



MODULE JAVA

Développement avancé

Alexandre CASES - 19-07-2022



1.

Introduction



Connaissances prérequis

- ▶ Base de JAVA (Variable, Structure conditionnelle & itérative, ...),
- ▶ Programmation objet en Java (Héritage, Implémentation, Polymorphisme, ...)
- ▶ Les classes essentiels de Java SE
- ▶ Les collections (ArrayList & HashMap)
- ▶ Gestion des exceptions
- ▶ Accès aux données avec JDBC

Environnement de développement

- ▶ **Java Development Kit (JDK)** : Ensemble de librairie de base pour la programmation Java

- ▶ Ajouter la variable d'environnement :
- ▶ Test java : `java -version`
- ▶ <https://jdk.java.net/18/>



- ▶ Eclipse
 - ▶ Eclipse.org
 - ▶ Eclipse IDE for Java Developers
 - ▶ Config « encoding » : utf8
 - ▶ <https://www.eclipse.org/downloads/>





2.

Aspects avancés du Java



La boucle for-each

- Une boucle “for” traditionnelle :

```
List couleurs = new ArrayList();  
couleurs.add("Vert");  
couleurs.add("Bleu");  
couleurs.add("Orange");  
for (int i=0; i <3; i++) {  
    System.out.println(couleurs.get(i));  
}
```

- Une boucle “for” amélioré, la boucle “for-each” :

```
List couleurs = new ArrayList();  
couleurs.add("Vert");  
couleurs.add("Bleu");  
couleurs.add("Orange");  
for (String couleur : couleurs) {  
    System.out.println(couleur);  
}
```

Les Itérateurs

- ▶ Vous devriez avoir vu cette notion avec JDBC pour itérer sur un `ResultSet` mais sachez que c'est aussi possible sur une liste avec une mécanique presque similaire:

- ▶ Une boucle "for" amélioré, la boucle "for-each" :

```
List couleurs = new ArrayList();
couleurs.add("Vert");
couleurs.add("Bleu");
couleurs.add("Orange");

Iterator<String> it = couleurs.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}
```

Le **hasNext()** permet de vérifier s'il y a un élément en suivant dans la liste, renvoi **true** ou **false**.

Le **next()** permet de déplacer le curseur sur l'élément suivant de la liste et renvoie l'item en question.



Exercice 1

- ▶ Créer un programme qui demande à l'utilisateur d'entrer des nombres jusqu'à ce que l'utilisateur tape 'q' (comme quitter).
 - ▷ A ce moment la, le programme doit :
 - ▷ Afficher la liste des nombres.
 - ▷ Afficher « Vous avez saisi <x> nombres. »
 - ▷ Refuser les doublons à la liste.
 - ▷ Si le nombre est déjà dans la liste: Affichée « Chiffre déjà ajouté à la saisie <x>. »
 - ▷ Quand le programme est terminé, affichez la liste avec une boucle for-each
 - ▷ Puis réaffichez la liste mais cette fois avec un itérateur

ATTENTION : Pensez à gérer les exceptions, votre programme ne doit pas planter



Exercice 1 V2 (Optionnel)

- ▶ Pour les plus rapide, faites le même programme que l'exercice précédent mais avec une HashMap
- ▶ Créer un relevé de note qui demande à l'utilisateur d'entrer le nom de l'élève et la note qu'il a eu jusqu'à ce que l'utilisateur tape 'q' (comme quitter) en utilisant une HashMap.
- ▶ A ce moment la, le programme doit :
 - ▷ Afficher la liste des notes "nom élève/note".
 - ▷ Afficher « Vous avez saisi <x> notes. »
 - ▷ Refuser les doublons d'élève.
 - ▷ Si l'élève est déjà dans la HashMap: Affichée « Élève déjà ajouté à la saisie <x>. »

La correction sera faite par une personne du groupe.

Classes anonymes

- ▶ Le problème : La syntaxe utilisée pour l'héritage de classe/mise en œuvre d'interface est trop verbeuse pour des codes simples.

```
interface Bird { void fly(); }
```

```
class MyBird implements Bird {  
    void fly() {  
        System.out.println("fly!");  
    }  
}
```

```
class Test {  
    void f() {  
        Bird bird = new MyBird();  
    }  
}
```

Nécessite une nouvelle classe,
donc un fichier,
le tout pour allouer
une **unique instance** et
peu de lignes de code

Classes anonymes

- ▶ Le problème : La syntaxe utilisée pour l'héritage de classe/mise en œuvre d'interface est trop verbeuse pour des codes simples.

```
interface Bird { void fly(); }
```

```
class MyBird implements Bird {  
    void fly() {  
        System.out.println("fly!");  
    }  
}
```

```
class Test {  
    void f() {  
        Bird bird = new MyBird();  
    }  
}
```

Nécessite une nouvelle classe,
donc un fichier,
le tout pour allouer
une **unique instance** et
peu de lignes de code




Classes anonymes

- ▶ But : simplifier la syntaxe utilisée pour l'héritage de classe ou la mise en œuvre d'interface pour les codes simples
 - Allocation d'une unique instance de la classe dans le code
 - Peu de méthodes et de lignes de code dans la classe
- ▶ Principes :
 - Omettre le nom de la classe
 - Donner le code de la mise en œuvre au moment de l'allocation

Classes anonymes

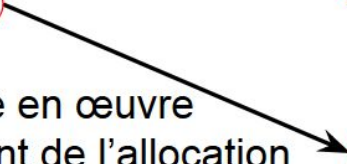
```
interface Bird { void fly(); }
```

Définit une nouvelle
classe qui hérite de `Bird`
et qui n'a pas de nom



```
class MyBird {  
    void fly() {  
        System.out...;  
    }  
}
```

Mise en œuvre
au moment de l'allocation



```
class Test {  
    void f() {  
        Bird bird = new MyBird();  
    }  
}
```

```
class Test {  
    void f() {  
        Bird bird = new Bird() {  
            void fly() {  
                System.out...  
            }  
        }  
    }  
}
```



Le try-with-resource

- La prise en charge de try-with-resources, introduite dans Java 7, nous permet de déclarer des ressources à utiliser dans un bloc try avec l'assurance que les ressources seront fermées après l'exécution de ce bloc.

