

FORMATION JAVA AVANCEE – NOUVEAUTES JAVA 8

B. NASS LAHSEN / N. BERRAF / P. DESPREZ

Le 30/09/2019



BNP PARIBAS

La banque d'un monde qui change

Sommaire & Organisation

Plan du cours

- Introduction à Java 8
- Rappel sur Maven (optionnel)
- Les expressions lambda
- La programmation fonctionnelle avancée
- L'API Stream
- Logging dans Java (optionnel)
- Nouvelle API Date/Time
- Java 8 Nashorn (deprecated)
- La classe Optional
- La persistance

Plan des TPs

- TP Lambda
- TP Référence de méthode
- TP Interfaces fonctionnelles
- TP Stream
- TP DateTime
- TP Optional
- TP JPA (optionnel)

Organisation

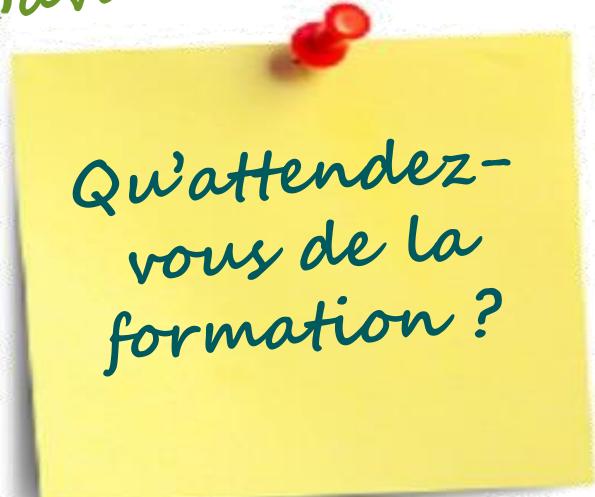
- Formation sur 2 jours
 - 50% de théorie
 - 50% de pratique



Tour de table

Expérience dans le
développement Java ?

Avez-vous travaillé
avec Java 8 ?



Qu'attendez-
vous de la
formation ?





INTRODUCTION À JAVA 8

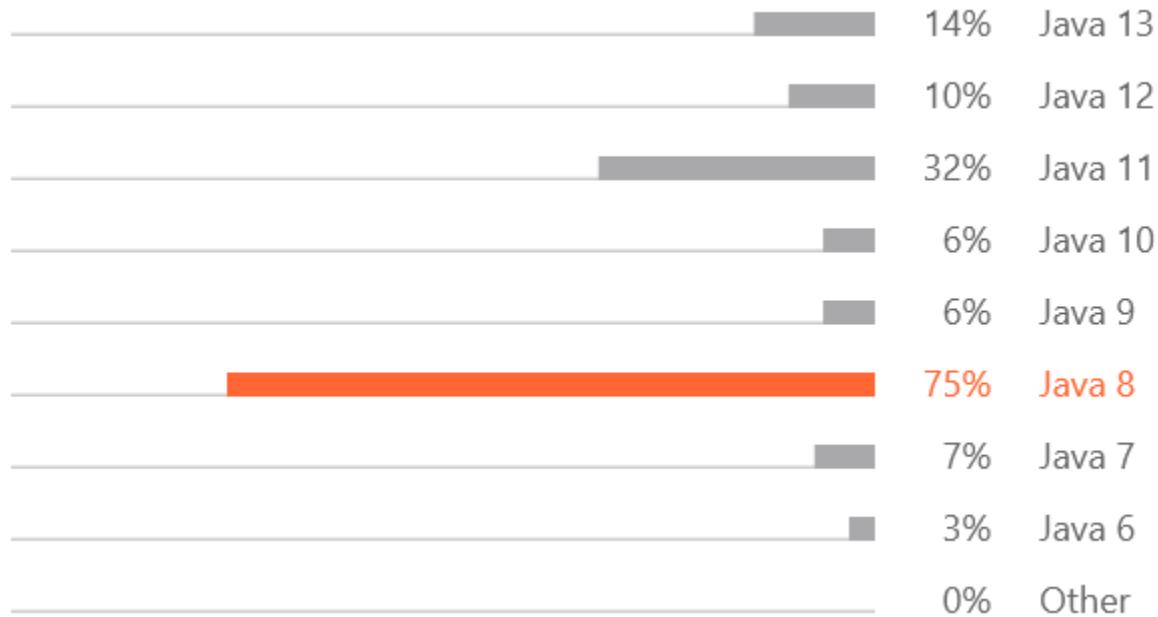


Introduction

- Java 8 est arrivé en Mars 2014
- Il s'agit de la mise à jour la plus importante depuis 15 ans que Java existe :
 - Ajout de la programmation fonctionnelle (concept de lambda)
 - Refonte API « Collections » : apparition du concept de *stream*
 - Simplification de la programmation parallèle
 - Forte adoption de Java SE 8 depuis son lancement.



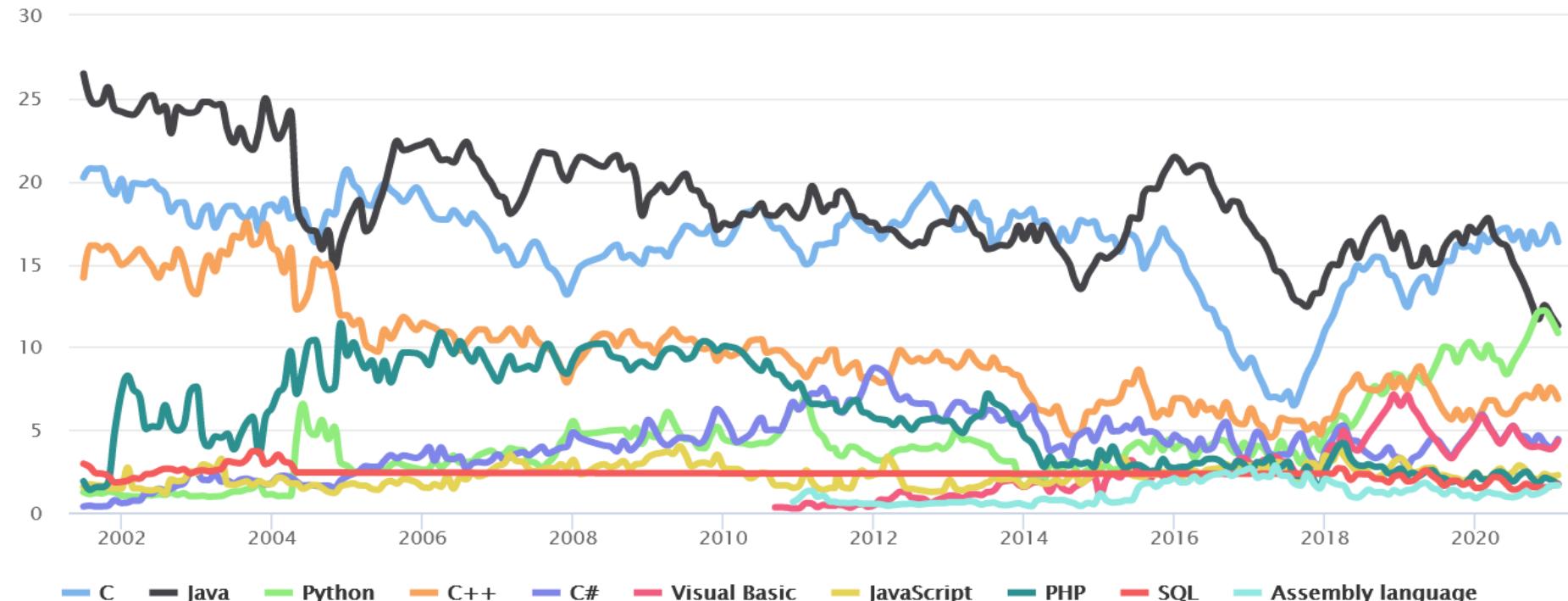
Java 8 adoption



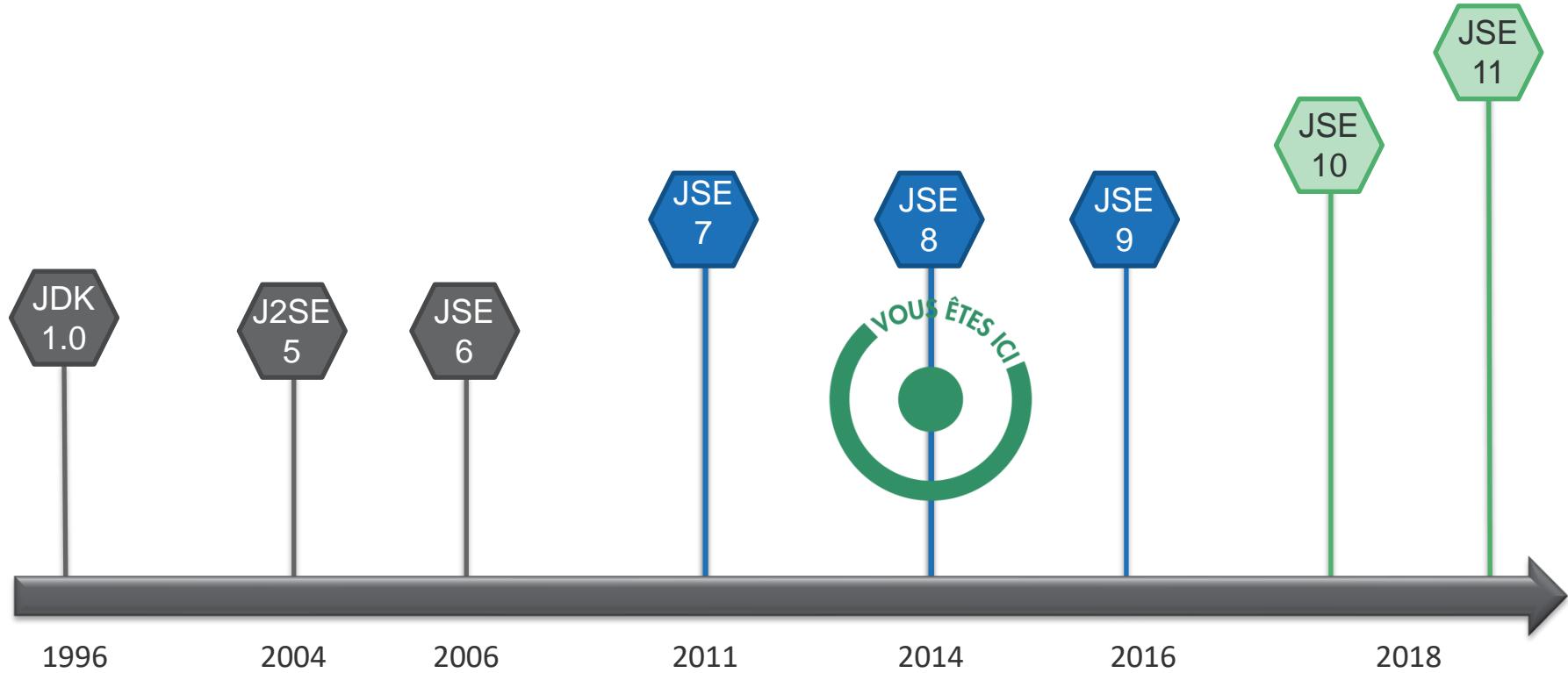
Java dans le peloton de tête des langages de programmation

TIOBE Programming Community Index

Source: www.tiobe.com



Un peu d'histoire



Evolution des versions de Java

Version	Date de release	Principales fonctionnalités
1.0	1996	La première version stable
1.1	1997	<ul style="list-style-type: none">• Un réaménagement important du modèle d'événement AWT• Classes internes ajoutées• Réflexion• I18n et Unicode Supportés
1.2(j2se)	1998	<ul style="list-style-type: none">• Framework Swing• Framework Collection
1.3	2000	<ul style="list-style-type: none">• Moteur de monitoring et de gestion de performance ajouté• Ajout de RMI
1.4	2002	<ul style="list-style-type: none">• Regular Expression Supportées• Gestion des exceptions haut niveau ajouté• Gestion non bloquante IO(NIO 1)• Java Web Start



Evolution des versions de Java

Version	Date de release	Principales fonctionnalités
1.5(J2SE 5)	2004-2009 22 mises à jours	<ul style="list-style-type: none">• Types Génériques• Annotations• Enumération• Varargs• Foreach
1.6(J2SE 6)	2006-2013 Plus de 50 mises à jours La dernière113	<ul style="list-style-type: none">• Amélioration des performances• Amélioration de la sécurité• JDBC 4
J2SE 7	2011-now Plus de 50 mises à jours La dernière101	<ul style="list-style-type: none">• String dans la clause switch• NIO 2• Types générique Diamond• API haut niveau de la gestion de concurrence



Evolution des versions de Java

Version	Date de release	Principales fonctionnalités
Java SE 8 (LTS)	2014-aujourd'hui	<ul style="list-style-type: none">• Expression lambda et la programmation fonctionnelle• Emulateur embarqué JavaScript• Nouvelles annotations• Méthode par défaut dans les interface
Java SE 9	Septembre 2017	<ul style="list-style-type: none">• Un meilleur support pour les Heaps de plusieurs Go• Système de module Java (système intégré OSGI)• Support HTTP 2.• Meilleur supports des flux• Meilleur support pour la parallélisations• API pour Json et la gestion des devises
Java SE 10	Mars 2018	<ul style="list-style-type: none">• API de gestion des monnaies et devises• Tableaux adressables 64 bits• Local-Variable (var) – Inférence de type



Evolution des versions de Java

Version	Date de release	Principales fonctionnalités
Java SE 11 (LTS)	Septembre 2018	<ul style="list-style-type: none">• Inférence de type pour les paramètres de lambdas• Nouveau client HTTP
Java SE 12	Mars 2019	<ul style="list-style-type: none">• TBD
Java SE 13	Septembre 2019	<ul style="list-style-type: none">• Amélioration du switch• Amélioration du CDS/AppCDS

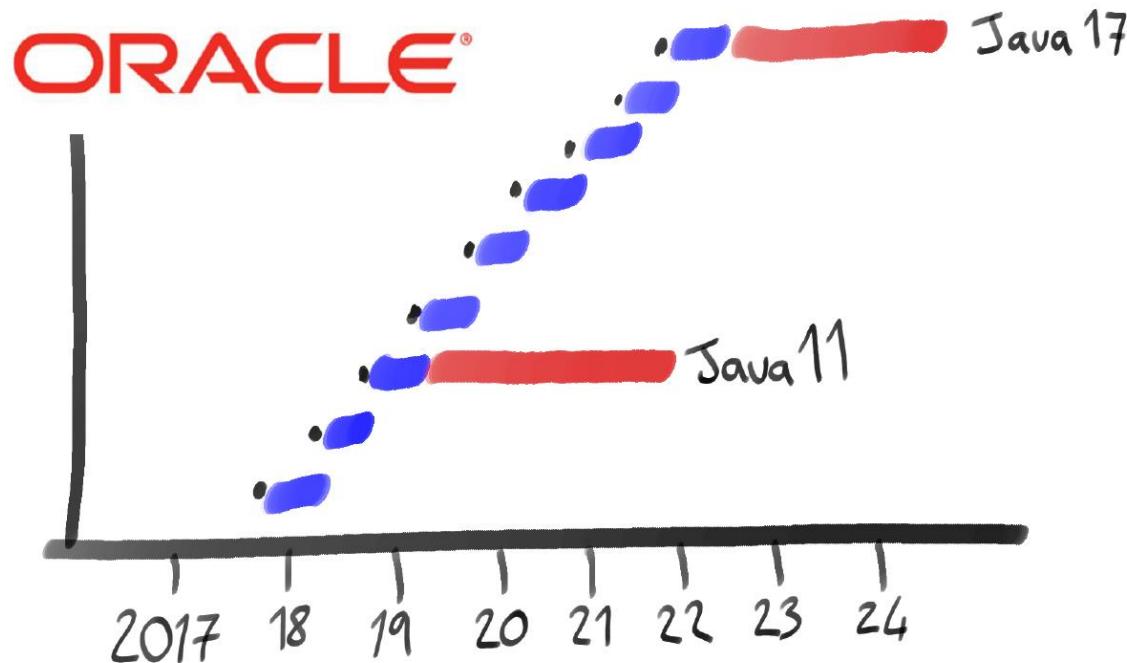


Evolution des versions de Java

Version	Date de release	Principales fonctionnalités
Java 14	Mars 2020	<ul style="list-style-type: none">• JEP 358 : Helpful NullPointerExceptions• JEP 359 : Records (Preview)• JEP 305 : Pattern Matching for instanceof (Preview)• JEP 361 : Switch Expressions (Standard)• JEP 368 : Text Blocks (Second Preview)
Java 15	Septembre 2020	<ul style="list-style-type: none">• Hidden Classes (JEP 371)• Retirer le moteur JavaScript Nashorn !!! (JEP 372)• Text Blocks (Standard) (JEP 378)
Java 17 (LTS)	Septembre 2021

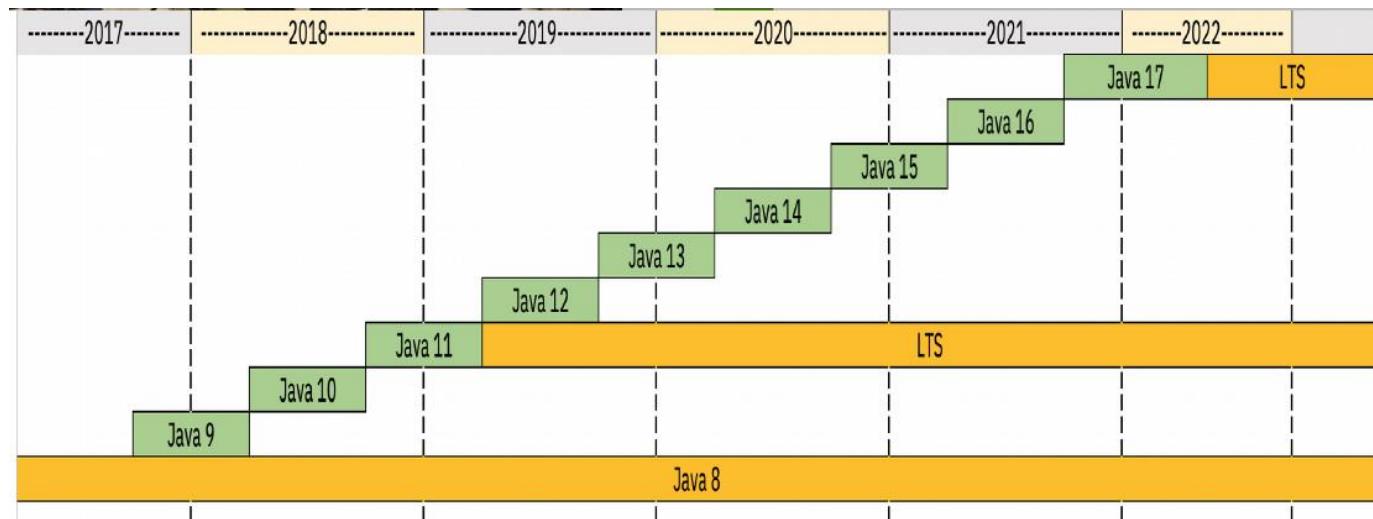


L'après java 8



Nouveau rythme

- 2 releases par an
 - 1 release sur 3 sera LTS (Long Term Support)
 - LTS = 3 ans de support par Oracle
 - Non LTS = 6 mois de support par Oracle
 - Par de recouvrement entre une LTS et la suivante



ET désormais plusieurs distributions OpenJDK



ORACLE®



IBM®



AdoptOpenJDK



BNP PARIBAS

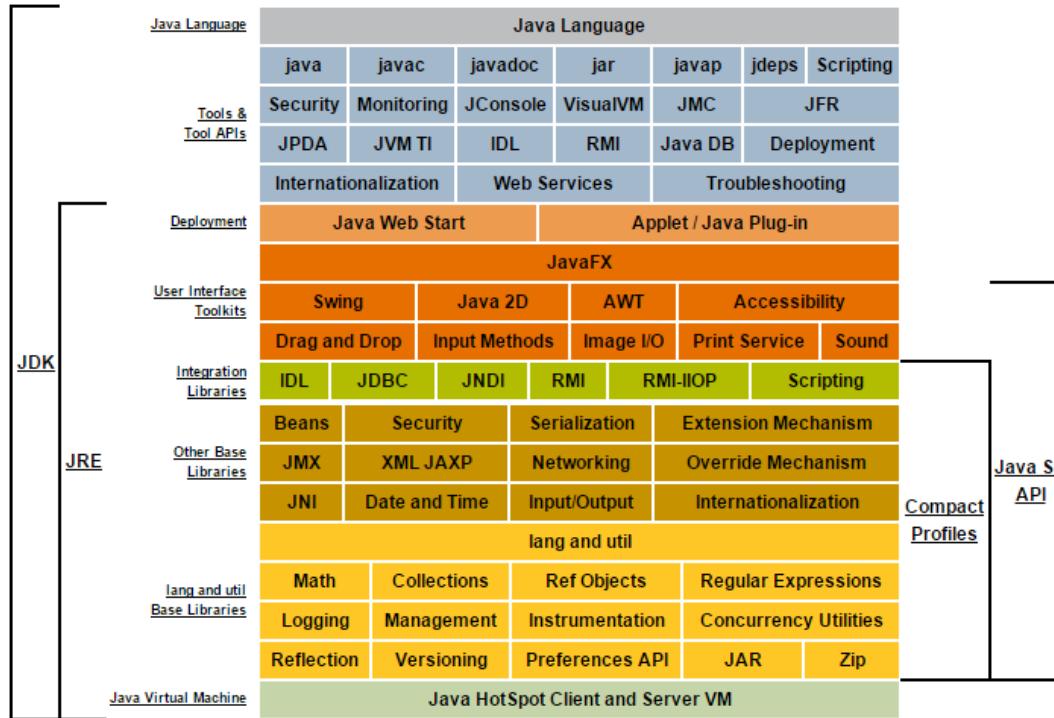
La banque d'un monde qui change

StampedLock Adder Suppression Perm gen
Type Annotations Compact Profiles Nashorn
Lambda Stream Date & Time
Default Method Interface fonctionnelle Base64
Accumulator Method references Parallel array

- Depuis combien de temps utilisez-vous Java 8 ?



