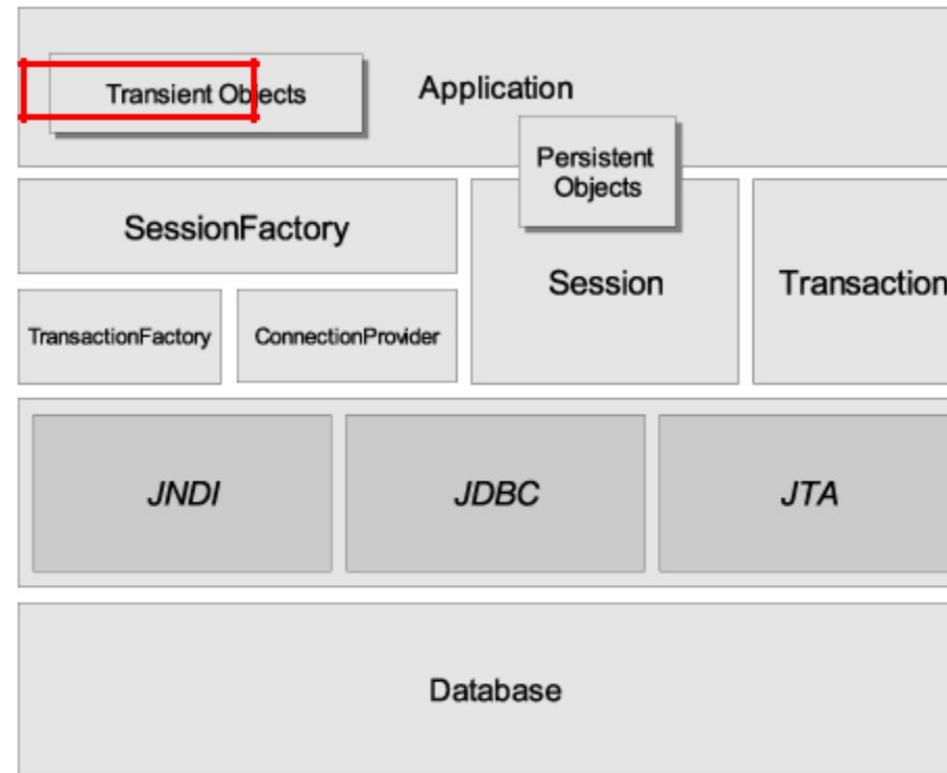
Hibernate – Aperçu



Les objets persistants

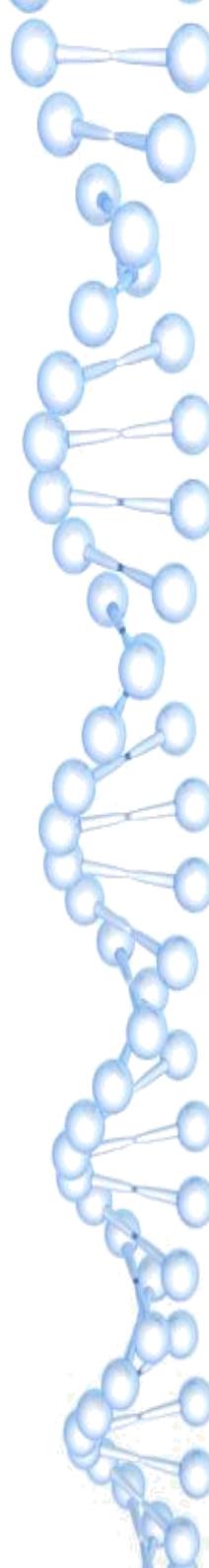
Ce sont de simple JavaBeans ou POJO, ils sont associés à une Session.

Hibernate – Aperçu

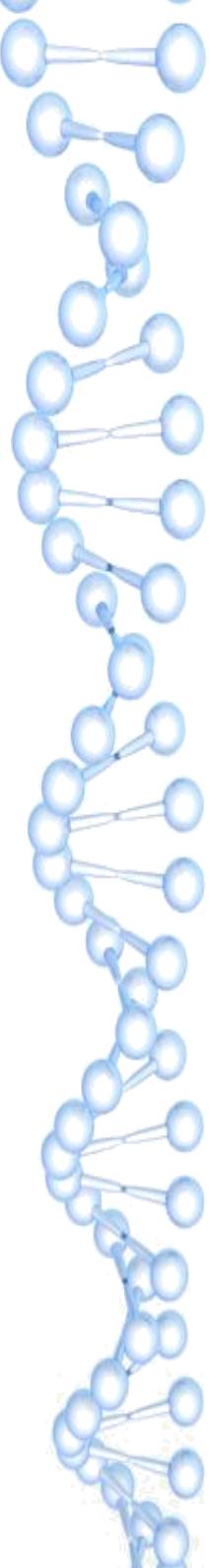


Objets Transient

Ce sont des instances de classes persistantes non-associées à une session

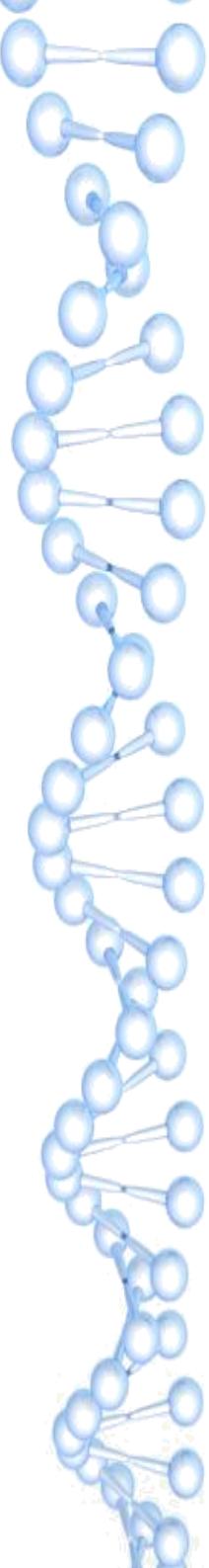


Stratégies de mapping



Les types de relations objet

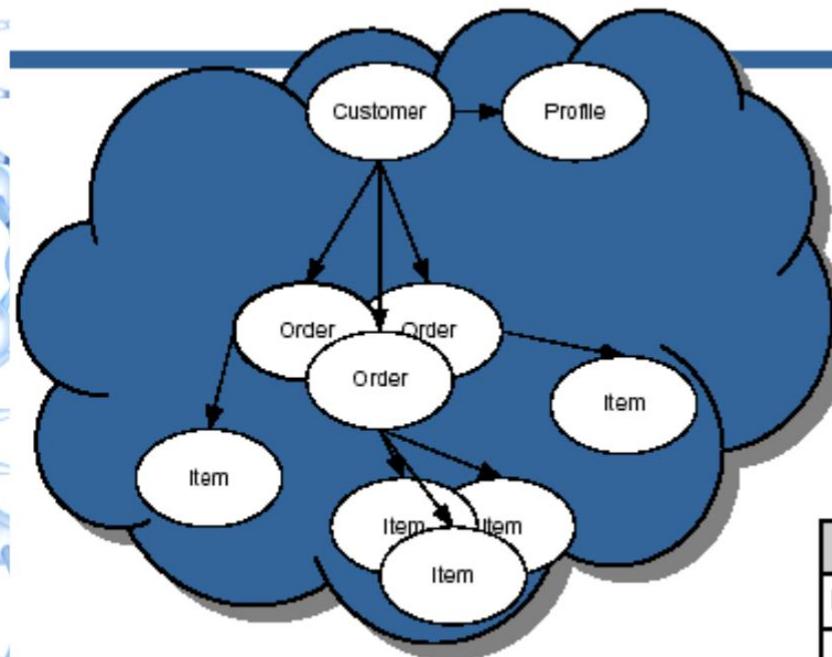
- La cardinalité
 - 1 vers 1 :
 - 1 vers plusieurs
 - plusieurs vers plusieurs
- La direction
 - Uni-directionnelle
 - bidirectionnelle



Le monde objet est plus complexe

- Pas de lien bidirectionnel implicite
- Difficulté d'identification de l'identité
- Pas (toujours) de classe d'association
- L'héritage
- L'encapsulation

Mapping de l'héritage

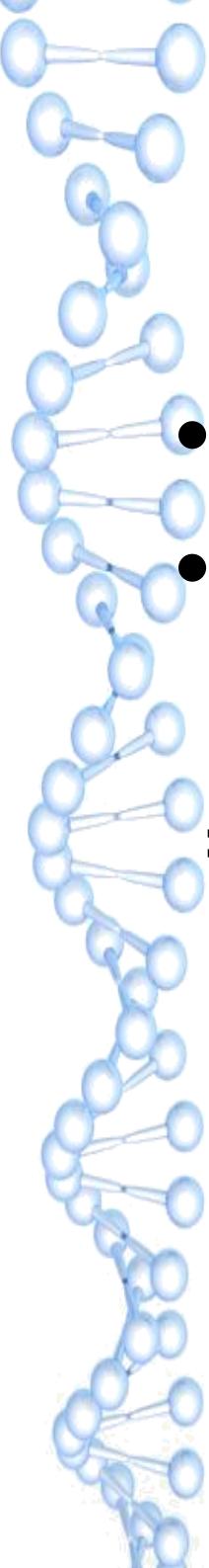


The Logical World

The Physical World

The Physical World diagram shows five database tables corresponding to the logical entities:

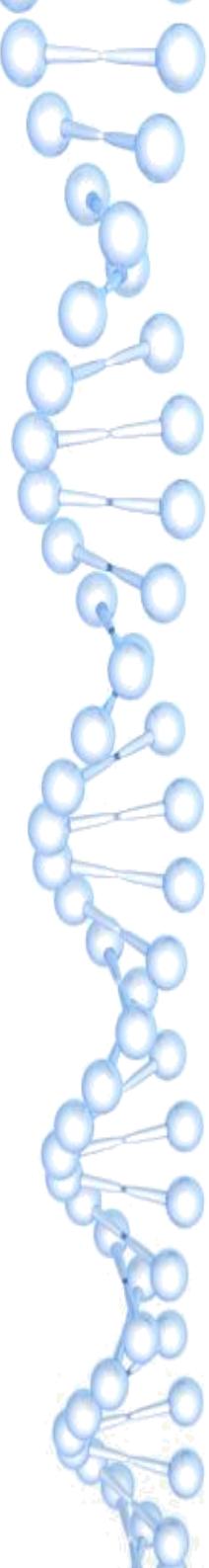
- Customer**: PK Oid. Columns: FirstName, LastName, CustomerNbr.
- Order**: PK Oid. Columns: Quantity, PONumber, ShippingInstructions, OrderDateTm.
- Item**: PK Oid. Columns: Description, Availability, LeadTimeMin, LeadTimeMax.
- CustomerProfile**: PK Oid. Columns: Type, ShowRecentOrder, ViewSpecials.
- LineItem**: PK Oid. Columns: Quantity, OrderUOM, OrderId, ItemId.