Try-Catch-Finally:

```
Scanner scanner = null;
try {
    scanner = new Scanner(new File("test.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

Try-With-Resources:

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
}
```

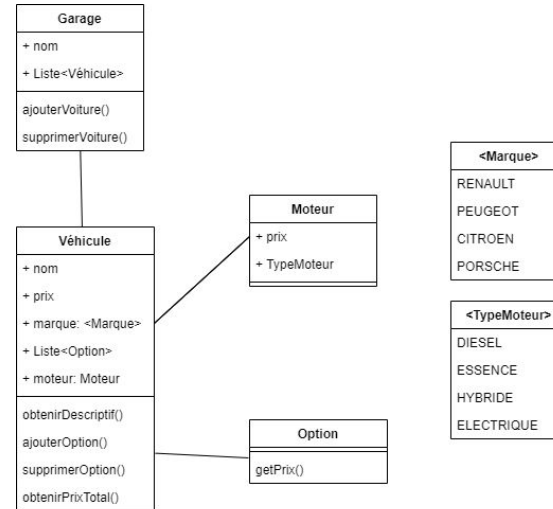


Exercice 2

- ▶ Créer une classe véhicule avec les propriétés : nom et vitesseMax et leurs getters et setters respectifs
- ▶ Créer une interface actionVehicule avec les signatures de méthodes : peutVoler(), peutRouler(), peutNaviguer() renvoyant un boolean
- ▶ Créer une classe voiture et bateau qui étendent la classe véhicule et qui implémente chacun l'interface actionVehicule
- ▶ Overider les méthodes de l'interface actionVéhicule et la méthode setName de la classe étendu
- ▶ Dans un main, créer un véhicule de chaque type, est afficher son nom ainsi que les actions qu'il peut faire (voler, rouler, naviguer)
- ▶ Dans le main, créer une classe Anonyme "Avion" implémentant l'interface actionVehicule et afficher les actions qu'il peut faire

Exercice 2 V2 (Optionnel)

- ▶ Créer un programme qui réponds au schéma suivant :
- ▶ Créer un garage et plusieurs véhicules qui s'ajoutent au garage.
- ▶ Ajouter/retirer des options en actualisant le prix. Faire pareil avec les motorisations.



Implements option

Climatisation	GPS
Prix = 499.99	Prix = 99.99
VitreElectrique	SiegeChauffant
Prix = 39	Prix = 175
BarresDeToit	
Prix = 49.99	

La correction sera faite par une personne du groupe.



Des éléments à approfondir ?

- ▶ Avant de passer à la suite, avez-vous des éléments qui ne sont pas encore assez clairs pour vous depuis vos débuts sur le langage JAVA ?



3.

Les Threads



Les Threads (1/2)

- Un Thread permet de gérer parallèlement **plusieurs objets** simultanément
- Les Threads permettent de faire des **exécutions simultanéments**
- Pour utiliser un Thread, il faut **étendre** la classe :

```
public class exemple extends Thread {
```

- Les Threads ont leur propres méthodes, pour la **surcharger** il faut utiliser le **décorateur** `@Override`

```
@Override  
public void run(){
```



Les Threads (2/2)

- Pour **instancier** un Thread, il faut le déclarer comme n'importe quel :

```
MaClasseThread x = new MaClasseThread();
```

- Pour **exécuter** un Thread, il faut appeler la méthode `start()` qui correspond à la méthode `run()` surchargé :

```
x.start();
```



Les Runnables

- Une classe Java ne peut étendre (extend) qu'une seule classe
- L'implémentation d'un Thread bloque des éventuels extends.
- Il existe une seconde méthode pour implémenter un Thread : **Le Runnable**
- Un Runnable s'utilise par **implémentation**
- Un Runnable n'expose qu'une seule méthode: **run()**
- L'instanciation est différente :

```
RunnableCompteur x = new RunnableCompteur();  
Thread t = new Thread(x);
```



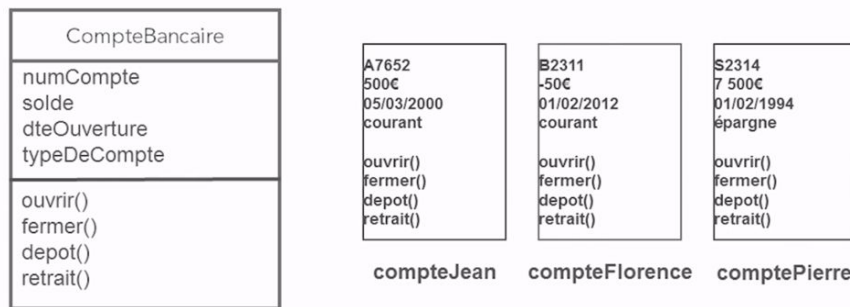
Exemples

□ Exemple d'implémentation d'un Thread avec un compteur :

1. Extend Thread
2. Implement Runnable

EXERCICE 3 (1/2) :

- Créer un programme qui instancie 3 comptes bancaire dans un main. L'objet CompteBancaire correspond au schéma ci-dessous et s'appuie sur un Thread



- L'appel de la méthode start() doit afficher le message suivant :

"Le compte A7652 a été créé le 05/03/2000 et a un solde initial de 500€. Il s'agit d'un compte Courant."