Modèle conceptuel de Java 8



<https://docs.oracle.com/javase/8/docs/>



- Java 8

- Date de sortie 18/03/2014
- Java 8 est le plus grand changement à Java depuis la création de la langue
- le lot de nouveautés le plus important depuis Java 5
 - Stream et parallèles streams sur les collections
 - API java.time
 - Méthodes par défaut

- JSR 335: Lambda Expressions

- Les Lambdas sont la nouvelle addition la plus importante. java joue le rattrapage: la plupart des langages de programmation majeurs possèdent déjà un support pour les expressions lambda
- Un grand défi était de d'introduire les lambdas sans nécessiter de recompilation des binaires existants



Nouveautés Java 8

● Lambda

○ Syntaxe:

- (paramètre) -> expression;
- (paramètre) -> {corps de méthode};

○ Privilégie l'inférence de type

L'**inférence de types** est un mécanisme qui permet à un compilateur ou un interpréteur de rechercher automatiquement les **types** associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source.

● API Stream

- Traitement sur une séquence d'éléments
- Bien adapté pour les collections, moins pour les maps
- Limiter l'utilisation de la méthode forEach()



Nouveautés Java 8

- Optional
 - Classe qui encapsule une valeur ou l'absence de valeur
 - Pour se forcer à se poser la question si l'objet est présent ou pas
 - Un peu de surcout mémoire et surtout pas Serializable.
 - OptionallInt, OptionalLong, OptionalDouble.
- Nashorn qui remplace Rhino en moteur javascript
- Date & Time
 - Ne plus utiliser Date et Calandar, ni même JodaTime
 - TemporalAdjuster: permet d'écrire du code pour ajuster une date.
 - Classe abstraite Clock : permet de fournir l'heure courante



Autres nouveautés Java 8

- Annotations améliorées
- Parallel arrays
 - Arrays.setAll(lambda) -> plus sympa à écrire
 - Arrays.parallelSetAll(lambda) -> pour paralléliser
- Réflexion sur les paramètres de la méthode
- Aucun PermGen dans JVM Hotspot
- JavaFX est enfin prêt à remplacer Swing
- Base64
- Contrôle amélioré des processus OS



Rappel sur Maven

- La notion de build
 - maven propose au développeur de définir lui-même à quoi correspond un 'build'.
 - compilation bien sûr, mais aussi lancement des tests unitaires examen de la couverture de test inspection du code
 - génération de la javadoc
 - création d'un livrable (jar, war, ear) Etc...
- Chaque tâche est exécutée via un plugin déclaré dans un fichier xml appelé pom.xml



Rappel sur Maven - La gestion des dépendance

- Maven propose une gestion fine des dépendances des bibliothèques à travers un dépôt central qui référence la plupart des bibliothèques et frameworks Java
- Le dépôt est accessible depuis internet
 - mvnrepository.apache.org

Rappel sur Maven - La gestion des dépendance

- La démarche est simple : le développeur se contente d'indiquer les dépendances directes dans une version donnée
 - junit 4.4 par exemple
- Maven prend en charge la récupération des dépendances indirectes



Rappel sur Maven - La gestion des dépendances

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework</groupId>
  <artifactId>gs-spring-boot</artifactId>
  <version>0.1.0</version>
  <properties>
    <java.version>1.8</java.version>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>2.1.7.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- DevTools -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <optional>true</optional>
    </dependency>
  </dependencies>
</project>
```

Rappel sur Maven - structure de projet

- Organisation des sources
- Maven propose d'avoir 4 sources folders :
 - src/main/java : classes java de l'application
 - src/test/java : classes java des tests unitaires
 - src/main/resources : fichiers xml, properties, etc... nécessaires pour l'exécution (donc à mettre dans le classpath)
 - src/test/resources : fichiers xml, properties, etc... nécessaires pour l'exécution des tests unitaires (donc à mettre dans le classpath)



Rappel sur Maven - mise en œuvre

- Utiliser Maven consiste à
 - Créez un projet de type maven
 - Renseignez les dépendances dans le fichier pom.xml à la racine du projet.
 - Utiliser un plugin maven si ce n'est pas le cas déjà (Q4E, MIA, m2votre IDE)
- Il est possible d'enrichir le cycle de projet par défaut en rajoutant d'autres plugins
 - génération du code
 - analyse du code
 - génération des rapports
 - génération de la documentation



Rappel sur Maven - mise en œuvre

- Les informations à renseigner sont les suivantes
- un groupId
- un artifactId
- une version
- un type de livrable (war, ear, jar) la définition du build
- un ensemble de dépendances
- les plugins



Rappel sur Maven - mise en œuvre

- L'ensemble des dépendances récupérées par maven sont stockées automatiquement dans le répertoire :
 - c:\documents and settings\.m2\repository
- Le résultat du build est stocké dans le répertoire target.



- Installation de JDK 8:
 - Lancer jdk-8u181-windows-x64.exe
 - Ajouter la variable d'environnement JAVA_HOME qui doit pointer sur le répertoire d'installation de JDK-8
 - Ajouter dans le path: %JAVA_HOME%\bin
 - echo %JAVA_HOME%
 - Vérifier l'installation: lancer la commande java –version
- Installation de 7zip
 - Lancer 7z1805-x64.exe
- Installation de Notepad++
 - Lancer npp.7.5.8.Installer.exe



Installation d'un environnement de développement

● Installation Maven

- Dans MesDocuments
 - Dézipper apache-maven-3.5.4-bin.zip dans MesDocuments(avec 7zip)
- Ajouter la variable d'environnement M2_HOME qui doit pointer sur le répertoire d'installation de Maven
- Ajouter dans le path: %M2_HOME%\bin
- Vérifier l'installation: mvn -version

● Installation votre IDE

- Créer le répertoire workspace dans MesDocuments
- Dézipper eclipse-java-photon-R-win32-x86_64.zip
- Lancer eclipse en utilisant le répertoire workspace créé précédemment
- Paramétrer eclipse pour pointer sur settings.xml de l'installation maven locale



Création d'un premier projet avec maven

- Créer un répertoire de travail maven-tp1 dans workspace
- Aller dans le répertoire workspace/maven-tp1 et lancer la commande:
- Ajouter le fichier .gitignore fourni à la racine du projet SampleProject

```
mvn archetype:generate -DgroupId=fr.training.samples -DartifactId=SampleProject  
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

- Importer le projet créé, comme un projet maven sous eclipse
- Lancer la classe de tester AppTest, afin de valider le projet

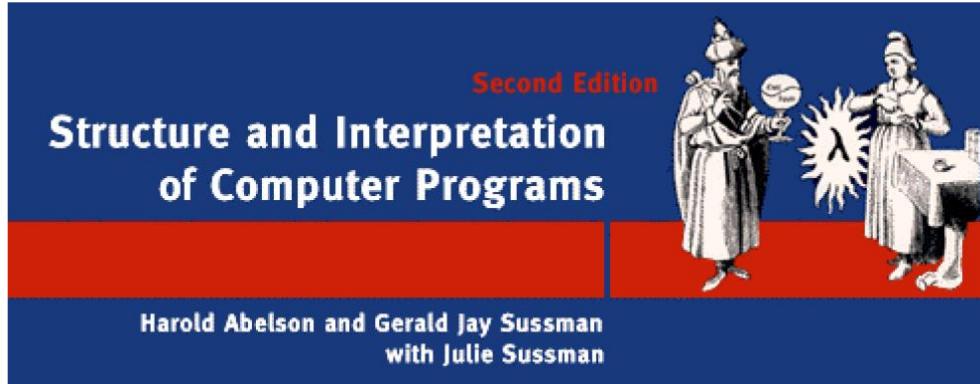


O LES EXPRESSIONS LAMBDA



Les langages fonctionnels

- lisp, scheme, scala, clojure, ocaml, haskell, ruby? python ?
- depuis 1958
- gros regain d'intérêt à cause du parallélisme



Les langages fonctionnels

- On a toujours besoin d'une classe et d'un objet de cette classe
- Mais le compilateur les crée à la volée à votre place;
- Création d'une classe interne anonyme
- Pas forcément de déclaration
- Une lambda peut remplacer toute valeur dont le type est *une interface fonctionnelle*.



Les expressions lambda. Pourquoi ?

- Ajout de programmation fonctionnelle
- Effacer le côté verbeux de JAVA
- Pression des langages «alternatifs» ? (Groovy, Scala, C#,...)



Qu'est-ce que une expression Lambda ?

- Une expression lambda = instance d'une « interface fonctionnelle »
- Possède 1 méthode
- Annotation @FunctionalInterface, pas obligatoire mais permet de valider les interfaces à la compilation
- Une expression lambda est utilisée pour représenter une interface fonctionnelle sous la forme d'une expression de la forme :
(arguments) -> corps

```
@FunctionalInterface  
public interface Consumer<T> {  
    public void accept(T t);  
}
```

```
a -> System.out.println(a)
```



Une expression lambda est-il un objet ?

La classe anonyme suivante :

```
Consumer<String> c = new Consumer<String>() {  
    @Override  
    public void accept(String s) {  
        System.out.println(s) ;  
    }  
} ;
```

Peut être implémenté sous la forme d'une expression lambda :

```
Consumer<String> c = s -> System.out.println(s) ;
```



Impact des expressions Lambdas

- Aussi appelé Closures (Groovy)
- Les constructions deviennent plus courtes et plus claires, surtout pour les classes anonymes
- Mise à jour requise pour les interfaces
 - Nouvelles méthodes requises dans les interfaces des Collections pour lambda
 - L'extension de toutes les interfaces existantes n'est pas une bonne idée
 - Ajouter des méthodes par défaut à l'interface: mot-clé 'par défaut' + méthode corps
 - De plus, des méthodes statiques ont été ajoutées, afin d'éviter de créer des classes utilitaires lorsqu'elles nécessitent des méthodes statiques liées à l'interface (mot-clé static')



Lambda dans Java 8

- La fonction à un nom:

- `public String printNumber(Integer i) { return i.toString(); }`

- Voici la même fonction sans nom

- `(Integer i) -> { return i.toString(); }`

- Syntaxe d'un lambda

- `[parameters] -> [body]`



Lambda dans Java 8

- Un lambda dans Java 8, est fondamentalement une méthode en Java sans déclaration habituellement écrite comme
- (paramètres) -> {body}. Exemples:
 - `(int x, int y) -> { return x + y; }`
 - `x -> x * x`
 - `() -> x`
- Un lambda peut avoir zéro ou plusieurs paramètres séparés par des virgules et leur type peut être explicitement déclaré ou déduit du contexte.
- Les parenthèses ne sont pas nécessaires autour d'un seul paramètre.
- () Est utilisé pour désigner des paramètres zéro.
- Le corps peut contenir zéro ou plus d'instructions.
- Les accolades ne sont pas nécessaires autour d'un corps de déclaration unique.



Lambda dans Java 8 - Exemples

- Fonction nommée:

- public String sayThanks() { return "Thank you"; }

- Lambda correspondant:

- () -> { return "Thank you"; }

- Pas de paramètres: ()

- Ou simplement:

- () -> "Thank you"

- Pour une seule expression, les accolades ne sont pas nécessaires

- Pour une déclaration unique qui renvoie une valeur, le « return » est facultatif

- MAIS si accolades, on doit également utiliser « return »

- () -> {"Thank you"; } ← Ne compile pas



Avantages de Lambdas en Java 8

- Activation de la programmation fonctionnelle
- Écrire un code plus compact et plus compact
- Faciliter la programmation en parallèle
- Développer plus d'API génériques, flexibles et réutilisables
- Être capable de transmettre des comportements ainsi que des données à des fonctions



Cas typiques d'utilisation

- Les classes anonymes (GUI)
- Runnables / Callables
- Comparateur
- Appliquer l'opération à une collection via la méthode foreach



Exemple 1

- Imprimer une liste d'entiers avec un lambda

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);  
intSeq.forEach(x -> System.out.println(x));
```

- On passe par une expression lambda qui définit une fonction anonyme avec un paramètre nommé x du type Entier



Exemple 2

- Lambda multi-lignes

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);
intSeq.forEach(x -> {
    x += 2;
    System.out.println(x);
});
```

- Les accolades sont nécessaires pour enfermer un corps multi-lignes dans une expression lambda.



Exemple 3

- Lambda avec une variable locale définie

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);
intSeq.forEach(x -> {
    int y = x * 2;
    System.out.println(y);
});
```

- Tout comme pour les fonctions ordinaires, vous pouvez définir des variables locales dans le corps d'une expression lambda



Exemple 4

- Lambda avec des paramètres

```
List<Integer> intSeq = Arrays.asList(1, 2, 3);
intSeq.forEach((x -> {
    x += 2;
    System.out.println(x);
}));
```

- Vous pouvez, si vous le souhaitez, spécifier le type de paramètre.



REX - Lambda

- N'utilisez pas de parenthèses de paramètres quand c'est optionnel

```
// prefer
str -> str.toUpperCase(Locale.US);

// avoid
(str) -> str.toUpperCase(Locale.US);
```



REX - Lambda

- Ne déclarons pas les variables locales comme «final»

```
public UnaryOperator<String> upperCaser(Locale locale) {  
    return str -> str.toUpperCase(locale);  
}
```

Do not declare as 'final'



REX - Lambda

- Préférer l'expression lambdas sur les blocs lambda
- Utiliser une méthode distincte si nécessaire

```
// prefer
str -> str.toUpperCase(Locale.US);

// use with care
str -> {
    return str.toUpperCase(Locale.US);
}
```



REX - Checked exceptions

- La plupart des interfaces fonctionnelles ne déclarent pas d'exceptions
- Aucune façon simple de mettre des exceptions vérifiées dans les lambdas

```
// does not compile!
public Function<String, Class> loader() {
    return className -> Class.forName(className);
}
```

Throws a checked exception



REX - Checked exceptions

- Écrivez ou trouvez une méthode utilitaire
- Convertit l'exception contrôlée en désactivé

```
public Function<String, Class> loader() {  
    return Unchecked.function(  
        className -> Class.forName(className));  
}
```



λ

Functional Programming



Lambda – TP1

- Il s'agit de remplacer le calcul de la somme fourni par la classe MathOperationNoJava8, en passant par les expressions lambda
 - Créer une classe MathOperationJava8
 - Passer par les expressions lambda pour obtenir le résultat



Lambda – TP2

- Il s'agit de remplacer l'exécution d'un thread en passant par les expressions Lambda
 - Remplacer l'implémentation de Runnable sur la classe LambdaRunnable en passant par les expressions Lambda
 - Remplacer l'implémentation de Thread sur la classe LambdaThread en passant par les expressions Lambda



Lambda – TP3

- La classe `LambdaComparator`, permet de trier par ordre croissant, une collection de personnes en fonction de leur âge.
- Transformer le code fourni en passant par les expressions Lambda et les API Java 8 pour obtenir le même résultat



Référence de méthode

- Cette fonctionnalité est liée aux expressions lambda.
- Parfois, les expressions lambda appellent une méthode existante. Dans ces cas, les références de méthodes apparaissent.
- Les références de méthode renvoient une méthode existante par nom.
- Une référence de méthode est décrite en utilisant le symbole "::".
- Type de méthode Références



Résumé: Référence de méthode

Type de méthode de référence	Syntaxe	Exemple
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString



Exemple: Référence de méthode

```
public class Main {  
  
    public static boolean predicate(Person elmt) {  
        return elmt.age < 19;  
    }  
  
    public static void main(String[] args) {  
  
        List<Person> persons = new LinkedList<Person>();  
        persons.add(new Person("fred", 20));  
        persons.add(new Person("xavier", 18));  
        persons.add(new Person("rémi", 15));  
        persons.add(new Person("pierre", 19));  
  
        CollectionUtils.filter(persons, Main::predicate);  
    }  
}
```

- Le compilateur détecte que la méthode référencée par Main::predicate est de la forme Person -> Boolean

- Sûre syntaxique ?

```
persons.forEach(System.out::println);
```



Exemple: Référence de méthode

- Référence de méthode

```
FileFilter x = File f -> f.canRead();
```



```
FileFilter x = File::canRead;
```

- Référence de constructeur

```
Factory<List<String>> f = () -> return new ArrayList<String>();
```



```
Factory<List<String>> f = ArrayList<String>::new;
```



Les limites des lambdas expressions

- Lambda ne peuvent pas lever checked exceptions





Référence de méthode - TP1

- Créer le package : com.training.methodreferences.tp1
- Créer la classe InstanceMethodReference avec une méthode main
- Ajouter la déclaration d'une liste:

```
List<String> fruits = Arrays.asList("Orange", "Apple", "Banana");
```
- Afficher l'ensemble des éléments de la liste de fruits en utilisant les expressions lambda
- Afficher l'ensemble des éléments la liste de fruits en utilisant les références de méthodes
- Afficher l'ensemble des éléments de la liste de fruits en les triant alphabétiquement
 - Il est possible de passer par la méthode compareToIgnoreCase de la classe String



Référence de méthode – TP2

com.training.methodreferences.tp2

- Remplacer l'expression lambda :

```
(delimiter, list) -> {
    StringBuilder sb = new StringBuilder(100);
    int size = list.size();
    for (int i = 0; i < size; i++) {
        sb.append(list.get(i));
        if (i < size - 1) {
            sb.append(delimiter);
        }
    }
    return sb.toString();
}
```

- Par une référence de méthode permettant de concaténer les chaînes de la liste

Indice →

* Utiliser la méthode join de la classe String



BNP PARIBAS

La banque d'un monde qui change

Référence de méthode – TP3

com.training.methodreferences.tp3

- Remplacer l'implémentation du Supplier

- Supplier<Collection<Integer>> supplier
- Par une référence du constructeur ArrayList

- Remplacer l'implémentation du Supplier

- Supplier<Collection<Integer>> supplier
- Par une référence de constructeur permettant d'avoir une liste triée





LA PROGRAMMATION FONCTIONNELLE AVANCÉE



Qu'est-ce que la programmation fonctionnelle?

- Un style de programmation qui traite le calcul comme l'évaluation des fonctions mathématiques
- Élimine les effets secondaires
- Gère les données comme immuables
- Les expressions ont une transparence référentielle
- Les fonctions peuvent prendre les fonctions en tant qu'arguments et fonctions de retour en tant que résultat
- Prévient la récurrence sur des boucles explicites



Pourquoi la programmation fonctionnelle?

- Nous permet d'écrire des programmes plus faciles à comprendre, plus déclaratifs et plus concis que la programmation impérative
- Permet de nous concentrer sur le problème plutôt que sur le code
- Facilite le parallélisme



Interface fonctionnelle

- L'idée : une interface qui définit une et une seule fonction
- Définition: interface qui a exactement une méthode abstraite
- Elle peut avoir une ou plusieurs méthodes par défaut



Interfaces fonctionnelles

- Une interface fonctionnelle est une interface Java avec exactement une méthode non-par défaut. Par exemple.,

```
public interface Consumer<T> {  
    void accept(T t);  
}
```

- Décision de conception: Les lambdas dans Java 8, se basent sur les interfaces fonctionnelles.
- Le package `java.util.function` ajoute plusieurs interfaces fonctionnelles très utiles



Interface fonctionnelle - SAM

- **SAM pour Single Abstract Method**
- Une interface fonctionnelle est une interface qui a une seule méthode abstraite, et représente donc un contrat de fonction unique. (Peut avoir d'autres méthodes avec des corps)
- Les classes abstraites peuvent être considérées à l'avenir
- L'annotation `@FunctionalInterface` contribue à garantir l'honneur du contrat Functional Interface
- Que se passe-t-il lorsque vous avez plus d'une méthode abstraite et utilisez `@FunctionalInterface`?



Interface fonctionnelle - Exemples

- Une interface avec une seule méthode abstraite
 - Runnable
 - Comparable
 - Callable
- Java SE 8 ajoute plusieurs interfaces fonctionnelles:
 - Function<T, R>
 - Predicate<T>
 - Supplier<T>
 - Consumer<T>



Fonctions prédéfinies

- Package `java.util.function`
- Quelques exemples:
 - `Function`: un paramètre, un résultat
 - `BiFunction`: deux paramètres, un résultat
 - `UnaryParameter`: identique à `Function`, mais le résultat est du même type que le paramètre
 - `BiOperator`: identique à `BiFunction`, mais les paramètres et les résultats sont du même type
 - `Predicate`: Fonction qui retourne vrai ou faux
 - `Supplier`: représente un fournisseur de résultats
 - `Consumer`: prend un paramètre, pas de résultat
 - `TypeSpecific`, p.ex. `IntFunction`, `IntToDoubleFunction`



java.util.functions

- C'est dans ce package que sont les interfaces fonctionnelles
- Il y en a 43
- Des versions existent pour les types primitifs (minimiser l'autoboxing)

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
<code>Runnable</code>	none	<code>void</code>	<code>run</code>	Runs an action without arguments or return value	
<code>Supplier<T></code>	none	<code>T</code>	<code>get</code>	Supplies a value of type T	
<code>Consumer<T></code>	<code>T</code>	<code>void</code>	<code>accept</code>	Consumes a value of type T	<code>chain</code>
<code>BiConsumer<T, U></code>	<code>T, U</code>	<code>void</code>	<code>accept</code>	Consumes values of types T and U	<code>chain</code>
<code>Function<T, R></code>	<code>T</code>	<code>R</code>	<code>apply</code>	A function with argument of type T	<code>compose,</code> <code>andThen,</code> <code>identity</code>
<code>BiFunction<T, U, R></code>	<code>T, U</code>	<code>R</code>	<code>apply</code>	A function with arguments of types T and U	<code>andThen</code>
<code>UnaryOperator<T></code>	<code>T</code>	<code>T</code>	<code>apply</code>	A unary operator on the type T	<code>compose,</code> <code>andThen,</code> <code>identity</code>



java.util.functions

Supplier

```
public interface Supplier<T> {  
    T get() ;  
}
```

Un supplier fournit un objet

Consumer

```
public interface Consumer<T> {  
    void accept(T t) ;  
}
```

Un consommateur consomme un objet

```
Consumer<String> c1 = s -> System.out.println(s) ;  
Consumer<String> c2 = ... ;  
Consumer<String> c3 = c1.andThen(c2) ;  
persons.stream().forEach(c3) ;
```



Java.util.functions

Function

```
public interface Function<T, R> {  
    R apply(T t) ;  
}
```

Predicate

```
public interface Predicate<T> {  
    boolean test(T t) ;  
}
```

- Les fonctions peuvent être chaînées et / ou composées
- BiFunction prend deux arguments au lieu d'un
- UnaryOperator et BinaryOperator opèrent sur un seul type

- Un Predicate prend un objet et retourne un booléen



Les implémentations par défaut des interfaces

- Initialement conçu pour ajouter des fonctionnalités aux interfaces existantes
 - list.sort(ordering) au lieu de Collections.sort(list, ordering)

```
* @since 1.8
*/
@unchecked, rawtypes/
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```



Méthodes par défaut

- Pour diminuer cette nécessité d'utilisation d'une classe abstraite, java 8 introduit la notion de « méthode par défaut » dans une interface.
 - Une interface peut contenir le corps d'une méthode (précédé de default) ;
 - Elle ne peut toujours pas contenir de variable d'instance (sinon, ce serait une classe) ;
 - Quand on implémente une telle interface, on n'est pas obligé de réécrire les méthodes par défaut.
 - Une classe peut implémenter *plusieurs interfaces* avec des méthodes par défaut
 - S'il y a conflit (deux méthodes par défaut de même signature sont héritées), elle doit les redéfinir ou c'est une erreur.



