



Leuville Objects

EXPERTS EN TECHNOLOGIES JAVA, SOA ET UML,
DU SERVEUR AU MOBILE

Nouveautés Java 8

Expressions lambda - Nashorn

API Date & Time

Evolutions streams

MapReduce

Leuville Objects

3 rue de la Porte de Buc
F-78000 Versailles
FRANCE

tel : + 33 (0) 1 39 50 2000
fax: + 33 (0) 1 39 50 2015

www.leuville.com
contact@leuville.com

© Leuville Objects, 2000-2015
29 rue Georges Clémenceau
F-91310 Leuville sur Orge
FRANCE

<http://www.leuville.com>

Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Les marques citées sont des marques commerciales déposées par leurs propriétaires respectifs.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A RÉPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

Table des matières

Les différentes offres Java	1
Les différentes éditions de Java	1-3
Java Standard Edition	1-5
Java Enterprise Edition	1-9
Java Micro Edition	1-11
Quelques APIs utiles	1-13
Aperçu des nouveautés introduites dans Java 8	17
Java 8	2-19
Les nouveautés du langage	2-21
Les mises à jour de l'API	2-31
Les lambda-expressions	37
Expression Lambda	3-39
Réification d'une fonction ou méthode	3-41
Dans les langages de programmation	3-43
Les lambda-expressions de Java 8	45
Foncteur	4-47
Lambda-expression	4-51
Aspects syntaxiques	4-53
Liens entre lambdas et interfaces	4-57
Inférence de type	4-61
Lambda expression pour manipuler les collections	4-63
Programmation fonctionnelle avancée	71
Interface Fonctionnelle	5-73
Usage avancé	5-79
Lambda et classe anonyme	5-81
Interfaces Fonctionnelles du JDK 8	5-85
BiFunction	5-89
Predicate	5-91
Référence de méthode	5-93
Référence vers une méthode statique	5-97
Référence vers une méthode d'instance	5-99
Référence vers Classe::méthode-d'instance	5-101
Référence avec super::	5-103
Référence à un constructeur	5-105
Référence à un constructeur de tableau	5-107
Supplier<T>	5-109
Consumer<T>	5-111
Interfaces avancées	5-113
Classes anonymes	115



Classe anonyme	6-117
Accès aux variables du contexte englobant	6-119
Nouvelle API Date and Time	121
Introduction	7-123
Idées de base	7-125
Les concepts temps machine et temps humain	7-129
Le package java.time	7-131
Classes essentielles	7-133
java.time.Instant	7-135
java.time.Duration	7-139
LocalDate et LocalTime	7-143
Adjuster	7-149
Troncature	7-151
Fuseaux horaires	7-153
Les classes fuseaux horaires	7-157
Périodes	7-161
Chronologies	7-163
Le reste de l'API	7-165
Formatage et parsing de dates	169
Introduction	8-171
Formatage	8-173
Parsing	8-175
Formats standards prédéfinis	8-177
Patterns pour formats spécifiques	8-181
Java 8 Nashorn	183
De Rhino à Nashorn	9-185
Fonctionnalités	9-187
La ligne de commande JJS	9-189
Les possibilités de JJS	9-193
Javascript depuis Java	9-199
Java depuis Javascript	9-207
Utilisation de l'objet JavaImporter	9-213
Utilisation de fonctions anonymes	9-215
Introduction au modèle MapReduce	217
Le modèle MapReduce	10-219
Deux opérations distinctes	10-221
Quatre étapes	10-223
Map	10-225
Reduce	10-227
Exemple	10-229
API Stream	231
Qu'est ce qu'un Stream	11-233



Intérêt	11-235
Principes d'utilisation d'un stream	11-239
Création d'un Stream	11-241
Opérations sur un Stream.....	11-243
Utiliser map-reduce avec Java 8	247
Principes.....	12-249
Map	12-251
Map Reduce	12-253
Mécanismes avancés des streams	255
Parallélisation des traitements	13-257
Contraintes	13-261
Liens avec les threads	13-263
Utiliser un pool de threads spécifique.....	13-267
Problèmes à éviter	13-269
Les améliorations JDBC	273
API JDBC	14-275
Impacts de Date Time	14-277
Statement::executeLargeUpdate	14-279
Les nouveaux outils	281
Jdeps.....	15-283
javapackager	15-287
Java Mission Control	15-291
Annexes	299
Dates et heures avec ISO 8601	301
Introduction.....	16-303
Norme	16-305
Hypothèses de base	16-307
Formats - dates.....	16-309
Formats - heures.....	16-313
Les classes imbriquées et anonymes en Java	317
Classe imbriquée.....	17-319
La classe imbriquée renforce l'encapsulation.....	17-321
Classe imbriquée et visibilité	17-323
Classe anonyme	17-327
JavaScript	331
Présentation.....	18-333
Instructions - Conditions - Boucles	18-335
Intégration de JavaScript dans une page HTML	18-341
Variables	18-343
Événements	18-345



Les objets du navigateur	18-347
DOM en JavaScript.....	18-349

Nouveautés Java 8

Les différentes offres Java

Version 3.0

- Les différentes éditions de Java
- Les API essentielles

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Les différentes éditions de Java

Trois éditions de Java

- Java Standard Edition (Java SE)
- Java Enterprise Edition (Java EE)
- Java Micro Edition (Java ME)

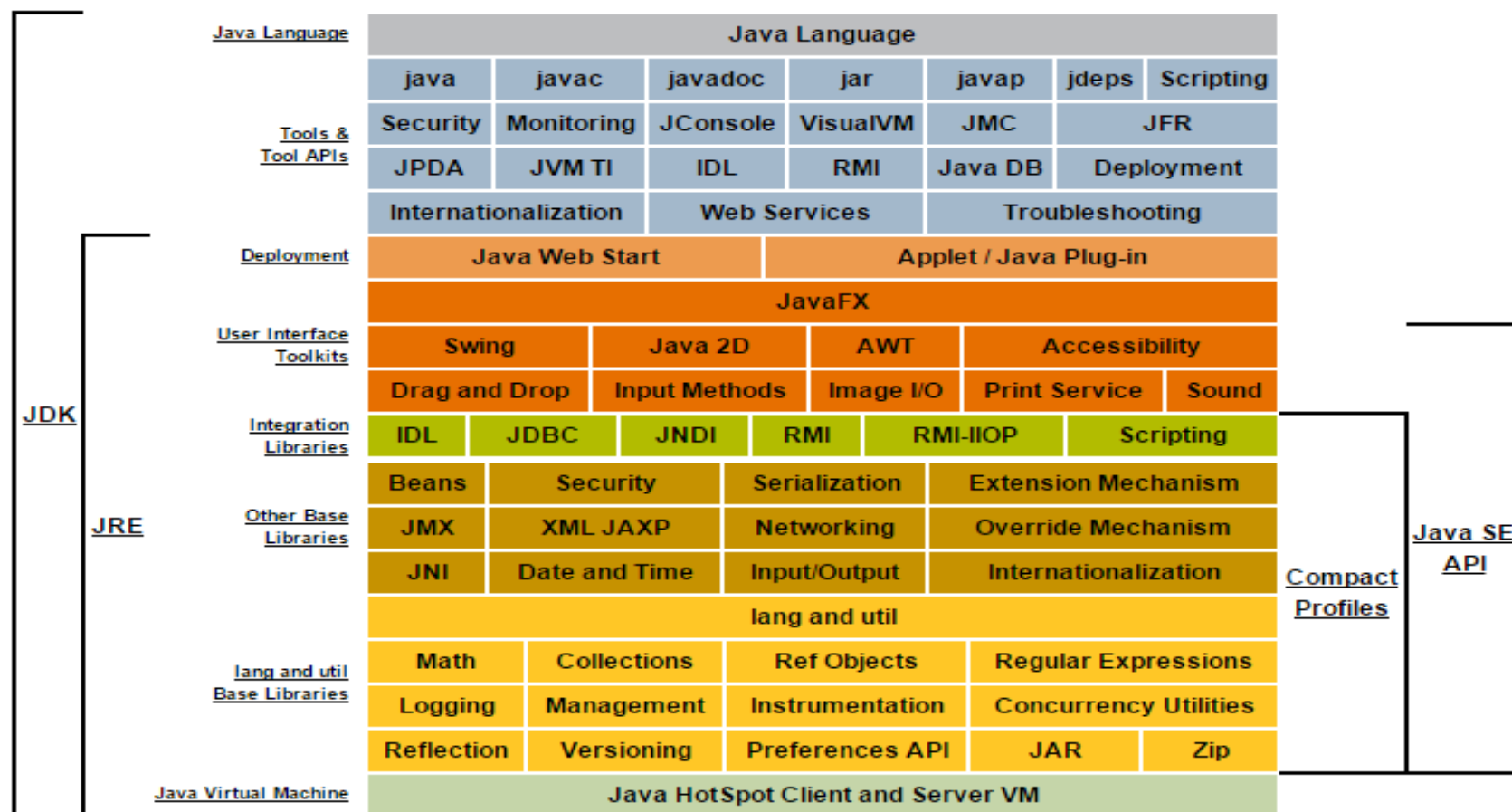
Les différentes éditions de Java

Notes

Java Standard Edition

Java 8 (Mars 2014)

- Contient l'ensemble des classes et outils de base pour le développement Java
- Se compose de plusieurs parties



Java Standard Edition

Notes

Les APIs (Application Programming Interface) sont des librairies de classes Java.

Java Standard Edition

Java Runtime Environment (JRE)

- Contient l'ensemble des éléments nécessaires à l'exécution d'une application Java
- Doit être installé sur le poste du client

Java Development Kit (JDK)

- Contient un JRE
- Contient les outils permettant de compiler, déployer les applications Java

Java SE API

- Ensemble des APIs Java permettant de mettre au point des applications

Java Standard Edition

Notes

Java Enterprise Edition

Contient l'ensemble des APIs nécessaires pour les applications d'entreprise

- Embarquée au sein des serveurs d'applications JavaEE tels que JBoss, Websphere, Weblogic ...

Couvre 4 domaines techniques

- *Web Service Technologies*
- *Web Application technologies*
- *Enterprise Application Technologies*
- *Management and Securities Technologies*

Implique une bonne connaissance des bases de Java SE

Java Enterprise Edition

Notes

Java Micro Edition

Prends en charge les aspects mobilité

Moins riche que l'édition standard

- Dédié à des environnements contraignants
 - Espace mémoire limité
 - Respect de contraintes temporelles
 - ...

Basée sur les concepts de programmation Java

Java Micro Edition

Notes

Quelques APIs utiles

JavaCard

- Gestion de cartes à puces

Java 3D

- Prise en charge de la géométrie 3D

Java Advanced Imaging (JAI)

- Amélioration de la gestion des images en Java

JavaBeans Activation Framework

- API permettant de déterminer un type de donnée pour définir comment y accéder
- Utilisé par JavaMail pour déterminer les types MIME

Quelques APIs utiles

Notes

Quelques APIs utiles

Java Communications

- Permet de faciliter les aspects communication entre plateformes (fax, modem ...)