Exercices

EXERCICE 3 (2/2) :

- Ajouter l'attribut *estOuvert* de type booléen, défini à False par défaut et qui devient True à la création d'un compte.
- Implémenter les 5 méthodes :
 - ouvrir() permet d'indiquer que le compte bancaire est ouvert
 - fermer() permet d'indiquer que le compte bancaire est fermé
 - depot() permet d'ajouter une somme définie au solde du compte
 - retrait() permet d'enlever une somme définie au solde du compte
 - virement() qui permet de transférer de l'argent vers un autre compte
- Les méthodes depot() et retrait() doivent afficher un message avec le nouveau solde.
- Ajouter une classe "Banque" respectant :
 - Une banque contient une liste de comptes bancaire. (ArrayList)
 - Une banque à une méthode "ajouterCompte" qui ajoute un compte à la liste.
- Dans le main, créer une banque et ajouter les 3 comptes bancaires.



Exercices

EXERCICE 4 :

- Reprendre le programme précédent, et ajouter :
 - Une interaction avec un utilisateur lui permettant de créer un compte en récupérant toutes les informations nécessaires.
 - Ajouter ce nouveau compte à la banque.
 - Laisser le choix à l'utilisateur de réaliser des opérations de dépôt, retrait, clôture, et virement (méthode à implémenter) vers un autre compte.

Utiliser l'interface Runnable à la place de l'objet Thread



Pour aller plus loin

Ressources supplémentaires :

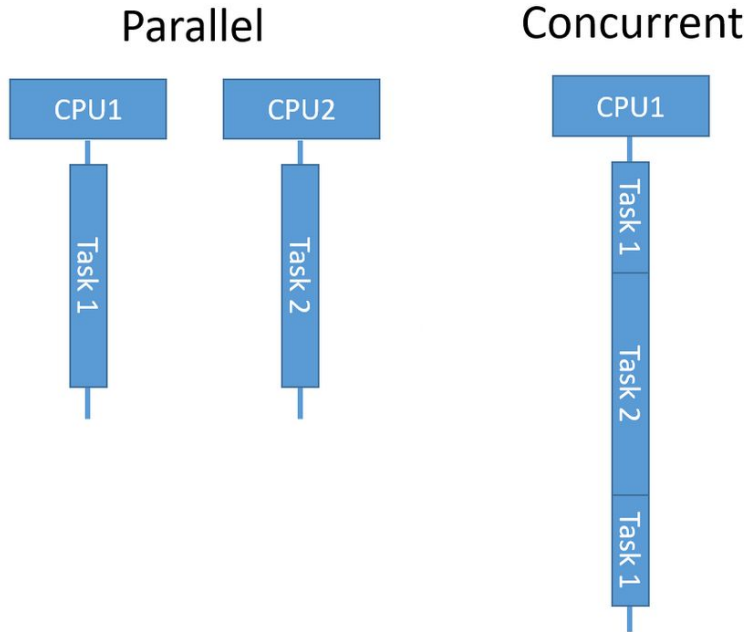
- ▶ <https://www.jmdoudoux.fr/java/dej/chap-threads.htm>



4.

L'asynchronisme

Threading VS Asynchronisme



Le "Parallèle" représente les threads et le fait qu'il travaille sur des tâches en parallèle.

Le "Concurrent" représente l'asynchronisme, c'est-à-dire qu'en attendant de recevoir un retour de la tâche 1, la tâche 2 est exécutée sans attendre le retour de la 1ère pour se lancer.



L'asynchronisme

- ▶ L'asynchronisme est une mécanique très utilisée en programmation côté Front.
- ▶ Elle permet de ne pas bloquer un programme même si on attend encore le retour d'un morceau de code.

Exemple :

Vous voulez faire un programme qui affiche des données météo, vous pouvez à l'aide de l'asynchronisme, demander les infos à l'API de Météo France, afficher votre page sans les données et afficher les données une fois le retour de l'API reçu.



L'asynchronisme en JAVA

- ▶ Depuis Java 8, dans le package : ***java.util.concurrent***

- ▶ ***Class CompletableFuture<T>***

Docs : <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html>

- ▶ En JAVA, cette mécanique est implémenté à l'aide de Thread ou de Runnable



Exemple

□ Exemple Asynchrone avec un compteur



5.

Les lambda expressions



Les lambda expressions ?

- ▶ Aussi appelé “Fonction anonyme”, sa définition se fait sans déclaration explicite du type de retour, ni de modificateurs d'accès ni de nom. C'est un raccourci syntaxique qui permet de définir une méthode directement à l'endroit où elle est utilisée.
- ▶ Permet d'écrire du code plus concis, donc plus rapide à écrire, à relire et à maintenir.
- ▶ Une expression lambda est donc un raccourci syntaxique qui simplifie l'écriture de traitements passés en paramètre. Elle est particulièrement utile notamment lorsque le traitement n'est utile qu'une seule fois : elle évite d'avoir à écrire une méthode dans une classe.