Méthodes par défaut

Exemple 1

```
public class C implements A, B {  
    public String a() { ... }  
}
```

La classe gagne !

```
public interface A {  
    default String a() { ... }  
}
```

```
public interface B {  
    default String a() { ... }  
}
```

Exemple 2

```
public class C implements A, B {  
    ...  
}
```

*Erreur de compilation :
« class C inherits unrelated
defaults for a() from types
A and B »*

```
public interface A {  
    default String a() { ... }  
}
```

```
public interface B {  
    default String a() { ... }  
}
```



Conflit avec plusieurs interfaces

- Étant donné que les classes de java peuvent implémenter plusieurs interfaces, il peut y avoir une situation dans laquelle 2 ou plus d'interfaces ont une valeur par défaut
- Méthode avec la même signature provoquant des conflits alors que java ne connaît pas quelle méthode utiliser

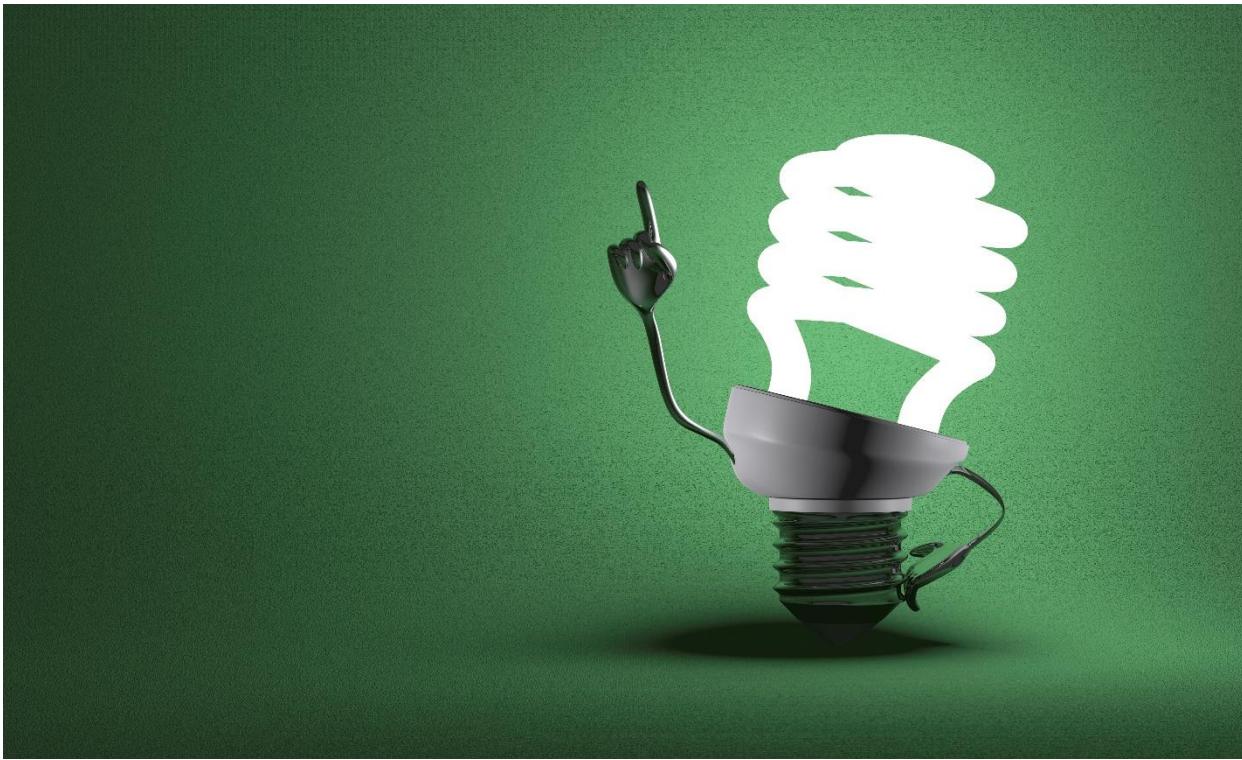
```
 MyClass inherits unrelated defaults for sayHi() from types InterfaceA and InterfaceB
```



Méthodes par défaut

- Ce que l'on a en Java est l'héritage multiple de type :
 - Java 8 amène l'héritage multiple d'implémentation
 - Ce que l'on a pas, c'est l'héritage multiple d'état ... et d'ailleurs, on n'en veut pas !
- 2 règles simples pour gérer les conflits de l'héritage multiple d'implémentation :
 - 1) La classe gagne
 - 2) Les implémentations les plus spécifiques gagnent





Interfaces fonctionnelles - TP1

- Soit la fonction f , qui converti la distance du Mile au Km:
 - $f(d) = 1.6 \times d$
 - Avec
 - d : Distance en Miles de type décimal
- Créer un package:
 - com.training.functionalinterface.tp1
- Créer une classe:
 - MilesToKmConverter
- Ecrire un programme qui affiche la conversion en Km, d'une distance en Miles, en passant par l'interface fonctionnelle Function
 - 3 miles = 4,80 kilometers



Interfaces fonctionnelles - TP2

- Soit la fonction f , qui calcule la surface d'un rectangle:
 - $f(L,l) = L \times l$
 - Avec
 - l : Largeur de type décimal
 - L : Longueur de type décimal
- Créer un package:
 - com.training.functionalinterface.tp2
- Créer une classe:
 - CalculSurface
- Ecrire un programme qui affiche la surface en passant par l'interface fonctionnelle BiFunction



Interfaces fonctionnelles - TP3

- Soit la fonction f , qui converti un degré Celsius en degré Fahrenheit:
 - $f(t) = 1.8 \times t + 32$
 - Avec
 - t : Température en Celsius de type entier
- Créer un package:
 - com.training.functionalinterface.tp3
- Créer une classe:
 - CelsiusToFahrenheitConverter
- Ecrire un programme qui affiche la température en passant par l'interface fonctionnelle `IntToDoubleFunction`





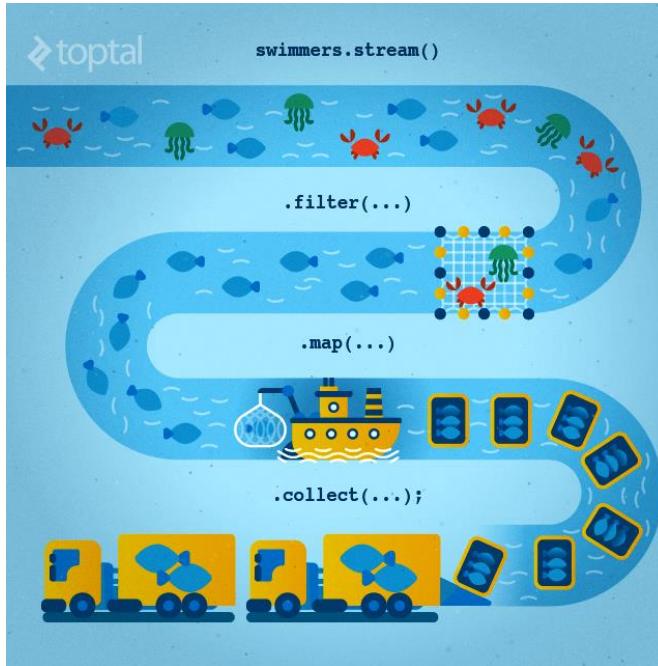
LES STREAMS



BNP PARIBAS

La banque d'un monde qui change

Les évolutions des Streams



Problématique du d'implémentation du Stream en Java8

La communauté Java voulait proposer les streams en java afin de faire simplement des calculs sur les éléments d'une collection :

Collection.stream() retourne un Stream : une nouvelle interface

```
// map / filter / reduce pattern sur Collection
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

Donc on a besoin d'une nouvelle méthode sur Collection

```
public interface Collection<E> {
    // nos bonnes vieilles méthodes
    Stream<E> stream() ;
}
```

Problème : ArrayList ne compile plus... Une solution doit être trouvée !



Interfaces java 8

- ArrayList a besoin de l'implémentation de stream()...
- Solution : mettons-les dans une interface ! *Méthodes par défaut*

```
public interface Collection<E> {  
    // nos bonnes vieilles méthodes  
    default Stream<E> stream() {  
        return ... ;  
    }  
}
```

- Une interface devient-elle une classe abstraite ? Non !
- Une classe abstraite peut avoir des champs et possède un constructeur
- Une implémentation par défaut est juste du code, lié à la compilation



API Stream

- Nouveau concept introduit dans Java 8
 - Permet de traiter efficacement de grands comme de petits volumes de données
- Un stream est un pipeline d'opérations
- Un Stream n'est pas une collection
 - Ne stocke pas de données
 - Ne viens pas polluer l'API Collection
- Techniquement,
 - Une interface :
 - Et des méthodes : filter, map, reduce, sorted, count, collect, forEach ...



API Stream

- Le nouveau package `java.util.stream` fournit des utilitaires pour prendre en charge les opérations de style fonctionnel sur des flux de valeurs.
- Un moyen commun d'obtenir un `stream` provient d'une collection:
 - `Stream<T> stream = collection.stream();`
- Les Streams peuvent être séquentiels ou parallèles.
- Les Streams sont utiles pour sélectionner des valeurs et effectuer des actions sur les résultats.



Stream: Une nouvelle Notion

- Un stream ne porte pas de donnée
- Un stream ne peut pas modifier sa source
- Une source peut être infinie, ou non bornée
- Un Stream traite ses données de façon lazy



Opérations sur les Streams

- Une opération intermédiaire maintient un flux ouvert pour d'autres opérations. Les opérations intermédiaires sont Lazy.
- Une opération de terminaison doit être l'opération finale sur un flux. Une fois qu'une opération de terminaison est invoquée, le flux est consommé et n'est plus utilisable.



L'API Stream – Comment construire un Stream ?

- De nombreuses façons de faire...

1. À partir d'une collection :

```
Collection<String> collection = ... ;  
Stream<String> stream = collection.stream() ;
```

2. À partir d'un tableau :

```
Stream<String> stream2 =  
Arrays.stream(new String [] {"one", "two", "three"}) ;
```

3. À partir des méthodes factory de Stream

```
Stream<String> stream1 = Stream.of("one", "two", "three") ;
```



L'API Stream – Comment construire un Stream ?

Encore quelques patterns :

```
Stream.empty() ; // Stream vide
Stream.of(T t) ; // un seul élément
Stream.generate(Supplier<T> s) ;
Stream.iterate(T seed, UnaryOperator<T> f) ;
```

Encore d'autres façons :

```
string.chars() ; // retourne un IntStream
lineNumberReader.lines() ; // retourne un Stream<String>
random.ints() ; // retourne un IntStream
```



L'API Stream – Les différents types d'opérations

On peut donc déclarer des opérations sur un Stream deux types d'opérations :

- 1)Les opérations intermédiaires
 - Exemple : `map`, `filter`
- 2)Les opérations terminales, qui déclenchent le traitement
 - Une seule opération terminale est autorisée
 - Un Stream ne peut être traité qu'une seule fois
 - Si besoin, un autre Stream doit être construit
 - Exemple : `reduce`



L'API Stream – Les différents types d'opérations

● Opération sur les collections, tableaux ou IO

○ Intermédiaires

- sorted(Comparator<? super T> comparator)
- map , filter, peek
- ...

○ Terminales

- allMatch(Predicate<? super T> predicate)
- collect(Collector<? super T,A,R> collector)
- get, forEach, max, min, findAny

● Pattern Map/Filter/ Reduce

```
Integer integer = persons.stream().map(p -> p.age = p.age + 10)
    .filter(age -> age > 20).reduce(Integer::sum).get();
```



L'API Stream – Utilisation

Un premier exemple : Retour sur notre map / filter / reduce

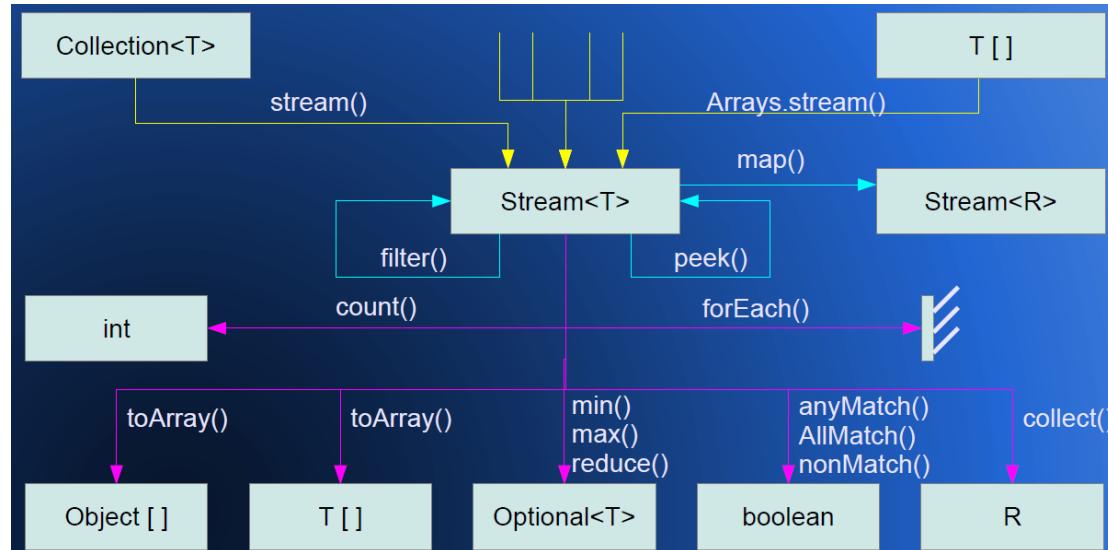
```
// map / filter / reduce pattern on collections
int sum = persons.stream()
    .map(p -> p.getAge())
    .filter(a -> a > 20)
    .reduce(0, (a1, a2) -> a1 + a2) ;
```

Un deuxième exemple : génération d'une chaîne de caractères

```
Random rand = new Random();
Stream<String> stream = Stream.generate( () ->
    Long.toHexString(rand.nextLong()));
```



API Stream



L'API Stream – Construction d'un Stream parallèle

Deux patterns :

- 1)Appeler parallelStream() au lieu de stream()

```
Stream<String> s = strings.parallelStream() ;
```

- 2)Appeler parallel() sur un stream existant

```
Stream<String> s = strings.stream().parallel() ;
```



forEach

- `forEach` méthode disponible sur les interfaces `Iterator` & `Map` et leurs implémentations
- Permet le contrôle interne de l'itération des éléments pour un éventuel fonctionnement en parallèle

```
List<String> names =  
    Arrays.asList("Alice", "Bob", "Charlie");  
names.forEach(e -> { System.out.println(e); });
```



- Permet de « revenir » des streams aux collections (ou à d'autres types).
 - Rassemble les résultats d'un traitement à l'aide d'un Collector
 - En pratique, on utilise les collectors définis dans la classe Collectors :
 - `toList()`
 - `toSet()`
 - `counting`
 - `groupingBy`
 - `Collectors.joining` : concaténation



Map & FILTER

- La méthode `map`: Applique une fonction à tous les éléments du stream, et renvoie un autre stream :

```
List<String> titres=  
    messages.stream()  
        .map(m -> m.getTitle())  
        .collect(Collectors.toList());
```

- La méthode `filter`: Exclue tous les éléments qui ne correspondent pas à un prédictat



reduce

- alternative à `collect` ; permet de combiner les éléments du stream en un seul à l'aide d'un opérateur ;
- soit T le type des éléments du stream, et une opération f :
 - $T \times T \rightarrow T$;
- alors `reduce` applique cette opération à tous les éléments du stream.
- Exemple:

```
List<Integer> l= Arrays.asList(3,10, 7,15,20);  
System.out.println(l.stream().reduce(1, (a,b)-> a*b));
```



Combinaison des opérations

```
Set<String> titresByToto=  
    messages.stream()  
        .filter(m-> m.getAuthors().equals("toto"))  
        .map(m -> m.getTitle())  
        .collect(Collectors.toSet());
```



Problème

- Factoriser le filtrage d'une collection suivant un prédictat

```
list<Person> persons = new LinkedList<Person>();  
persons.add(new Person("fred", 20));  
persons.add(new Person("xavier", 18));  
persons.add(new Person("rémi", 15));  
persons.add(new Person("pierre", 19));  
  
List<Person> filteredList = new ArrayList<Person>(persons);  
  
for (Person person : persons) {  
    if (!(person.age < 19)) {  
        filteredList.remove(person);  
    }  
}
```

Comment factoriser cette partie de code ?

Éviter de dupliquer ce code pour tous les filtrages

```
for (Person person : persons) {  
    if (!(person.name.startsWith("p"))){  
        filteredList.remove(person);  
    }  
  
for (Animal a : animals) {  
    if (!(a.poids > 200 )) {  
        filteredList.remove(a);  
    }  
}
```



Solution 1 : Classe Abstraite

- Classe abstraite avec une fonction abstraite « predicate »

```
public abstract class AbstractCollectionsUtil<T> {  
  
    1  public List<T> filter(List<T> list) {  
        List<T> filteredList = new ArrayList<T>(list);  
        if (list != null) {  
            for (T elmt : list) {  
                if (!predicate(elmt)) {  
                    filteredList.remove(elmt);  
                }  
            }  
        }  
        return filteredList;  
    }  
  
    2  public abstract boolean predicate(T elmt);  
}  
  
  3  public class PersonCollectionsUtil  
extends AbstractCollectionsUtil<Person> {  
  
    /**  
     * {@inheritDoc}  
     */  
    @Override  
    public boolean predicate(Person elmt) {  
        return elmt.age < 19;  
    }  
}
```

- Instanciation :

 `List<Person> filteredList = PersonCollectionsUtil.getInstance().filter(persons);`



Solution 2 : Interface

- Fonction prenant en paramètre : une liste et une interface

```
1  public static <T> boolean filter(final Iterable<T> collection, final Predicate<? super T> predicate) {  
    boolean result = false;  
    if (collection != null && predicate != null) {  
        for (final Iterator<T> it = collection.iterator(); it.hasNext(); ) {  
            if (!predicate.evaluate(it.next())) {  
                it.remove();  
                result = true;  
            }  
        }  
    }  
    return result;  
}  
  
2  public interface Predicate<T> {  
    /**  
     * Use the specified parameter to perform a test that returns true or false.  
     *  
     * @param object the object to evaluate, should not be changed  
     * @return true or false  
     * @throws ClassCastException (runtime) if the input is the wrong class  
     * @throws IllegalArgumentException (runtime) if the input is invalid  
     * @throws FunctorException (runtime) if the predicate encounters a problem  
     */  
    boolean evaluate(T object);  
}
```

- Création d'une classe anonyme pour exécuter le filtrage

★ CollectionUtils.filter(persons, new Predicate<Person>() {
 @Override
 public boolean evaluate(Person _object) {
 return _object.age > 19;
 }
});



Solution 3 : JDK8 / Lambda (1/2)

- La fonction définie précédemment ne change pas

1

```
public static <T> boolean filter(final Iterable<T> collection, final Predicate<? super T> predicate) {  
    boolean result = false;  
    if (collection != null && predicate != null) {  
        for (final Iterator<T> it = collection.iterator(); it.hasNext(); ) {  
            if (!predicate.evaluate(it.next())) {  
                it.remove();  
                result = true;  
            }  
        }  
    }  
    return result;  
}
```

- L'interface définie précédemment ne change pas

2

```
@FunctionalInterface  
public interface Predicate<T> {  
  
    boolean evaluate(T object);  
}
```

→ Optionnel

Solution 3 : JDK8 / Lambda (2/2)

 Solution 2

```
CollectionUtils.filter(persons, new Predicate<Person>() {  
    @Override  
    public boolean evaluate(Person _object) {  
        return _object.age > 19;  
    }  
});
```

```
CollectionUtils.filter(persons, (Person p) -> p.age > 19);
```

- Le compilateur voit cette lambda expression comme une instance de Predicat.
- Type non obligatoire :

```
CollectionUtils.filter(persons, p -> p.age > 19);
```



Questions

- Détection d'erreurs à la compilation ?