Mapping de l'héritage

- SGBDR pas d'héritage
- Dans le monde Objet : héritage

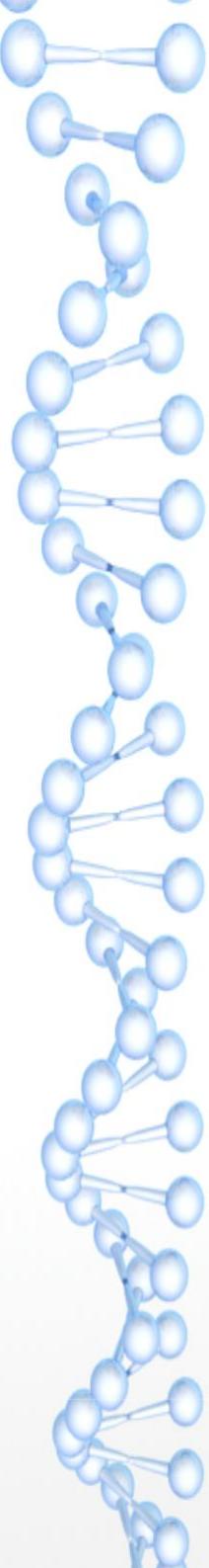
=> Il faut « aplatisir » le graphe pour le ranger dans la base de données relationnelle



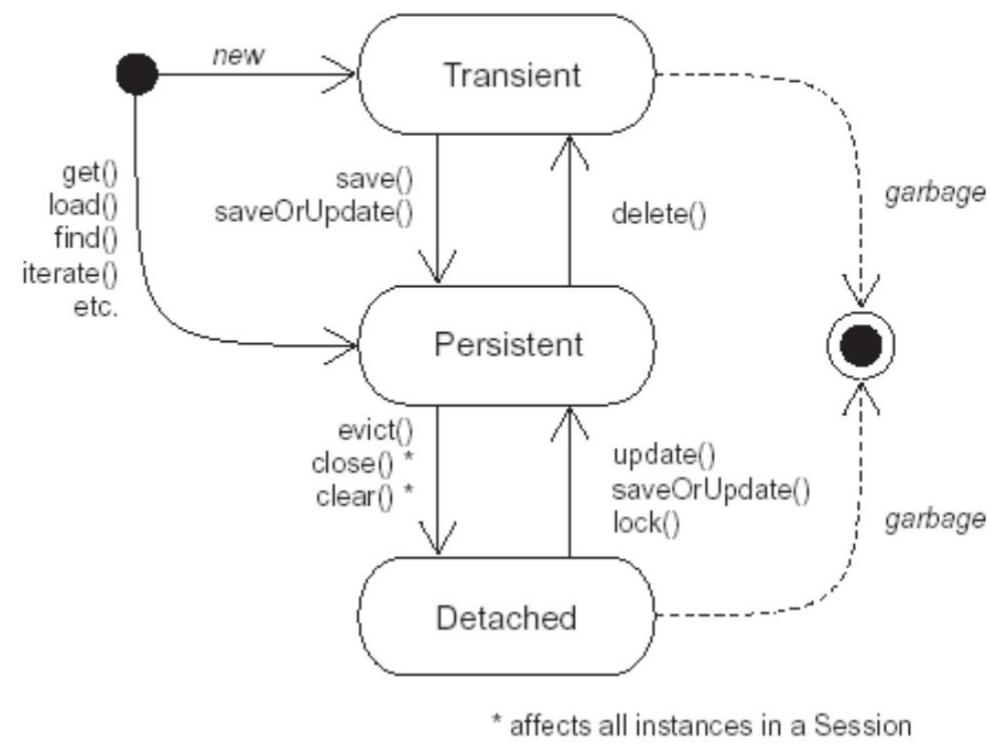
Mapping de l'héritage : 3+1Solutions

- Une table pour toute la hiérarchie
- Une table pour chaque classe concrète
- Une table par classe
- Schéma générique

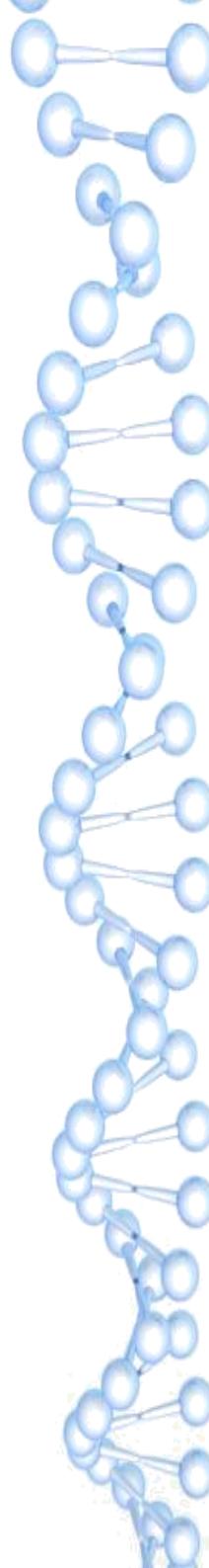
=> La plus commune : une table par classe