La syntaxe d'une expression lambda

- ▶ Un des avantages des expressions lambda est d'avoir une syntaxe très simple.
- ▶ La syntaxe d'une expression lambda est composée de trois parties :
 - ▷ un ensemble de paramètres, d'aucun à plusieurs
 - ▷ l'opérateur ->
 - ▷ le corps de la fonction
- ▶ Elle peut prendre deux formes principales :
 - ▷ (paramètres) -> expression;
 - ▷ (paramètres) -> { traitements; }

La syntaxe d'une expression lambda

- ▶ L'écriture d'une expression lambda doit respecter plusieurs règles générales :
 - ▷ zéro, un ou plusieurs paramètres dont le type peut être déclaré explicitement ou inféré par le compilateur selon le contexte
 - ▷ les paramètres sont entourés par des parenthèses et séparés par des virgules. Des parenthèses vides indiquent qu'il n'y a pas de paramètre
 - ▷ lorsqu'il n'y a qu'un seul paramètre et que son type est inféré alors les parenthèses ne sont pas obligatoires
 - ▷ le corps de l'expression peut contenir zéro, une ou plusieurs instructions. Si le corps ne contient d'une seule instruction, les accolades ne sont pas obligatoires et le type de retour correspond à celui de l'instruction. Lorsqu'il y a plusieurs instructions alors elles doivent être entourées avec des accolades



Exemple

- ▶ Exemple sur Eclipse
 - ▷ Fonction additionner
 - ▷ Fonction afficher



Exercice 5

- Créer un fichier JAVA
- Ecrire les fonctions suivantes à l'aides lambdas expressions
 1. multiplier
 2. compter : affiche les numéros de 1 à X, X étant un chiffre passé en paramètre
- Tester leurs bon fonctionnement



Pour aller plus loin

Ressources supplémentaires :

- ▶ <https://www.jmdoudoux.fr/java/dej/chap-lambdas.htm>



6.

Les streams



Qu'est-ce qu'un Stream ?

- ▶ Stream n'est pas une collection ou une structure de donnée de manière générale. C'est une séquence d'éléments sur laquelle on peut effectuer un groupe d'opérations de manière séquentielle ou parallèle.
- ▶ Il existe deux types d'opérations sur les Streams :
 - ▷ Les opérations intermédiaires : elles transforment un Stream en un autre Stream
 - ▷ Les opérations finales : elles produisent un résultat



Comment les utiliser ?

- ▶ À la manière du SQL sur une table , les éléments d'un Stream vont passer à travers un pipeline de prédicats, de comparateurs, de fonctions...
- ▶ Il existe deux types d'opérations sur les Streams :
 - ▷ Les opérations intermédiaires : elles transforment un Stream en un autre Stream
 - ▷ Les opérations finales : elles produisent un résultat



Comment les utiliser ?

- Récupération de la liste triée sans l'utilisation des streams :

```
List<Order> orderList = new ArrayList<>();

orderList.add(new Order(1, OrderType.BUY, 100.0, "phone"));
orderList.add(new Order(2, OrderType.SELL, 50.0, "mouse"));
orderList.add(new Order(3, OrderType.SELL, 150.0, "bike"));
orderList.add(new Order(4, OrderType.BUY, 500.0, "laptop"));
orderList.add(new Order(5, OrderType.SELL, 40.0, "keyboard"));

List<Order> sellOrderList = new ArrayList<>();

for (Order order : orderList) {
    if (order.getType() == Order.OrderType.SELL) {
        sellOrderList.add(order);
    }
}
```

```
Collections.sort(sellOrderList, (o1, o2) -> o1.getPrice().compareTo(o2.getPrice()));

List<String> products = new ArrayList<>();

for (Order o : sellOrderList) {
    products.add(o.getProduct());
    System.out.println(o.getProduct());
}
```



Comment les utiliser ?

- Récupération de la liste triée avec l'utilisation de Stream :

```
List<String> products2 = orderList.stream()  
    .filter(o -> o.getType() == OrderType.SELL)  
    .sorted(Comparator.comparing(Order::getPrice))  
    .map(Order::getProduct)  
    .collect(toList());
```



Exemple

□ Exemple avec le model Realisateur de la médiathèque



Exercice 6

- Reproduisez l'exemple avec les films en filtrant les films avec une durée plus grande que 120 minutes et trier la liste en fonction du titre

Utilisé les fonctionnalités des Streams



7.

Présentation des services Web



7.1.

Présentation des architectures distribuées



Architectures ?

- ▶ Architecture **centralisée** : « client/serveur »
 - ▶ 1 serveur / N clients
- ▶ Architecture distribuée : « peer to peer »
 - ▶ N serveurs / N clients



Rappel sur l'architecture **centralisée**

- ▶ **Client/serveur** : distinction stricte entre le rôle de client et le rôle de serveur
 - ▶ 1 **client** effectue une requête pour un service donné sur un serveur et attend une réponse
 - ▶ 1 **serveur** reçoit une demande de service, la traite et retourne une réponse au client



Rappel sur l'architecture **centralisée**

- ▶ **Caractéristiques du client :**
 - ▶ Actif
 - ▶ Connecté à un serveur
 - ▶ Envoie des requêtes à un serveur
 - ▶ Attend et traite les réponses du serveur
 - ▶ Interagit avec un utilisateur final (par exemple avec une IHM)



Rappel sur l'architecture **centralisée**

- ▶ **Caractéristiques du serveur :**
 - ▶ Passif
 - ▶ A l'écoute des requêtes clients
 - ▶ Traite les requêtes et fournit une réponse
 - ▶ Pas d'interaction directe avec les utilisateurs finaux