```
CollectionUtils.filter(persons, (Person p) -> p.age = 15);
```

Type mismatch: cannot convert from Integer to boolean
Press 'F2' for focus

- Puis-je mettre une Lambda dans une variable ?

```
Predicate<Person> predicate = p -> p.age > 19;  
CollectionUtils.filter(persons, predicate);
```

- Une lambda peut-elle prendre plusieurs paramètres?

Cas d'utilisation	Exemple de lambda	Functional interface JDK8
Combine two values	(int a, int b) -> a * b	IntBinaryOperator
Creating objects	() -> new Apple(10)	Supplier<Apple>



Les Streams – Pattern Iterator

```
public class Person {  
    private String name;  
    private int age;  
    // constructors  
    // getters / setters  
}
```

Un bon vieux bean ...

... et une bonne vieille liste

```
List<Person> list = new ArrayList<>()  
;
```

Calculons la moyenne des âges des personnes

```
int sum = 0 ;  
// I need a default value in case  
// the list is empty  
int average = 0 ;  
for (Person person : list) {  
    sum += person.getAge() ;  
}  
if (! list.isEmpty()) {  
    average = sum / list.size() ;  
}
```



Les Streams – Pattern Iterator

Plus dur : pour les personnes de plus de 20 ans

```
int sum = 0 ;
int n = 0 ;
int average = 0 ;
for (Person person : list) {
    if (person.getAge() > 20) {
        n++ ;
        sum += person.getAge() ;
    }
}
if (n > 0) {
    average = sum / n ;
}
```

```
select avg(age)
from Person
where age > 20
```

Dans ce cas la base de données
mène le calcul
comme elle l'entend :
Ici on décrit le résultat

« programmation impérative »



« programmation déclarative »



BNP PARIBAS

La banque d'un monde qui change

Les Streams – Pattern Map/Filter/Reduce



Mapping :

- prend une liste d'un type donné
- retourne une liste d'un autre type
- possède le même nombre d'éléments

Filtrage :

- prend une liste d'un type donné
- retourne une liste du même type
- mais avec moins d'éléments

Réduction :

- agrégation des éléments d'une liste dans un seul élément
- Ex : moyenne, somme, min, max, etc...



Les Streams – Mise en œuvre Pattern Map/Filter/Reduce

On crée une interface pour modéliser le mapper...

```
public interface Mapper<T, V> {  
    public V map(T t) ;  
}
```

... et on crée une classe anonyme

```
new Mapper<Person, Integer>() {  
    public Integer map(Person p) {  
        return p.getAge() ;  
    }  
}
```



On peut faire la même chose pour le filtrage

```
public interface Predicate<T> {  
    public boolean filter(T t) ;  
}
```

```
new Predicate<Integer>() {  
    public boolean filter(Integer i) {  
        return i > 20 ;  
    }  
}
```

Et enfin pour la réduction

```
public interface Reducer<T> {  
    public T reduce(T t1, T t2) ;  
}
```

```
new Reducer<Integer>() {  
    public Integer reduce(Integer i1, Integer i2) {  
        return i1 + i2 ;  
    }  
}
```



Les Streams – Mise en œuvre Pattern Map/Filter/Reduce

Et on applique...

```
List<Person> persons = ... ;  
int sum =  
persons.map(  
    new Mapper<Person, Integer>() {  
        public Integer map(Person p) {  
            return p.getAge() ;  
        }  
    })  
.filter(  
    new Filter<Integer>() {  
        public boolean filter(Integer age) {  
            return age > 20 ;  
        }  
    })  
.reduce(0,  
    new Reducer<Integer>() {  
        public Integer recude(Integer i1, Integer i2) {  
            return i1 + i2 ;  
        }  
    })  
);
```



Les Streams – Mise en œuvre Pattern Map/Filter/Reduce

Prenons l'exemple du Mapper

```
mapper = new Mapper<Person, Integer>() {  
    public Integer map(Person person) {// 1 méthode  
        return person.getAge();  
    }  
}
```

```
mapper = (Person person) -> person.getAge();
```

Le compilateur reconnaît cette expression comme une implémentation du mapper



Les Streams – Mise en œuvre Pattern Map/Filter/Reduce

Que se passe-t-il si ...
... il y a plus d'une ligne de code ?

```
mapper = (Person person) -> {  
    System.out.println("Mapping " + person) ;  
    return person.getAge() ;  
} // Accolades, un return explicite
```

...le type de retour est void ?

```
consumer = (Person person) -> p.setAge(p.getAge() + 1) ;
```

...la méthode prend plus d'un argument ?

```
reducer = (int i1, int i2) -> {  
    return i1 + i2 ;  
}
```

ou

```
reducer = (int i1, int i2) -> i1 + i2 ;
```

Comment le compilateur reconnaît-il l'implémentation du mapper ?

```
mapper = (Person person) -> person.getAge() ;
```

- 1) Il ne faut qu'une méthode dans le mapper
- 2) Les types des paramètres et le type de retour doivent être compatibles
- 3) Les exceptions levées doivent être compatibles



Les Streams – Mise en œuvre Pattern Map/Filter/Reduce

On peut écrire d'autres lambdas facilement :

```
mapper = (Person person) -> person.getAge() ; // mapper
filter = (int age) -> age > 20 ; // filter
reducer = (int i1, int i2) -> i1 + i2 ; // reducer
```

Et la plupart du temps, le compilateur reconnaît ceci :

```
mapper = person -> person.getAge() ; // mapper
filter = age -> age > 20 ; // filter
reducer = (i1, i2) -> i1 + i2 ; // reducer
```

Le type des paramètres peut être omis

```
reducer = (int i1, int i2) -> {
    return i1 + i2 ;
} // reducer = (i1, i2) -> i1 + i2 ;
```



Le concept de lambda – La réduction

A handwritten derivation on a chalkboard illustrating the reduction of a lambda expression. At the top, a sequence of numbers is shown in boxes: $3 | 4 | 2 | 6 | 7 | \dots | 9 | 0$. Below this, two terms are shown: i_1, i_2 and $i_1 + i_2$. A red bracket groups i_1, i_2 , and a blue bracket groups $i_1 + i_2$. A red arrow points from i_1, i_2 to $i_1 + i_2$. A blue arrow points from $i_1 + i_2$ down to $i_1 + i_2$. To the right, the text "Red(i₁, Red(i₂, i₃))" is written above "Red(Red(i₁, i₂), i₃)". Below this, the lambda expression $\lambda = (i_1, i_2) \rightarrow i_1 + i_2$ is written in green.



Le concept de lambda – La réduction

2 exemples :

```
Reducer r1 = (i1, i2) -> i1 + i2 ; // Ok Reducer  
  
r2 = (i1, i2) -> i1*i1 + i2*i2 ; // Oooops
```

Attention :

- L'opération doit être associative
- Sinon le résultat est toujours reproductible en série mais il ne l'est en général pas en parallèle



Multithreading

La machine virtuelle se charge de diviser le calcul en thread et d'utiliser le maximum de cœurs du processeur.

- Test 1

```
for (int i = 1_000_000; i < 4_000_000; i++) {  
    integer.add(i);  
}  
  
Map<Object, List<Integer>> groupByPrimary = numbers.stream().collect(  
    Collectors.groupingBy(s -> Primes.isPrime(s)));
```

- Test 2

```
for (int i = 1_000_000; i < 15_000_000; i++) {  
    integer.add(i);  
}  
  
integer.stream().parallel().filter(s -> Primes.isPrime(s)).count();
```



REX - API Stream

- Exécution de traitement sur une séquence d'éléments
 - Obtenu d'une source finie ou infinie
 - Exécution d'un pipeline d'opérations
 - Exécution séquentielle ou en //
- Requiert de réfléchir en fonctionnel



REX - API Stream

- Attention à l'ordre des opérations intermédiaires
 - Ex : filter() + sorted() vs sorted() + filter() => performances
- Ne pas abuser des Streams
- Bien adapté pour les collections, moins pour les map
- Limiter l'utilisation du `forEach` (raisonnement impératif et pas fonctionnel)
- Déboguer un Stream
 - Plutôt difficile
 - Utiliser la méthode `peek()`
 - Ou utiliser une référence de méthode + point d'arrêt



REX - API Stream

- Avec des données primitives, utilisez par DoubleStream, IntStream ... il y'a gros impacts sur les perfs à cause de l'auto-boxing
- Utilisation des Stream infinis
 - Penser à utiliser des limites (ex : limit())



REX - Les Streams parallèles

- Facilité de mise en œuvre qui ne rime pas forcément avec performance
- Utilise le framework Fork/Join et son pool par défaut
- Les opérations intermédiaires de type stateful vont dégrader un peu les perfs
- Attention au Spliterator
 - Certaines sources de données (ex : LinkedList et I/O) sont peu performantes)
- Attention aux Collectors
 - Performance (ex : grouping())
 - Utilisation d'objets ayant un support de la concurrence => pas de HashMap
- Attention au surcoût de la parallélisations
 - Doit être compensé par le volume de données à traiter



QUESTIONS

- Un stream est un pipeline d'opérations
- Les Streams transforment les données
- Les streams sont une structure de données
- Les streams ne peuvent pas modifier les données



Les streams...

Sont à usage unique

Possèdent des opérations intermédiaires et terminales

Sont sensibles à l'ordre des opérations

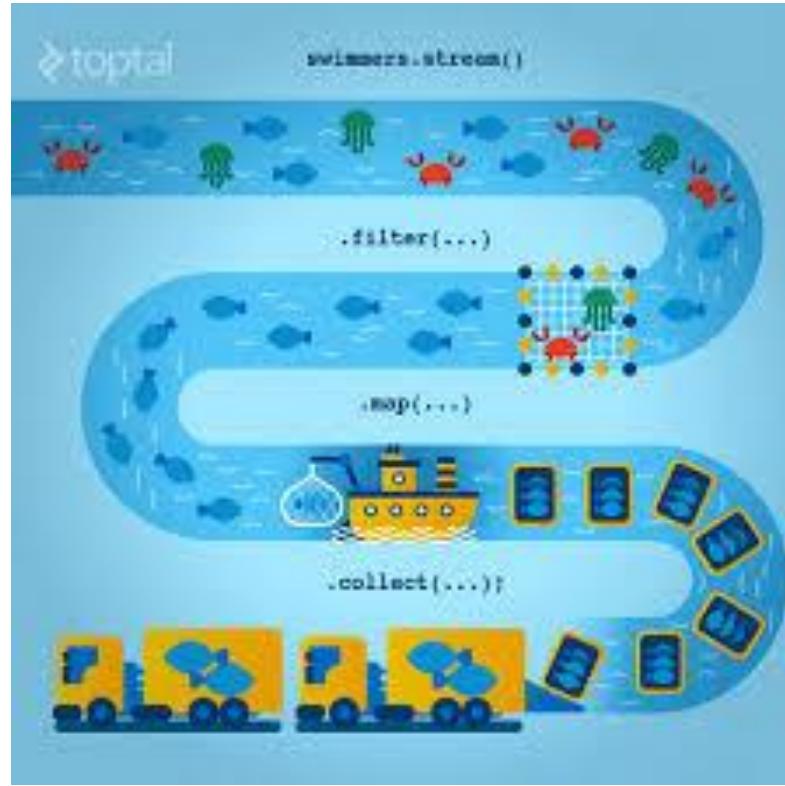
Produisent du code concis et **lisible**

Permettent de faire des opérations simples, simplement!

Sont un **pipeline d'opérations** pas une structure de données
(non mutable ☺)



TP Stream



Stream - TP1

- Sur la classe StreamCollect1, remplacer dans la méthode collect:
 - Les expressions lambda
 - Par des références de méthode



Stream - TP2

- Sur la classe StreamCollect2, remplacer dans la méthode collect:
 - Les références de méthode par des expressions lambda



Stream - TP3

- Sur la classe StreamConcat, concaténer les streams *stream1* et *stream2*
- Afficher le résultat en passant par l'api Java 8



- Créer une classe `StreamFilter` dans le package `fr.ib.training.stream.tp4`
- En passant par l'API **Stream** et la méthode `filter`:
 - lister tous les fichiers « `.java` » dans le répertoire parent et les sous-répertoires



Stream - TP5 - Optionnel

- Dans la classe `School`, implémenter la méthode `calculateAvgAge`, qui permet de calculer l'age moyen des enseignants.



Stream - TP6 - Optionnel

- Dans la classe `PersonStream`, en passant par l'API Stream:
 - Trouver les personnes qui ont un nom Jeff
 - Trouver l'âge maximal des personnes
 - Convertir la liste de personnes en une `Map` avec comme clé l'âge et la valeur le nom de la personne
 - `List<Person> vers Map<Integer, String>`
 - Afficher tous les éléments de la `Map` en passant par les API java 8



Stream - TP7 - Optionnel

- Créer une classe `StreamExercice` dans le package `fr.ib.training.stream.tp7`
- Voici la chaîne étudiée:

- `List<String> words = Arrays.asList("hi", "hello", "hola", "bye", "goodbye", "adios");`

1. Produire une chaîne unique qui résulte de la concaténation des éléments du tableau en majuscules:
 - Ex., le résultat devrait être "HIHELLO ...".
 - Utilisez une seule opération de réduction, sans utiliser `map`.
2. Produire la même chaîne que ci-dessus, mais cette fois par une opération `map` qui transforme les mots en majuscule, suivie d'une opération de réduction qui les concatène.
3. Produire une chaîne concaténée tout les éléments de la liste, avec une virgule comme séparateur
4. Trouver le nombre total de chaînes de caractères dans la liste.
5. Trouver le nombre de mots contenant un "h".



Stream - TP8 - Optionnel

- En utilisant l'API Stream
 - Afficher la liste des employées triées par numéro d'employée par ordre descendant
 - Afficher la liste des employées triées par nom puis par prénom



O LE LOGGING EN JAVA



La présentation du logging

- Le logging est une activité technique utile et nécessaire dans une application pour :
 - Déboguer : pratique lorsque la mise en œuvre d'un débogueur n'est pas facile.
 - Obtenir des traces d'exécution (démarrage/arrêt, informations, avertissements, erreurs d'exécution, ...)
 - Faciliter la recherche d'une source d'anomalie (stacktrace, ...)
 - Comprendre ou vérifier le flux des traitements exécutés : traces des entrées/sorties dans les méthodes, affichage de la pile d'appels, ...



La présentation du logging

- L'importance du logging croît avec la taille et la complexité de l'application qui l'utilise.
- Une API de logging fait généralement intervenir trois composants principaux :
 - Logger : invoqué pour émettre grâce au framework un message généralement avec un niveau de gravité associé
 - Formatter : utilisé pour formater le contenu du message
 - Appender : utilisé pour envoyer le message à une cible de stockage (console, fichier, base de données, email, ...)



La présentation du logging

- Le logging doit faire partie intégrante des fonctionnalités d'une application. Bien sûr le niveau de gravité des messages n'est pas le même en développement et en production mais le code de l'application doit rester le même. Seule la configuration du logging doit changer dans les différents environnements.
- Généralement la configuration peut être externalisée dans un fichier ce qui rend l'utilisation de l'API plus souple et flexible.
- La modification de la configuration du logging en cours d'exécution de l'application (soit dynamiquement soit par recharge de la configuration) est importante pour permettre d'avoir couramment un niveau de log acceptable et, au besoin, un niveau de log plus fin sans devoir relancer l'application.



La présentation du logging

- Les API de logging ont plusieurs inconvénients :
 - Il faut définir avec précision les messages à ajouter dans les journaux et la pertinence des informations qu'ils contiennent
 - Il faut définir avec précision le niveau de gravité des messages
 - L'utilisation d'une API de logging peut dégrader les performances d'une application



La présentation du logging

- Le logging est particulièrement important dans une application notamment côté serveur mais une utilisation à outrance ou une mauvaise utilisation de cette fonctionnalité peut dégrader les performances générales de l'application.
- Les frameworks de logging sont conçus pour limiter la consommation en ressources nécessaires à leur mise en œuvre mais cette consommation existe tout de même et croît naturellement avec le nombre de messages émis.
- L'utilisation d'une API de Logging implique donc une surcharge de consommation de ressources (CPU, mémoire, ...) mais elle se justifie par l'apport des informations fournies en cas de problème sous réserve que ces informations aient été judicieusement choisies.



Recommandations lors de la mise en œuvre

- Voici quelques règles pour une bonne mise en œuvre du logging :
 - Chaque message doit contenir la date/heure d'émission et la classe émettrice
 - Ne jamais utiliser de System.out pour afficher des messages mais utiliser une API de Logging
 - Ne jamais utiliser la méthode printStackTrace() de la classe Exception pour afficher des messages mais utiliser une API de Logging
 - Eviter les messages émis trop fréquemment (par exemple dans une boucle avec un nombre important d'itérations ou dans une méthode fréquemment invoquée, ...)
 - Utiliser le niveau de gravité en adéquation avec le message



Recommandations lors de la mise en œuvre

- Pour des traces d'exécution, il est pratique d'émettre un message en début d'une méthode qui affiche les paramètres en entrée et un message à la fin de la méthode avec la valeur de retour
- Il est fortement recommandé d'utiliser une API de logging plutôt que d'utiliser la méthode `System.out.println()` pour plusieurs raisons :
 - Une API de logging permet un contrôle sur le format des messages en proposant un format standard pouvant inclure des données telles que la date/heure, la classe, le thread, ...
 - Une API de logging permet de gérer différentes cibles de stockage des messages
 - Une API de logging permet de modifier à l'exécution le niveau de gravité des messages pris en compte
- Sur des applications utilisées par plusieurs utilisateurs, par exemple une application web, il peut être très utile de faire figurer dans le message une identité sur le responsable de l'action (par exemple, l'adresse IP d'une requête http).

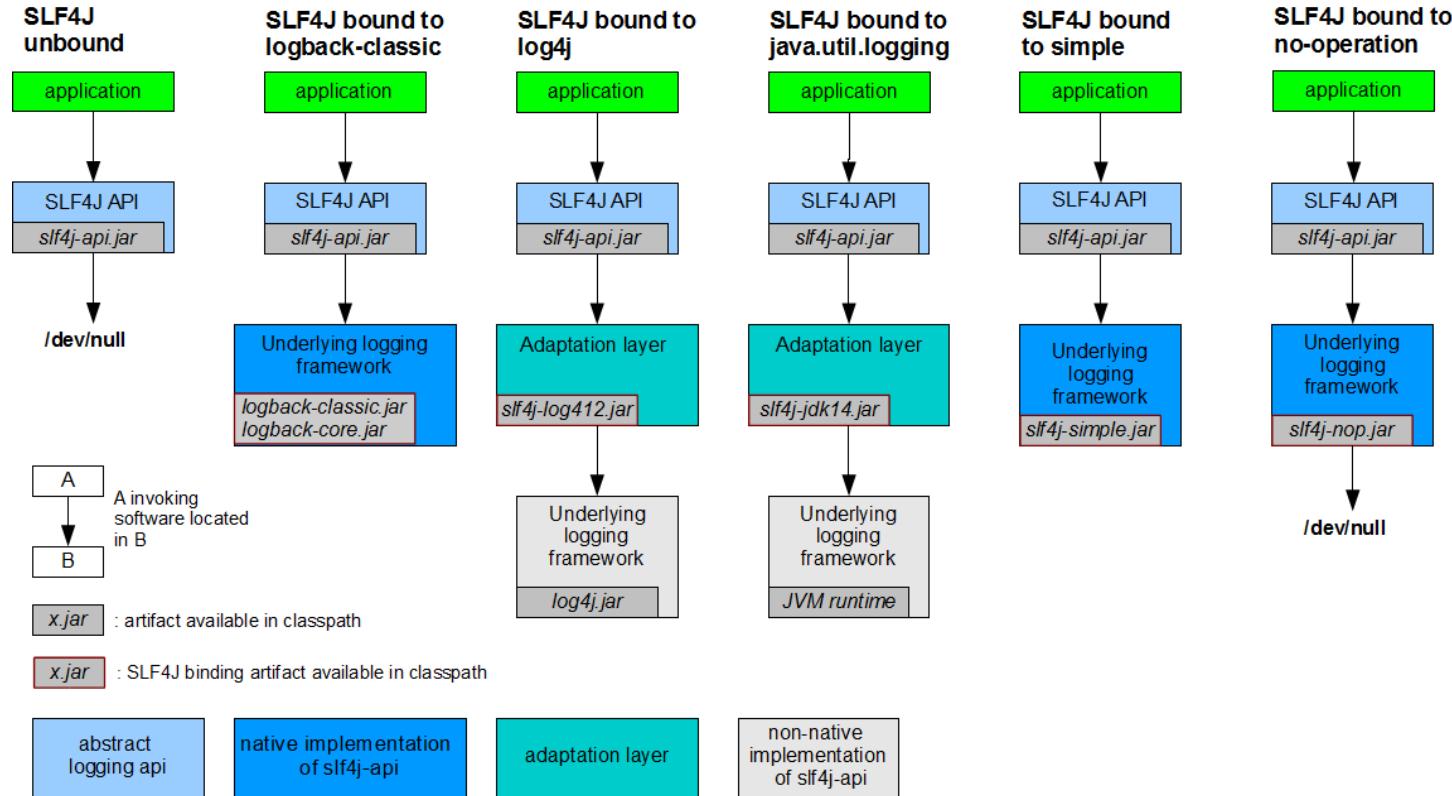


Les différents frameworks

- De nombreux frameworks existent pour mettre en oeuvre le logging dont :
 - Log4j
 - Java Logging
 - Jlog
 - Protomatter
 - SLF4J
 - LogBack



Level : SLF4J une couche d'abstraction



Level : SLF4J et les abstraits



slf4j	Log4j	Log4j2	Logback	java.util.logging
FATAL	FATAL	FATAL		
ERROR	ERROR	ERROR	ERROR	SEVERE
WARN	WARN	WARN	WARN	WARNING
INFO	INFO	INFO	INFO	INFO
				CONFIG
DEBUG	DEBUG	DEBUG	DEBUG	FINE
				FINER
TRACE	TRACE	TRACE	TRACE	FINEST



Logs > Niveaux de sévérité consensuels

Level	Description
TRACE	Journalisation verbeuse. Peut être utilisé occasionnellement pour traquer un problème coriace. Ex : enregistrer le contenu d'un objet (dumping object), E/S méthodes, contenu buffers...
DEBUG	Journalisation potentiellement verbeuse. C'est un niveau indispensable en phase de développement/qualification, parfois activé en production pour une surveillance occasionnelle.
INFO	Journalisation peu verbeuse. C'est l'endroit où le programme rend compte de tout ce qui est considéré normal et signifiant . Ex: début/fin traitement, appel BdD, requête/réponse service tiers...
WARN	Journalisation très peu verbeuse. Enregistrer les erreurs et les exceptions produites lors de l'exécution provoquant un comportement anormal de votre programme que vous aimerez traquer mais ne nécessitant une intervention corrective immédiate . Ex: paramètre attendu non reçu mais un paramètre par défaut est défini, service distant injoignable
ERROR	Journalisation très peu verbeuse. Enregistrer les erreurs et les exceptions internes produites lors de l'exécution provoquant un comportement anormal de votre programme et nécessitant une intervention corrective immédiate . Ex: persistance d'un enregistrement en échec, incapacité de transmettre une notification, dépassement mémoire...



Logs > Sévérité par environnement

Levl/Env.	Dev.	Int.	Qua.	Prev.	Bench	Prod.
TRACE	Parfois	Non	Non	Non	Non	Jamais
DEBUG	Oui	Oui	Oui	Oui	Non	Non
INFO	Oui	Oui	Oui	Oui	Oui	Oui
WARN	Oui	Oui	Oui	Oui	Oui	Oui
ERROR	Oui	Oui	Oui	Oui	Oui	Oui

Oui : activé par défaut

Non : activable occasionnellement

Jamais : n'est jamais activé



Bonnes pratiques

- Une application doit générer des logs
- Les logs doivent être concis, compréhensibles par l'humain et exploitables par les traitements
- Ne jamais utiliser de `System.out`. S'appuyer exclusivement sur les fonctionnalités offertes par votre framework de logs préféré
- Référencer l'ensemble des codes erreur d'une application dans le dossier d'exploitation (entre les mains des ops)
- Votre logging ne doit pas s'appuyer directement sur les API de votre framework de logging. Utilisez une couche d'isolation
- SLF4J est la façade populaire et standard à privilégier pour isoler logback, log4j, Java Util Loggin...
- Les niveaux de严重性 doivent être utiliser avec rigueur
- Les sévérité choisis doivent faire partie de la liste : Trace, Debug, Info, Warning, Error
- Le niveau FATAL est hors scope. Il est confondu avec Error.



Bonnes pratiques

- Ne loggez pas aveuglément. Posez une stratégie dès le départ pour un logging organisé et optimal. Choisissez avec rigueur les parties nécessitant une log
- Un juste milieu DOIT être trouvé entre aucune log et log à chaque ligne de code. La dégradation des performances (CPU, I/O, mémoire...) résultant du logging doit être justifiée
- Evitez de faire faire des traitements riches aux fonctions de logs afin de déduire le contenu de la log. Ceci peut amener :
 - Risque de dégradation des performances
 - Risque de produire des erreurs supplémentaires
 - Risque d'aboutir sur des valeurs nulles et perdre le sens de la log
- Un système de logging ne doit en aucun cas pénaliser l'application qu'il sert (surconsommation, blocage, dépassement mémoire...)
- Le format de la log doit être UTF8
- Les données sensibles non utiles au diagnostic ne doivent pas être loggées (mdp, iban, n° carte...)



Bonnes pratiques

- La log ne doit pas être à l'intérieur d'une boucle
- Logger ne doit pas influer sur l'état des autres objets de l'application
- Utiliser l'anglais comme langue principale pour la partie message
- Le fichier de configuration de votre système de log doit être à l'extérieur de votre application
 - La prise en compte du nouveau paramétrage ne doit pas nécessiter de relancer l'application
- Déclarer votre logger comme static final :
`private static final Logger log = LoggerFactory.getLogger(Foo.class);`
 - Final : ce qui est vrai car la valeur du logger n'est pas modifiée au sein de la classe
 - static : ce qui est vrai car on utilise généralement un logger par classe
 - Private : éviter le détournement de l'objet par une autre classe (hijack)
- Tout échange avec un service externe doit être loggé
- Toute lecture/écriture dans un fichier doit être loggée
- Redéfinir la méthode `toString` afin de permettre une sérialisation de votre objet pour une meilleure relecture dans la log
 - `logger.info("Info {}", Customer)` fait appel à la méthode `Customer.toString()` si présente



Bonnes pratiques

- Structurer le file system accueillant les fichiers logs comme suit (normes groupe : ref. RI : 000090) :
 - /applis/logs/[codeAF]-[codeAT]-[id]/wlc
 - codeAF : AP + partie numérique sur 5 positions (géré par l'application REFI)
 - codeAT: partie alphanumérique sur 5 positions définissant l'application technique
 - Id : identifiant optionnel
 - wlc : WebSphere Liberty Core
 - /applis/logs/[codeAP]-[code5car]-[id]/ihs
 - Ihs : serveur web
- Utilisez le format suivant pour nommer votre fichier log :
 - {codeAF}-{codeAT}.log
- Une rotation de fichier logs doit être prévue. Elle doit prendre en compte la taille des fichiers et la durée souhaitée pour leur conservation.
- Utilisez le format suivant pour nommer les archives de votre fichier log :
 - {codeAF}-{codeAT}_{timestamp}_backup.{index}.log.gz
 - Timestamp : timestamp de la rotation (date génération du fichier archive)
 - Index : numéro ordre du fichier archive



Bonnes pratiques

- L'objet Exception (catch) doit être passé en paramètre de la méthode de logging