Session et Cycle de vie de l'objet persistant

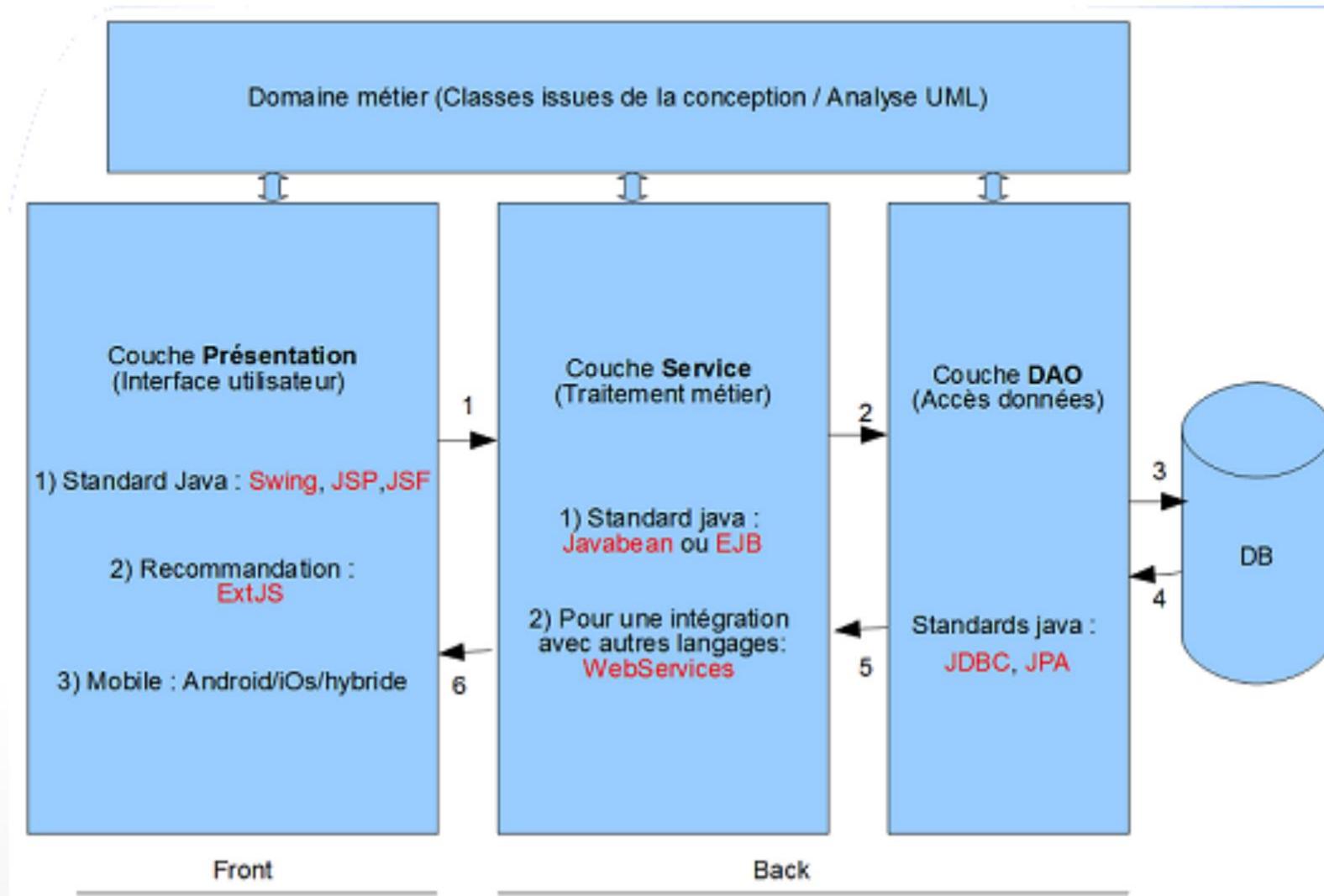


- Transient
- Persistent
- Detached

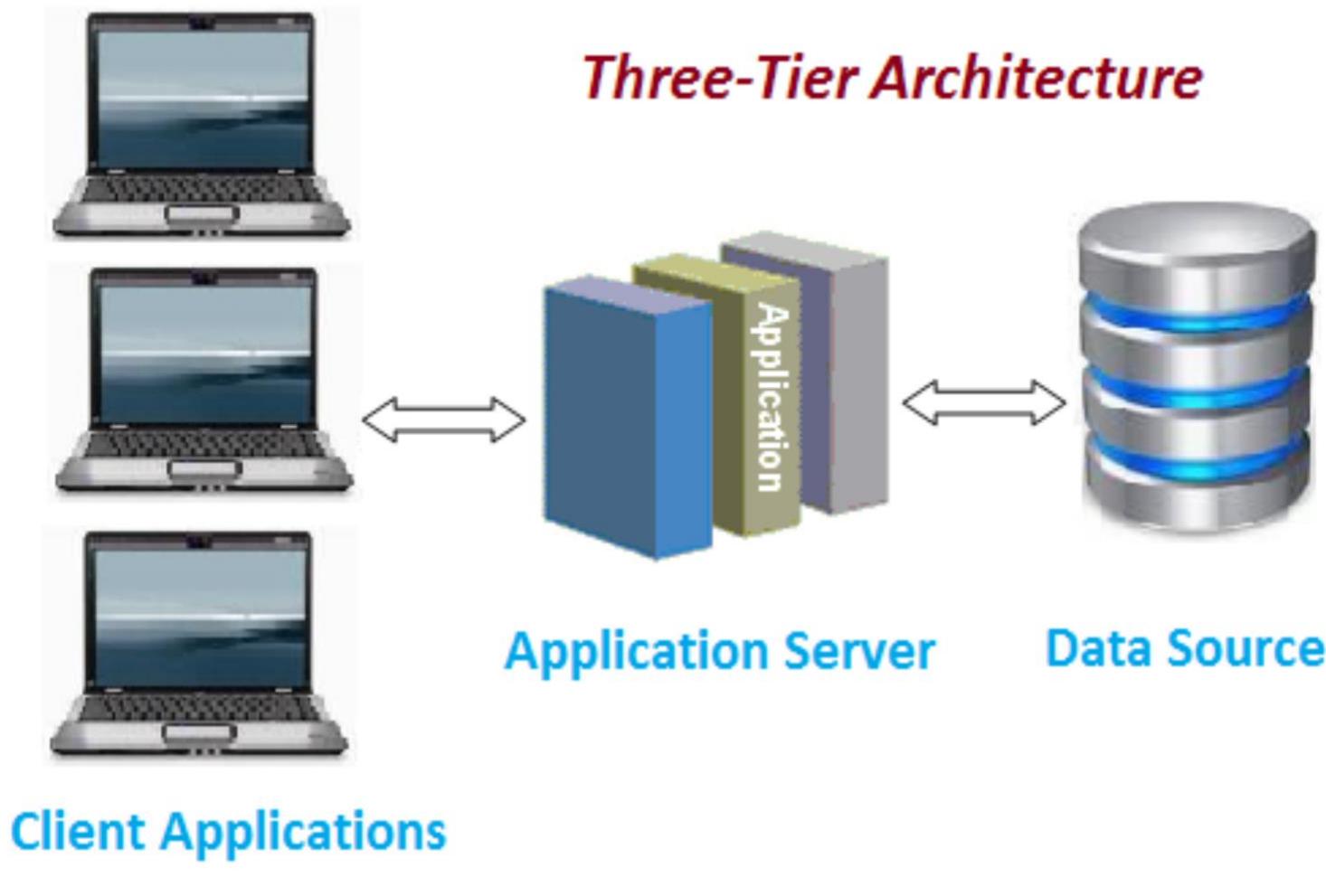


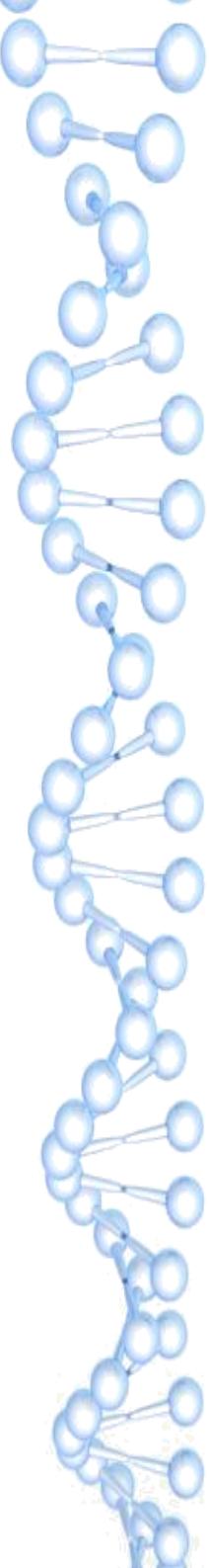
Applications d'Entreprise

Enjeux du développement JEE



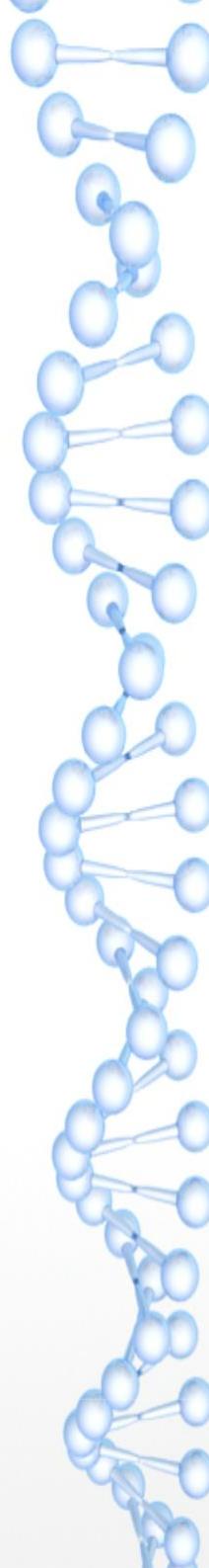
Enjeux du développement JEE





Architecture JEE

- . Application JEE met en œuvre des éléments très techniques.
- . Log, persistance, accès aux ressources, sécurité
- . Évolution des technologies
- . Ne doit pas compliquer le code
- . Ne doit pas remettre en cause le code
- . Testable
- . Séparer le code métier



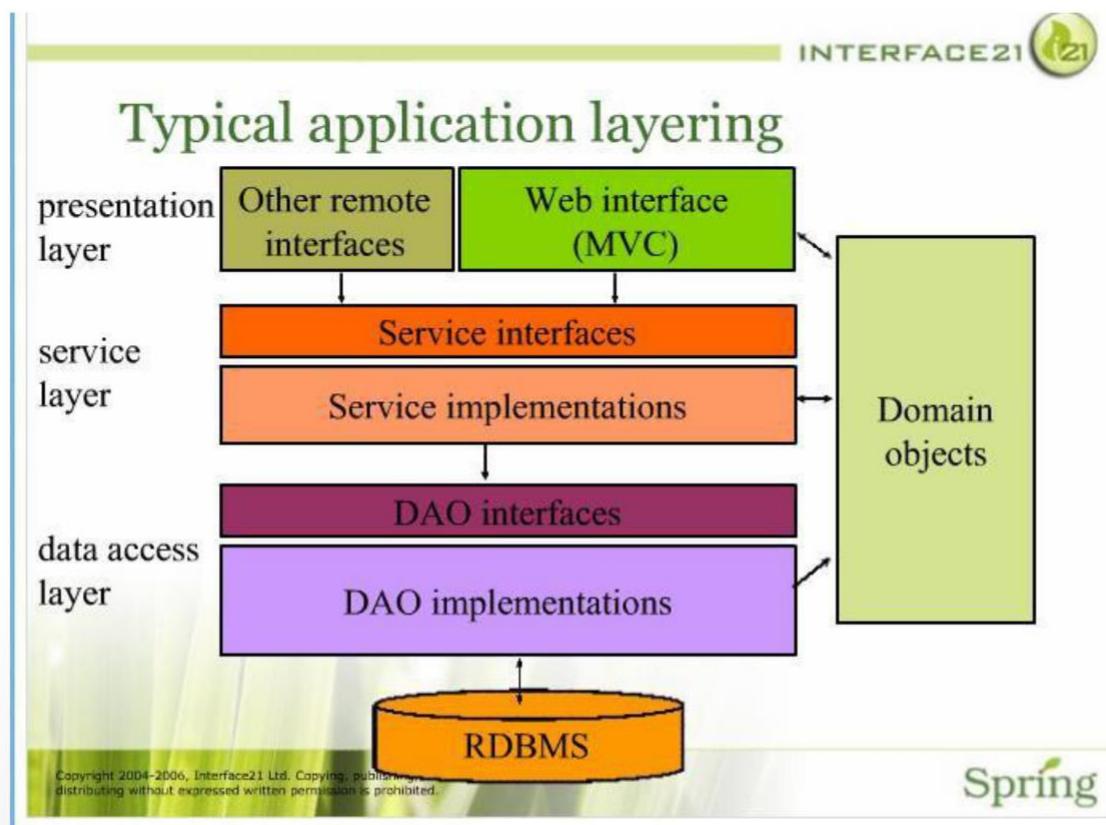
Architecture JEE

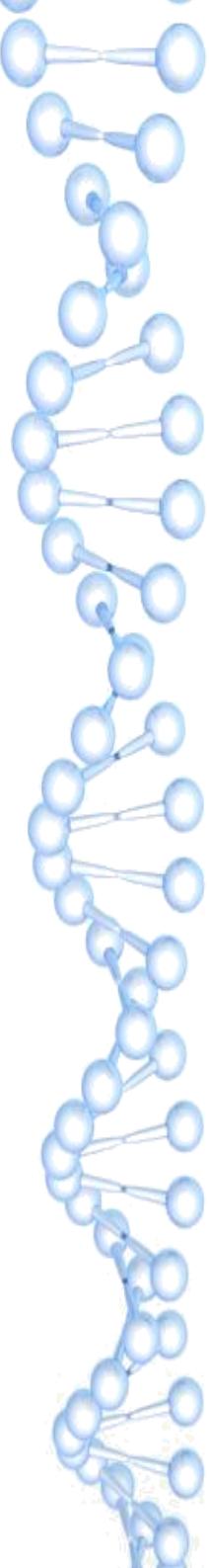
- . L'objectif : Éviter ça !



Architecture JEE

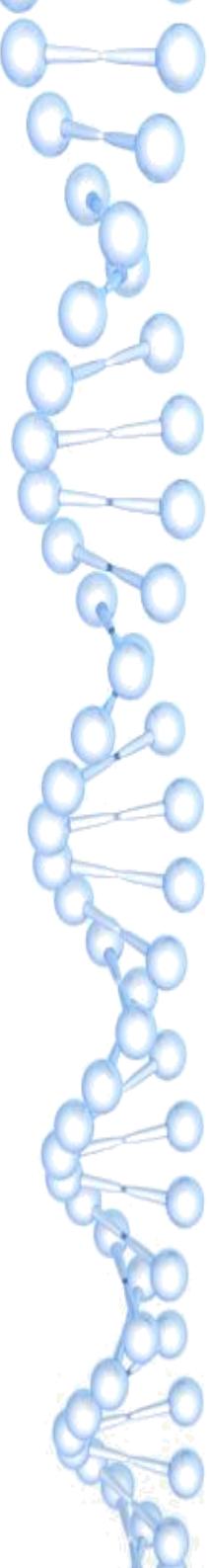
- . Architecture en couches
 - Permet de séparer les grandes responsabilités





Architecture JEE

- La couche du dessus utilise la couche du dessous
- La couche du dessous ne connaît pas la couche du dessus.
- En Java, la communication entre couches s'effectue en établissant des contrats via des interfaces.



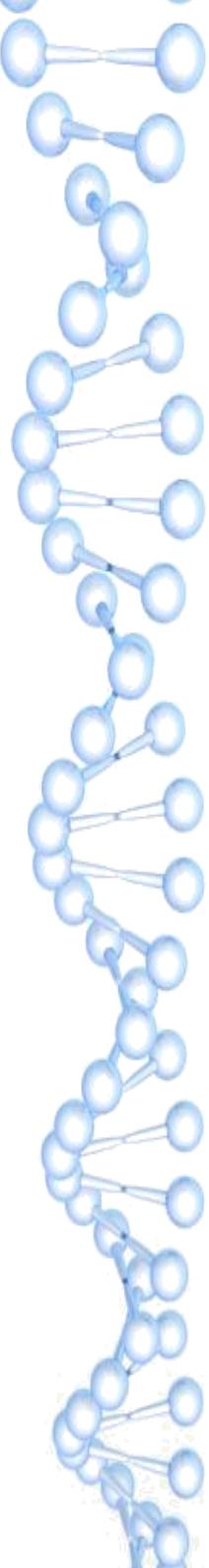
Architecture JEE

- . La couche métier

Représente des concepts génériques et stables et les règles de gestion associées : Le modèle métier.