Rappel sur l'architecture **centralisée**

- ▶ **Exemples d'architecture centralisée client/serveur :**
 - ▶ Consultation de pages web (envoi de requêtes HTTP depuis un navigateur à un serveur pour consulter les pages)
 - ▶ Gestion des mails (client pour envoyer et recevoir les mail, serveur pour la gestion : SMTP, POP, IMAP)



Rappel sur l'architecture **centralisée**

- ▶ **Découpage en couches :**
 - ▶ **Présentation** : affichage, dialogue avec un utilisateur final
 - ▶ **Service** : traitements, règles de gestion et logique applicative
 - ▶ **Données** : DAO, persistance des données



Rappel sur l'architecture **centralisée**

- ▶ **Découpage en couches :**
 - ▶ La **répartition** de ces couches entre les **rôles** de client et de serveur permet de distinguer entre les différents types d'architecture client/serveur
 - ▶ 2 tiers
 - ▶ 3 tiers
 - ▶ N tiers



Architectures **distribuées**

- ▶ **Relations d'égal à égal :**
 - ▶ Pas de connaissance globale du réseau
 - ▶ Pas de coordination globale des nœuds
 - ▶ Chaque nœud ne connaît que les nœuds constituant son voisinage
 - ▶ Toutes les données sont accessibles depuis n'importe quel nœud
 - ▶ Les nœuds sont volatiles



Architectures **distribuées**

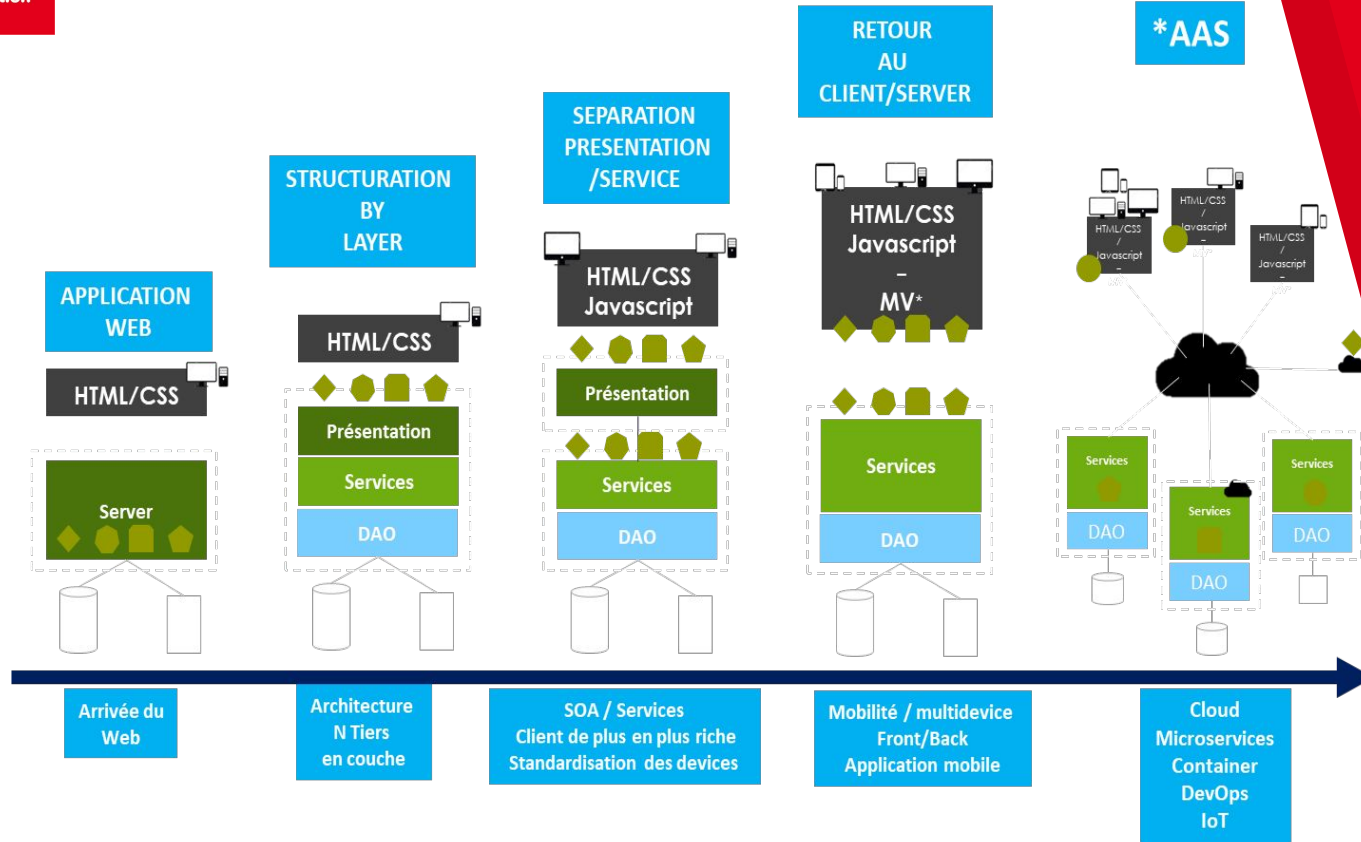
- ▶ **Avantages :**
 - ▶ Plus adapté à la montée en charge (scalabilité)
 - ▶ Meilleure robustesse en cas de panne (réplication, pas de SPOF : « single point of failure »)



Architectures **distribuées**

- ▶ **Inconvénients :**
 - ▶ Problématiques spécifiques
 - ▶ Concurrency
 - ▶ Fragmentation des données
 - ▶ Gestion de la réplication
 - ▶ ...

Evolution des architectures au cours du temps





7.2.