```
String s = "Hello world";
try {
    Integer i = Integer.valueOf(s);
} catch (NumberFormatException e) {
    logger.error("Failed to format {}", s, e);
}
```

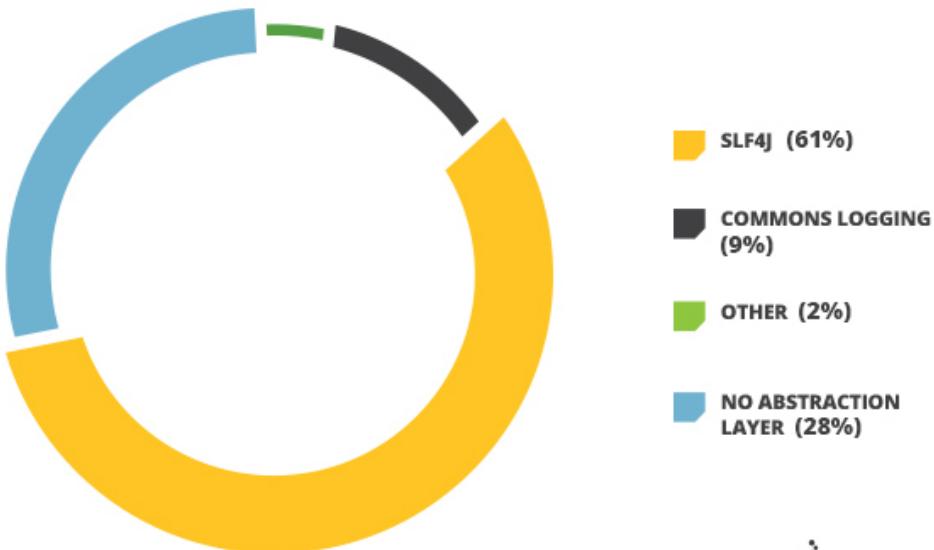
- Conserver un nombre raisonnable d'archives de log
- L'emplacement des fichiers logs doit être extérieur à votre package
- Structurez la log à minima comme suit :

- {Horodatage}\t{Level}\t{IP}\t{CorrelationID}\t{UserID}\t{Localisation}\t{Message}\t{ErrorCode}
 - Séparateur : \t : tabulation (lisible et léger). Exige le remplacement des \t présent dans les textes loggés
 - 1. Horodatage : ISO 8601 incluant le fuseau horaire (yyyy-MM-dd HH:mm:ss,SSSZ). Ex: 2017-06-30 10:04:08,207+0200
 - 2. Level : T:TRACE, D:DEBUG, I:INFO, W:WARN, E:ERROR
 - 3. IP : IP de l'utilisateur
 - 4. CorrelationID : UUID (RFC 4122)
 - 5. UserID : Identifiant de l'utilisateur. Dépend de ce que manipule l'application (alias, identifiant télématique, identifiant RP)
 - 6. Localisation : localisation dans le programme (Class Method Line)
 - 7. Message : Message simple et explicite. Il doit être unique au sein d'une application afin de faciliter l'analyse
 - 8. ErrorCode : un code unique pour l'erreur si erreur
- D'autres données peuvent être intégrées si jugées utiles à l'application



SLF4J et les autres

ABSTRACTION LAYER USAGE



 REBELLABS



BNP PARIBAS

La banque d'un monde qui change



NOUVELLE API DATE ET TIME



API Date et Time

- Nouvelle date et heure API - JSR 310
- Couvre les dates, les heures, les instants, les périodes, les durées
- Apporte 80% + de Joda-Time au JDK
- Corrige les erreurs dans Joda-Time



- Deux conceptions du temps :
 - Temps machine (entier augmentant depuis le 1^{er} janvier 1970)
 - Temps humain. (plusieurs champs, jour –mois –année,...)
- 4 principes architecturaux :
 - ImmuTable et Thread Safe
 - Chaînable
 - Clarté
 - Extensibilité



API Date et Time

Class	Date	Time	ZoneOffset	Zoneld	Example
LocalDate	✓	✗	✗	✗	2015-12-03
LocalTime	✗	✓	✗	✗	11:30
LocalDateTime	✓	✓	✗	✗	2015-12-03T11:30
OffsetDateTime	✓	✓	✓	✗	2015-12-03T11:30+01:00
ZonedDateTime	✓	✓	✓	✓	2015-12-03T11:30+01:00 [Europe/London]
Instant	✗	✗	✗	✗	123456789 nanos from 1970-01-01T00:00Z



java.time.Instant: Temps Machine

- Instant.EPOCH → 1970-01-01T00:00:00Z
- Instant.MIN → -1000000000-01-01T00:00:00Z
- Instant.MAX → +1000000000-12-31T23:59:59.999999999Z
- Instant.now(); → 2014-01-15T20:40:27.093Z
- D'autres méthodes (isAfter(),isBefore(),parse(),...)



java.time: Temps Humain

- LocalDate.MIN// -999999999-01-01
- LocalDate.MAX// +999999999-12-31
- LocalDate.now() // 2014-01-15
- LocalTime.now() // 22:09:46.553
- LocalTime.NOON// 12:00
- LocalTime.MIDNIGHT// 00:00
- LocalDateTime.now() // 2014-01-15T22:10:49.852
- D'autres méthodes (minusX, plusX, parse, getMonth,...)



java.time.Duration: Temps Machine

- Duration.ZERO//représente 0.
- .plusDays(long x) // ajoute x jours à la durée (plusHours, plusNanos,...)
- .minusDays(long x) // retire x jours à la durée (minusHours, minusNanos,...)



Enumérations

- Java 8 ajoute plusieurs énumérations, tels que **java.time.temporal.ChronoUnit** pour exprimer des choses comme "jours" et "heures" au lieu des constantes entières utilisées dans l'API Calendrier. Par exemple:
 - LocalDate today = LocalDate.now();
 - LocalDate nextWeek = today.plus(1, ChronoUnit.WEEKS);
 - LocalDate nextMonth = today.plus(1, ChronoUnit.MONTHS);
 - LocalDate nextYear = today.plus(1, ChronoUnit.YEARS);
 - LocalDate nextDecade = today.plus(1, ChronoUnit.DECADES);
- Il y a aussi les énumérations `java.time.DayOfWeek` et `java.time.Month`.



Clock

- L'horloge peut être utilisée en conjonction avec les dates et les heures pour aider à construire vos tests.
- Pendant la production, une horloge normale peut être utilisée et une autre pendant les tests.
- Pour obtenir l'horloge par défaut, utilisez ce qui suit:
 - `Clock.systemDefaultZone();`
- L'horloge peut ensuite passer dans les méthodes d'usine par exemple:
 - `LocalTime time=LocalTime.now();`



Période et Durée 1/2

- Java 8 a deux types pour représenter les différences de temps à mesure que les humains les comprennent, Période et Durée.
 - La période est un mesure de temps basée sur la date, tel que «2 ans, 3 mois et 4 jours».
 - La durée est une mesure de temps en secondes ou nanosecondes, tel que '34 .5 secondes'.
- Les périodes et les durées peuvent être déterminés en utilisant les méthodes between
 - `Period p=Period.between(date1, date2);`
 - `Duration d=Duration.between(time1,time2);`



Période et Durée 2/2

- Ils peuvent également être créés à l'aide de méthodes statiques. Par exemple, des Durations peuvent être créées pour toute quantité de secondes, minutes, heures ou jours

- Duration twoHours=Duration.ofHours(2);
- Duration tenMinutes=Duration.ofMinutes(10);
- Duration thirtySecs=Duration.ofSeconds(30);

- Périodes et Durations peuvent être ajoutés ou soustraits à partir de types de données Java 8. Par exemple:

- LocalTime t2 = time.plus(twoHours);



Temporal Adjusters

- Un TemporalAdjuster peut être utilisé pour accomplir une tâche complexe traitant de la date qui est populaire dans les applications métier.
- Par exemple, ils peuvent être utilisés pour trouver le «premier lundi du mois» ou «mardi prochain».
- La classe `java.time.temporal.TemporalAdjusters` contient une multitude de méthodes utiles pour créer des TemporalAdjusters.



Temporal Adjusters - EXEMPLES

- En voici quelques uns:
 - firstDayOfMonth()
 - firstDayOfNextMonth()
 - firstInMonth(DayOfWeek)
 - lastDayOfMonth()
 - next(DayOfWeek)
 - nextOrSame(DayOfWeek)
 - previous(DayOfWeek)
 - previousOrSame(DayOfWeek)
- Pour utiliser un TemporalAdjuster, utilisez la méthode with. Cette méthode renvoie une copie ajustée de l'objet date-heure ou date.
- Par exemple:
 - import static java.time.temporal.TemporalAdjusters.*;
 - LocalDate
 - nextTuesday=LocalDate.now().with(next(DayOfWeek.TUESDAY));



TimeZones

- Les zones horaires sont représentées par la classe `java.time.ZoneId`.
- Il existe deux types de ZoneIds, des décalages fixes et des régions géographiques. C'est pour compenser des choses comme "l'heure d'été" qui peut être très complexe.
- Nous pouvons obtenir une instance d'un `ZoneId` de plusieurs façons, y compris les deux suivants:
 - `ZoneId mountainTime = ZoneId.of ("America / Denver") ;`
 - `ZoneId myZone = ZoneId.systemDefault () ;`
- Pour imprimer toutes les ID disponibles, utilisez `getAvailableZoneIds ()`;
 - `System.out.println (ZoneId.getAvailableZoneIds ()) ;`





DateTime - TP1

- La classe `CalculDuration`, affiche la durée globale d' exécution en millisecondes
- Convertir le calcul de la durée d'exécution de la classe en passant par les nouvelles API Java 8
 - Instant
 - Duration



DateTime - TP2

- En utilisant l'API `LocalDate`
 - Créer un package `com.training.datetime.tp2`
 - Créer une classe `LocalDateUtils`
 - Afficher la date du jour: `today`
 - Afficher le jour du mois courant
 - Afficher le nombre de jours du mois
 - Ajouter 1 jours à la date courante puis afficher le résultat: `tomorrow`
 - Ajouter une décennie à la date courante puis afficher le résultat: `nextDecade`
 - Calculer la période entre les deux derniers résultats: `nextDecade et tomorrow`



DateTime - TP3

- En utilisant l'API `DateTimeFormatter`
 - Créer un package `com.training.datetime.tp3`
 - Créer une classe `DateTimeFormatterSample`
 - Afficher la date du jour en utilisant le pattern: `dd MMMM , yyyy HH:mm:ss`
 - Afficher la date du jour en passant par l'énumération `FormatStyle.SHORT`



DateTime - TP4

- Nous considérons une application de gestion de temps, qui récolte les informations suivantes:
 - Date et heure d'arrivée le matin (27/04/17 à 09:29:30)
 - Date et heure de départ le soir (27/04/17 à 18:25:24)
 - Date et heure de début déjeuner (27/04/17 à 12:30:30)
 - Date et heure de fin de déjeuner (27/04/17 à 13:20:14)
- La Durée de pause est de 20 min sur la journée
- Créer un package com.training.datetime.tp4
- Créer une classe CalculDureeTravail
- En utilisant l'API Date & Time, calculer la durée effective travaillée qui se définit comme suit:
 - dureeGlobale - dureeDejeuner - dureePause
 - Avec:
 - dureeGlobale = Date et heure de départ le soir - Date et heure d'arrivée le matin
 - dureeDejeuner = Date et heure de fin de déjeuner - Date et heure de début déjeuner



REX - Date & Time

- Enfin une API riche et complexe
- Thread-safe car classes immuables
- Bonnes pratiques :
 - Ne plus utiliser Date/Calendar ni Joda Time
 - Bien choisir le type à utiliser selon les données temporelles requises
 - Lors de la déclaration de variables, ne pas utiliser les interfaces (ex : Temporal), mais directement les classes (ex : DateTime)
 - Utilisation des TemporalAdjuster
 - Classe Clock
 - Facilite les tests automatisés
 - Par défaut, renvoie la date/heure système
 - Pour les tests, injecter une instance obtenue par `Clock.fixed()`. Permet de fixer le temps pour avoir des tests reproductibles
- Utiliser les types suivants:
 - LocalDate, LocalTime, ZonedDateTime, Instant
- Pour les échanges réseaux sous format XML/JSON passer par les types:
 - OffsetTime, OffsetDateTime



○ JAVA 8 NASHORN



Historique de Nashorn

- Nashorn, prononcé «nass-horn», est l'allemand pour «rhinocéros».
- C'est aussi le nom du remplacement - introduit avec Java 8 - pour l'ancien moteur Rhino JavaScript. Rhino et Nashorn sont des implémentations du langage JavaScript écrit pour s'exécuter sur la machine virtuelle Java ou JVM.
- Conception et implémentation d'une nouvelle mise en œuvre légère et performante de JavaScript, et intégration dans le JDK.
- Le nouveau moteur est mis à la disposition des applications Java via l'API javax.script existante, et plus généralement via un nouvel outil de ligne de commande.



Nashorn

- Remplacement du moteur JavaScript, intégré dans Java 7 de Mozilla Rhino
- Conformité aux spécifications linguistiques 5.1 ECMAScript-262 Edition 5.1
- Collaboration entre Oracle, IBM et RedHat
- Utilise largement l'invocation dynamique
- 20x plus rapide que Rhino
- Beaucoup plus petit - peut fonctionner sur des périphériques intégrés
- Open Source
- Page de projet:
 - <http://openjdk.java.net/projects/nashorn/>



- Outil de commande jjs et jrungscript
- Nashorn peut être utilisé à partir du code Java
- Exécuter Java à partir de JavaScript
- Exécuter JavaFX avec Nashorn
- Mode standalone ou bien encapsulé dans une application Java



Nashorn: Moteur JavaScript

- Moteur léger et performant à haute performance
 - Intégré dans JRE
- Utilisez l'API **javax.script** (JSR 223)
- Nouvel outil de ligne de commande, jjs pour exécuter JavaScript
- Messages d'erreur et documentation internationalisés



JavaScript depuis Java

- Nashorn prend en charge l'invocation des fonctions JavaScript définies dans nos fichiers de script directement à partir du code java.
- Passer les objets java en tant qu'archives de la fonction et renvoyer les données de la fonction à la méthode java appelante.