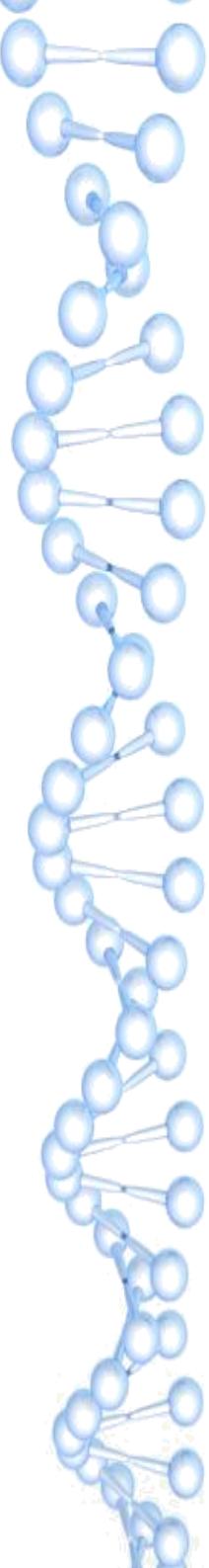
- . la couche d'accès aux données

Gère l'accès aux référentiels de données de l'application
assure le découpage entre le tiers métier et
les différentes technologies d'accès aux données



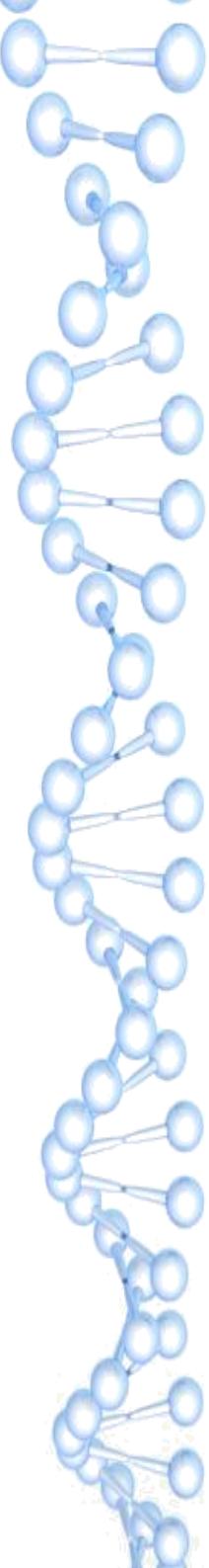
Architecture JEE

- . La couche applicative ou service
 - Offre des services directement utilisables par la couche présentation. Assemble les services métier. Par nature, peu réutilisable
- . La couche présentation
 - Mise en forme des données en provenance de la couche applicative et leur présentation aux clients de l'application.
 - Prend en charge l'affichage des pages et l'interaction avec les périphériques manipulés par l'utilisateur.
 - S'adresse uniquement à la couche applicative



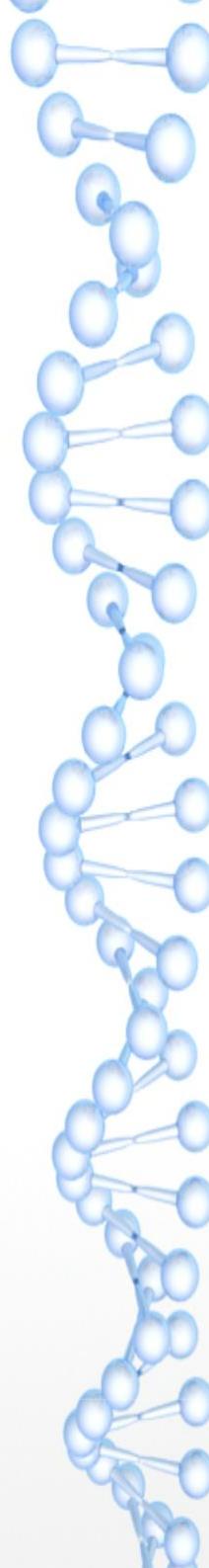
Architecture JEE

- . Préoccupations transverses
 - Localisation des composants des couches supérieures et inférieures
 - Gestion des transactions
 - Gestion des exceptions
 - Gestion des logs
 - Sécurité



Architecture JEE

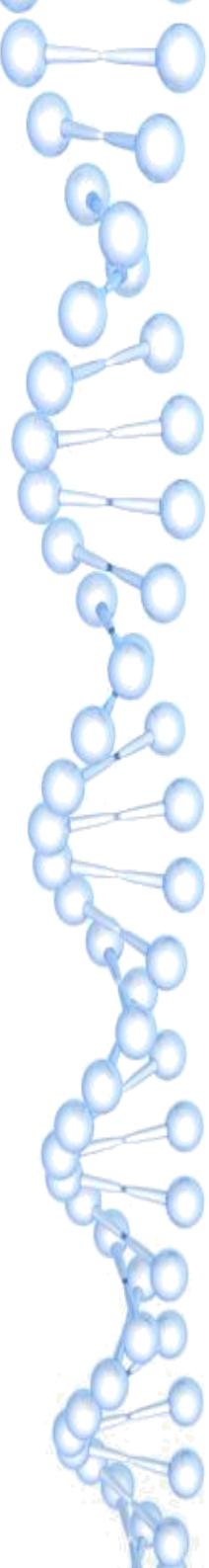
- . Réduire les dépendances.
 - Suivre les évolutions techniques sans impacter le code métier.
 - Tendre vers un couplage faible (interfaces). Le code doit pouvoir être facilement testé



Architecture JEE

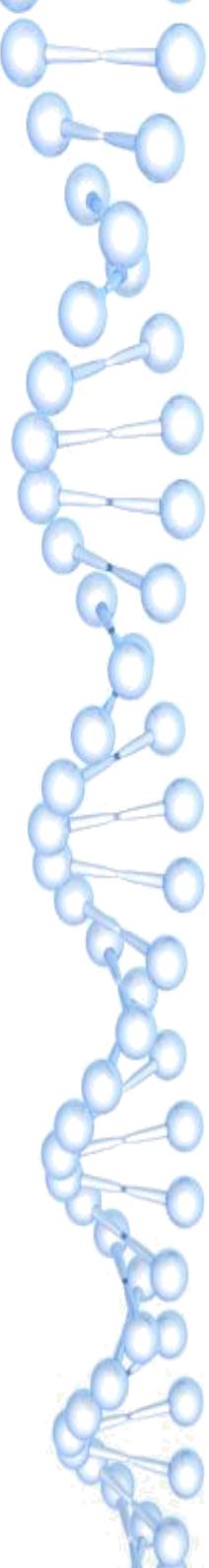
- . Besoin de Cohésion :
Chaque classe assume uniquement sa propre responsabilité





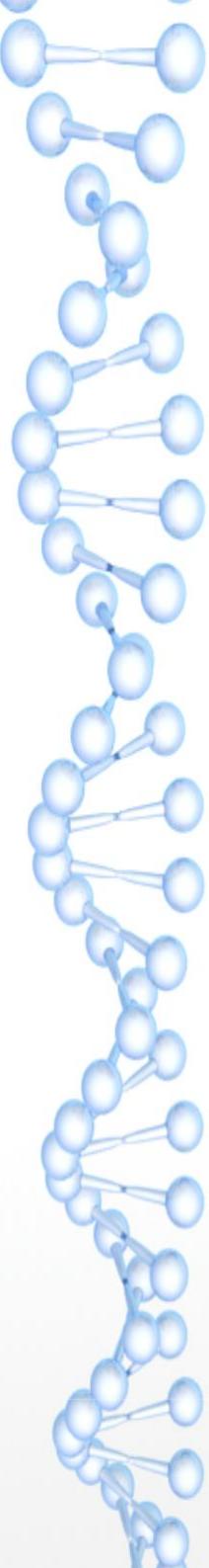
Architecture JEE

- . Separation Of Concerns
 - Consiste à isoler les différentes problématiques à traiter, par
 - La Présentation (IHM, ...)
 - Les Services Métiers
 - La Persistance
 - Bref, des composants spécialisés



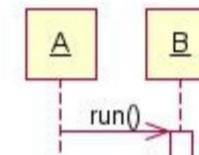
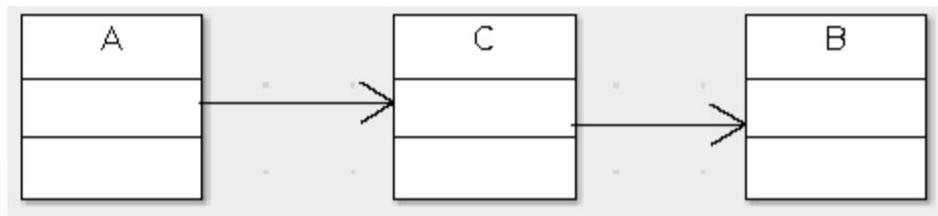
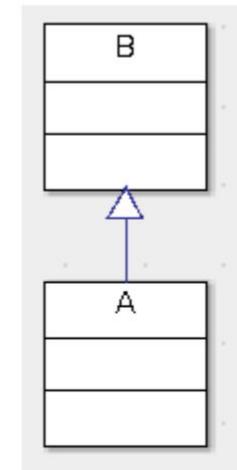
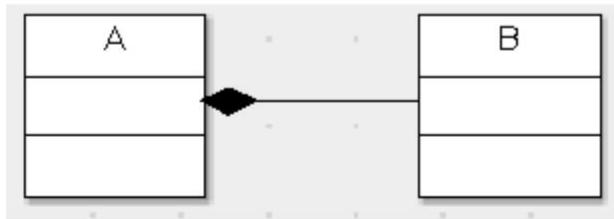
Architecture JEE

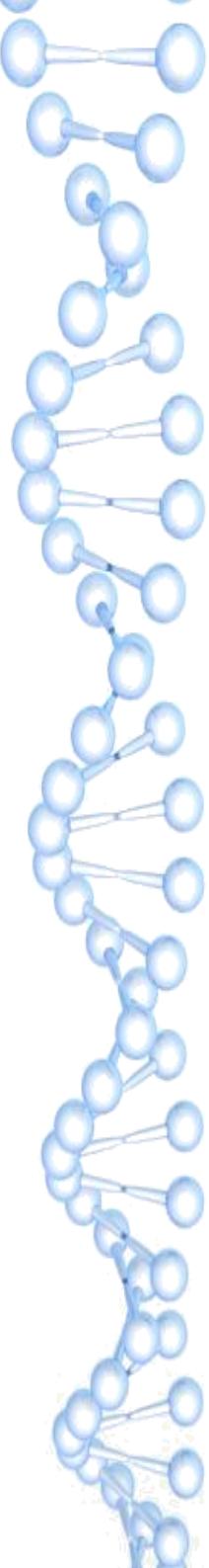
- . Besoin : limiter les dépendances :
- . En programmation objet, un objet de type A dépend d'un objet de type B si au moins une de ces conditions est vérifiée :
 - 1.A possède un attribut de type B (dépendance par composition)
 - 2.A est de type B (dépendance par héritage)
 - 3.A dépend d'un autre objet de type C qui dépend d'un objet de type B (dépendance transitive)
 - 4.une méthode de A appelle une méthode de B



Le framework Spring :Architecture JEE

Dépendances



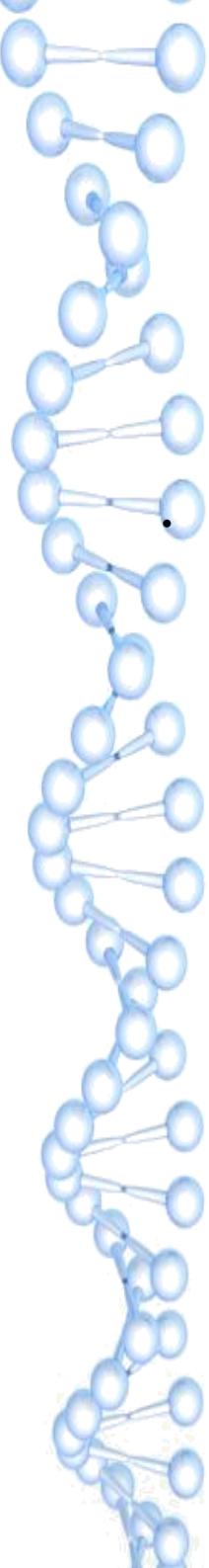


Architecture JEE

Approche "naïve" :