JavaMail

- Gestion de mails par l'intermédiaire de Java

Java Media Framework (JMF)

- Support des médias audios, vidéos ...

Java Speech

- Reconnaissance de la parole

Quelques APIs utiles

Notes

Nouveautés Java 8

Version 3.0

Aperçu des nouveautés introduites dans Java 8

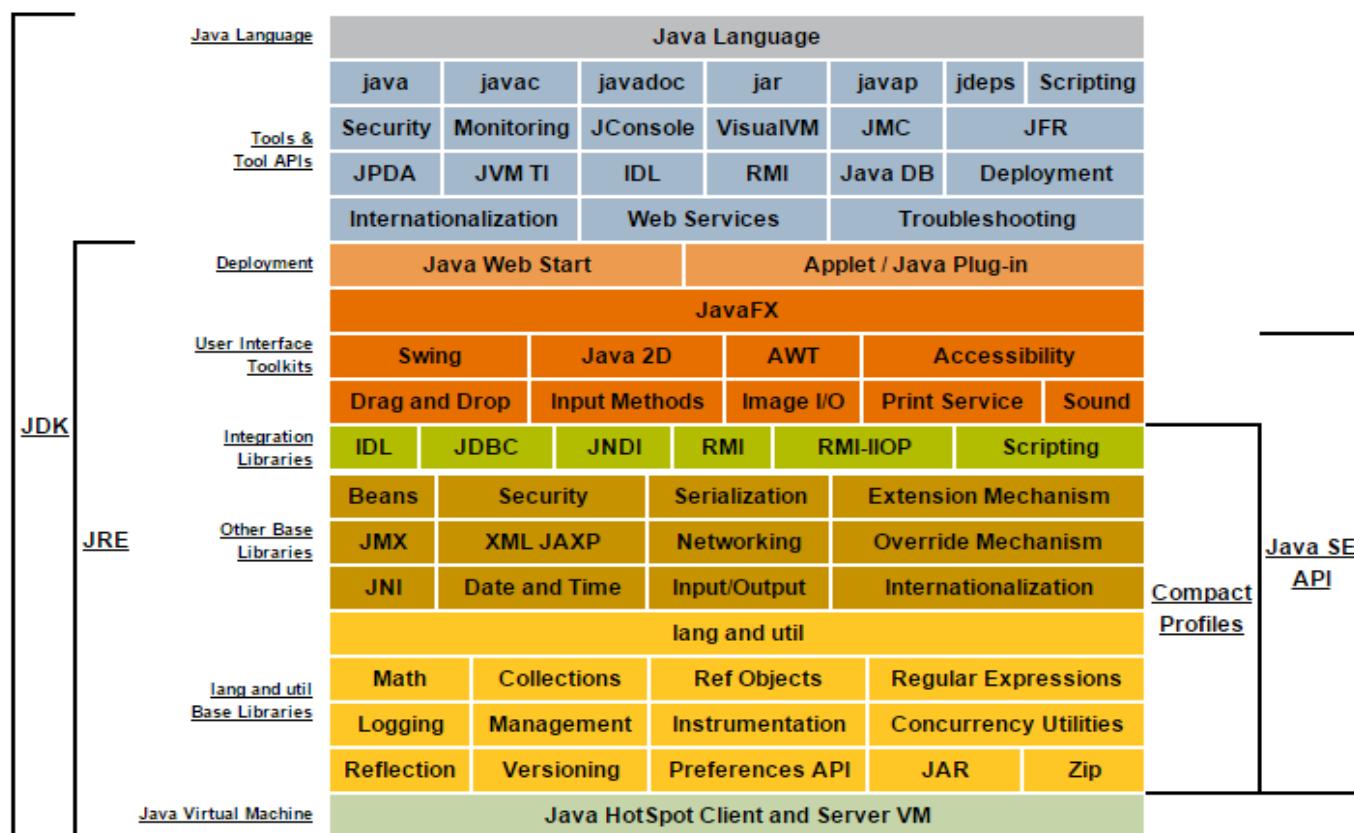
- Présentation
- Les nouveautés du langage
- Les mises à jour de l'API
- ...

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Java 8

- Version du langage Java sortie en mars 2014
- Version majeure du langage Java car introduit de nouveaux concepts et de nouvelles syntaxes dans le langage.
- Liste de nouveautés :
 - <http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>



Java 8

Notes

Les nouveautés du langage

Lambda ou lambda-expression

- Plus grosse nouveauté de Java 8 : apporte la puissance de la programmation fonctionnelle dans Java

```
String[] testStrings = ...;
// Forme longue :
Arrays.sort(testStrings, (String s1, String s2) -> { return s1.length() - s2.length(); });

// Forme courte :
Arrays.sort(testStrings, (String s1, String s2) -> s1.length() - s2.length());

// Forme courte avec type implicite des paramètres
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());

//Lambda avec une interface fonctionnelle
Runnable r1 = () -> { System.out.println("My Runnable"); };
```

- Décrite par la JSR 335

Les nouveautés du langage

Notes

Les nouveautés du langage

Interface fonctionnelle

```
@FunctionalInterface  
public interface ExampleInterface {  
    void doSomething();  
}
```

- Définition d'interface qui ne possède qu'une seule méthode d'instance abstraite
- Permet de manipuler des blocs de code comme les lambdas
- Evolution liée aux lambdas

Les nouveautés du langage

Notes

Les nouveautés du langage

Référence à une méthode

- Objectif : rendre le code plus simple et plus lisible
- Mécanisme présent dans d'autres langages (C++)

```
Supplier<Double> random = Math::random;
double result = random.get();

Random r = new Random();
Supplier<Double> random2 = r::nextDouble;
double result2 = random2.get(); // r.nextDouble();

Function<Random, Double> random3 = Random::nextDouble;
double result3 = random3.apply(r); // r.nextDouble();

Function<String, Thread> factory = Thread::new;
Thread t = factory.apply("name");
```

Les nouveautés du langage

Notes

Les nouveautés du langage

Méthode par défaut

- Permet de définir une implémentation par défaut au sein d'une interface
- L'implémentation de l'interface n'oblige pas à fournir d'implémentation d'une telle méthode

```
interface Person {  
    void sayGoodBye();  
    default void sayHello() {  
        System.out.println("Hello there!");  
    }  
}
```

Les nouveautés du langage

Notes

Les nouveautés du langage

Méthode statique dans les interfaces

- Permet d'éviter la création de classes sans attribut et ne comportant que des méthodes statiques, comme les fabriques

```
public interface Path {  
    public static Path get(String first, String... more) {  
        return FileSystems.getDefault().getPath(first, more);  
    }  
    ...  
}
```

- Plus de nécessité de couples interface/classe tels que:
 - Collection / Collections
 - Path / Paths
 - ...

Les nouveautés du langage

Notes

Les mises à jour de l'API

Streams et streams parallèles sur les collections

- Représente un flux de données que l'on peut manipuler à la volée

API `java.time`

- Nouvelle API date, heure et calendrier basée sur la JSR 310, conforme à la norme ISO 8601
- Les classes sont désormais immuables et thread-safe
- Différencie deux modèles de temps : le temps machine et le temps humain

Annotations multiples

- Possibilité de répéter plusieurs fois une annotation de même type pour un élément donné d'un programme

```
@Schedule(dayOfMonth="last")  
@Schedule(dayOfWeek="Fri", hour="23")  
public void doPeriodicCleanup() { ... }
```

Nashorn

- Nouveau moteur Javascript (<http://openjdk.java.net/jeps/174>)

Les mises à jour de l'API

Notes

Les mises à jour de l'API

JavaFX

- Permet de réaliser des clients riches
- Disponible pour les plates-formes ARM
- Possibilité d'embarquer des composants Swing
- Nouveau thème Modena, nombreuses nouveautés dans l'API

Securité

- TLS 2.1 activé par défaut coté client
- Amélioration des algorithmes pour le chiffrement des mots de passe
- ...

Les mises à jour de l'API

Notes

Les mises à jour de l'API

JDBC

- JDBC 4.2 introduit de nouvelles fonctionnalités
- Suppression du JDBC-ODBC Bridge
- Java DB 10.10 inclus dans le JDK
- Possibilité d'employer les nouveaux streams dans les ResultSet
- Utilisation de la nouvelle API DateTime pour manipuler les dates

Les mises à jour de l'API

Notes

Nouveautés Java 8

Les lambda-expressions

Version 3.0

- Concepts
- Utilité

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Expression Lambda

Concept de programmation fonctionnelle

- Paradigme de programmation axé sur la notion de fonction (au sens mathématique)
 - fonction = relation entre deux ensembles de valeurs (surjection)
- Caractéristiques de la fonction en programmation fonctionnelle
 - indépendante (ne dépend d'aucun état d'exécution en dehors du sien),
 - sans état (ne possède aucun état d'exécution interne qui est conservé entre les appels),
 - déterministe (renvoie toujours le même résultat pourvu qu'on lui ait fourni les mêmes arguments), on parle aussi de transparence référentielle.
- Une application peut être considéré comme un ensemble de fonctions opérant sur des données.
- Lambda-expression
 - fonction anonyme
 - manipulable comme un objet
 - appelé également closure

Expression Lambda

Réification d'une fonction ou méthode

Simplifie la programmation fonctionnelle

- Une fonction est un objet
- Peut être stockée dans une variable
- Peut opérer sur des paramètres qui sont des fonctions
- Peut retourner un résultat qui est une fonction
- Approche connue des Design Patterns : Foncteur

Avantages

- Robustesse du code
- Optimisation possible par mémoization
 - terme défini par Donald Michie en 1968 à partir du latin memorandum
 - consiste à stocker les associations (paramètres, résultats) dans une table afin d'éviter les recalculs inutiles

Réification d'une fonction ou méthode

Dans les langages de programmation

Java

- Introduit en Java 8

```
(int x) -> x * 2
```

C++

- Introduit en C++11

```
auto f = [] (string name) {  
    cout << "Hello " + name << endl;  
};  
f("Toto");
```

C#

- Une expression lambda est une fonction anonyme de la forme (paramètres) => expression

```
(int x, string s) => s.Length > x
```

Autres langages

- Objective-C : code blocks
- Javascript : closures

Dans les langages de programmation

Nouveautés Java 8

Les lambda-expressions de Java 8

Version 3.0

- Aspects syntaxiques
- Exemples d'utilisation
- Concept de foncteur
- Lambda Expression pour manipuler les Collections

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Foncteur

Présentation du Pattern

- Encore appelé Pattern function Object, permet d'encapsuler une fonction dans un objet afin de rendre son utilisation interchangeable.
- Différent du design Pattern Strategy qui permet d'encapsuler un algorithme dans un objet afin de rendre son utilisation interchangeable.