Positionnement des Web services

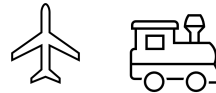
Les Services Web

- ▶ Exemple d'une agence de voyage :

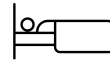
- ▶ Un produit « **voyage** » = une combinaison de plusieurs produits



- ▶ Réservation des billets de transport



- ▶ Réservation des nuits d'hôtel



- ▶ Réservation des locations de voiture



- ▶ ...



Les Services Web

- ▶ Exemple d'une agence de voyage :
 - ▶ La construction d'un produit « voyage » est le résultat d'informations récupérées auprès de différents fournisseurs :
 - ▶ Compagnies aériennes
 - ▶ Compagnies ferroviaires
 - ▶ Loueurs de voiture
 - ▶ Chaînes hôtelières
 - ▶ ...



Les Services Web

- ▶ Exemple d'une agence de voyage :
 - ▶ Une **application** de réservation de voyage sollicite d'autres applications réparties pour satisfaire la demande !
 - ▶ 2 types de **sollicitations** :
 - ▶ **Transformation** : adaptation du dialogue en fonction du profil utilisateur
 - ▶ **Agrégation** : appel à des applications proposées par des partenaires ou fournisseurs



Les Services Web

- ▶ Exemple d'une agence de voyage :
 - ▶ Pour réaliser cela, on s'appuie des Services Web !



Les Services Web

- ▶ Un Service Web, c'est quoi ?
 - ▶ Fonction **distante** mise à disposition sur un réseau □
accessibilité
 - ▶ Infrastructure **souple** pour des échanges entre des systèmes distribués hétérogènes
 - ▶ **Localisable** à partir de registres
 - ▶ **Couplage faible**



Les Services Web

- ▶ Un Service Web, c'est quoi ?
 - ▶ Répond à la problématique **B2B** (SOA -> architecture orientée service)
 - ▶ Un service **résout un problème** donnée
 - ▶ **Combinaison possible** pour résoudre des problèmes complexes



Les Services Web

- ▶ Idée générale
 - ▶ Un client a un **besoin**
 - ▶ Pour un besoin, plusieurs **services** et donc **fournisseurs** peuvent exister (avec ses propres caractéristiques)
 - ▶ Le client **choisit** un fournisseur pour pouvoir utiliser son service (celui qui correspond à son besoin et qui est compatible avec ses **exigences** (coût, performance, ...))



Les Services Web

- ▶ Pourquoi peut-on avoir besoin de Services Web ?
 - ▶ Besoin d'interopérabilité dans des environnements applicatifs distribués
 - ▶ Echanges sur des protocoles standards (HTTP, SMTP, ...)
 - ▶ Échanges entre des systèmes hétérogènes (environnements différents, langages différents)



Les Services Web

- ▶ Les usages
 - ▶ Assemblage de composants faiblement couplés
 - ▶ Définition indépendantes mais interaction
 - ▶ Adapté pour les applications orientées messages
 - ▶ Asynchronisme



Les Services Web

- ▶ Les acteurs
 - ▶ Le **client** : celui qui invoque le service web
 - ▶ Le **fournisseur** : celui qui fournit et met à disposition le service web
 - ▶ **L'annuaire** : celui qui détient et partage les informations sur les services web



Les Services Web

- ▶ Les acteurs
 - ▶ Le fournisseur :
 - ▶ Serveur d'application (par exemple JEE)
 - ▶ Expose un ou plusieurs services (EJB, servlets, enveloppés d'une couche « service »)



Les Services Web

- ▶ Les acteurs
 - ▶ L'annuaire :
 - ▶ Déclaration dans un annuaire = publication



7.3.

Approches SOAP et REST

Les types de **service web**

- ▶ Services web de type **SOAP**



- ▶ Services web de type **REST**





Les types de **service web**

- ▶ Services web de type **SOAP**
 - ▶ SOAP est un **protocole**
 - ▶ SOAP = Simple Object Access **Protocol**
 - ▶ Initialement conçu pour que des applications développées avec différents langages sur différentes plateformes puissent **communiquer**



Les types de **service web**

- ▶ Services web de type **SOAP**
 - ▶ **Protocole** = règles imposées qui augmentent la complexité et les coûts
 - ▶ **Mais**, standards qui assurent la **conformité** et sont privilégiés pour certaines applications en **entreprise**



Les types de **service web**

- ▶ Services web de type **SOAP**
 - ▶ Les **standards** de conformité intégrés incluent la **sécurité**, l'**atomicité**, la **cohérence**, l'**isolement** et la **durabilité** (ACID), et un ensemble de propriétés qui permet d'assurer des **transactions** de base de données fiables



Les types de **service web**

- ▶ Services web de type **SOAP**
 - ▶ Les principales **spécifications** :
 - ▶ **WS-Security** : standardise la manière dont les messages sont sécurisés et transférés via des identifiants uniques appelés jetons
 - ▶ **WS-ReliableMessaging** : standardise la gestion des erreurs entre les messages transférés par le biais d'une infrastructure informatique non fiable



Les types de **service web**

- ▶ Services web de type **SOAP**
 - ▶ Les principales **spécifications** :
 - ▶ **WS-Adressing** : ajoute les informations de routage des paquets en tant que métadonnées dans des en-têtes SOAP, au lieu de les conserver plus en profondeur dans le réseau
 - ▶ **WSDL** (Web Services Description Language) : décrit la fonction d'un service web ainsi que ses limites



Les types de **service web**

- ▶ Services web de type **SOAP**
 - ▶ Lorsqu'une requête de données est envoyée à une API SOAP, elle peut être gérée par n'importe quel **protocole** de couches de l'application : HTTP, SMTP, TCP, ...