`B b = new B()`

- Complètement couplé
- Pas de singleton
- A dépend de la classe concrète
- Signature du constructeur ne peut pas évoluer



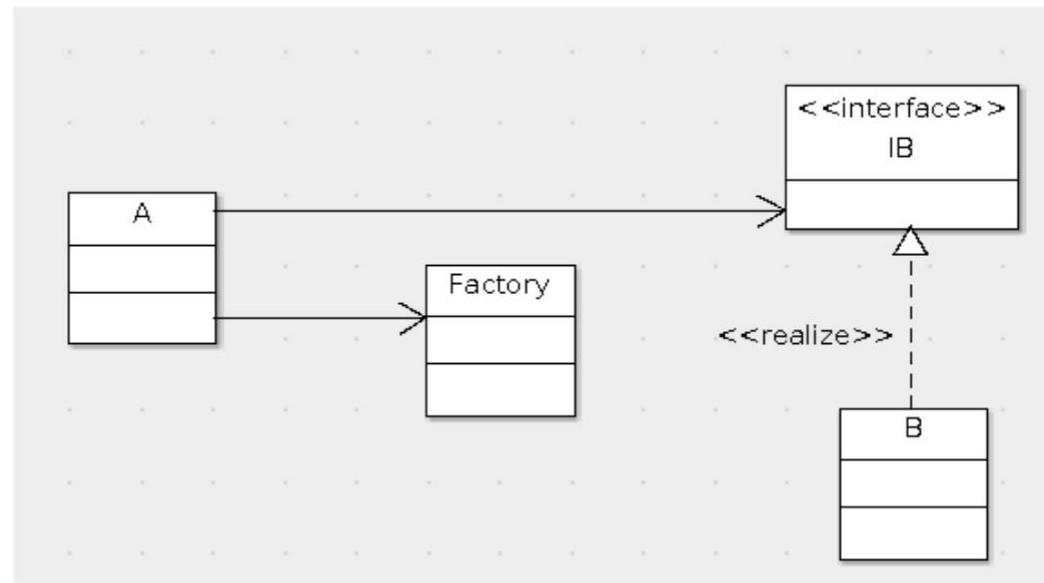
Spring :Architecture JEE

- . Les dépendances ne permettent pas :
 - D'installer A sans installer B
 - De réutiliser A sans réutiliser B
 - Une modification de B
 - . Peut entraîner une modification de A
 - . Une recompilation de A

Spring : Architecture JEE

Mieux : Interface et Factory

```
A { IB b = Factory.getB() ; }
```

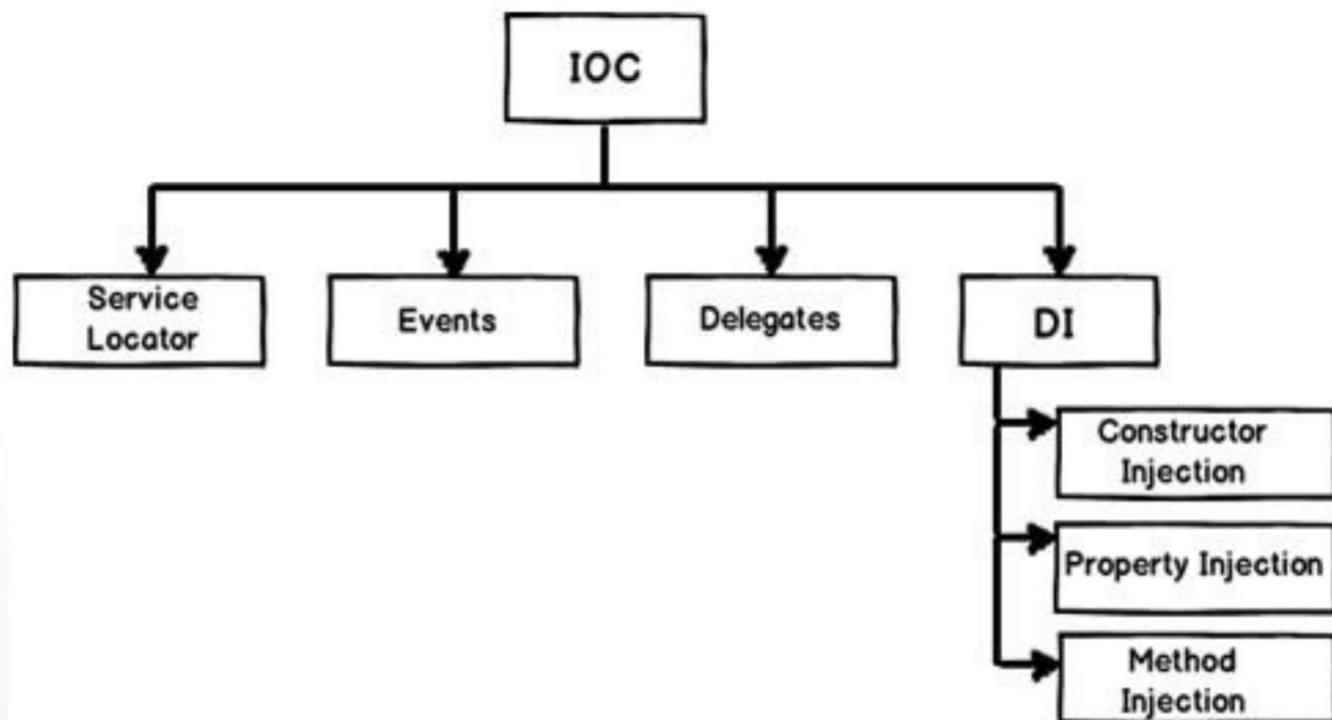


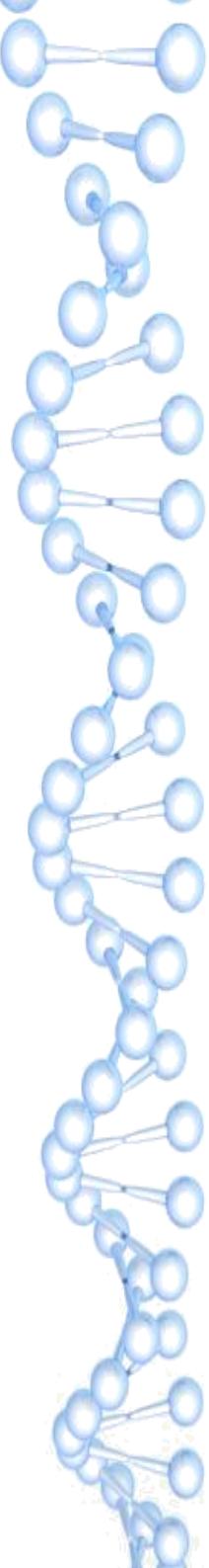
B peut évoluer sans impacter A

Problème restant : Le composant A est à l'initiative de la demande

Spring :Architecture JEE

- . Solution : l'inversion de contrôle
 - Le conteneur gère les dépendances à la place du composant



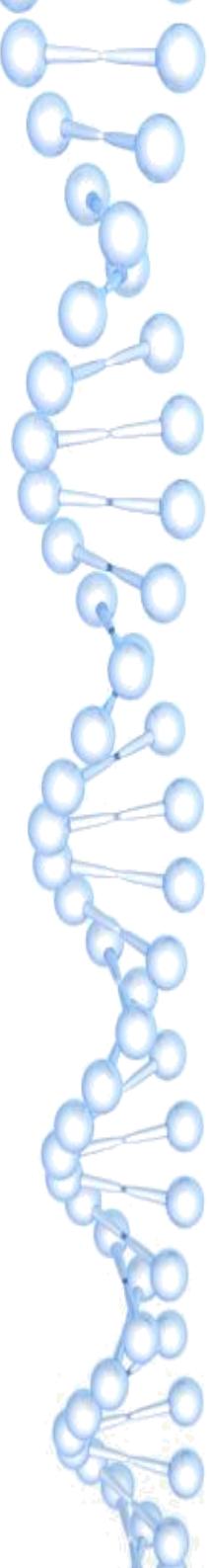


Spring :Architecture JEE

- . Solution : l'inversion de contrôle
 - Le conteneur gère les dépendances à la place du composant
 - Recherche de dépendance
 - Utilisation d'une entité de résolution (JNDI par exemple)
 - InitialContext ctx = new InitialContext();
ds = (DataSource)ctx.lookup(JndiDataSourceName)

Problème :

- méthode à l'initiative du composant donc intrusive
- pas de typage

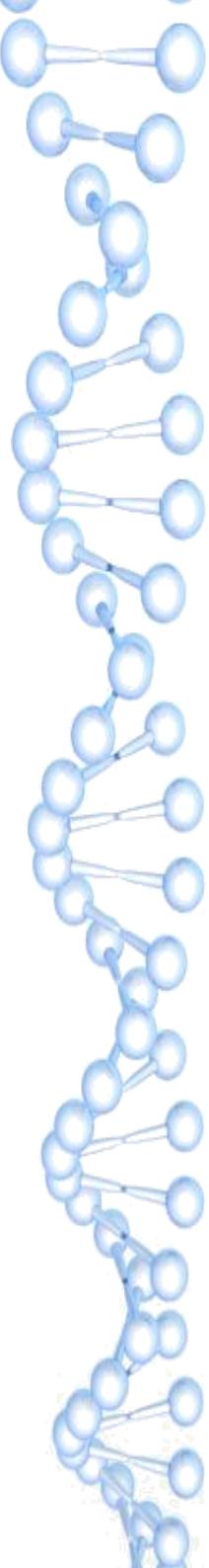


Spring :Architecture JEE

Solution :

IOC par DI : Injection de Dépendance

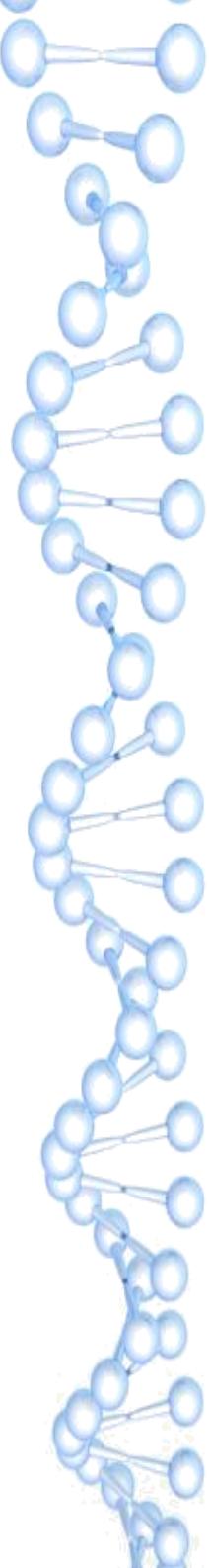
- . A n'est pas à l'initiative
- . Hollywood principle :
 - Don't call me ; I'll call you !
 - C'est le conteneur qui s'en charge