Objectif

- Créer une interface (générique) dont la responsabilité est de représenter une fonction, avec ses paramètres d'entrée et sa valeur de retour.
- Exemple :

```
public interface Function<A, R> {  
    public R invoke(A... args);  
}
```

Foncteur

Objectif

- Dans l'exemple ci-dessus :
 - R est le type de retour de la méthode (Return).
 - A est le type des paramètres d'entrée (Arguments).

Foncteur

Concepts

- Pureté de la fonction: Fonction qui retourne la même valeur, pourvu qu'on lui fournisse les mêmes paramètres.
- Immutabilité (ou Immuabilité): Fonction qui respecte le principe de la pureté d'une fonction et en plus ne change pas l'état des variables qui lui sont passées en paramètre.
- Expressivité : Fonction prenant d'autres fonctions en paramètre et / ou qui renvoie une autre fonction.
- Composabilité : Fonction ayant la capacité de composer des fonctions ensemble pour obtenir une fonction.

Foncteur

Lambda-expression

Définitions

- Traduction des concepts de programmation fonctionnelle et foncteur dans le monde Java
- Bloc de code potentiellement paramétré
- Remplacement des classes anonymes, plus simple à utiliser
- Permet de simplifier les codes exploitant des classes anonymes, ou des objets typés à partir d'interfaces ne comportant qu'une seule opération

Exemple

- Associer un EventHandler à un bouton en Java FX

```
button.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent event) {  
        System.out.println("Thanks for clicking!");  
    }  
});
```

- ->

```
Button red = new Button("Red");  
red.setOnAction(event -> message.setTextFill(Color.RED));
```

Lambda-expression

Aspects syntaxiques

Syntaxe

- (Paramètres) -> expression
- (Paramètres) -> {Bloc d'expression ;}
- Typage optionnel (inférence de type par le compilateur)

Exemples

- (int x) -> x * 2
- (x , y) -> {return x+y;}
- () -> 10
- (String y) -> System.out.println(y)

Aspects syntaxiques

Notes

- paramètres : représente le ou les paramètres de l'expression lambda.
- expression : représente le corps de l'expression. Lorsqu'il y a plusieurs instruction dans le corps on met les accolades. Dans ce cas il faut préciser la valeur retournée avec le mot clé return.

Aspects syntaxiques

Paramètres

- Le type peut être inféré à la compilation
- Si le type est inférable et le paramètre unique, les parenthèses peuvent être oubliées

```
Button red = new Button("Red");  
red.setOnAction(event -> message.setTextFill(Color.RED));
```

Bloc d'expression

- Les accolades peuvent être omises s'il s'agit de retourner le résultat d'une unique expression
- Sinon, il faut veiller à ce que toutes les branches retournent un résultat de même type

```
() -> { for (int i = 0; i < 1000; i++) doWork(); }
```

```
(String first, String second) -> {  
    if (first.length() < second.length()) return -1;  
    else if (first.length() > second.length()) return 1;  
    else return 0;  
}
```

Aspects syntaxiques

Liens entre lambdas et interfaces

Une lambda-expression permet d'implémenter une interface

- A partir de ces interfaces:

```
public interface Exemple1{  
    int carre (int x);  
}  
  
public interface Exemple2{  
    int operation (int x, int y);  
}  
  
public interface Exemple3{  
    int nombre ();  
}
```

- On peut écrire:

```
Exemple1 ex1 = (int x) -> x * x;  
Exemple2 ex2_somme = (x,y) -> x+y;  
Exemple2 ex2_moins = (x,y) -> x - y;  
Exemple2 ex2_produit = (x,y) -> x * y;  
Exemple3 ex3_dix = () -> 10;  
Exemple3 ex3_million = () -> ex1.carre(1000);
```

Liens entre lambdas et interfaces

Liens entre lambdas et interfaces

Utilisation des implémentations

```
int nombre1 = 5; int nombre2 = 3;

System.out.println("ex1.carre(nombre1) = : " + ex1.carre(nombre1));
System.out.println("ex2_somme.operation(nombre1, nombre2) = : " + ex2_somme.operation(nombre1, nombre2));
System.out.println("ex2_soustraction.operation(nombre1, nombre2)= : " + ex2_moins.operation(nombre1, nombre2));
System.out.println("ex2_produit.operation(nombre1, nombre2) = : " + ex2_produit.operation(nombre1, nombre2));
System.out.println("ex3_dix.nombre() = : " + ex3_dix.nombre());
System.out.println("ex3_million.nombre() = : " + ex3_million.nombre());
```

- Le résultat obtenu:

```
ex1.carre(nombre1) = : 25
ex2_somme.operation(nombre1, nombre2) = : 8
ex2_moins.operation(nombre1, nombre2)= : 2
ex2_produit.operation(nombre1, nombre2) = : 15
ex3_dix.nombre() = : 10
ex3_million.nombre() = : 1000000
```

```
Exemple1 ex1 = (int x) -> x * x;
Exemple2 ex2_somme = (x,y) -> x+y;
Exemple2 ex2_moins = (x,y) -> x - y;
Exemple2 ex2_produit = (x,y) -> x * y;
Exemple3 ex3_dix = () -> 10;
```

Liens entre lambdas et interfaces

Inférence de type

A la compilation

- Exemple
 - La lambda-expression est affectée à un `Comparator<String>`
 - Le compilateur en déduit le type des arguments `first` et `second`

```
Comparator<String> comp
    = (first, second) // Same as (String first, String second)
      -> Integer.compare(first.length(), second.length());
```

Inférence de type

Lambda expression pour manipuler les collections

API Collection

- Une collection: regroupement d'objets généralement de même nature.
- 2 grandes Familles: [java.util.Collection](#) et [java.util.Map](#).
- Types de collection : List , Map, Set, Queue

Création d'une Collection

```
List<Etudiant> etudiants = new ArrayList<> ();
```

Opération sur les collections

- Utilisation des fonctions de base pour manipuler une collection. (Soit la liste étudiants)

```
[nom=victorien, age=20], [nom=fabrice, age=30], [nom=laurent, age=25],  
[nom=alain, age=18], [nom=jean, age=15], [nom=max, age=22]
```

Lambda expression pour manipuler les collections

Lambda expression pour manipuler les collections

- Afficher le nom de chaque étudiant de la liste "etudiants": En Java 7

```
for(Etudiant etu : etudiants){  
    System.out.println(etu.getName());  
}
```

- En Java 8

```
etudiants.forEach( (etu) -> System.out.println( etu.getName( ) ) );
```

- Résultat à la console

```
victorien  
fabrice  
laurent  
alain  
jean  
max
```

Lambda expression pour manipuler les collections

Notes

Il est possible de simplifier encore le code d'itération en utilisant une référence de méthode. Ce concept sera abordé ultérieurement.

Lambda expression pour manipuler les collections

- La liste des étudiants par ordre alphabétique : En Java 7

```
String[] noms = { "victorien", "fabrice", "laurent", "alain", "jean", "max" };
Arrays.sort(noms, new Comparator<String>() {
    @Override
    public int compare(String etu1, String etu2) {
        return etu1.compareTo(etu2);
    }
});
List<String> nomsEtudiant = Arrays.asList(noms);
for(String etu : nomsEtudiant){
    System.out.println(etu);
}
```

- Résultat

```
alain
fabrice
jean
laurent
max
victorien
```

Lambda expression pour manipuler les collections

Lambda expression pour manipuler les collections

- En java 8 :

```
Arrays.sort(noms,(n1,n2) -> (n1.compareTo(n2)));  
List<String> nomsEtudiant = Arrays.asList(noms);  
nomsEtudiant.forEach( System.out::println);
```

- Pour l’affichage, nous avons utilisé une référence à la méthode println() de l’objet System.out

Lambda expression pour manipuler les collections

Nouveautés Java 8

Programmation fonctionnelle avancée

Version 3.0

- Interface fonctionnelle et `@FunctionalInterface`
- Interfaces fonctionnelles du JDK 8
- Référence de méthode et de constructeur
- Implémentation par défaut dans une interface

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Interface Fonctionnelle

Besoins

- De nombreux langages disposant de mécanismes de programmation fonctionnelle permettent de définir des variables ayant le type d'une fonction
- Exemple en C++11

```
auto f = [] (string name) {  
    cout << "Hello " + name << endl;  
};  
f("Toto");
```

- C'est impossible en Java 8, qui conserve une forte orientation Objet
- Java définit et utilise de nombreuses interfaces comportant uniquement une opération
- Runnable, Comparator, ...
- Il est souhaitable qu'il y ait une compatibilité entre ces interfaces et les lambda-expressions, de façon à pouvoir utiliser ces dernières partout où on utilise ces interfaces

Concept

- Une interface fonctionnelle est une interface comportant une seule opération abstraite
- Une lambda-expression peut être typée en interface fonctionnelle tout en restant un objet

Interface Fonctionnelle

Interface Fonctionnelle

Définition

- Interface avec exactement une méthode abstraite.
- attention, une méthode par défaut = non abstraite.
- Annotation : `@java.lang.FunctionalInterface`
- optionnelle, permet de vérifier que la contrainte de présence d'une méthode abstraite est respectée
- utilisation conseillée
- Exemple :

```
public class Test1 {  
  
    @FunctionalInterface  
    interface Truc<T> {  
        public T changeIt(T data);  
    }  
  
    public static void main(String[] args) {  
  
        Truc<String> lambda1 = (uneString) -> uneString.toUpperCase();  
  
        System.out.println(lambda1.changeIt("toto"));  
    }  
}
```

Interface Fonctionnelle

Notes

- L'annotation `@FunctionalInterface` est optionnelle (Comme l'annotation `@Override`).
- Elle sert à apporter une sécurité supplémentaire à la compilation.

Interface Fonctionnelle

Adaptation de nombreuses interfaces de base du JDK

- Runnable
- Comparator
- ...