```
ScriptEngineManager m = new ScriptEngineManager();
ScriptEngine nashorn = m.getEngineByName("nashorn");
try {
    nashorn.eval("print('Hello, world')");
} catch (ScriptException e) {
}
```



Java depuis JavaScript

- L'invocation de méthodes java à partir de javascript est assez simple. Nous définissons d'abord une méthode java statique :

```
static String fun1(String name) {  
    System.out.format("Hi there from Java, %s", name);  
    return "greetings from java";  
}
```

- Les classes Java peuvent être référencées à partir de javascript via l'extension API Java.type:

```
var MyJavaClass = Java.type('my.package.MyJavaClass');  
var result = MyJavaClass.fun1('John Doe');  
print(result);
```

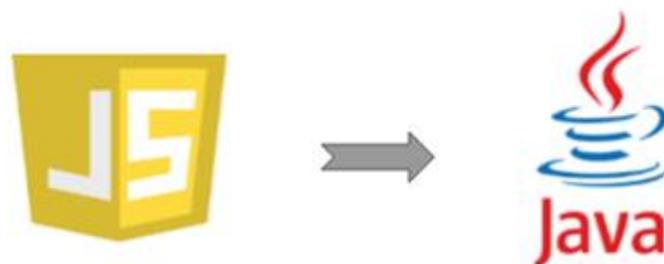


L'outils jjs

- Moteur javascript Nashorn
- Commande jjs en ligne de commande
- Se trouve sous \bin
- Interpréteur encapsulable dans les applications Java



JavaScript in Java



- En vous inspirant de la classe ScriptEngineDemo
 - Créer un package fr.ib.training.nashorn.tp1
 - Créer une classe ScriptEngineSample
 - Exécuter les fichiers JavaScript suivant en utilisant le moteur Nashorn
 - test1.js
 - test2.js
 - stats.js





LES AUTRES ÉVOLUTIONS DE JAVA 8



Rappel sur la notion de Multi-tâches

- Multi-tâches : exécution de plusieurs processus simultanément.
 - Un processus est un programme en cours d'exécution.
 - Le système d'exploitation distribue le temps CPU entre les processus
- Un processus peut être dans différents états.
 - En exécution (running) : il utilise le processeur
 - Prêt : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en exécution)
 - Bloqué

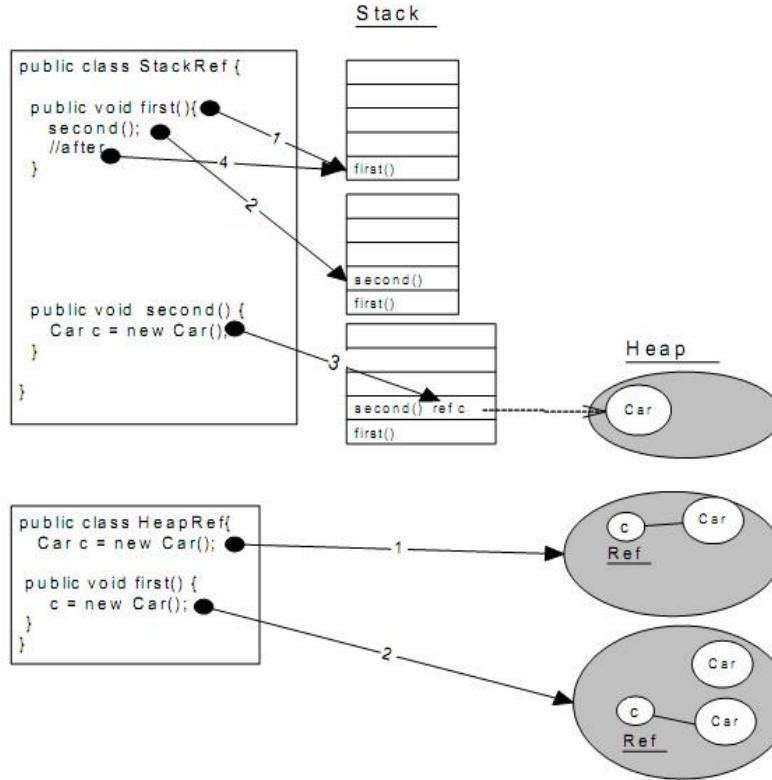


Rappel sur la notion de Parallélisme

- Parallélisme : pouvoir faire exécuter plusieurs tâches à un ordinateur avec plusieurs processeurs.
- Si l'ordinateur possède moins de processeurs que de processus à exécuter :
 - division du temps d'utilisation du processeur en tranches de temps (time slice en anglais)
 - attribution des tranches de temps à chacune des tâches de façon telle qu'on ait l'impression que les tâches se déroulent en parallèle.
- Les systèmes d'exploitation modernes gèrent le muti-tâches et le parallélisme



Heap (Tas) Vs Stack (Pile)

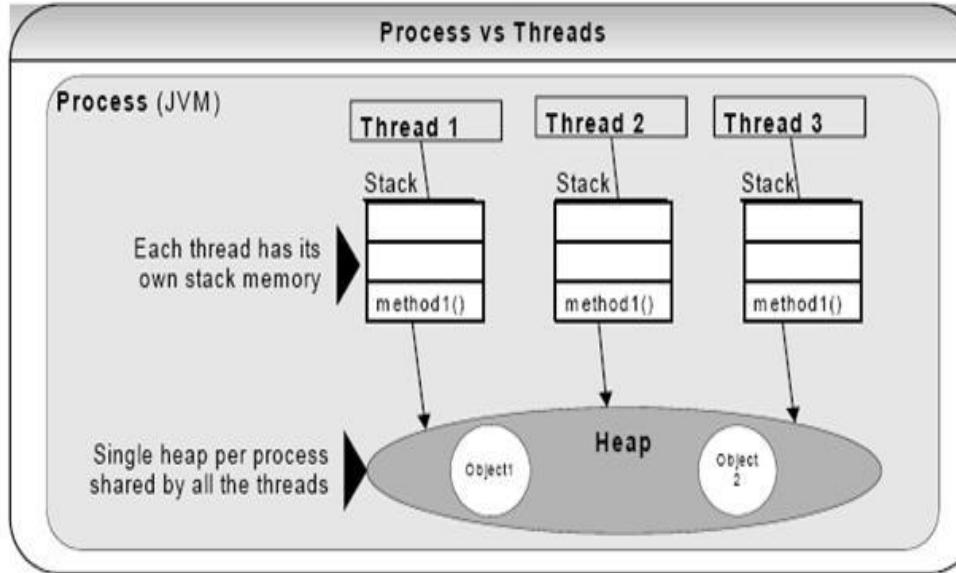


Gestion de la mémoire

- Le plus grand différence entre le Heap et le Stack est
 - Stack utilise pour le stockage des variables locales et les appels de méthode
 - Heap stocke les objets Java. Tous les objets sont créés dans le Heap
 - Le paramétrage Heap de la JVM peut être spécifié avec :
 - Xms taille de démarrage
 - Xmx taille maximale
- Chaque thread à son propre Stack.
 - Cela peut être spécifié avec la variable XSS de la JVM.
- S'il n'y a plus assez d'espace en stack, on aura une erreur JVM:
`java.lang.StackOverflowError` (boucle sur la même méthode)
- S'il n'y a plus de heap pour créer les objets, on aura l'erreur:
`Java.lang.OutOfMemoryError`
- Les variables dans le Stack sont visibles uniquement par le thread propriétaire uniquement.



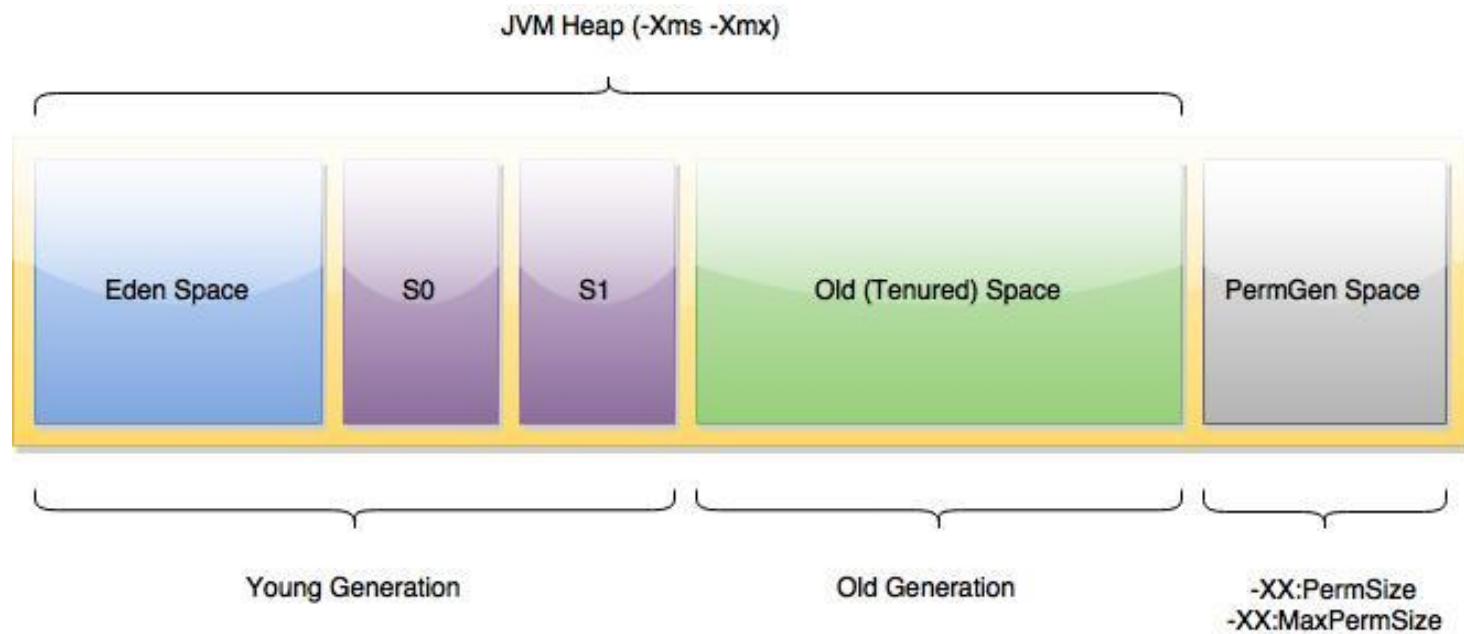
Process Vs Thread



- Les variables dans le Stack sont visibles uniquement par le thread propriétaire uniquement.
- Cependant les objets créés dans le Heap sont visible par tous les thread.
- Autrement dit, la mémoire stack est espace privé de la mémoire Java, alors que le heap est partagé par tous les thread.



La gestion de mémoire avant Java 8

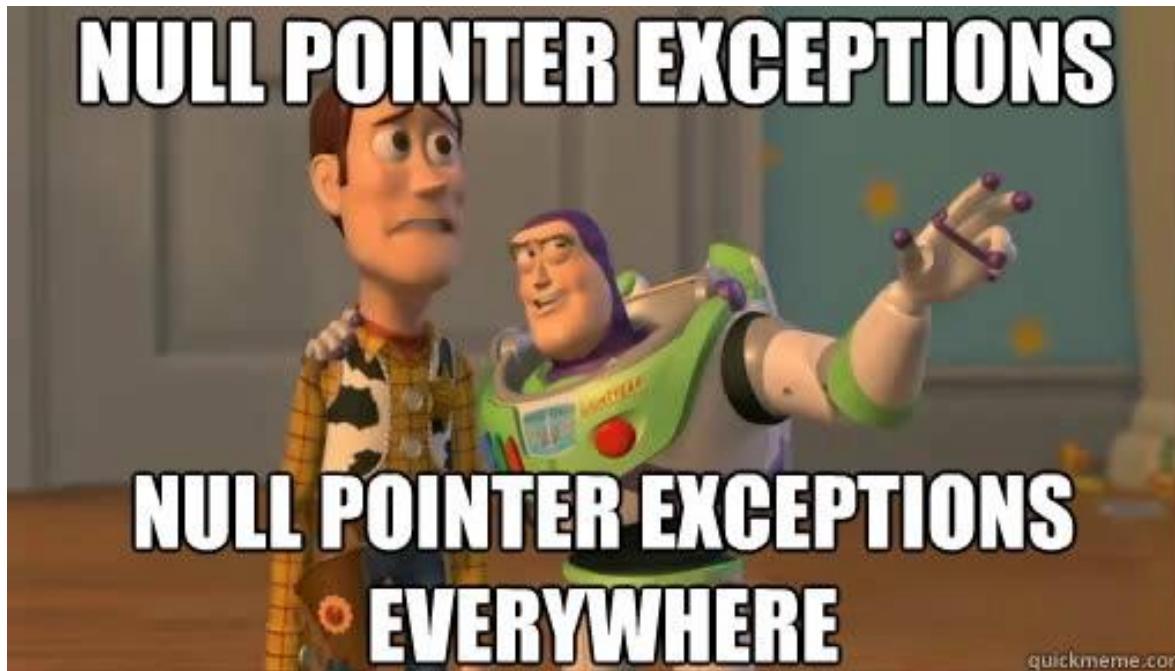


Suppression de la Génération permanente (PermGen)

- En tant que développeur Java, il est très probable que vous rencontriez une erreur Java de l'espace de génération permanente:
`java.lang.OutOfMemoryError: Espace PermGen.`
- Dans les versions HotSpot JDK 1.7 et anciennes, nous avons besoin d'un dimensionnement et d'un réglage appropriés de cet espace mémoire particulier.
- JDK 8 HotSpot JVM utilise maintenant la mémoire native pour la représentation des métadonnées de classe appelées Metaspace. Ce JDK par défaut élimine cette exigence d'accord.



Optional = Null pointer exception killer



Mot clé Optional

- `Optional` un utilitaire rusé pour empêcher `NullPointerException`.
- `Optional` est un conteneur simple pour une valeur qui peut être nulle ou non nulle.
- Pensez à une méthode qui peut renvoyer un résultat non nul, mais parfois ne renvoie rien. Au lieu de renvoyer, vous renvoyez un optionnel en Java 8.



Mot clé Optional

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true
optional.get();                // "bam"
optional.orElse("fallback");   // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));
```



Mot clé Optional

- une méthode retourne une valeur ou reçoit un argument ;
- dans certains cas, on veut dire que cette valeur peut être absente ;
- solution usuelle : renvoyer (ou passer) null.

Problème

Ça n'est pas explicite. Du coup, le programmeur se méfie de tout argument ou toute valeur retournée de type objet.

- Optional permet de le rendre explicite ;
- ça ne résoud que partiellement le problème (il est au niveau du langage lui-même) ;



Mot clé Optional

Construction

```
Optional<Integer> a= Optional.of(3);  
Optional<Integer> b= Optional.empty();
```

méthodes

- `isPresent()` : renvoie vrai si on a une valeur ;
- `filter/map` : comme pour un stream ;
- `orElse(autreValeur)` : la valeur (si elle est là), sinon une valeur par défaut ;
- `ifPresent(Consumer consumer)` : prend comme argument une fonction qui fera quelque chose avec notre élément.
- `get()` : renvoie la valeur (ou lève une exception).



Les Optionals...

Expriment clairement les types de retour

Forcent le développeur à se poser les bonnes questions

Améliorent l'expressivité du code

Plus de NullPointerException@Runtime

Plus de code plomberie pour vérifier les null



TP Optional



TP Optional

- En utilisant l'API Optional
 - Le programme PossibleNullAsIs, affiche le nom de rue d'une entreprise
 - On souhaite rendre la donnée nom de rue Optionnelle
 - Modifier la classe Address, en passant par le mot clé Optional
 - Adapter la classe PossibleNullAsIs , pour gérer le cas où le nom de rue n'est pas renseigné



REX - Optional

- Classe qui encapsule une valeur ou l'absence de valeur.
- L'utilisation d'Optional rend le code plus robuste (au détriment de la performance).
- Limitations : classe finale et pas Serializable
- Sujet le plus controversé alors qu'il ne s'agit que d'une classe.
- Une variable de type Optional ne doit **jamais être null**



REX - Optional

- Utilisation comme valeur de retour :
 - Nécessaire dans certaines circonstances
 - A éviter dans les getters de bean
- Utilisation dans les paramètres :
 - Pas recommandé car pollue la signature
 - Plus complexe pour l'appelant.
 - Contournement : possibilité d'utilisé la surcharge de méthode pour éviter de passer des paramètres null.



REX - Optional

- Utilisation comme variable d'instance :
 - A éviter
 - Support par certains frameworks
- Utilisation comme variable locale :
 - Jamais
- Bonne pratiques :
 - Passer par une fabriques of(), ofNullable(), empty()
 - Essayer de limiter le caractère optionnel d'une valeur
 - Pour les primitifs : OptionallInt, OptionLong ...
 - Attention à l'utilisation de la méthode get() => NoSuchElementException si pas de valeur.
 - Utilisation de orElse() pour passer une valeur par défaut.
 - Eviter d'utiliser Optional avec une collection ou un tableau => utiliser une collection ou un tableau vide avec isEmpty() ou length()



Java.util.Optional

Où et quand utiliser les optionals ?

- Comme un champ d'un bean ou paramètre de méthode
 - La javadoc stipule qu'un Optional ne doit pas être utilisé comme champ
- Comme type de retour de méthode
- Dans une optique de sérialisation
 - Optional n'est pas Sérializable
- Pour éviter les NullPointerException



REX - Optional

Où et quand utiliser les Optionals ?

- Utiliser partout
- Utilisez au lieu de null sur les API publiques, les entrées et sorties
- Utiliser dans quelques endroits sélectionnés
- Utiliser au lieu de null sur les types de retour publics
- Ne pas utiliser



L'outil jdeps

- Un outil pour analyser les dépendances entre les classes
- Se lance sur une classe et recherche les dépendances à partir de celle-ci
- Exemple d'options disponibles
 - jdeps -P Deps.class -> trouver les profiles
 - jdeps -v Deps.class -> Récursif



L'outil javac -h

- Génère les entêtes natifs
- Plus besoin de passer par javah
- Les entêtes sont générés pour toute classe qui possède des méthodes natives ou qui utilise l'annotation Native



Profiles compact

Full SE API	Beans	JNI	JAX-WS
	Preferences	Accessibility	IDL
	RMI-IIOP	CORBA	Print Service
	Sound	Swing	Java 2D
	AWT	Drag and Drop	Input Methods
	Image I/O		
compact3	Security ¹	JMX	
	XML JAXP ²	Management	Instrumentation
compact2	JDBC	RMI	XML JAXP
compact1	Core (java.lang.*)	Security	Serialization
	Networking	Ref Objects	Regular Expressions
	Date and Time	Input/Output	Collections
	Logging	Concurrency	Reflection
	JAR	ZIP	Versioning
	Internationalization	JNDI	Override Mechanism
	Extension Mechanism	Scripting	

1. Adds kerberos, acl, and sasl to compact1 Security.
2. Adds crypto to compact2 XML JAXP.

SUMMARY: NESTED | FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.util.function

Interface Function<T,R>

Type Parameters:

T the type of the input to the function
R the type of the output from the function

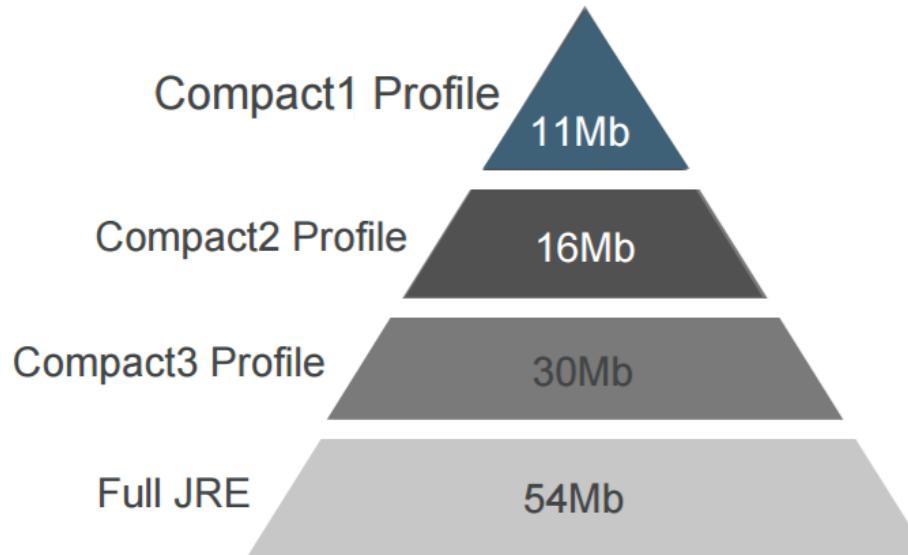


Profiles compact

- Sous-ensembles de l'API Java SE pour réduire l'utilisation de la mémoire
 - Surtout pour les systèmes embarqués
- Trois profils: compact1 < compact2 < compact3 < API complète
- Support d'outils
 - `javac -profile [profile]` → Valide que seule l'API sur le profil donné a été utilisée
 - `jdeps -P` → Liste des exigences de profil utilisées par les cours compilés



Evaluation de la taille de chaque profil



Parameter Names (JEP 118)

- Les noms de paramètres pour les paramètres de méthode / constructeur peuvent être récupérés à l'exécution par réflexion

```
Method method = ...  
Parameter param =  
method.getParameters()[0];  
System.out.println(param.getName());
```



Base64 Encoding/Decoding (JEP 135)