Spring :Architecture JEE

. Avantages

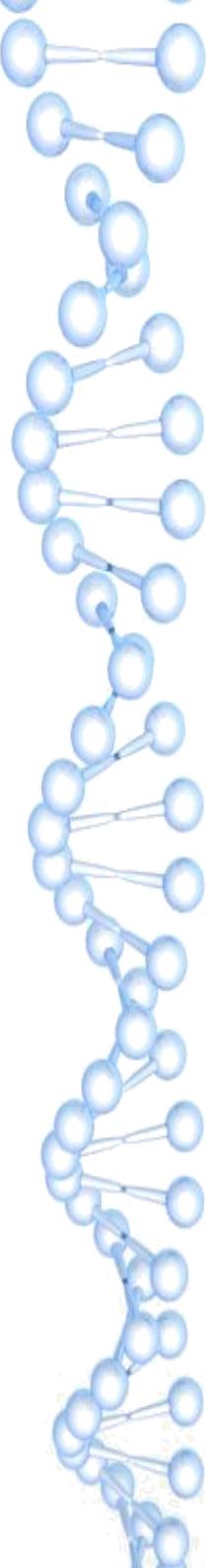
- Construction d'applications par assemblage de composants.
- Couplage faible
- Pas de dépendance avec le conteneur :
- Pas besoin d'hériter de classes du conteneur.
- Les composants sont de simples POJOs



Spring :Architecture JEE

. Avantages

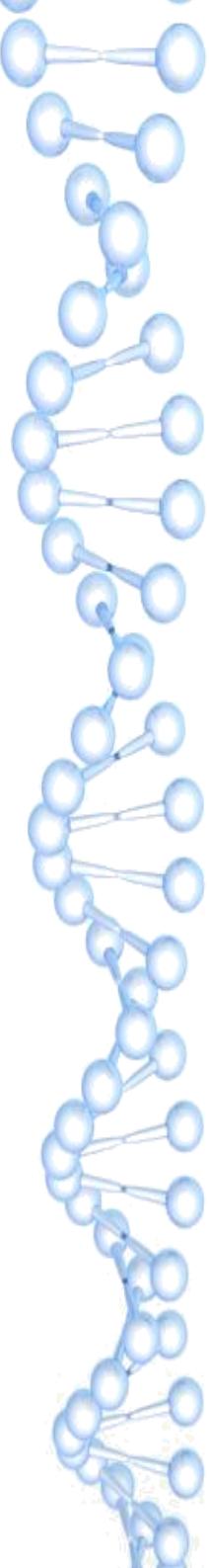
- Code métier simple
- Utilisation des interfaces
- Masquage de l'implémentation
- Test et bouchonnage facile
- Externalisation du comportement des factories
- Possibilité d'utiliser des intercepteurs Log, Cache, Sécurité



Spring :Architecture JEE

. Avantages

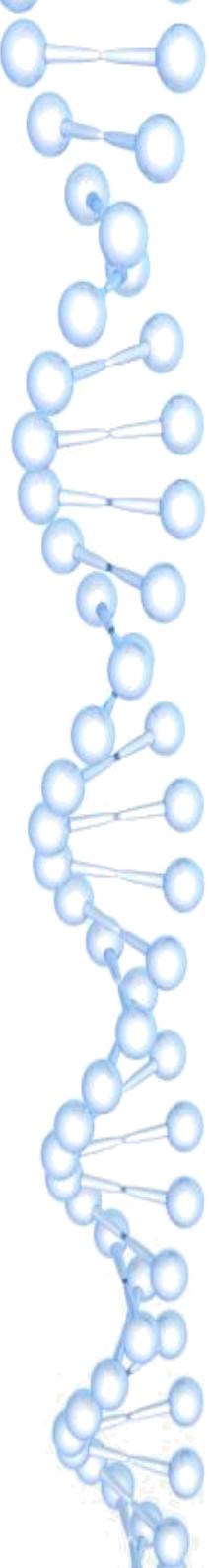
- Construction d'applications par assemblage de composants.
- Couplage faible
- Pas de dépendance avec le conteneur :
- Pas besoin d'hériter de classes du conteneur.
- Les composants sont de simples POJOs



Spring Framework

Objectifs de Spring

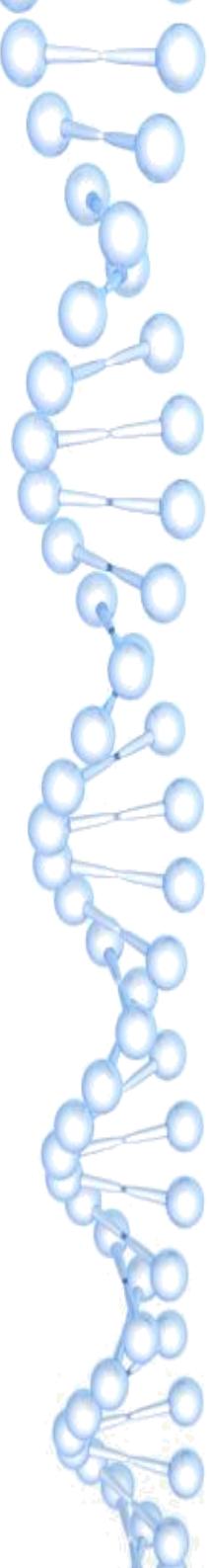
- . Simplifier les développements Java / JEE
- . Faciliter la mise en œuvre de bonnes pratiques
- . Alléger l'architecture des applications JEE
- . Favoriser le découplage des composants
- . Favoriser la mise en œuvre des tests unitaires
- . Réaliser simplement les traitements simples
- . Permettre l'intégration de traitements plus complexes



Spring Framework

Objectifs de Spring

- . Fondé sur les POJO (plain old java object)
- . Injection de dépendances (IoC) et programmation orientée Aspect (AOP)
- . Minimiser les dépendances au framework
- . Spécification de comportements par déclaration
- . Gestion simplifiée des exceptions
- . Testabilité des composants



Spring Framework

- . Spring est un conteneur « léger »

Par opposition aux conteneurs dits "lourds" JEE (notamment avant les EJB 3) :

Environnement inadapté pour gérer des composants simples

Règles de développement et de déploiement contraignantes

- . Spring est adapté au développement JEE

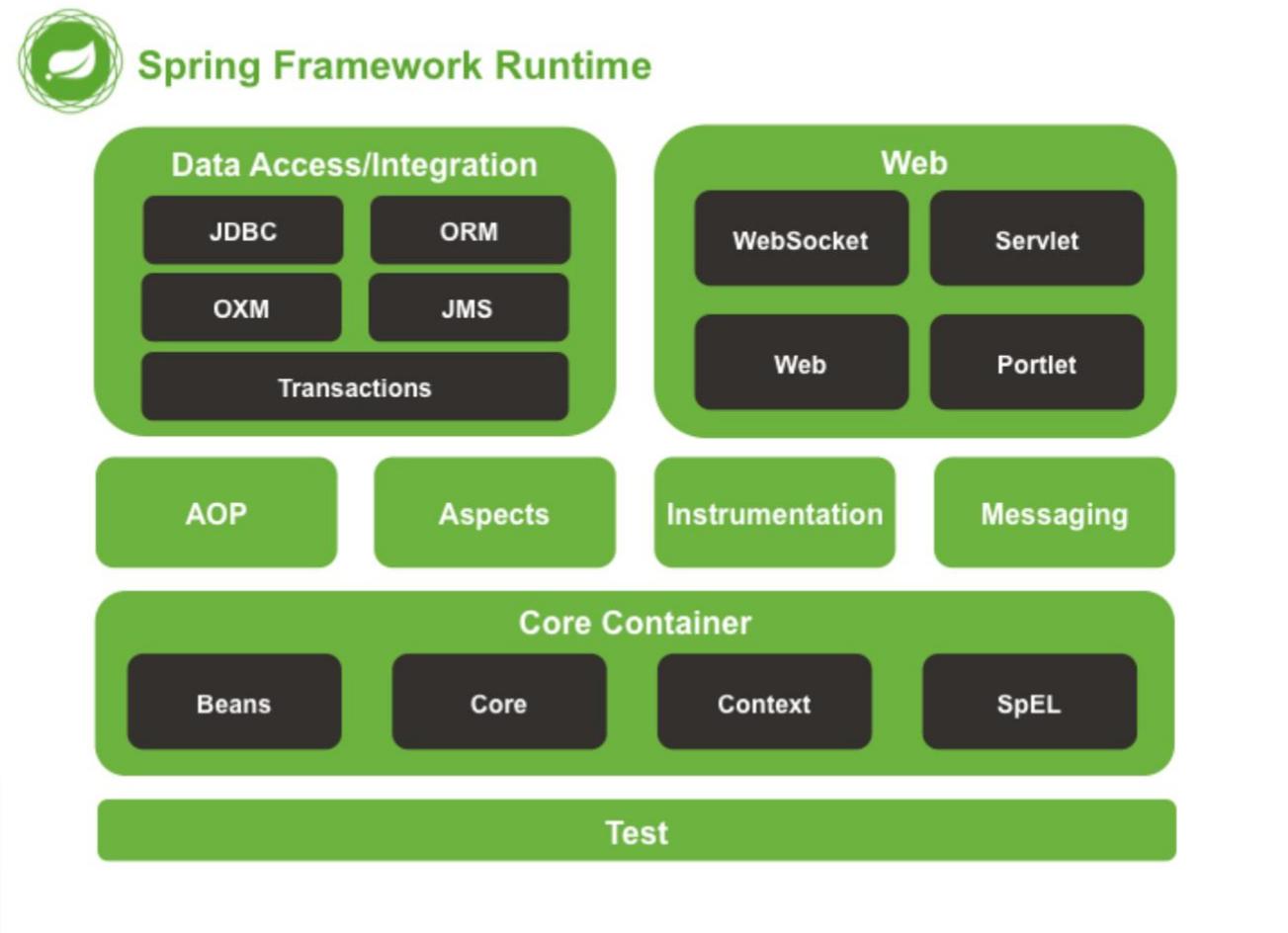
Infrastructure similaire à un serveur d'application JEE

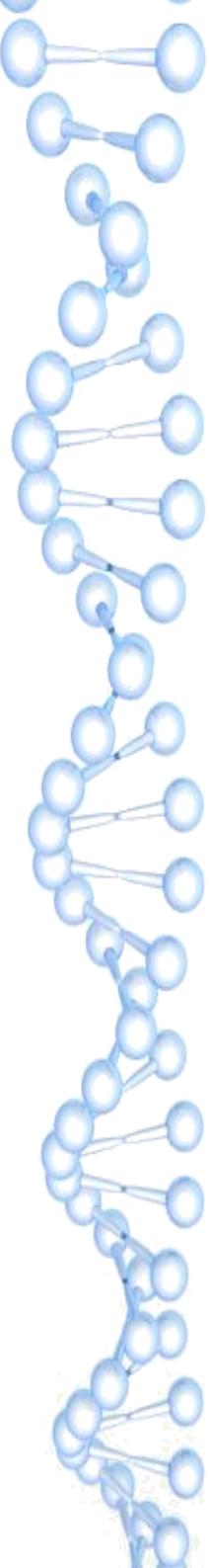
Prend en charge la création et la mise en relation d'objets par l'intermédiaire d'une configuration