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface Runnable is used
     * to create a thread, starting the thread causes the object's
     * run method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method run is that it may
     * take any action whatsoever.
     *
     * @see      java.lang.Thread#run()
     */
    public abstract void run();
}
```

Ce qui permet d'écrire

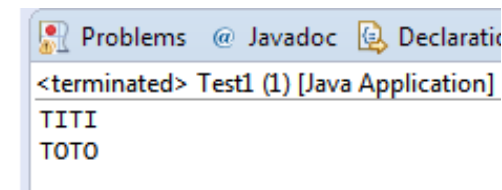
```
new Thread(() -> System.out.println("Bonjour")).start();
```

Interface Fonctionnelle

Usage avancé

- Passage en paramètre d'une lambda
- Appel d'une lambda dans le corps d'une autre lambda

```
public class Test1 {  
  
    @FunctionalInterface  
    interface Truc<T> {  
        public T changeIt(T data);  
    }  
  
    public static <T> T doIt(Truc<T> lambda, T data) {  
        return lambda.changeIt(data);  
    }  
  
    public static void main(String[] args) {  
  
        Truc<String> lambda1 = (uneString) -> uneString.toUpperCase();  
  
        System.out.println(doIt(lambda1, "titi"));  
  
        new Thread(() -> {System.out.println(doIt(lambda1, "toto"));}).start();  
  
    }  
}
```



Usage avancé

Notes

Les lambda-expressions s'utilisent comme des objets usuels.

Lambda et classe anonyme

Compatibilité

- En première approche, une lambda-expression est une instance d'une classe anonyme
- Cela permet d'assurer la compatibilité avec les mécanismes Java basés sur les interfaces ne comportant qu'une seule opération

Mais

- La JVM est libre d'implémenter le mécanisme des lambda-expressions comme elle le souhaite
 - il n'est pas requis de passer systématiquement par de nouvelles classes / nouvelles instances comme cela l'était avec les classes anonymes
- Lors du développement, il peut être souhaitable de considérer une lambda-expression en tant que fonction plutôt qu'en tant qu'objet comme on le fait avec les classes anonymes

Lambda et classe anonyme

Lambda et classe anonyme

Exemple réalisé avec le JDK 1.8.0_45 d'Oracle

```
public class Test2 {  
  
    @FunctionalInterface  
    interface Truc<T> { public T changeIt(T data); }  
  
    public static void printInfo(Object x) {  
        System.out.println(x + "\n" +  
            x.getClass().getName() + "\n" +  
            x.getClass().getSimpleName() + "\n" +  
            x.getClass().getCanonicalName() + "\n" +  
            x.getClass().getTypeName()  
        );  
    }  
  
    public static void main(String[] args) {  
        printInfo(new Test2());  
  
        Truc<String> lambda1 = (uneString) -> uneString.toUpperCase();  
        printInfo(lambda1);  
  
        Truc<String> lambda2 = (uneString) -> uneString.toUpperCase();  
        lambda2.changeIt("tata");  
        printInfo(lambda2);  
    }  
}
```

```
<terminated> Test2 [Java Application] C:\Program  
Test2@659e0bfd  
Test2  
Test2  
Test2  
Test2  
Test2$$Lambda$1/424058530@31befd9f  
Test2$$Lambda$1/424058530  
Test2$$Lambda$1/424058530  
Test2$$Lambda$1/424058530  
Test2$$Lambda$1/424058530  
Test2$$Lambda$2/471910020@1fb3ebeb  
Test2$$Lambda$2/471910020  
Test2$$Lambda$2/471910020  
Test2$$Lambda$2/471910020  
Test2$$Lambda$2/471910020
```

Lambda et classe anonyme

Interfaces Fonctionnelles du JDK 8

De nouvelles interfaces fonctionnelles dans le package `java.util.function`

- Interfaces fonctionnelles génériques
- Permettent de typer les lambda-expressions sans créer d'interface spécifique
- Regroupent les interfaces fonctionnelles en fonction de leur nature et leur modes opératoires
- Plusieurs interfaces disponibles
 - `Function<T, R>` pour les fonctions mono-paramétrées
 - `BiFunction<T, U, R>` pour les fonctions à deux paramètres
 - `Supplier<T>` qui définit une opération: `T get();`
 - `Consumer<T>` qui définit une opération `void accept(T);`
 - `Predicate<T>` pour les fonctions mono-paramétrées à retour booléen
 - `BiPredicate<T, U>`
 - ...

Interfaces Fonctionnelles du JDK 8

Il existe de nombreuses autres sous-interfaces spécialisées au sein de ce package.

Function

java.util.function.Function

- Function<T, R> : opération qui accepte un argument (type T) et retourne un résultat (type R)

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

- Exemple
 - Déclaration et affectation

```
Function<Etudiant,Integer> func = etu ->etu.getAge();
```

- Utilisation : afficher les ages des étudiants de la liste "etudiants"

```
etudiants.forEach((etudiant) ->System.out.print("; " + func.apply(etudiant)));
```

Function

BiFunction

java.util.function.BiFunction<T, U, R>

- T et U sont les types des deux arguments
- R est le type du retour

```
public interface BiFunction <T, U, R>{  
    R apply(T t, U u);  
}
```

- Exemple
 - Un comparateur est une BiFunction<T,U>

```
BiFunction<String, String, Integer> comp  
    = (first, second) -> Integer.compare(first.length(), second.length());
```

BiFunction

Predicate

- Predicate<T> : Opération qui accepte une valeur (type T) et retourne un booléen.

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

- Exemple

```
Predicate<Etudiant> majeurPredicate = e -> e.getAge()>18;  
  
Etudiant etu1 = new Etudiant("fabrice", 30);  
Etudiant etu2 = new Etudiant("laurent", 15);  
Etudiant etu3 = new Etudiant("alain", 23 );  
Etudiant etu4 = new Etudiant("serge", 17);  
  
System.out.println(majeurPredicate.test(etu1));  
System.out.println(majeurPredicate.test(etu2));  
System.out.println(majeurPredicate.test(etu3));  
System.out.println(majeurPredicate.test(etu4));  
  
//Affiche: true false true false
```

Predicate

Référence de méthode

Mécanisme de simplification du code

- Permet de se concentrer sur ce qu'on veut faire, et pas forcément sur la façon de le faire
- Exemple:
 - Cette écriture, déjà simple

```
button.setOnAction(event -> System.out.println(event));
```

- peut le devenir encore plus

```
button.setOnAction(System.out::println);
```

- grâce à la possibilité de référencer la méthode println() disponible sur l'objet System.out

Avec l'opérateur de portée ::, on peut référencer

- Des méthodes
- Des constructeurs
- et utiliser ensuite ces références comme des objets classiques

Référence de méthode

Référence de méthode

Equivalence avec les lambda-expressions

- A partir de cet exemple

```
button.setOnAction(System.out::println);
```

- `System.out::println` est une référence de méthode équivalente à la lambda-expression:

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

- Autres exemples
 - `Math::pow` est équivalente à: `(x, y) -> Math.pow(x,y)`
 - `String::compareToIgnoreCase` équivaut à: `(x, y) -> x.compareToIgnoreCase(y)`

Il est donc possible de faire des références de trois types:

- `objet::méthode`-d'instance \rightarrow lambda-expression(param) avec corps = **`méthode`**(param)
- `classe::méthode`-statique \rightarrow lambda-expression(paramètres) avec corps = **`méthode`**(paramètres)
- `classe::méthode`-d'instance \rightarrow lambda-expression(p1, p2, ...) avec corps = p1.**`méthode`**(p2, ...)

Référence de méthode

Notes

En cas de surcharge de la méthode appelée, le compilateur choisit la cible par inférence de type en fonction du contexte.

Référence vers une méthode statique

NomClasse : : NomMéthode

- classe::**méthode**-statique -> lambda-expression(paramètres) avec corps = **méthode**(paramètres)
- Exemple avec une référence sur Math::abs()

```
import java.util.function.Function;

public class Static {

    public static void main(String[] args) {
        Integer[] values = { -8, 12, 3, 0, -3, 9, -17 };

        Function<Integer, Integer> function = Math::abs;

        for (int v : values) {
            System.out.println(function.apply(v));
        }
    }
}
```

Référence vers une méthode statique

Notes

- la syntaxe est `NomClasse : : NomMethode`
- `NomClasse` : nom de la classe
- `NomMethode` : nom de la méthode statique de la classe

Référence vers une méthode d'instance

NomClasse::NomMéthode

- objet::**méthode**-d'instance → lambda-expression(param) avec corps = **méthode**(param)
- Exemple 1

```
button.setOnAction(System.out::println);
```

- Exemple 2

```
public class This {  
    public void doIt() {  
        System.out.println("This::doIt");  
    }  
  
    public static void main(String[] args) {  
        This me = new This();  
        new Thread(me::doIt).start();  
    }  
}
```

Référence vers une méthode d'instance

Référence vers Classe::méthode-d'instance

Avec au moins deux paramètres

- Exemple

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

- où

String::compareToIgnoreCase équivaut à: $(x, y) \rightarrow x.compareToIgnoreCase(y)$

Référence vers Classe::méthode-d'instance

Référence avec super::

super::méthode

- this est la cible de l'envoi de message
- La méthode exécutée est celle de la superclasse

```
class A {  
    public void doIt() {  
        System.out.println("A::doIt");  
    }  
}  
class B extends A {  
    public void doIt() {  
        System.out.println("B::doIt");  
    }  
}  
class C extends B {  
    public void doIt() {  
        System.out.println("C::doIt");  
        new Thread(super::doIt).start();  
    }  
}  
  
public class Super {  
  
    public static void main(String[] args) {  
        A a = new C();  
        a.doIt();  
    }  
}
```

```
<terminated> Super  
C::doIt  
B::doIt
```

Référence avec `super::`

Référence à un constructeur

NomClasse::new

- Référence à un constructeur
- Si la classe comporte plusieurs constructeurs, le bon est inféré selon le contexte et les types employés

```
public class Test1 {  
  
    public void print() {  
        System.out.println(getClass().getSimpleName() + " " + this);  
    }  
  
    public static void main(String[] args) {  
  
        Supplier<Test1> builder = Test1::new;  
        Test1[] tab = { builder.get(), builder.get(), builder.get() };  
        List<Test1> list = Arrays.asList(tab);  
        list.forEach(Test1::print);  
    }  
}
```

Référence à un constructeur

Référence à un constructeur de tableau

Type[]::new

- Le paramètre est la taille du tableau
- Exemples
 - `int[]::new` équivalent à: `x -> new int[x]`
 - `Button[]::new` équivalent à: `x -> new Button[x]`

```
Button[] buttons = stream.toArray(Button[]::new);
```

Référence à un constructeur de tableau

Supplier<T>

java.util.function.Supplier

- Supplier<T> : Opération qui ne prend pas d'argument et qui retourne un résultat (type T)

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

- Exemple

```
Supplier<Etudiant> etuSupplier = Etudiant::new; // référence au constructeur de Etudiant  
Etudiant etud = etuSupplier.get();  
Etudiant etud2 = etuSupplier.get();  
etud.setName("supplier");  
etud2.setAge(5);
```

Supplier<T>

Consumer<T>

java.util.function.Consumer

- Consumer<T> : Opération qui accepte un argument et ne retourne pas de résultat

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

- Exemple

```
Consumer<Etudiant> etudConsumer = (u) -> System.out.println("Nom : " + u.getName());  
etudiants.forEach(etudConsumer::accept);
```

Consumer<T>

Soit La liste "etudiants" suivante :

```
[nom=victorien, age=20], [nom=fabrice, age=30], [nom=laurent, age=25],  
[nom=alain, age=18], [nom=jean, age=15], [nom=max, age=22]
```


Interfaces avancées

Méthodes par défaut

- Permet de proposer une implémentation par défaut aux méthodes déclarées dans les interfaces.