- Avant, le développeur passait par des API non Public:
 - sun.misc.BASE64Encoder
 - sun.misc.BASE64Decoder
- Avec Java 8, on passe par une API standard pour codage et décodage Base64:
 - **java.util.Base64.Encoder**
 - **java.util.Base64.Decoder**
 - encode, encodeToString, decode, wrap methods



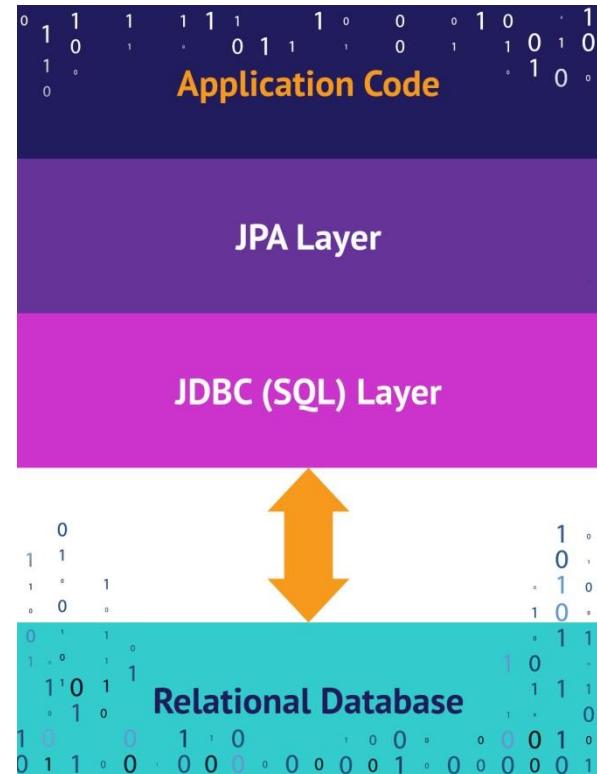


LA PERSISTANCE AVEC HIBERNATE

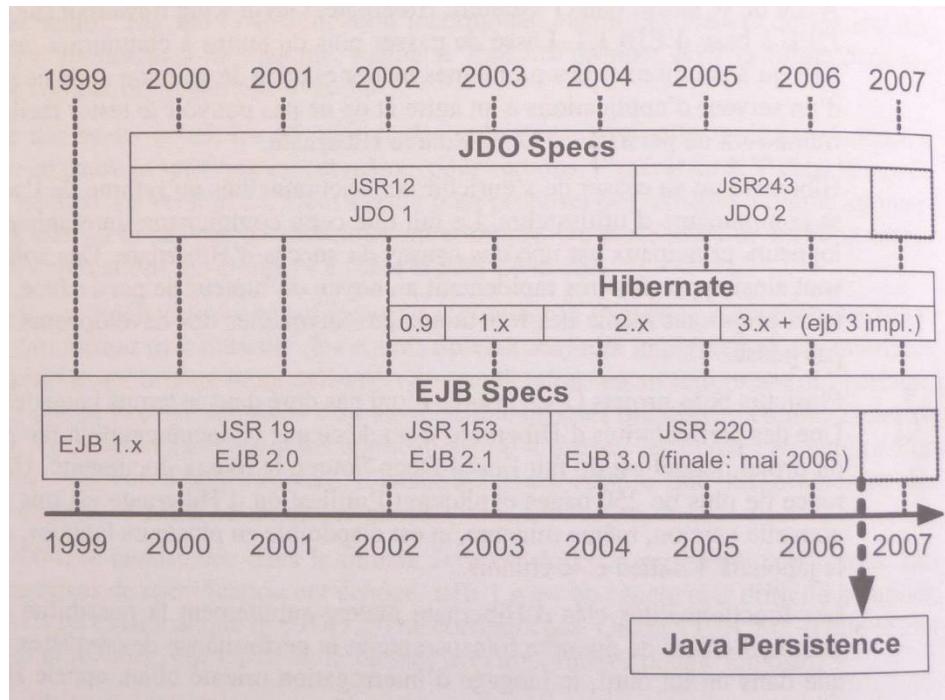


Objectif des outils d'Object Relational Mapping (ORM)

- Productivité, maintenance pour permettre au code métier d'extraire et mettre à jour des données d'un SGBD.
- Fonctions usuelles :
 - génération du code SQL
 - langage de manipulation objet
 - peuplement des données extraites (le ResultSet devient invisible) en grappe
 - cache



Historique



- La JSR 317 sur JPA 2.0 liée à la spécification EJB 3.1 (2009) pourrait bien faire un grand pas permettant une large adoption.
- JPA s'utilise de manière libre avec ou sans la couche EJB3.x.
- JSR 317 : Java Persistence 2.0 : 10 décembre 2009
- JSR 318: Enterprise JavaBeans 3.1 : 10 décembre 2009
- La JSR 317 complète les manques de JPA 1.0, en ajoutant certaines fonctionnalités déjà disponibles avec Hibernate 3 mais non reprises en JPA1.0 faute de temps :
 - API Criteria,
 - UserType
 - Query Language amélioré
 - accès à l'état d'une entité
 - evict()
 - Bulk update, bulk delete



Pourquoi avons-nous besoin d'Hibernate?

● Les avantages d'Hibernate

- réduit le code par 3
- propre
- API souple pour faire simple, ou riche pour faire complexe
- fonctions périphériques : cache, pool ...
- outils de génération accélérant encore la prise en main
- standard de fait
- stable
- bref ... incontournable

● Quelques exemples de code JDBC standard

● Les mêmes exemples en Hibernate



Les Utilisations d' Hibernate

- On peut utiliser Hibernate de 3 manières différentes :
 - Hibernate natif avec configuration XML
 - Hibernate natif avec configuration par annotations JPA
 - JPA + annotations + Hibernate

JPA	Hibernate	JDBC
EntityManager (em)	Session	Connexion
EntityManagerFactory	SessionFactory	DataSource
API JPA sur em	API Hibernate sur la session	API SQL sur la connection



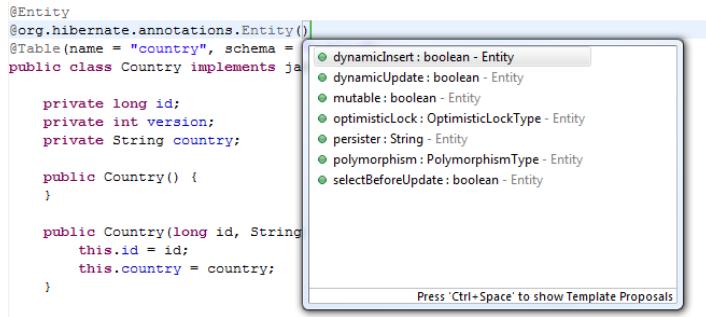
Mapping par annotations

- Les annotations sont standards et proviennent de javax.persistence

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Version;

@Entity
@Table(name = "country", schema = "public")
public class Country implements java.io.Serializable {
```

- Hibernate fournit une annotation @Entity à ajouter en complément si nécessaire (deprecated en version 4)



Mapping par annotations

- @GeneratedValue permet de générer la valeur de la clé primaire

- cas le plus simple (JPA) :

```
@Id  
@GeneratedValue(strategy = GenerationType.AUTO)  
private int id;
```

- cas standard (JPA):

```
@Id  
@Column(name = "id", unique = true, nullable = false)  
@GeneratedValue(strategy = GenerationType.TABLE, generator = "countryGen")  
@TableGenerator(name = "countryGen",  
               table = "ID_GENERATOR",  
               pkColumnName = "GEN_KEY",  
               valueColumnName = "GEN_VALUE",  
               pkColumnValue = "COUNTRY_ID",  
               allocationSize = 50)  
public long getId() {  
    return this.id;  
}
```

- cas spécifique :

```
@GenericGenerator(strategy = "org.hibernate.id.enhanced.TableGenerator", name = "countryGen", parameters = {  
    @Parameter(name = "table_name", value = "ID_GENERATOR"),  
    @Parameter(name = "value_column_name", value = "GEN_VALUE"),  
    @Parameter(name = "segment_column_name", value = "GEN_ID"),  
    @Parameter(name = "segment_value", value = "COUNTRY_ID"),  
    @Parameter(name = "increment_size", value = "50"), @Parameter(name = "optimizer", value = "pooled") })  
@Id  
@Column(name = "id", unique = true, nullable = false)  
@GeneratedValue(generator = "countryGen")
```

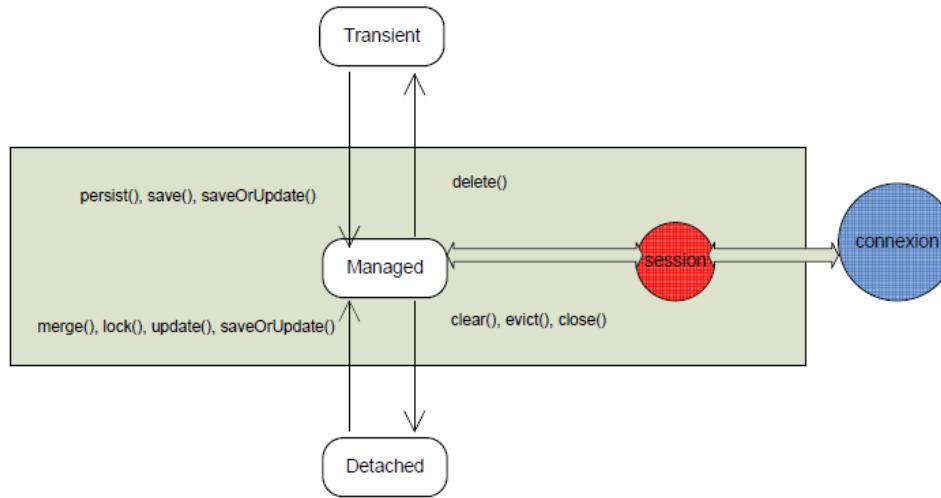


Mapping par annotations

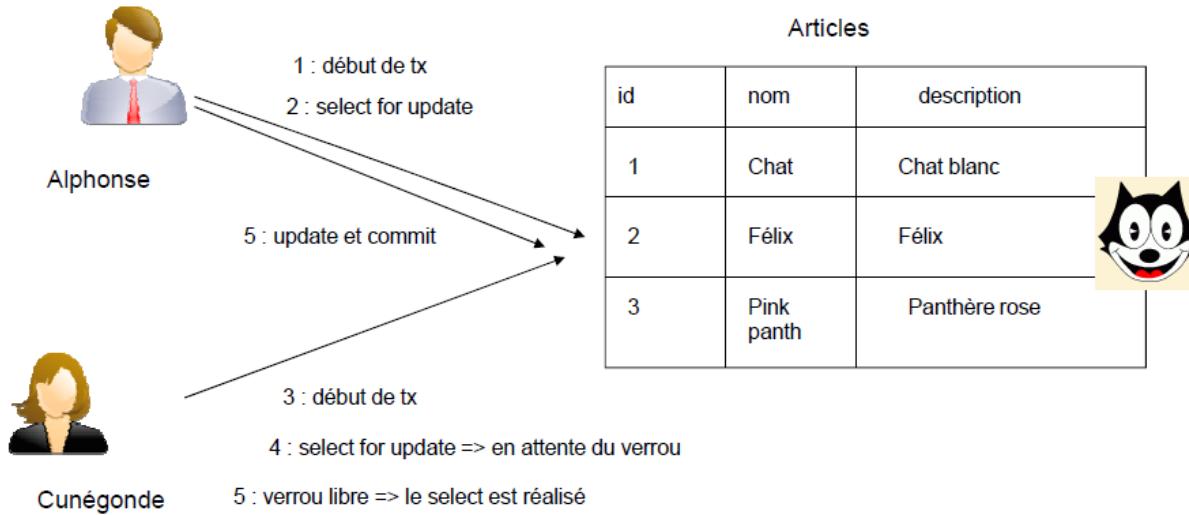
- Annotations des propriétés
 - @Basic (optionnel)
 - @Lob
 - @Temporal (TemporalType.DATE ou TIME ou TIMESTAMP)
 - @Enumerated (ORDINAL ou STRING) par défaut ORDINAL
 - @Version
- Propriété non persistante :
 - @Transient
- Note : c'est la position de @Id qui décide si Hibernate accède les valeurs par les getter/setter (mode property) ou par réflexion sur l'attribut (mode field)



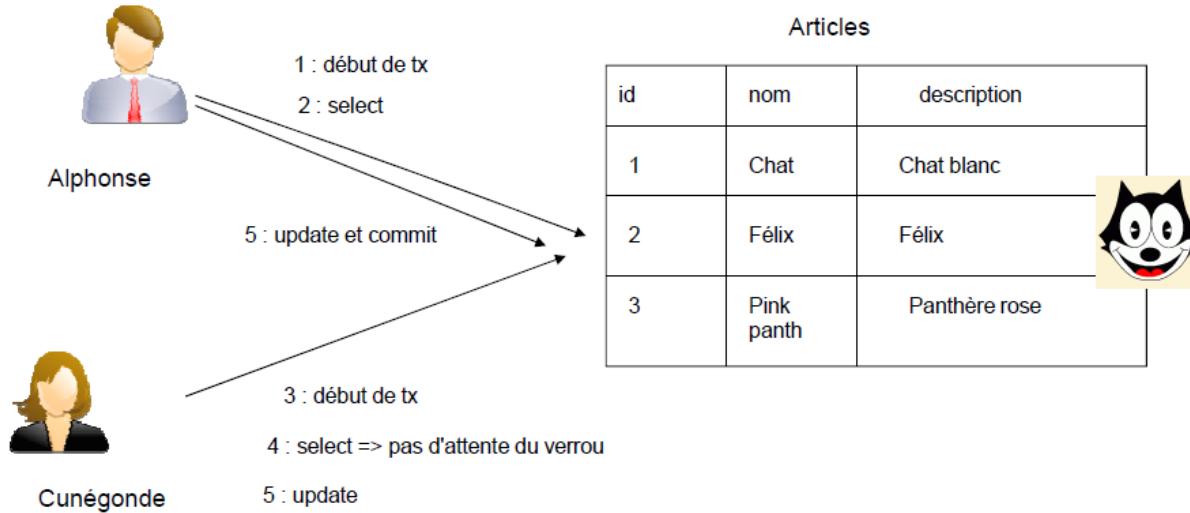
Cycle de vie des entités



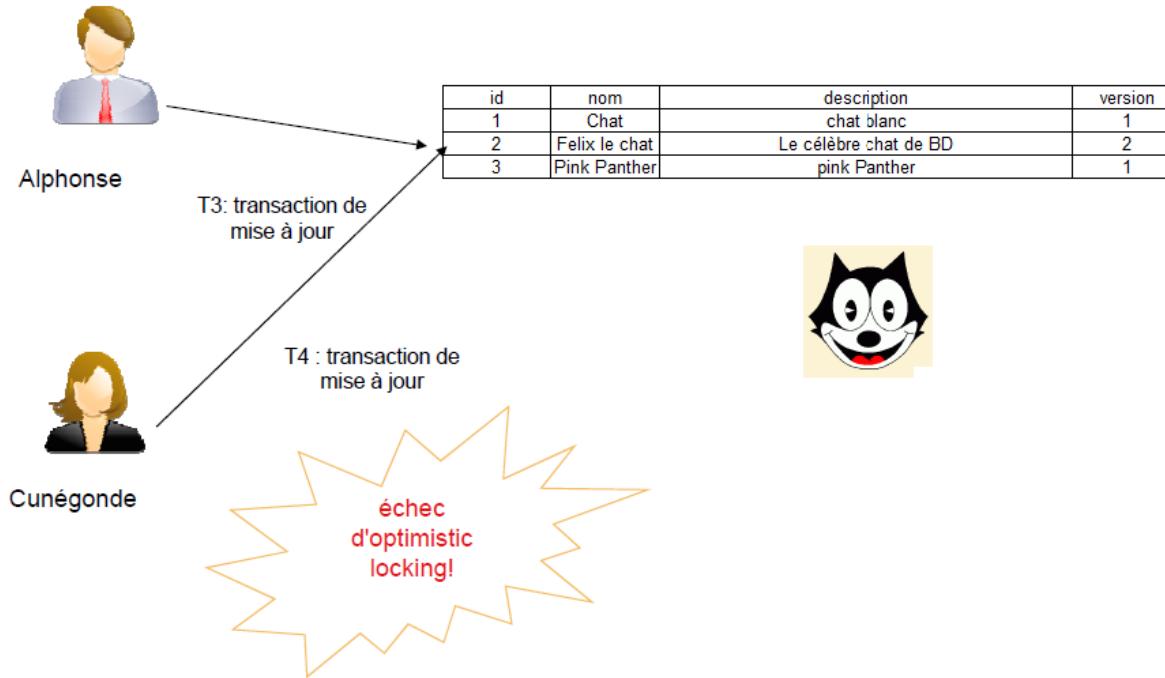
Verrous pessimistes : select for update



Sans verrou



Verrou optimiste : Optimist Lock



Les relations

- Les relations entre entités sont définies par les annotations de JPA:

Annotation	Relation
@OneToOne	1-1
@OneToMany	1-n
@ManyToOne	n-1
@ManyToMany	n-n

- Les relations sont définies par le type de cascade pour contrôler la persistence:
- PERSIST: quand une nouvelle entité est insérée, toutes ses relations sont alors insérées
- MERGE: quand une entité détachée est mise à jour, toutes ses relations sont alors enregistrées
- REMOVE: quand une entité est supprimée, ses relations sont alors supprimées
- ALL: ce type de cascade est l'union des trois types précédents
- Exemple de déclaration dans une annotation :
`@OneToMany(CascadeType={CascadeType.MERGE, CascadeType.REMOVE})`



Les relations ManyToOne / OneToMany

```
@Entity  
@Table(name="EMP")  
public class Employee {  
  
    @Id  
    private int id;  
  
    @ManyToOne  
    @JoinColumn(name="DEPT_ID")  
    private Department d;  
  
    // getters & setters  
    ...  
}
```

```
@Entity  
public class Department {  
  
    @Id  
    private int id;  
  
    private String dname;  
  
    @OneToMany(mappedBy="d")  
    private Collection<Employee> emps;  
  
    // getters & setters  
    ...  
}
```

EMP

ID	DEPT_ID		
PK	FK		

DEPARTMENT

ID	DNAME		
PK			



Mapping many-to-one par annotations

- @ManyToOne

```
@Entity
public class City implements java.io.Serializable {

    private long id;
    private int version;
    private Country country;
    private String city;

    @Id
    public long getId() {
        return this.id;
    }

    @ManyToOne
    public Country getCountry() {
        return this.country;
    }
}
```



Mapping many-to-one par annotations

- Le nom par défaut de la colonne recevant la FK est :
 - nom de la propriété annotée + _+ nom de la clé primaire jointe
- Dans notre cas :
=> la colonne de la table CITY doit s'appeler: COUNTRY_ID car
 - le getter s'appelle get + Country
 - La clé primaire de la table COUNTRY est ID

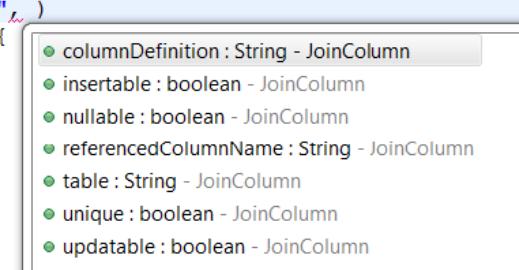
```
@ManyToOne
public Country getCountry() {
    return this.country;
}
```



Mapping many-to-one par annotations

- On peut customisé le nom de colonne de la FK avec :
 - @JoinColumn (name="FK_COLUMN")
- @JoinColumn est optionnel,
- name est optionnel, définit le nom de la colonne recevant la clé étrangère

```
@ManyToOne  
@JoinColumn(name = "country_id")  
public Country getMaCountry() {  
    return this.country;  
}
```



columnDefinition : String - JoinColumn
insertable : boolean - JoinColumn
nullable : boolean - JoinColumn
referencedColumnName : String - JoinColumn
table : String - JoinColumn
unique : boolean - JoinColumn
updatable : boolean - JoinColumn



Mapping many-to-one par annotations

- Autres paramètres du @ManyToOne

```
@ManyToOne(fetch = FetchType.LAZY, optional = false, cascade = CascadeType.ALL)
@JoinColumn(name = "country_id", nullable = false)
public Country getCountry() {
    return this.country;
}
```

- FetchType.EAGER (mode par défaut) ou FetchType.LAZY
- CascadeType. ALL | MERGE | PERSIST | REFRESH | REMOVE
 - On peut aussi utiliser l'annotation spécifique Hibernate :
org.hibernate.annotations.Cascade qui permet de cascader les ordre hibernate natifs