Pas de "couplage" entre le conteneur et les composants :

Les classes n'ont pas à implémenter une quelconque interface pour être prises en charge par le framework

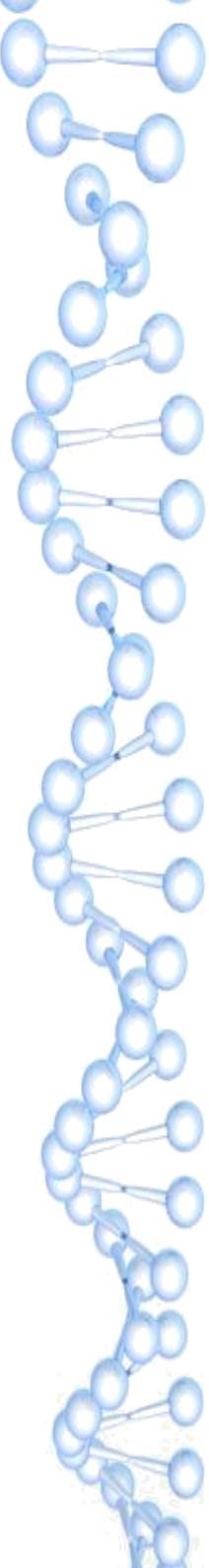
Spring: Les Composants





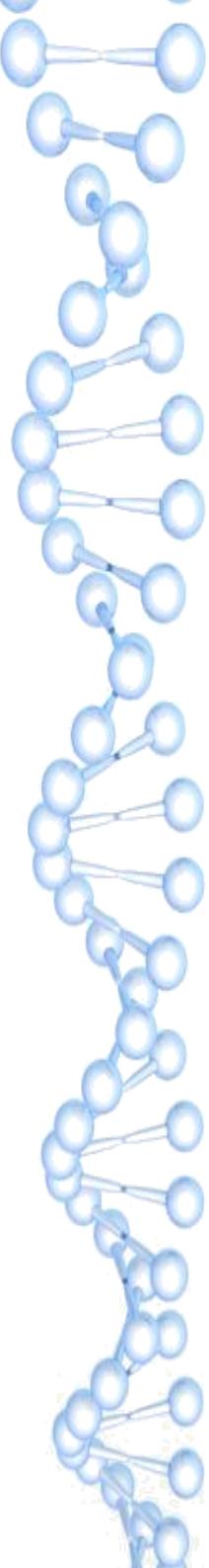
Spring:IoC

- . l'inversion de contrôle
 - . deux variantes
 - recherche de dépendance
 - utilisation d'une entité de résolution (JNDI par exemple)
 - méthode à l'initiative du composant donc intrusive
 - pas de typage
 - . injection de dépendance (DI)
 - dépendance fournie au composant
 - méthode transparente pour le composant
 - pas de connaissance du framework d'injection
 - vérification du typage par le framework



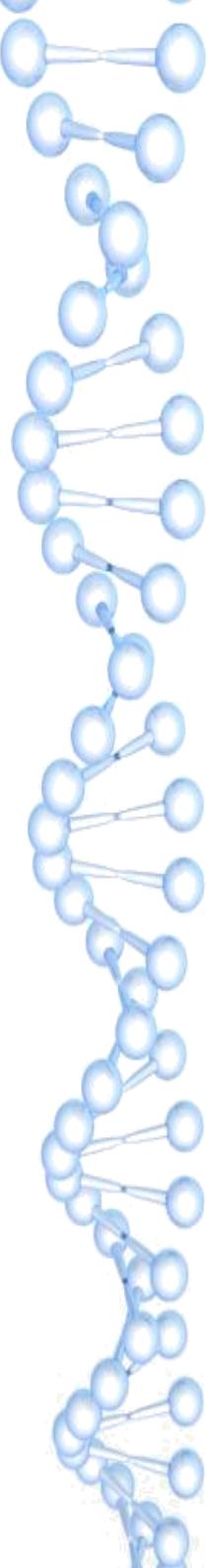
Spring:IoC

- . l'inversion de contrôle
 - . deux variantes
 - recherche de dépendance
 - utilisation d'une entité de résolution (JNDI par exemple)
 - méthode à l'initiative du composant donc intrusive
 - pas de typage
 - . injection de dépendance (DI)
 - dépendance fournie au composant
 - méthode transparente pour le composant
 - pas de connaissance du framework d'injection
 - vérification du typage par le framework



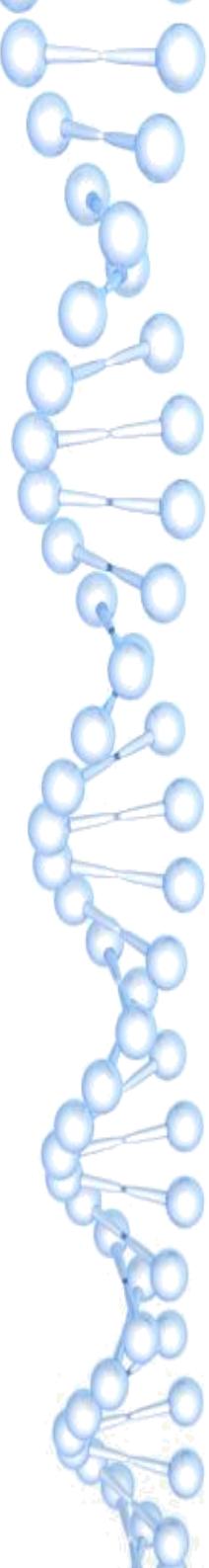
Spring:IoC

- . Pour pouvoir fonctionner, un composant a généralement besoin d'autres composants ...
- . L'injection de dépendance permet de
- . Fabriquer les composants
- . Leur "injecter" automatiquement les composants nécessaires



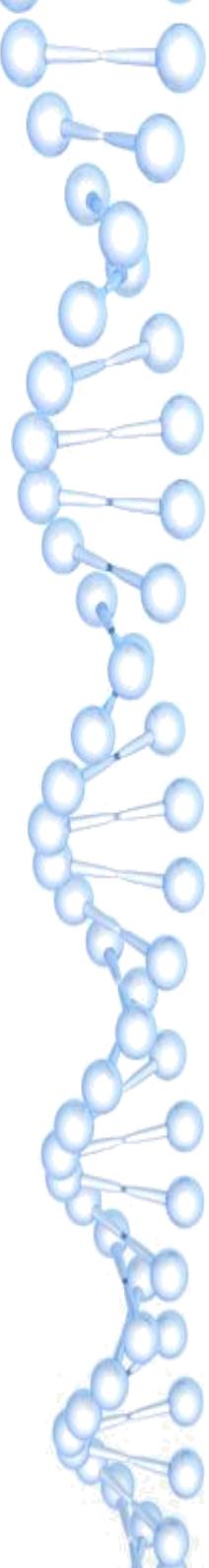
Spring:IoC

- . si A dépend de B, pour créer A, on a besoin de B
- . Pour supprimer la dépendance, on peut :
 - Créer une interface I qui contient toutes les méthodes que A peut appeler sur B
 - Indiquer de B implémente I
 - Remplacer toutes les références de A à B par des références à I
- . On veut disposer dans A d'un objet implementant I alors qu'on ne sait pas l'instancier.



Spring:IoC

- . Solution :
 - Créer un objet « b » de type B et l'injecter dans A.
- . L'injection peut être faite :
 - à l'instanciation : on passe l'objet b au constructeur de A
 - par modificateur : on passe l'objet b à une méthode de A qui va modifier un attribut (« setter »)



Dependency Injection

- . Objectifs :

- Configurer dynamiquement les dépendances entre objets

- Décrire les ressources dans un fichier de configuration

- . type 1

- injection à travers une interface spécifique au framework

- . type 2

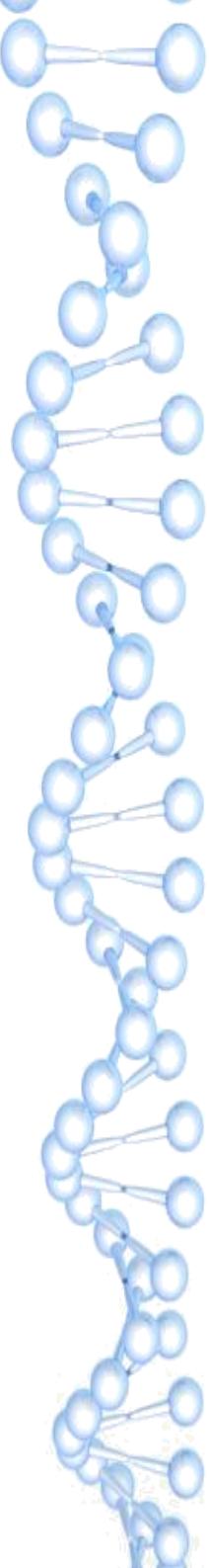
- exploitation du pattern JavaBean
 - injection via les setters

- . type 3

- injection via les constructeurs

Martin Fowler

<http://martinfowler.com/articles/injection.html#InterfaceInjection>



Dependency Injection

Type 1 IoC (interface injection)

I'interface définit la méthode à appeler pour l'injection

```
public interface InjectFinder {  
    void injectFinder(MovieFinder finder);  
}  
  
class MovieLister implements InjectFinder  
{  
    public void injectFinder(MovieFinder finder) {  
        this.finder = finder;  
    }  
}
```

Dependency Injection

Type 2 IoC (setter injection)

```
ServiceB serviceB = new ServiceB(d, e);
ServiceC serviceC = new ServiceC();

MonComposantA a = new MonComposantA();

a.setServiceB(serviceB);

a.setServiceC(serviceC);

a.doMyStuff();

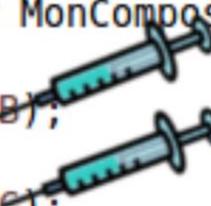
ServiceB serviceB = new ServiceB(d, e);
ServiceC serviceC = new ServiceC();

MonComposantA a = new MonComposantA();

a.setServiceB(serviceB);

a.setServiceC(serviceC);

a.doMyStuff();
```



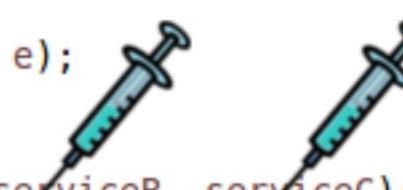
Dependency Injection

. Type 3 IoC (constructor injection)

```
ServiceB serviceB = new ServiceB(d, e);
ServiceC serviceC = new ServiceC();

MonComposantA a = new MonComposantA(serviceB, serviceC);

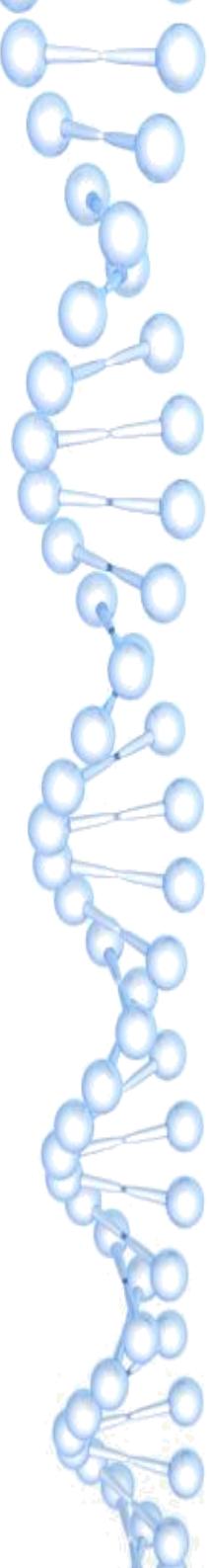
a.doMyStuff();
```



```
ServiceB serviceB = new ServiceB(d, e);
ServiceC serviceC = new ServiceC();

MonComposantA a = new MonComposantA(serviceB, serviceC);

a.doMyStuff();
```