```
public interface Etudiant{  
  
    public void afficheStatut();  
  
    default void afficheEtudiant(){  
        system.out.println("Je suis un Etudiant");  
    }  
}
```

Méthode static dans les interfaces

```
public interface MonInterface{  
    public static void main(String[] args){  
        System.out.println("Bonjour le Monde");  
    }  
}
```

Interfaces avancées

- Le mot clé `default` est utilisé pour définir une implémentation par défaut.

Nouveautés Java 8

Classes anonymes

Version 3.0

- Inférence de type et accès aux variables locales englobantes

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Classe anonyme

Rappels

- Classe déclarée et instanciée directement
- Elle est implicitement final
- Ne peut pas contenir de constructeur

Exemple

- A partir d'une interface

```
public class MaClasse{
    AutreClasse ac = new AutreClasse( ){
        public void read( a ) {
            System.out.println( a );
        }
    };
}
```

```
public interface AutreClasse {
    public void read ( a );
}

//UTILISATION :
ac.read( "Bonjour" ); // Affiche Bonjour
```

Classe anonyme

Notes

- `ac` (`AutreClasse`) est déclarée puis instancié.
- Cette classe sera nommée `MaClasse$1` par le compilateur.

Accès aux variables du contexte englobant

Accès variable locale depuis le corps d'une méthode d'une classe anonyme

- Interdit (jusqu'à Java 7 compris) si la variable n'est pas explicitement finale
- Autorisé en Java 8 si la variable est effectivement finale, cela étant déduit par le compilateur (inférence)
- La modification est toujours impossible

```
1
2 public class Test1 {
3
4     interface Truc<T> {
5         public void doIt(T data);
6     }
7
8     public static void main(String[] args) {
9
10        int x = 5;
11
12        Truc<String> foo = new Truc<String>() {
13
14            @Override
15            public void doIt(String data) {
16                System.out.println(data + x);
17            }
18
19        };
20
21    }
```

Accès aux variables du contexte englobant

Notes

Le fait de modifier la valeur de x entraînera une erreur de compilation. (local variables referenced from a lambda expression must be final or effectively final)

Nouveautés Java 8

Nouvelle API Date and Time

Version 3.0

- Vue d'ensemble de l'API et des nouveaux packages
- Temps machine et temps humain
- Dates, durées et fuseaux horaires

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Introduction

Nombreux problèmes avec l'API existante

- Les classes existantes, telles que `java.util.Date` et `SimpleDateFormat`, ne sont pas thread-safe, conduisant à des problèmes potentiels de concurrence.
- Certaines des classes de date et d'heure présentent également une conception assez pauvre.
 - Par exemple dans `java.util.Date`, certains choix manquent de cohérence
 - "Les années commencent en 1900,
 - "Les mois commencent à 1,
 - "Et les jours commencent à 0.
- Ces problèmes ont conduit à la popularité bibliothèques tierces, comme Joda-Time.

Une nouvelle API Date et Time a été conçue pour Java SE 8

- Le projet a été mené conjointement par l'auteur de Joda-Time (Stephen Colebourne) et Oracle, sous JSR 310, et sera présent dans le nouveau paquet `java.time` de Java SE 8.

Introduction

Idées de base

Concepts essentiels

- Immuabilité et thread safety
- Méthodes chaînables
- Homogénéité de l'API
- Généralisation des fabriques
- Conformité calendrier ISO-8601

ISO_ZONED_DATE_TIME	With zone offset and zone ID	1969-07-16T09:32:00-05:00[America/New_York]
---------------------	---------------------------------	---

- Possibilité d'utiliser des calendriers non-conformes, ou d'en créer de nouveaux

Idées de base

- La nouvelle API est pilotée par ces idées fondamentales:
 - **Immuabilité et thread safety** : Toutes les classes centrales de l'API Date and Time sont immuables, ce qui nous assure de ne pas avoir à nous soucier de problèmes de concurrence. De plus, qui dit objets immuables, dit des objets simples à créer, à utiliser et à tester.
 - **Chaînage** : les méthodes chaînables rendent le code plus lisible et elles sont aussi plus simples à apprendre. Quant aux méthodes de type factory (par exemple: `now()`, `from()`, etc.) elles sont utilisées en lieu et place de constructeurs.
 - **Clarté** : Chaque méthode définit clairement ce qu'elle fait. De plus, hormis dans quelques cas particuliers, passer un paramètre nul à une méthode provoquera la levée d'un `NullPointerException`. Les méthodes de validation prenant des objets en paramètre et retournant un booléen retournent généralement false lorsque null est passé.
 - **Extensibilité** : Le design pattern Stratégie utilisé à travers l'API permet son extension en évitant toute confusion. Par exemple, bien que les classes de l' API soient basées sur le système de calendrier ISO-8601, nous pouvons aussi utiliser les calendriers non-ISO - tel que le calendrier Impérial Japonais - qui sont inclus dans l'API, ou même créer notre propre calendrier.

Idées de base

Création des objets

- Toutes les classes de base à la nouvelle API sont construites par des procédés courants de Factory.
- Lors de la construction d'une valeur par ses champs constitutifs, la Factory est appelée *of*.
- Lors de la conversion depuis un autre type, l'usine est appelé *from*.
- Il y a aussi des méthode de parsing qui prennent des chaînes comme paramètres.

Accès aux données

- Les getters conventionnels du Standard Java sont utilisés afin d'obtenir les valeurs des instances.

Modification

- Il est également possible de modifier les attributs des objets. Puisque dans la nouvelle API toutes les classes fondamentales sont immuables, ces méthodes retournent de nouveaux objets.
 - plusieurs de ces méthodes sont préfixées par *with*
- Il existe également des méthodes de calculs basés sur les différents champs.

Idées de base

Les concepts temps machine et temps humain

- La nouvelle API différencie également deux modèles de temps : le temps machine et le temps humain.

Le temps machine

- Pour une machine, le temps n'est qu'un entier augmentant depuis l'époque (01 janvier 1970 00h00min00s0ms0ns).
- Il correspond réellement à l'idée que l'on se fait du temps et de la durée.
- Il est défini de façon unique pour tous les développeurs du monde.
- A chaque instant réel correspond un temps machine et réciproquement.
- Ce temps sera manipulé à l'aide des classes `Instant` et `Duration`.

Le temps humain

- Il correspond à la manière dont on utilise les heures et les dates de façon usuelle et plus ou moins "locale".
- Il s'agit d'une succession de champs ayant une unité (année, mois, jours, heure, etc.). À noter également que le mois "0" n'existe plus et le mois "1" correspond au mois de janvier.
- Certaines notions comme l'année et le mois ne correspondent pas à une durée fixe puisque tous les mois n'ont pas le même nombre de jours et que certaines années sont bissextiles.
- Par ailleurs l'heure associée localement à un instant donné s'exprime différemment suivant le fuseau horaire concerné.

Les concepts temps machine et temps humain

Le package `java.time`

Un package primaire, `java.time`

- **`java.time`**: le noyau de l'API pour la représentation de la date et l'heure.
 - Il inclut les classes `Date`, `Hour`, la date et l'heure combinée, les fuseaux horaires, les instants, la durée et l'horloges.
 - Ces classes sont basées sur le système de calendrier défini dans la norme ISO-8601 et sont immuables et thread-safes.

Quatre packages complémentaires

- **`java.time.chrono`**: L'API pour représenter les systèmes de calendrier autres que celui par défaut, ISO-8601. Il est possible de définir son propre système de calendrier.
- **`java.time.format`**: Classes pour le formatage et le parsing des dates et heures.
- **`java.time.temporal`**: API étendue, principalement à destination des développeurs d'API et de frameworks, permettant l'interopérabilité entre les classes `Date` et `Time`, les requêtes et les `Adjustments`.
- **`java.time.zone`**: Les classes qui prennent en charge les fuseaux horaires, les décalages de fuseaux horaires, et les règles de fuseaux horaires. La plupart des développeurs n'auront besoin que de des classes `ZonedDateTime`, `IDZone` ou `ZoneOffset`.

Le package `java.time`

Classes essentielles

Nouvelle API de Java 8

- Instant = point sur l'axe des temps (similaire à une Date)
- Duration = différence entre deux instants
- LocalDateTime = date sans information de fuseau horaire
- TemporalAdjuster = moyen de faire des calculs à partir de dates
- ZonedDateTime = date relative à un fuseau horaire (similaire à GregorianCalendar)
- Period = durée tenant compte des spécificités telles que l'heure d'été
- DateTimeFormatter = pour formater et parser des dates et heures

Classes essentielles

java.time.Instant

La classe Instant représente un point précis dans le temps depuis le 01/01/1970 00h00

- Nombre de nanosecondes depuis l'Epoch (1er Janvier 1970). C'est ce qui se rapproche le plus de java.util.Date.
- **Instant.EPOCH** : représente l'epoch
- **Instant.MIN** : représente la plus petite valeur (pour les dates avant Jésus-Christ)
- **Instant.MAX** : représente la plus grande valeur possible (31 décembre un trillion).
- **Instant.parse()** : retourne un objet de type Instant à partir d'une chaîne de caractère représentant une date au format ISO-8601.
- **Instant.ofEpochSecond()/Instant.ofEpochMilliSecond()** : retourne un objet de type Instant à partir d'un offset de x secondes ou millisecondes par rapport à l'epoch.
- **Instant.now()** : retourne un objet de type Instant représentant la date UTC actuelle.
- **Instant.now().getNano()** : retourne le nombre de nano secondes de la date actuelle

java.time.Instant

Notes

- **Instant.parse()** : retourne un objet de type Instant à partir d'une chaîne de caractère représentant une date au format ISO-8601. Si la chaîne de caractères ne représente pas une valeur valide, une exception de type `java.time.format.DateTimeParseException` sera levée (le standard ISO-8601 spécifie que la lettre "T" désigne l'heure qu'elle précède et "Z" une data UTC).
- Les méthodes `isAfter()`, `isBefore()`, `minusSeconds()`, etc. font ce que leur nom indique.

java.time.Instant