Les types de **service web**

- ▶ Services web de type **SOAP**
 - ▶ Les messages SOAP doivent être envoyés sous la forme d'un document **XML**
 - ▶ Une fois finalisée, une requête destinée à une API SOAP **ne peut pas être mise en cache** par un navigateur (pas possible d'y accéder plus tard sans la renvoyer vers l'API)



Les types de **service web**

- ▶ Services web de type **REST**
 - ▶ REST **n'est pas un protocole**
 - ▶ REST est un ensemble de **principes** architecturaux adapté aux besoins des services web et applications mobiles légers
 - ▶ La mise en place de ces recommandations est laissée à **l'appréciation** des développeurs



Les types de **service web**

- ▶ Services web de type **REST**
 - ▶ L'envoi d'une requête à une API REST se fait généralement par le protocole **HTTP**
 - ▶ À la réception de la requête, les API développées selon les principes REST (appelées API ou services web RESTful) peuvent renvoyer des messages dans **différents formats** : HTML, XML, texte brut, JSON



Les types de **service web**

- ▶ Services web de type **REST**
 - ▶ Le format **JSON** (JavaScript Object Notation) est le plus utilisé pour les messages : **léger**, **lisible** par tous les langages de programmation et les humains
 - ▶ Les API **REST** sont plus flexibles et plus faciles à mettre en place

Le format **JSON** (exemple)

```
{
  "listAuteurs": {
    "count": "3",
    "0": {
      "id": 2,
      "firstName": "Joe",
      "lastName": "Goncalves",
      "phone": "0102030405",
      "email": "gonzalves@gmail.c",
    },
    "0": {
      "id": 4,
      "firstName": "Claude",
      "lastName": "Delannoy",
      "phone": "0677889900",
      "email": "claudio@delannooy.com",
    },
    "0": {
      "id": 10,
      "firstName": "Stefan",
      "lastName": "Zweig",
      "phone": "0660606060",
      "email": "stefan.zweig@lejoueurdechecs.de",
    },
  },
}
```



8.

Annotations



Annotations

- ▶ Depuis Java 5, les annotations apportent une standardisation des métadonnées dans un but généraliste. Ces métadonnées associées aux entités Java peuvent être exploitées à la compilation ou à l'exécution.
- ▶ Java propose plusieurs annotations standard et permet la création de ses propres annotations.
- ▶ Une annotation précède l'entité qu'elle concerne. Elle est désignée par un nom précédé du caractère @.
- ▶ Il existe plusieurs catégories d'annotations :
 - ▷ les marqueurs (markers) : ces annotations ne possèdent pas d'attribut (exemple : `@Deprecated`, `@Override`, ...)
 - ▷ les annotations paramétrées (single value annotations) : ces annotations ne possèdent qu'un seul attribut (exemple : `@MonAnnotation("test")`)
 - ▷ les annotations multi paramétrées (full annotations) : ces annotations possèdent plusieurs attributs (exemple : `@MonAnnotation(arg1="test 3", arg2="test 2", arg3="test3")`)



L'utilisation des annotations

- ▶ Les annotations prennent une place de plus en plus importante dans la plate-forme Java et dans de nombreuses API open source.
- ▶ Les utilisations des annotations concernent plusieurs fonctionnalités :
 - ▷ Utilisation par le compilateur pour détecter des erreurs ou ignorer des avertissements
 - ▷ Documentation
 - ▷ Génération de code
 - ▷ Génération de fichiers

Les API qui utilisent les annotations

De nombreuses API standard utilisent les annotations depuis leur intégration dans Java notamment :

- JAXB 2.0 : JSR 222 (Java Architecture for XML Binding 2.0)
- Les services web de Java 6 (JAX-WS) : JSR 181 (Web Services Metadata for the Java Platform) et JSR 224 (Java APIs for XML Web Services 2.0 API)
- Les EJB 3.0 et JPA : JSR 220 (Enterprise JavaBeans 3.0)
- Servlets 3.0, CDI
- ...

De nombreuses API open source utilisent aussi les annotations notamment JUnit, TestNG, Hibernate, ...



Les annotations standard

- ▶ **L'annotation @Deprecated** : C'est un marqueur qui précise que l'entité concernée est obsolète et qu'il ne faudrait plus l'utiliser. Elle peut être utilisée avec une classe, une interface ou un membre (méthode ou champ).
- ▶ **L'annotation @Override** : Cette annotation est un marqueur utilisé par le compilateur pour vérifier la réécriture de méthodes héritées. @Override s'utilise pour annoter une méthode qui est une réécriture d'une méthode héritée. Le compilateur lève une erreur si aucune méthode héritée ne correspond.
- ▶ **L'annotation @SuppressWarnings** : Les compilateurs peuvent détecter des cas qui sont potentiellement suspects, si vous êtes sûr que le warning peut être ignoré sans risque alors il est possible d'utiliser l'annotation @SuppressWarnings pour demander au compilateur de l'ignorer.