Dependency Injection

L'injection de dépendance dans Spring peut se faire

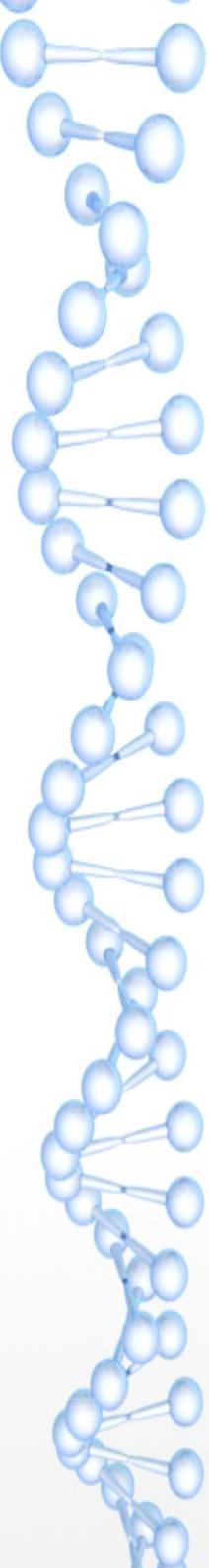
- Injection par constructeur
- Injection par setters

Dependency Injection

Injection par constructeur :
mise en œuvre (Collaborateurs)

```
public class MonComposantA implements IComposantA {  
  
    ServiceB serviceB;  
    ServiceC serviceC;  
  
    public MonComposantA(ServiceB serviceB, ServiceC serviceC) {  
        // Code here ...  
    }  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframew  
  
    <bean id="hello" class="org.formation.spring.firstOne.MonComposantA">  
        <constructor-arg ref="myServiceC"/>  
        <constructor-arg ref="myServiceB"/>  
    </bean>
```



Dependency Injection

Injection par constructeur :
mise en œuvre (index des champs)

```
public class ServiceB {  
    public ServiceB(int d, int e) {  
        // Code here ...  
    }  
  
<bean id="hello" class="org.formation.spring.firstOne.MonComposantA">  
    <constructor-arg ref="myServiceC"/>  
    <constructor-arg ref="myServiceB"/>  
</bean>  
<bean id="myServiceC" class="org.formation.spring.firstOne.ServiceC"></bean>  
<bean id="myServiceB" class="org.formation.spring.firstOne.ServiceB">  
    <constructor-arg value="0" index="0"/>  
    <constructor-arg value="12" index="1"/>  
</bean>
```

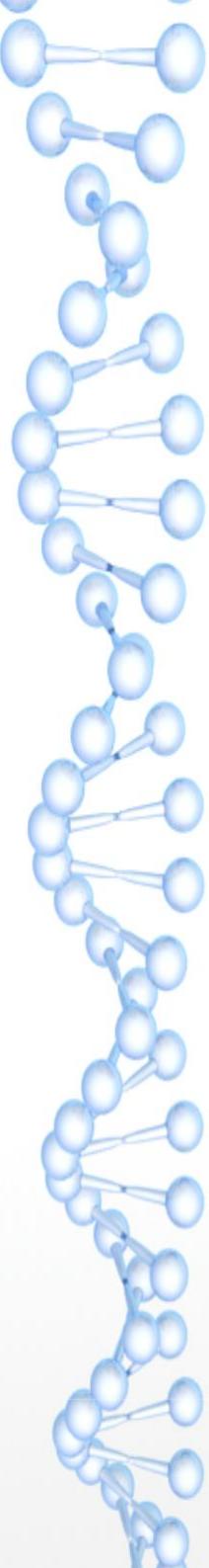
Dependency Injection

Injection par constructeur :
mise en œuvre (nom des champs)

```
public class ServiceB {  
    @ConstructorProperties({"d", "e"})  
    public ServiceB(int d, int e) {  
        // Code here ...  
    }  
  
<bean id="myServiceB" class="org.formation.spring.firstOne.ServiceB">  
    <constructor-arg name="d" value="12"/>  
    <constructor-arg name="e" value="145"/>  
</bean>
```

(Type des champs)

```
public ServiceB(int d, String e) {  
    // Code here ...  
}  
  
<bean id="myServiceB" class="org.formation.spring.firstOne.ServiceB">  
    <constructor-arg name="d" value="12"/>  
    <constructor-arg type="java.lang.String" value="145"/>  
</bean>
```



Dependency Injection

Injection par setters : mise en œuvre
Collaborateurs et injection de valeur

```
public class MonComposantA implements IComposantA {

    ServiceB serviceB;
    ServiceC serviceC;
    int myValue;

    public void setMyValue(int myValue) {
        this.myValue = myValue;
    }

    public void setServiceB(ServiceB serviceB) {
        this.serviceB = serviceB;
    }

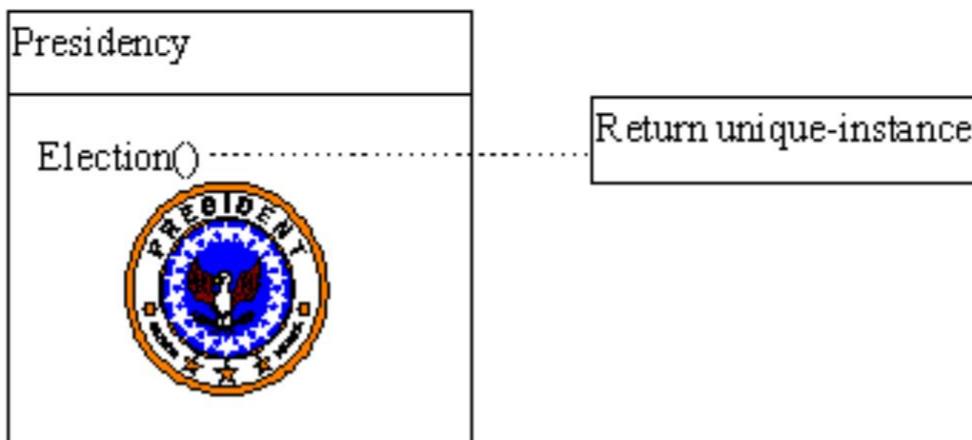
    public void setServiceC(ServiceC serviceC) {
        this.serviceC = serviceC;
    }

<bean id="hello2" class="org.formation.spring.firstOne.MonComposantA">
    <property name="serviceB" ref="myServiceB"/>
    <property name="serviceC" ref="myServiceC"/>
    <property name="myValue" value="42"/></property>
</bean>
```

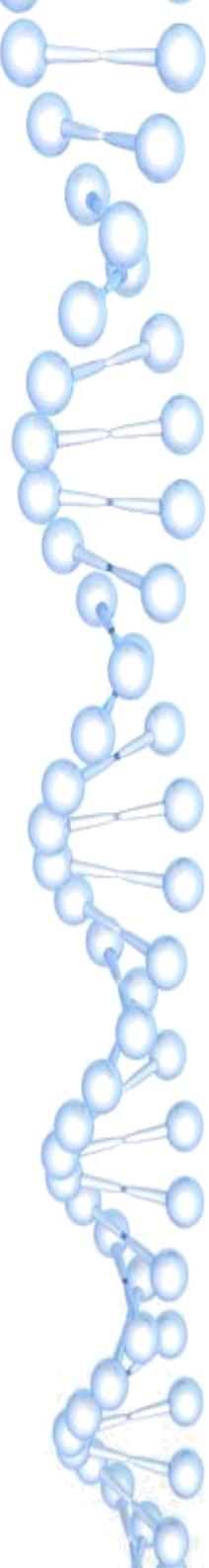
Spring:Scope

Spring crée par défaut les beans comme singleton.

Pattern Singleton : une instance unique.



Les beans Spring sont généralement des services

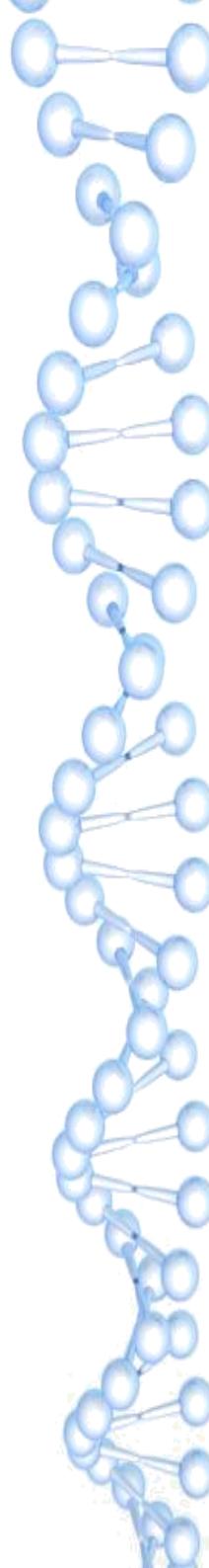


Spring:Scope

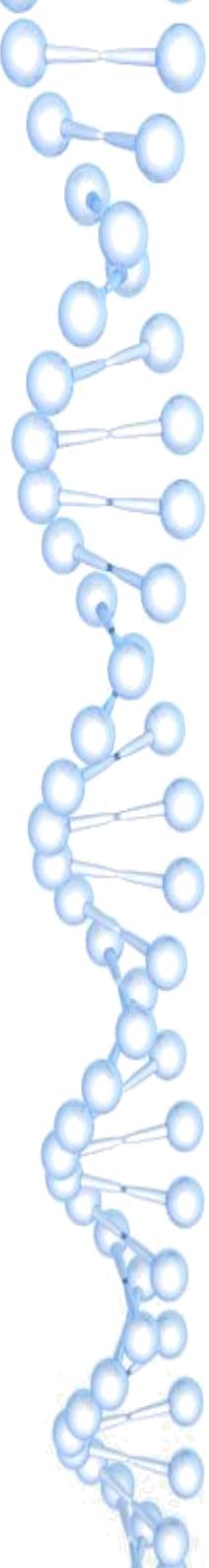
Portée des beans :

Stratégie de création et de conservation des instances des beans gérés par Spring

- . **singleton** : une seule instance existe pour le bean (portée par défaut)
- . **prototype** : une nouvelle instance est créée à chaque injection
- . **request** : une instance est associée à chaque requête (application web uniquement)
- . **session** : une instance est associée à chaque session (application web uniquement)



Merci



En Bonus

Il est possible d'ajouter de courtes présentations d'une technologie, d'une API ou de nouveautés Java 8 , par exemple Programmation fonctionnelle sous Java, JPA, DAO..

Nous mettons à votre disposition une liste bibliographique et webographique sur les sujets abordés.

Vous trouverez sur un répertoire partagé quelques PDFs en rapport avec cette session.