```

36 @Test
37 public void instant() {
38     //---- Instant.EPOCH
39     Assert.assertEquals("1970-01-01T00:00:00Z", Instant.EPOCH.toString());
40     Assert.assertEquals(Instant.parse("1970-01-01T00:00:00Z"), Instant.EPOCH);
41     Assert.assertEquals(Instant.ofEpochSecond(0), Instant.EPOCH);
42     //---- Instant.MIN
43     Assert.assertEquals(Instant.parse("-1000000000-01-01T00:00:00Z"), Instant.MIN);
44     //---- Instant.MAX
45     Assert.assertEquals(Instant.parse("+1000000000-12-31T23:59:59.999999999Z"), Instant.MAX);
46     //---- Few instance methods
47     final Instant instant = Instant.now();
48     // prints the current time
49     // e.g. 2013-05-26T21:37Z (Coordinated Universal Time)
50     System.out.println("Instant.now() : " + instant);
51     // print the number of nano seconds
52     System.out.println("Instant.now().getNano() : " + instant.getNano());
53     //-- Working with 2 instants
54     // 2013-05-26T23:10:40Z & 1530-05-26T23:10:40Z
55     final Instant instant20130526_231040 = Instant.parse("2013-05-26T23:10:40Z");
56     final Instant instant15300526_231040 = Instant.parse("1530-05-26T23:10:40Z");
57     // 2013-05-26T23:10:40Z is After 1530-05-26T23:10:40Z
58     Assert.assertTrue(instant20130526_231040.isAfter(instant15300526_231040));
59     // 2013-05-26T23:10:40Z is NOT Before 1530-05-26T23:10:40Z
60     Assert.assertFalse(instant20130526_231040.isBefore(instant15300526_231040));
61     // 2013-05-26T23:10:40Z minus 1 hour (3600s)
62     Assert.assertEquals(Instant.parse("2013-05-26T22:10:40Z"),
63         instant20130526_231040.minusSeconds(3600));
64 }

```

java.time.Instant

java.time.Duration

Durée = différence entre deux Instants

```
Instant start = Instant.now();  
runAlgorithm();  
Instant end = Instant.now();  
Duration timeElapsed = Duration.between(start, end);
```

- **Immutable:** les méthodes de "modification" créent en réalité de nouvelles instances
- **Duration.ZERO** : représente une durée nulle.
- **Duration.parse()** : retourne un objet de type Duration à partir d'une chaîne de caractère représentant une durée au format ISO-8601.
- **Duration.ofMillis()** : retourne un objet de type Duration à partir d'une chaîne de caractères représentant une durée en millisecondes.

java.time.Duration

Notes

- **Duration.parse()** : retourne un objet de type Duration à partir d'une chaîne de caractère représentant une durée au format ISO-8601. Si la chaîne de caractères ne représente pas une valeur valide, une exception de type java.time.format.DateTimeParseException sera levée (selon le standard ISO-8601 une durée commence P – pour Period- et est suivie d'une valeur comme par exemple 4DT11H9M8S – pour Days, Hours, Minutes et Seconds – où la date et l'heure sont séparées par la lettre "T" – pour Time -). Le standard permet aussi de définir un nombre d'années et de mois, ce que ne permet pas la méthode parse().
- La classe Duration, à l'instar de Instant, possède, différents getters et méthodes de manipulation telles que plusX(), minusX(), etc. où X correspond à Days, Hours, Minutes, etc. ainsi que withY() où Y correspond à Seconds ou Nanos (ces méthodes retournent une copie de la durée ajustée avec la valeur passée en paramètre).
- Les classes Instant et Duration étant immuables, les méthodes plusX(), minusX(), withY(), etc. retournent toujours de nouvelles instances.

java.time.Duration

Exemples

```
6 @Test
7 public void duration() {
8     //---- Duration.ZERO
9     Assert.assertEquals(Duration.parse("PT0S"), Duration.ZERO);
10    Assert.assertEquals("PT0S", Duration.ZERO.toString());
11    //---- 2h 5min 30s 345ms = 7_530_345ms
12    final Duration duration = Duration.ofMillis(7_530_345);
13    Assert.assertEquals("PT2H5M30.345S", duration.toString());
14    Assert.assertEquals(7530, duration.getSeconds());
15    Assert.assertEquals(345000000, duration.getNano());
16 }
17 }
```

java.time.Duration

LocalDate et LocalTime

Date et Heure sans notion de fuseau horaire

- La problématique des fuseaux horaires est ici mise de côté.
- Ces classes locales devront être utilisées lorsque le contexte n'est pas nécessaire.
- Une application Desktop pourrait être un exemple d'utilisation.

```
15 LocalDateTime TIMEPOINT = LocalDateTime.now(); // la date et l'heure
16 LocalDate localDate1 = LocalDate.of(2012, Month.DECEMBER, 12);
17 LocalDate localDate2 = LocalDate.ofEpochDay(150);
18 LocalTime localTime1 = LocalTime.of(17, 18);
19 LocalTime localTime2 = LocalTime.parse("10:15:30");
20
21 System.out.println(TIMEPOINT);
22 System.out.println(localDate1);
23 System.out.println(localDate2);
24 System.out.println(localTime1);
25 System.out.println(localTime2);
26
```

```
2015-04-15T14:53:06.942
2012-12-12
1970-05-31
17:18
10:15:30
```

- Il existe aussi une classe composite appelée LocalDateTime , qui est un jumelage de LocalDate et LocalTime

LocalDate et LocalTime

LocalDate et LocalTime

Accès aux informations

- Par des getters

```
@Test
public void test2() {

    LocalDateTime timePoint = LocalDateTime.now();
    LocalDate theDate = timePoint.toLocalDate();
    Month month = timePoint.getMonth();
    int day = timePoint.getDayOfMonth();
    timePoint.getSecond();
    System.out.println(timePoint);
    System.out.println(theDate);
    System.out.println(month);
    System.out.println(day);
}
```

LocalDate et LocalTime

Notes

LocalDate et LocalTime

Calculs

```
45 @Test
46 public void test3() {
47
48     LocalDateTime timePoint = LocalDateTime.now();
49     LocalDateTime thePast = timePoint.withDayOfMonth (10).withYear (2010);
50     LocalDateTime yetAnother = thePast.plusWeeks (3).plus (3, ChronoUnit.WEEKS);
51     System.out.println(timePoint);
52     System.out.println(thePast);
53     System.out.println(yetAnother); |
54 }
55 }
56
```

2015-04-15T16:00:47.988
2010-04-10T16:00:47.988
2010-05-22T16:00:47.988

LocalDate et LocalTime

Adjuster

Permettent d'effectuer des calculs, réglages ou ajustements

- Exemple: déterminer le premier mardi de chaque mois

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(  
    TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

- Des Adjusters embarqués sont définis dans la nouvelle API.

```
import static java.time.temporal.TemporalAdjusters.*;  
  
LocalDateTime timePoint = ...  
foo = timePoint.with(lastDayOfMonth());  
bar = timePoint.with(previousOrSame(ChronoUnit.WEDNESDAY));  
timePoint.with(LocalTime.now());
```

Adjuster

Troncature

Adapter la précision à ses besoins

- La nouvelle API prend en charge différents degrés de précision pour représenter une date, une heure et la date avec heure, mais de toute évidence il y a des notions de précision plus fines.
- Le méthode *truncatedTo* permet de tronquer une valeur à un champ.

```
@Test
public void test5() {

    LocalDateTime timePoint = LocalDateTime.now();
    LocalDateTime truncatedTime = timePoint.truncatedTo(ChronoUnit.SECONDS);
    System.out.println(timePoint);
    System.out.println(truncatedTime);

}
```

```
16:50:18.191
16:50:18
```

Troncature

Fuseaux horaires

- Les classes locales que nous avons examinées précédemment mettent de côté la complexité qui peut être introduite par les fuseaux horaires.
- Un fuseau horaire est un ensemble de règles, correspondant à une région dans laquelle le temps standard est le même. Il y en a environ 40.
- Les fuseaux horaires sont définis par leur décalage par rapport au temps universel (UTC).
 - Le signe de départ est positif (+) lorsque l'heure locale est en avance sur le temps universel (UTC) (fuseaux horaires à l'Est du méridien horaire de Greenwich : fuseaux horaires d'Europe centrale, Afrique centrale et de l'Est, Asie, Pacifique Ouest et central).
 - Négatif (-) sinon (fuseaux horaires de quelques pays d'Europe occidentale ou d'Afrique occidentale, Amériques, Pacifique Est).
- Ils évoluent d'une façon synchrone et avec une différence constante et connue.
- Les fuseaux horaires peuvent être désignés par deux identifiants, abrégé, par exemple, "PLT", ou long, par exemple, "Asie / Karachi."

Fuseaux horaires

Notes

Pour éviter d'éventuelles confusions, il s'agit bien d'un décalage exprimé *depuis* l'heure UTC (et non pas *vers*). Ce qui est logique puisqu'il s'agit d'une indication de fuseau horaire. Ainsi T09:00:00+01:00 correspond à T08:00:00Z (9 heures dans le fuseau ayant une heure de plus que l'UTC donc 8 heures en UTC). 09:00+01:00 ne représente pas l'addition de 09:00 et de 01:00, ce qui aurait été encore plus logique, mais plutôt cette addition-ci: **09:00 = UTC +01:00**.

Fuseaux horaires

- Lors de la conception d'une application, il faut identifier les scénarios qui nécessitent l'utilisation de fuseaux horaires et quels décalages faut-il appliquer.
- **ZoneId**: est un identifiant pour une région donnée. Chaque ZoneId correspond à certaines règles qui définissent le fuseau horaire pour cet emplacement.

```
id = ZoneId.of("Europe/Paris");  
ZonedDateTime zoned = ZonedDateTime.of(dateTime, id);  
assertEquals(id, ZoneId.from(zoned));
```

- **ZoneOffset**: est la période de temps représentant la différence entre Greenwich / UTC et un fuseau horaire

```
ZoneOffset offset = ZoneOffset.of("+02:00");
```

Fuseaux horaires

Les classes fuseaux horaires

***ZonedDateTime* est une date et heure avec un fuseau horaire complet.**

- La règle de base est que si on veut représenter une date et heure sans s'appuyer sur le contexte d'un serveur spécifique, il faut utiliser *ZonedDateTime* .

```
ZonedDateTime.parse ("2007-12-03T10: 15: 30+01:00[Europe/ Paris]);
```

***OffsetDateTime* est une date et heure avec un décalage résolu.**

- Ceci est utile pour la sérialisation des données dans une base de données.
- Il devrait également être utilisé comme format de sérialisation pour les Timestamp de logging sur des serveurs situés dans des fuseaux horaires différents

Les classes fuseaux horaires

Notes

Les classes fuseaux horaires

OffsetTime est un temps avec un décalage résolu.

```
OffsetTime time = OffsetTime.now();  
// Modifie le décalage en gardant la meme position sur timeline  
OffsetTime sameTimeDifferentOffset = time.withOffsetSameInstant (offset);  
// Modifie le décalage et la position sur timeline  
OffsetTime changeTimeWithNewOffset = time.withOffsetSameLocal (offset);  
changeTimeWithNewOffset.withHour(3).plusSeconds(2);
```

- Il existe déjà une classe de fuseau horaire dans Java, `java.util.TimeZone`, mais elle n'est pas utilisé par Java SE 8 parce qu'elle n'est pas immuable.

Les classes fuseaux horaires

Périodes

- Une période représente une valeur telle que "trois mois et un jour", qui se traduit par une distance sur la timeline.