Les annotations personnalisées

- ▶ Sur la plate-forme Java, une annotation est une interface lors de sa déclaration et est une instance d'une classe qui implémente cette interface lors de son utilisation.
- ▶ La définition d'une annotation nécessite une syntaxe particulière utilisant le mot clé `@interface`. Une annotation se déclare donc de façon similaire à une interface.

```
Alex.java ✕  
1 package exemples;  
2  
3 public @interface Alex {  
4  
5 }
```

- ▶ Une fois compilée, cette annotation peut être utilisée dans le code. Pour utiliser une annotation, il faut importer l'annotation et l'appeler dans le code en la faisant précéder du caractère `@`.

```
AlexClass.java ✕  
1 package exemples;  
2  
3 @Alex  
4 public class AlexClass {  
5  
6 }  
7
```



Pour aller plus loin

Ressources supplémentaires :

- ▶ <https://www.jmdoudoux.fr/java/dej/chap-annotations.htm>

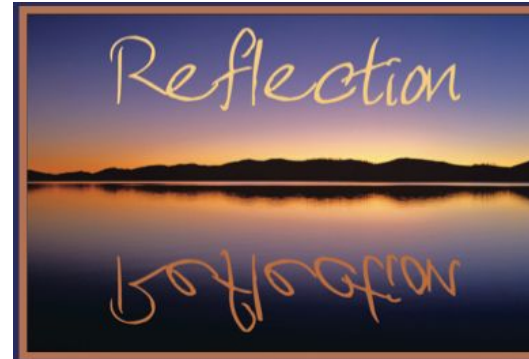


9.

Java Reflection API

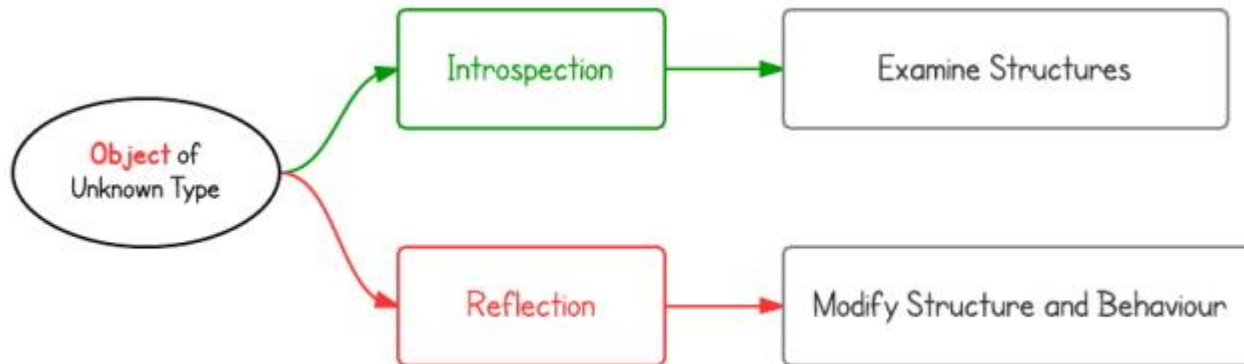
Qu'est ce que Java Reflection ?

- ▶ Java utilise le mot "Java Reflection" pour nommer une API importante dans la bibliothèque standard de Java
- ▶ La Reflection est l'image de réflexion d'un objet. Par exemple, votre image dans le miroir, ou la réflexion d'un arbre dans l'eau du lac. Le mot "Java Reflection" est simplement d'observer différemment, une autre manière d'accès aux objets de Java.



Qu'est ce que Java Reflection ?

- ▶ **Java** peut généralement être appelé **Java Introspection** (Introspection), qui peut évaluer la structure d'un objet lors de l'exécution (Runtime).
- ▶ Avec **Java Reflection**, le programme peut évaluer la structure d'un objet lors de l'exécution et modifier la structure et le comportement de l'objet.





Exemple

□ Exemple JAVA Reflection



10.

Connaître JMS et JMX



JMS

- ▶ JMS, acronyme de Java Message Service, est une API pour permettre un dialogue standard entre des applications ou des composants grâce à des brokers de messages. Elle permet donc d'utiliser des services de messaging dans des applications Java comme le fait l'API JDBC pour les bases de données.
- ▶ JMS définit deux modes pour la diffusion des messages :
 - ▷ Point à point (Point to point) : dans ce mode un message est envoyé par un producteur et est reçu par un unique consommateur.
 - ▷ Publication / souscription (publish/subscribe) : dans ce mode un message est envoyé par un producteur et est reçu par un ou plusieurs consommateurs.



JMS

- ▶ Les messages sont asynchrones mais JMS définit deux modes pour consommer un message :
 - ▷ Mode synchrone : ce mode nécessite l'appel de la méthode `receive()` ou d'une de ses surcharges. Dans ce cas, l'application est arrêtée jusqu'à l'arrivée du message. Une version surchargée de cette méthode permet de rendre la main après un certain timeout.
 - ▷ Mode asynchrone : il faut définir un listener qui va lancer un thread attendant les messages et exécutant une méthode à leur arrivée.
- ▶ JMS propose un support pour différents types de messages : texte brut, flux d'octets, objets Java sérialisés, ...



JMX

- ▶ JMX est l'acronyme de Java Management Extensions
- ▶ JMX est une spécification qui définit une architecture, une API et des services pour permettre de surveiller et de gérer des ressources Java. JMX permet de mettre en place, en utilisant un standard, un système de surveillance et de gestion d'une application, d'un service ou d'une ressource sans avoir à fournir beaucoup d'effort.
- ▶ JMX peut permettre de configurer, gérer et maintenir une application durant son exécution en fonction des fonctionnalités développées. Il peut aussi favoriser l'anticipation de certains problèmes par une information sur les événements critiques de l'application ou du système.



Pour aller plus loin

Ressources supplémentaires :

- ▶ <https://www.jmdoudoux.fr/java/dej/chap-jms.htm>
- ▶ <https://www.jmdoudoux.fr/java/dej/chap-jmx.htm>



11.

Journée TP



Journée TP

- ▶ Voir sujet sur Google Doc