```
@ManyToOne(fetch = FetchType.LAZY)
@Cascade(CascadeType.SAVE_UPDATE)
public Country getCountry() {
    return this.country;
}
```



Conséquence du Mode Eager

- Si les many-to-one sont paramétrés en "standard" avec juste l'annotation @ManyToOne le mode est EAGER.
- => Hibernate sélectionne donc automatiquement l'objet du coté "un" . Mais que se passe il si cet objet lui-même à un many-to-one vers un autre objet ?
- Hibernate va récupérer aussi cet autre objet jusqu'à une profondeur paramétrable

hibernate.max_fetch_depth

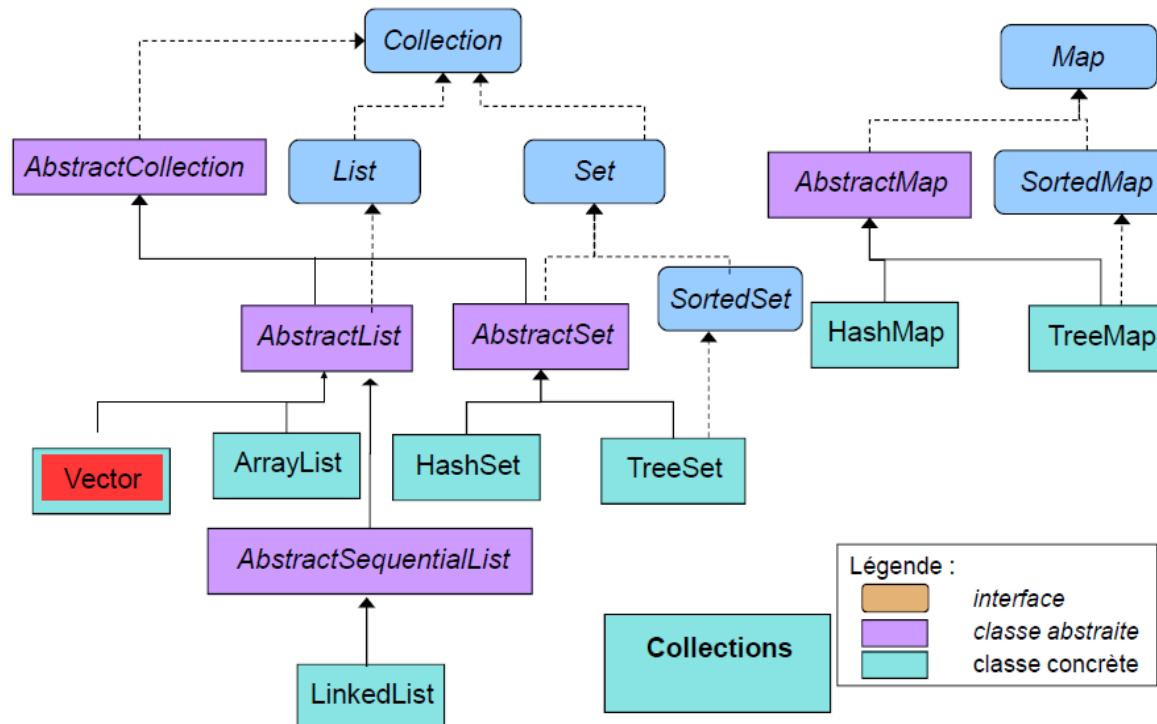
Sets a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A 0 disables default outer join fetching.

e.g. recommended values between 0 and 3

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="business.hb" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${db.hibernate.dialect}</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.cglib.use_reflection_optimizer">false</prop>
            <prop key="hibernate.query.substitutions">true 1, false 0, yes 'Y', no 'N'</prop>
            <prop key="hibernate.connection.isolation">2</prop>
            <prop key="hibernate.max_fetch_depth">1</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>
```



Schéma du framework de collection



one-to-many : création paramètre : inverse=true/false

- Si hbm.xml est utilisé, inverse="true|false" précise à Hibernate le lien à utiliser pour gérer la relation
- Si inverse="true" le lien utilisé est sur l'entité 1 (comme en SQL)



- Si inverse="false" le lien utilisé est la collection :



Synthèse pour one-to-many

- En général, la gestion du côté collection ("many") provoque un select massif de la collection quand on ajoute un élément à la collection. Seul le bag ne pose pas de problème de sélection massive (sur un simple ajout).
 - `france.getCities().add(new City("Lyon"))`
- Raison : Hibernate fait tout pour respecter la sémantique des interfaces :
 - Un Set ne doit pas contenir de doublon=> un select afin de vérifier qu'il n'y a pas de doublons en java
 - Une List est ordonnée => un select afin d'ordonner les items en java
 - Pour un bag, pas de select, car un bag peut contenir des doublons et n'est pas ordonné.



La relation OneToOne

```
@Entity
@Table(name="EMP")
public class Employee {

    @Id
    private int id;

    @OneToOne(mappedBy="P_SPACE")
    private ParkingSpace space;

    // getters & setters
    ...
}
```

```
@Entity
public class ParkingSpace {

    @Id
    private int id;

    private int lot;

    private String location;

    @OneToOne(mappedBy="space")
    private Employee emp;

    // getters & setters
    ...
}
```



EMP

ID	P_SPACE		
PK	FK		

+O

PARKINGSPACE

ID	LOT	LOCATION	
PK			

O+



La relation ManyToMany

```
@Entity
@Table(name="EMP")
public class Employee {

    @Id
    private int id;

    @JoinTable(name="EMP_PROJ",
               joinColumns=
                   @JoinColumn(name="EMP_ID"),
               inverseJoinColumns=
                   @ManyToManyColumn(name="PROJ_ID"))
    private Collection<Project> p;
}
```

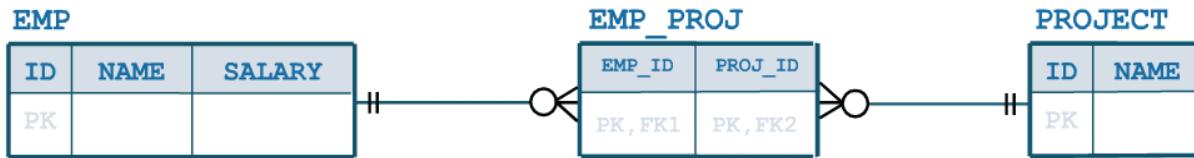
```
@Entity
public class Project {

    @Id
    private int id;

    private String name;

    @ManyToMany(mappedBy="p")
    private Collection<Employee> e;

    // getters & setters
    ...
}
```



Mapping @ pour many-to-many

- Le modèle de référence UNIDIRECTIONNEL pour le many-to-many avec annotations :

```
@ManyToMany(fetch = FetchType.LAZY)
public Set<Actor> getActors() {
    return this.actors;
}
```

- Du côté opposé :
 - soit ne pas coder de getter pour la propriété de retour (cas usuel)
 - soit, mettre @Transient pour que le getter soit ignoré
- La table de jointure doit s'appeler entiteMaitre+entiteCible SANS de "_" entre les 2 noms d'entité, sinon ajouter @JoinTable.
- Les clés de cette table seront actors_id et films_id : nomDuGetter + '_' + nomDeLaPK

```
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable( name = "film_actor", schema = "public",
            joinColumns = { @JoinColumn(name = "film_id", nullable = false, updatable = false) },
            inverseJoinColumns = { @JoinColumn(name = "actor_id", nullable = false, updatable = false) })
public Set<Actor> getActors() {
    return this.actors;
}
```



Mapping @ pour many-to-many

- Le modèle de référence bidirectionnelle pour le many-to-many avec annotations :

```
@ManyToMany(fetch = FetchType.LAZY)
public Set<Actor> getActors() {
    return this.actors;
}
```

- Du coté opposé (ne pas oublier mappedBy) :

```
@ManyToMany(fetch = FetchType.LAZY, mappedBy = "actors")
public Set<Film> getFilms() {
    return this.films;
}
```

- La table de jointure par défaut doit être appelée : entiteMaitre_entiteInverse (avec attribut "mappedBy") soit dans notre exemple : film_actor car Film est maître. Sinon ajouter @JoinTable pour customiser ce nom
- Les clés de cette table seront actors_id et films_id : nomDuGetter + '_' + nomDeLaPK



Double Mapping @ pour many-to-many

- Pour gérer simplement les liens many-to-many on peut définir une entité "virtuelle" sur la table de jointure et deux one-to-many de part et d'autres.
- le many-to-many permet de faire des jointures directes dans les requêtes HQL ou Criteria
- l'entité centrale permet d'ajouter et de détruire simplement ces liens
- Malheureusement définir 2 many-to-many de chaque côté avec mappedBy est interdit
 - On ne peut pas donc désactiver complètement la gestion par liste. Le côté ne portant pas l'attribut mappedBy (Film dans notre exemple) va être en mesure de gérer en mode liste : film1.getActors().add(unActor)



Double Mapping @ pour many-to-many

● Séquence de création de ce mapping

- Choisir le nom de la table de jointure ainsi que ceux des clés étrangères. Il y a conflit entre le nommage du one-to-many qui prend les noms au singulier (film_id) et celui du many-to-many qui les prend au pluriel (films_id) => faire son choix, les 2 solutions sont possibles. Pour ce powerpoint je choisi le singulier : film_id
- Régler le mapping many-to-many afin d'obtenir les nommages désirés :
 - Sur la classe Film

```
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(joinColumns = { @JoinColumn(name = "film_id", nullable = false, updatable = false) },
           inverseJoinColumns = { @JoinColumn(name = "actor_id", nullable = false, updatable = false) })
public Set<Actor> getActors() {
    return this.actors;
}
```

- Sur la classe Actor, l'attribut mappedBy signifie "celui qui gère la relation est de l'autre côté", l'annotations @JoinTable n'est pas acceptée, (elle serait redondante avec celle posée dans la classe Film)

```
@ManyToMany(mappedBy = "actors")
public Set<Film> getFilms() {
    return this.films;
}
```



Double Mapping en @ pour many-to-many

- Ajouter l'entité FilmActor (attention le nom de table par défaut aurait été filmactor par défaut, il faut donc bien utiliser @Table)

```
@Entity
@Table(name = "film_actor")
public class FilmActor implements java.io.Serializable {

    private FilmActorId filmActorId;

    @Id
    public FilmActorId getId() {
        return filmActorId;
    }

    public FilmActor() {
    }
```



Double Mapping en @ pour many-to-many

- La clé composite est définie dans une classe à part

```
@Embeddable
public class FilmActorId implements java.io.Serializable {

    private Film film;
    private Actor actor;

    @ManyToOne(fetch = FetchType.LAZY)
    public Film getFilm() {
        return this.film;
    }

    @ManyToOne(fetch = FetchType.LAZY)
    public Actor getActor() {
        return this.actor;
    }
}
```



Double Mapping en @ pour many-to-many

- Définir un one-to-many supplémentaire de chaque côté
 - Exemple dans Film

```
@OneToMany(mappedBy = "id.film")
public Set<FilmActor> getFilmActors() {
    return filmActors;
}
```

- Le mappedBy vaut "id.film" car
 - il faut appliquer le getter getId() qui retourne une instance de FilmActor
 - Puis appliquer getFilm() sur cette instance pour extraire le Film



Hibernate Query Language

- Les requêtes sont indépendantes de la BD
- Les requêtes HQL doivent être écrites selon la structure des classes
- Les requêtes HQL sont au niveau Objet. Elles retournent des objets
- Hibernate convertit les requêtes HQL en SQL et les envoi à la BD.
- Ne masque pas la force de SQL:
 - Jointures et produit cartésien
 - Projections
 - Agrégation (max, avg) et groupe
 - Ordre
 - Sous-requêtes



API Criteria

- Criteria API: Permet de d'exécuter des requêtes sur des objets. Il représente une alternative au HQL.
- Criteria: On se base pour cela sur:
 - Le nom du Bean
 - Ajout d'objet criterion
- Criterion: L'objet Criterion peut être créé de plusieurs façons:
- Restrictions:
 - ge() [\geq]
 - ne() [\neq]
 - lt() [$<$]
 - gt() [$>$]
 - le() [\leq]
 - like() [%]
 - and() [et]
 - or() [ou]
- Expression.sql(): On peut écrire ici des requêtes spécifiques à la base de données



Stratégie générale de mapping

- Principe 1 : Appliquer un mode lazy sur les collections
- Principe 2 : Appliquer un mode lazy les entités
- Principe 3 : Donner aux entités la gestion de leurs clés étrangères
 - # Une entité se gère elle-même ses clés (l'entité du côté "n" est maître)
 - # le coté qui détient une collection ne gère pas la clé étrangère qui est dans l'entité en vis-à-vis.
- Principe 4 : Coder les méthodes tx en gérant les entités avec un "esprit SQL"
 - # ne pas naviguer dans les collections comme si cela ne coûtait rien
 - # faire des save() ou update() explicites sur les entités
 - # initialiser les collections utilisées dès le départ, et non par navigation aléatoire
- Principe 5 : Éviter les cascades



Utilisation de cette stratégie générale

- Les conséquences du paramétrage exposé précédemment sont :
 - les entités du côté "one" sont lazy : elles ne sont jamais chargées automatiquement quelle que soit l'API de sélection utilisée (get, Criteria, HQL)
 - les collections sont lazy
 - l'ajout d'un élément dans une collection ne se fait pas en manipulant la collection :
 - france.getCities().add(myCity);
 - mais du côté "one" :
 - myCity.setCountry(france);



Installation d'un environnement de développement

- Installation de MySQL :

- Lancer mysql-installer-community-5.7.19.0.msi
 - Choisir MySQL Server et mysql-workbench
 - Mot de passe: admin
- Si besoin, il faudra lancer au préalable selon l'architecture 32/64 bits :
 - Lancer vc_redist.x86.exe ou vc_redist.x64.exe
 - Lancer vcredist_x86.exe et/ou vcredist_x64.exe
 - Créer le schéma en utilisant mysql-workbench: formationdb
- Définir ce schéma comme schéma par défaut



TP1: JPA avec hbm.xml

- Copier le projet helloworld-hibernate, dans votre workspace/jpa-tp1
- Importer le projet helloworld-hibernate sous votre IDE:
- Copier les fichiers suivants dans src/main/java:
 - Country.java
 - Main.java
- Copier les fichiers suivants dans src/main/resources:
 - Country.hbm.xml
 - hibernatepagila.cfg.xml
 - log4j.properties.
- Lancer le Main
- Vérifier la création de la table Country et l'insertion d'une ligne dans cette table.
- Lancer une 2eme exécution => résultat ?
- Solution ?



TP2: JPA avec annotations

- Créer un répertoire jpa-tp2 dans votre workspace
- Copier le projet helloworld-hibernate dans jpa-tp2
- Supprimer l'ancien projet jpa-tp1 uniquement du workspace, puis importer le nouveau projet jpa-tp2 dans votre IDE
- Nous allons reconduire le même exemple en annotations:
 - Codage en java des entités métier annotations java (Voir le slide d'après pour un exemple)
 - Codage d'un main qui insère les données en utilisant la classe Country



```
package model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Version;

@SuppressWarnings("serial")
@Entity
public class Country implements java.io.Serializable {

    private Long id;
    private int version;
    private String name;

    @Override
    public String toString() {
        return "Country [id=" + id + ", version=" + version + ", name=" + name + "]";
    }

    public Country() {
    }

    public Country(String _name) {
        super();
        name = _name;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getId() {
        return this.id;
    }

    @Version
    public int getVersion() {
        return this.version;
    }

    // @Column(length = 10, unique = true)
    @Column(length = 10)
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

TP3: JPA avec annotations

- Créer un répertoire jpa-tp3 dans workspace
- Copier le projet helloworld-hibernate du répertoire jpa-tp2 vers jpa-tp3
- Supprimer l'ancien projet jpa-tp2 uniquement du worksapce, puis importer le nouveau projet jpa-tp3 dans votre IDE
- Importer les classes fournies:
 - CountryDao
 - HibernateUtils
- Créer la classe CountryDaolmpl dans le package DAO, qui implémente CountryDao
- Implémenter les méthodes:
 - Country insert(Country _Country);
 - List<Country> findAll();
 - Country findCountryByName(String _name);
 - Country findOneById(Long _id);
 - void delete(Long _id);
 - Country update(Country _Country);
- Adapter la classe Main, afin d'invoquer le CountryDao pour effectuer:
 - L'insertion d'un pays
 - La consultation de la liste des pays en base



TP4: @OneTOMANY

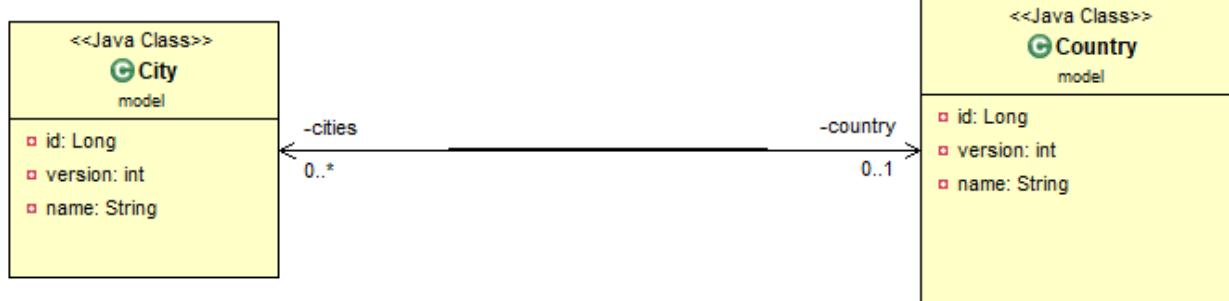
- Créer un répertoire jpa-tp4 dans workspace
- Copier le projet helloworld-hibernate du répertoire jpa-tp3 dans jpa-tp4
- Supprimer l'ancien projet jpa-tp3 uniquement du worksapce, puis importer le nouveau projet jpa-tp4 dans votre IDE
- Ajouter la dépendance suivante au niveau du pom.xml

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.3.2.Final</version>
</dependency>
```



TP4: @OneTOMANY

- Ajouter la classe City au modèle, selon de diagramme de classe suivant:



- Copier la classe CityDao dans le package dao
- Créer l'implémentation CityDaoImpl, de la classe CityDao, dans le package dao
- Implémenter toutes les méthodes
- Tester en utilisant la classe main:
 - Ajouter deux villes à un pays **Country**
 - Insérer ces deux villes
 - Lister toutes les villes du pays **Country**



CONCLUSION



En conclusion sur Java 8

- Un grand pas pour la programmation fonctionnelle, mais un peu tardivement
- Un grand manque sur les tests unitaires, pris en charge dans la version JUnit 5
- On ne peut pas déclarer lambdas comme paramètres d'entrée ou types de retour
- Les variables mentionnées dans lambdas doivent être effectivement finales
- Le code Java écrit et le bytecode compilé sont de plus en plus éloignés
- Stacktraces moins significatives et débogage difficile (surtout pour Streams)

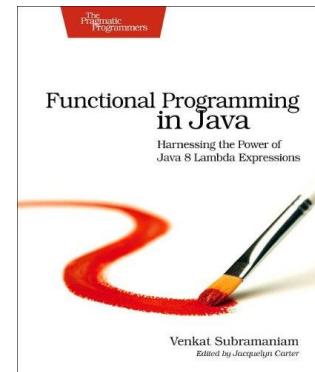
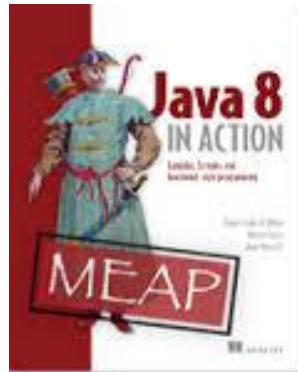
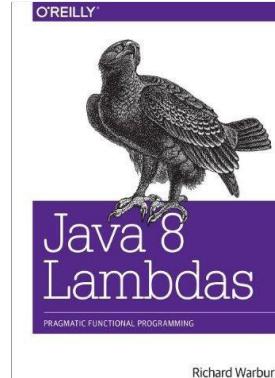
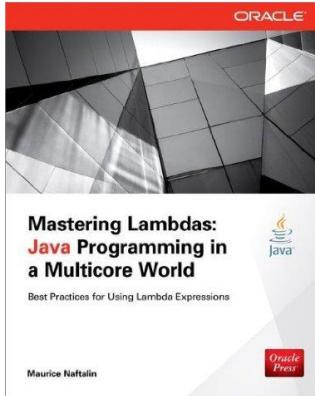
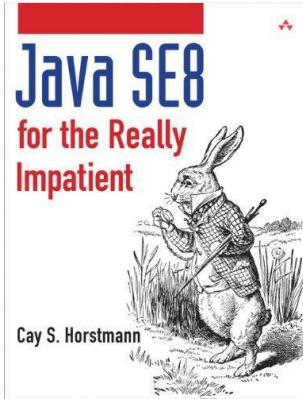


Conclusion

- Pourquoi les lambdas ont-ils été introduits dans Java 8 ?
 - Parce que les lambdas autorisent des nouveaux patterns qui permettent de paralléliser les traitements simplement et de façon sûre
 - Cela répond aux besoins des applications et des concepteurs d'API
- Migrer vers Java 8 nécessite du travail pour les développeurs
 - auto-formation
 - changement des habitudes de programmer



Annexe – Pour aller plus loin



Références

Support dans Lambdas dans JDK 8:

- <http://jdk8.java.net/lambda/> - lambda support
- <http://jdk8.java.net/download.html> - no lambda support

Articles sur les Lambdas:

- <http://www.oraclejavamagazine-digital.com/javamagazine/20121112?pg=35#pg35>
- http://www.angelikalanger.com/Conferences/Slides/jf12_LambdasInJava8-1.pdf
- <http://datumedge.blogspot.com/2012/06/java-8-lambdas.html>
- <http://www.infoq.com/articles/java-8-vs-scala>



Références

L'API Stream

- <http://cr.openjdk.java.net/~briangoetz/lambda/sotc3.html>
- <http://aruld.info/java-8-this-aint-your-grandpas-java/>
- <http://java.dzone.com/articles/exciting-ideas-java-8-streams>

La programmation fonctionnelle en Java

- <http://code.google.com/p/functionaljava/>
- <http://shop.oreilly.com/product/0636920021667.do>
- <http://apocalisp.wordpress.com/2008/06/18/parallel-strategies-and-the-callable-monad/>





Qu'avez-vous
retenue de la
formation ?

Si il y avait
une chose à
changer, ce
serait...