```
// 3 ans, 2 mois, 1 jour
Period period = Period.of(3, 2, 1);

// Il est possible d'utiliser les périodes pour modifier la date
LocalDate newDate = oldDate.plus(period);
ZonedDateTime newDateTime = oldDateTime.minus(period);

assertEquals(1, period.get(ChronoUnit.DAYS));
```

Périodes

Notes

Chronologies

- Afin de répondre aux besoins des développeurs qui utilisent des systèmes de gestion d'agenda non-ISO, Java SE 8 introduit le concept de chronologie.
- Il représente un système de gestion d'agenda et agit comme une Factory pour les points de temps dans ce système.
- Il y a aussi des interfaces qui correspondent aux classes de points de temps de base, mais sont paramétrées par:
 - *Chronology*:
 - *ChronoLocalDate*
 - *ChronoLocalDateTime*
 - *ChronoZonedDateTime*
- Ces classes sont là uniquement pour les développeurs qui travaillent sur des applications très internationalisées qui doivent prendre en compte les systèmes de gestion d'agendas locaux, et elles ne devraient pas être utilisées par les développeurs n'ayant pas ce genre d'exigences.

Chronologies

Notes

Le reste de l'API

Java SE 8 introduit aussi des classes pour certains autres cas d'utilisation communs.

- La Classe *MonthDay*, contient une paire de Month et Day et est utile pour représenter les anniversaires.
- La Classe *YearMonth* couvre les scénarios et cas d'utilisation liés aux Date de début et la date d'expiration des cartes de crédit (date sans jour spécifié).
- Ces nouveaux types seront supportés par la JDBC de Java SE 8.

Le reste de l'API

Notes

Le reste de l'API

- Ces types peuvent être mappées avec des types de bases de données ou des types ANSI SQL; par exemple, la cartographie ANSI :

Table 1: La cartographie ANSI

ANSI SQL	JAVA SE 8
DATE	LocalDate
TIME	LocalTime
TIMESTAMP	LocalDateTime
TIME WITH TIMEZONE	OffsetTime
TIMESTAMP WITH TIMEZONE	OffsetDateTime

Le reste de l'API

Notes

Nouveautés Java 8

Formatage et parsing de dates

Version 3.0

- Les formats employés
- `DateTimeFormatter`

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Introduction

Possibilités

- `DateTimeFormatter` = pour formater et parser des dates et heures
- Méthodes `parse()` et `format()` réparties dans les différentes classes de date et heure

Formatage

- Standard = prévu plutôt pour l'exploitation par des applications
- Spécifiques = plutôt pour l'exploitation directe par des humains
- 4 styles possibles: `SHORT`, `MEDIUM`, `LONG`, `FULL`

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT

Introduction

Formatage

Standard

- A partir des formats prédéfinis

```
String formatted = DateTimeFormatter.ISO_DATE_TIME.format(apollo11launch);  
// 1969-07-16T09:32:00-05:00[America/New_York]
```

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);  
String formatted = formatter.format(apollo11launch);  
// July 16, 1969 9:32:00 AM EDT
```

Spécifique

- A partir d'un pattern

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

Formatage

Parsing

Avec un formatage standard

- ISO_LOCAL_DATE

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
```

Avec un formatage spécifique

```
ZonedDateTime apollo11launch =  
    ZonedDateTime.parse("1969-07-16 03:32:00-0400",  
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

Parsing

Formats standards prédéfinis

1/2

Formatter	Description	Example
BASIC_ISO_DATE	Year, month, day, zone offset without separators	19690716-0500
ISO_LOCAL_DATE, ISO_LOCAL_TIME, ISO_LOCAL_DATE_TIME	Separators -, :, T	1969-07-16, 09:32:00, 1969-07-16T09:32:00
ISO_OFFSET_DATE, ISO_OFFSET_TIME, ISO_OFFSET_DATE_TIME	Like ISO_LOCAL_XXX, but with zone offset	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00
ISO_ZONED_DATE_TIME	With zone offset and zone ID	1969-07-16T09:32:00-05:00[America/New_York]
ISO_INSTANT	In UTC, denoted by the Z zone ID	1969-07-16T14:32:00Z

Formats standards prédéfinis

Formats standards prédéfinis

2/2

ISO_DATE, ISO_TIME, ISO_DATE_TIME	Like ISO_OFFSET_DATE, ISO_OFFSET_TIME, ISO_ZONED_DATE_TIME, but the zone information is optional	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00[America/New_York]
ISO_ORDINAL_DATE	The year and day of year, for LocalDate	1969-197
ISO_WEEK_DATE	The year, week, and day of week, for LocalDate	1969-W29-3
RFC_1123_DATE_TIME	The standard for email timestamps, codified in RFC 822 and updated to four digits for the year in RFC 1123	Wed, 16 Jul 1969 09:32:00 -0500

Formats standards prédéfinis

Patterns pour formats spécifiques

ChronoField or Purpose	Examples
ERA	G: AD, GGGG: Anno Domini, GGGGG: A
YEAR_OF_ERA	yy: 69, yyyy: 1969
MONTH_OF_YEAR	M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J
DAY_OF_MONTH	d: 6, dd: 06
DAY_OF_WEEK	e: 3, E: Wed, EEEE: Wednesday, EEEEE: W
HOUR_OF_DAY	H: 9, HH: 09
CLOCK_HOUR_OF_AM_PM	K: 9, KK: 09
AMPM_OF_DAY	a: AM
MINUTE_OF_HOUR	mm: 02
SECOND_OF_MINUTE	ss: 00
NANO_OF_SECOND	nnnnnn: 000000
Time zone ID	VV: America/New_York
Time zone name	z: EDT, zzzz: Eastern Daylight Time
Zone offset	x: -04, xx: -0400, xxx: -04:00, XXX: same, but use Z for zero
Localized zone offset	O: GMT-4, OOOO: GMT-04:00

Patterns pour formats spécifiques

Nouveautés Java 8

Java 8 Nashorn

Version 3.0

- Présentation du moteur Nashorn
- Fonctionnalités
- La ligne de commande JJS
- Appel du code Javascript depuis Java et inversement
- Utilisation de l'objet Packages
- Simulation d'import
- Utilisation de JavaImporter et des fonctions anonymes

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

De Rhino à Nashorn

Nashorn est un moteur JavaScript développé en Java par Oracle

- Il est basé sur l'ECMAScript-262, implémenté entièrement en Java et intégré au JDK.
- C'est le successeur de Rhino, le moteur d'exécution JavaScript créé par Mozilla.
- Il est une implémentation légère et haute performance de Javascript.
- Améliorer les performances de Nashorn a aidé à trouver de nouvelles opportunités d'optimisation de la JVM pour les implémentations de langages dynamiques.
- En plus, Nashorn fournit à présent une manière très agréable d'utiliser Java pour JavaScript en donnant un accès simple aux classes Java, permettant à des applications JavaFX entières d'être écrites en JavaScript et de tourner sur la JVM.
- Ce type d'interopérabilité entre langages dynamiques tournant sur la JVM et Java offre une manière efficace d'écrire de telles applications.
- Javascript est exécutable depuis la ligne de commande avec JJS (intégré aux binaires de java 8)
- Il est également accessible au sein du code Java

De Rhino à Nashorn

Notes

Fonctionnalités

Grâce à Nashorn, il est possible de faire interagir un programme Java et un programme Javascript

- Autrement dit, on peut :
 - faire transiter des variables entre ces deux mondes,
 - permettre à Java d'appeler des fonctions Javascript et inversement.
- On peut l'utiliser:
 - A partir de la ligne de commande JJS
 - A partir du code java
 - A partir du code javascript

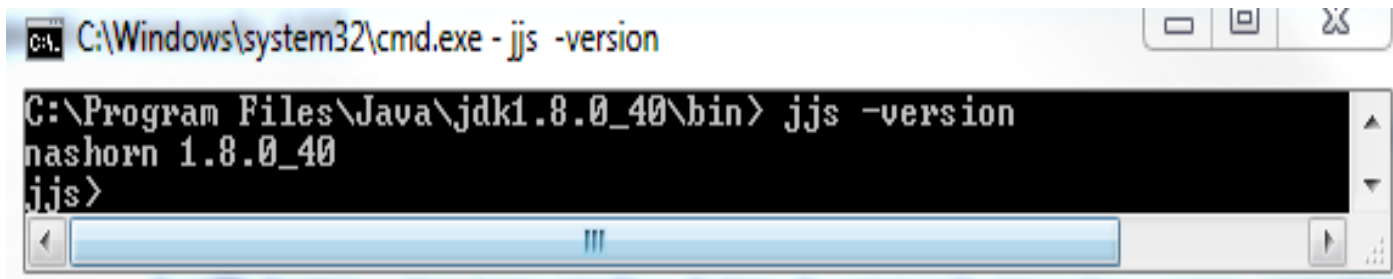
Fonctionnalités

Notes

La ligne de commande JJS

Pour interpréter du Javascript

- Il est possible d'interpréter du js en ligne de commande avec l'utilitaire jjs (Java / JavaScript) disponible dans le jdk/jre.
- Il suffit de taper 'jjs' et une interface de ligne de commande interactive s'ouvre, permettant l'exécution et le test des extraits de code js, sans avoir à les appeler directement depuis du code Java
- Support de la notation shebang `#!/`, permettant ainsi le codage et l'exécution de scripts shell.
- Vérifier que jjs est accessible en ligne de commande:



```
C:\Windows\system32\cmd.exe - jjs -version  
  
C:\Program Files\Java\jdk1.8.0_40\bin> jjs -version  
nashorn 1.8.0_40  
jjs>
```

La ligne de commande JJS

Notes

La ligne de commande JJS

```
jjs [<options>] <files> [-- <arguments>]
  -D <-Dname=value. Set a system property. This option can be repeated.>
  -cp, -classpath <-cp path. Specify where to find user class files.>
  -doe, -dump-on-error <Dump a stack trace on errors.>
    param: [true|false]  default: false
  -fv, -fullversion <Print full version info of Nashorn.>
    param: [true|false]  default: false
  -fx <Launch script as an fx application.>
    param: [true|false]  default: false
  -h, -help <Print help for command line flags.>
    param: [true|false]  default: false
  --language <Specify ECMAScript language version.>
    param: [es5|es6]    default: es5
  -ot, --optimistic-types <Use optimistic type assumptions with deoptimizing recompilation.
    This makes the compiler try, for any program symbol whose type cannot
    be proven at compile time, to type it as narrow and primitive as
    possible. If the runtime encounters an error because symbol type
    is too narrow, a wider method will be generated until steady stage
    is reached. While this produces as optimal Java Bytecode as possible,
    erroneous type guesses will lead to longer warmup. Optimistic typing
    is currently disabled by default, but can be enabled for significantly
    better peak performance.>
    param: [true|false]  default: false
  -scripting <Enable scripting features.>
    param: [true|false]  default: false
  -strict <Run scripts in strict mode.>
    param: [true|false]  default: false
  -t, -timezone <Set timezone for script execution.>
    param: <timezone>    default: Europe/Paris
  -v, -version <Print version info of Nashorn.>
    param: [true|false]  default: false
```

La ligne de commande JJS

Notes

Les possibilités de JJS

Fonctionnement de type REPL (Read-Eval-Print Loop)

- Evaluation et affichage de la valeur d'une expression

```
jjs> 'Hello, World!'.length  
13
```

- Définition de fonctions et invocation

```
jjs> function factorial(n) { return n <= 1 ? 1 : n * factorial(n - 1) }  
function factorial(n) { return n <= 1 ? 1 : n * factorial(n - 1) }  
jjs> factorial(10)  
3628800
```

- Appel de méthodes Java

```
var input = new java.util.Scanner(  
    new java.net.URL('http://horstmann.com').openStream())  
input.useDelimiter('$')  
var contents = input.next()
```

- La saisie de contents affiche le contenu de la page correspondant à l'URL

Les possibilités de JJS

Notes

Les possibilités de JJS

Interpréter des fichiers javascript

- Créer un fichier .js contenant le code Javascript

```
print('Hello Nashorn!');
```

- Ouvrir la console et taper la commande suivante

```
>jjs hello.js
```

- Résultat :

```
Hello Nashorn!
```

Les possibilités de JJS

Notes

Les possibilités de JJS

Passer des arguments

- Exemple
 - Ouvrir la console et taper la commande suivante

```
> jjs -- a b c  
  
jjs> print('letters: ' +arguments.join(", "))
```

- Résultat :

```
letters: a, b, c  
jjs>
```

Les possibilités de JJS

Notes

Javascript depuis Java

Avec l'API de Scripting (JSR 223, Java 6)

- Accès au moteur JavaScript Nashorn inclus dans Java 8
- Evaluation de code Javascript

```
ScriptEngineManager manager = new ScriptEngineManager();  
ScriptEngine engine = manager.getEngineByName("nashorn");  
Object result = engine.eval("'Hello, World!'.length");  
System.out.println(result);
```

Javascript depuis Java

Notes

- Pour récupérer une instance de `ScriptEngine`, il est aussi possible de passer “JavaScript” ou “ECMAScript” en paramètre de la méthode `getEngineByName()`, à la place de “nashorn”. Le résultat sera identique.
- La méthode `eval()` quant à elle évalue la chaîne de caractère et l’exécute.

Javascript depuis Java

Appel de fonctions définies dans un fichier .js

```
30 @Test
31 public void test2() throws FileNotFoundException, ScriptException, NoSuchMethodException {
32
33
34     ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
35     engine.eval(new FileReader("resources\\script.js"));
36     try {
37         Invocable invocable = (Invocable) engine;
38
39         Object result = invocable.invokeFunction("fun1", "Peter Parker");
40         System.out.println(result);
41         System.out.println(result.getClass());
42     }
43     catch (ScriptException e) {
44     }
```

- Nashorn supporte l'invocation des fonctions javascript définies dans un fichier de scripts directement à partir du code java.
- On peut passer des objets Java comme arguments et renvoyer le résultat de la fonction à la méthode java appelante.

Javascript depuis Java

Notes

- Pour appeler une fonction on doit d'abord convertir le type 'ScriptEngine' en 'Invocable'.
- L'interface 'invocable' est implémentée par Nashorn ScriptEngine et définit une fonction 'invokeFunction' pour appeler une fonction javascript d'un nom donné.

Javascript depuis Java

Appel de fonctions définies dans un fichier .js

- Ce code Java utilisé avec la fonction Javascript ci-dessous

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("resources\\script.js"));
try {
    Invocable invocable = (Invocable) engine;

    Object result = invocable.invokeFunction("fun1", "Peter Parker");
    System.out.println(result);
    System.out.println(result.getClass());
}
```

```
var fun1 = function(name) {
    print('Hi there from Javascript, ' + name);
    return "greetings from javascript";
};
```

- produit ces affichages

```
<terminated> JsTets.test2 [JUnit] C:\Program Files\Java\jre1.8.0_40\bin\javaw.exe (15 avr. 2015 11:25:06)
Hi there from Javascript, Peter Parker
greetings from javascript
class java.lang.String
```

Javascript depuis Java

Notes

Javascript depuis Java

Appel de fonctions définies dans un fichier .js

- Passage de paramètres Java aux fonctions Javascript

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("./test.js"));
Invocable invocable = (Invocable) engine;

invocable.invokeFunction("fun2", new Date());

invocable.invokeFunction("fun2", LocalDateTime.now());

invocable.invokeFunction("fun2", new Test());
```

```
var fun2 = function (object) {
    print("JS Class Definition: " + Object.prototype.toString.call(object));
};
```

- Affiche

```
<terminated> Test [Java Application] C:\Program Files\Java\jre1.8.0_45\
JS Class Definition: [object java.util.Date]
JS Class Definition: [object java.time.LocalDateTime]
JS Class Definition: [object javascript.Test]
```

Javascript depuis Java

Notes

- Les objets Java peuvent être passés en paramètres sans perdre aucune information de type du côté javascript.
- Depuis que le script s'exécute nativement sur la machine virtuelle Java, on peut utiliser toute la puissance de l'API Java ou des bibliothèques externes sur nashorn.

Java depuis Javascript

Java.type()

- Permet "d'importer" un type Java au sein d'un script JS
- Exemple

```
public class Test2 {  
    public static class Truc {  
        private int x;  
  
        public Truc(int x) {  
            this.x = x;  
        }  
        public int getX() {  
            return x;  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");  
        engine.eval(new FileReader("./test2.js"));  
        Invocable invocable = (Invocable) engine;  
  
        System.out.println(invocable.invokeFunction("test", 100));  
    }  
}
```

```
4 var test = function(val) {  
5     print('objet de type java avec param ' + val);  
6     var Truc = Java.type("javascript.Test2.Truc");  
7     var truc = new Truc(val);  
8     print(truc.getX());  
9     return "ok";  
10 };
```

```
<terminated> Test2 (1) [Java Application] C:\P  
objet de type java avec param 100  
100  
ok
```

Java depuis Javascript

Notes

Java depuis Javascript

Java.type()

- Il est possible d'importer tous les types Java au sein d'un script
- Attention aux conflits de noms avec les types Javascript

```
var URL = Java.type('java.net.URL')
var JMath = Java.type('java.lang.Math')
// Avoids conflict with JavaScript Math object
```

- Pas obligatoire pour les packages commençant par java, javax, javafx, com, org et edu
- Alternative
 - Packages, employé ci-dessous dans le même contexte que l'exemple précédent

```
4 var test = function(val) {
5     print('objet de type java avec param ' + val);
6     var Truc = Packages.javascript.Test2.Truc;
7     var truc = new Truc(val);
8     print(truc.getX());
9     return "ok";
10 };
```

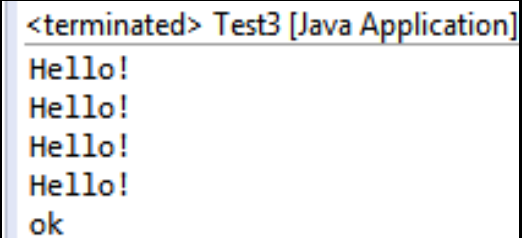
Java depuis Javascript

Notes

Java depuis Javascript

Packages

```
var test = function() {  
    var java = Packages.java;  
  
    var task = new java.util.TimerTask() {  
        run: function() {  
            print('Hello!');  
        }  
    }  
  
    var timer = new java.util.Timer();  
    timer.schedule(task, 0, 1000);  
    java.lang.Thread.sleep(3000);  
    timer.cancel();  
  
    return "ok";  
};
```



```
<terminated> Test3 [Java Application]  
Hello!  
Hello!  
Hello!  
Hello!  
ok
```

Java depuis Javascript

Notes

Utilisation de l'objet `JavaImporter`

Permet d'utiliser des types Java sans préciser leurs noms complets

- Exemple

```
var test = function() {  
    var pkgs = new JavaImporter(java.lang, java.util);  
    with(pkgs) {  
        var task = new TimerTask() {  
            run: function() {  
                print('Hello!');  
            }  
        }  
        var timer = new Timer();  
        timer.schedule(task, 0, 1000);  
        Thread.sleep(3000);  
        timer.cancel();  
    }  
  
    return "ok";  
};
```

Utilisation de l'objet `JavaImporter`

Notes

Utilisation de fonctions anonymes

- Le JavaScript permet d'utiliser des fonctions anonymes sans la contrainte d'interfaces fonctionnelles
- Nous pouvons remplacer la redéfinition de la méthode run() de la classe TimerTask ,qui est une classe abstraite par une fonction anonyme.

```
var java = Packages.java;
var TimerTask = java.util.TimerTask;
var Timer = java.util.Timer;
var Thread = java.lang.Thread;

var timer = new Timer();
timer.schedule(
    function() {
        print('Hello!');
    }, 0, 1000);

Thread.sleep(10000);
timer.cancel();
```

Utilisation de fonctions anonymes

Notes

Nouveautés Java 8

Introduction au modèle MapReduce

Version 3.0

- Présentation
- Exemples

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Le modèle MapReduce

Principe

- Pour exécuter un problème large de manière distribuée, il faut pouvoir le découper en plusieurs problèmes de taille réduite, souvent appelés « tâches »
- Chaque tâche est alors exécutée sur une machine d'un cluster
- MapReduce est un paradigme (un modèle) visant à généraliser les approches existantes pour produire une approche unique applicable à tous les problèmes
- MapReduce existe depuis longtemps, notamment dans les langages fonctionnels (Lisp, Scheme)

2004 : « MapReduce: Simplified Data Processing on Large Clusters »

- La présentation du modèle généralisable à tous les problèmes et orientée calcul distribué est attribuable à un *whitepaper* issu du département de recherche de Google publié en 2004 : « MapReduce: Simplified Data Processing on Large Clusters »

Le modèle MapReduce

Notes

Deux opérations distinctes

MAP

- Transforme les données d'entrée en une série de couples clef/valeur
- Regroupe les données en les associant à des clefs (les couples clef/valeur doivent avoir un sens par rapport au problème à résoudre)

Cette opération doit être parallélisable : il doit être possible de découper les données d'entrée en plusieurs fragments et faire exécuter l'opération MAP sur chaque machine du cluster sur un fragment distinct.

REDUCE

- Applique un traitement à toutes les valeurs de chacune des clefs distinctes produites par l'opération MAP
- Au terme de cette opération, chaque clef distincte aura un résultat

Chaque machine du cluster effectue l'opération REDUCE sur une des clefs uniques en utilisant la listes des valeurs associées.

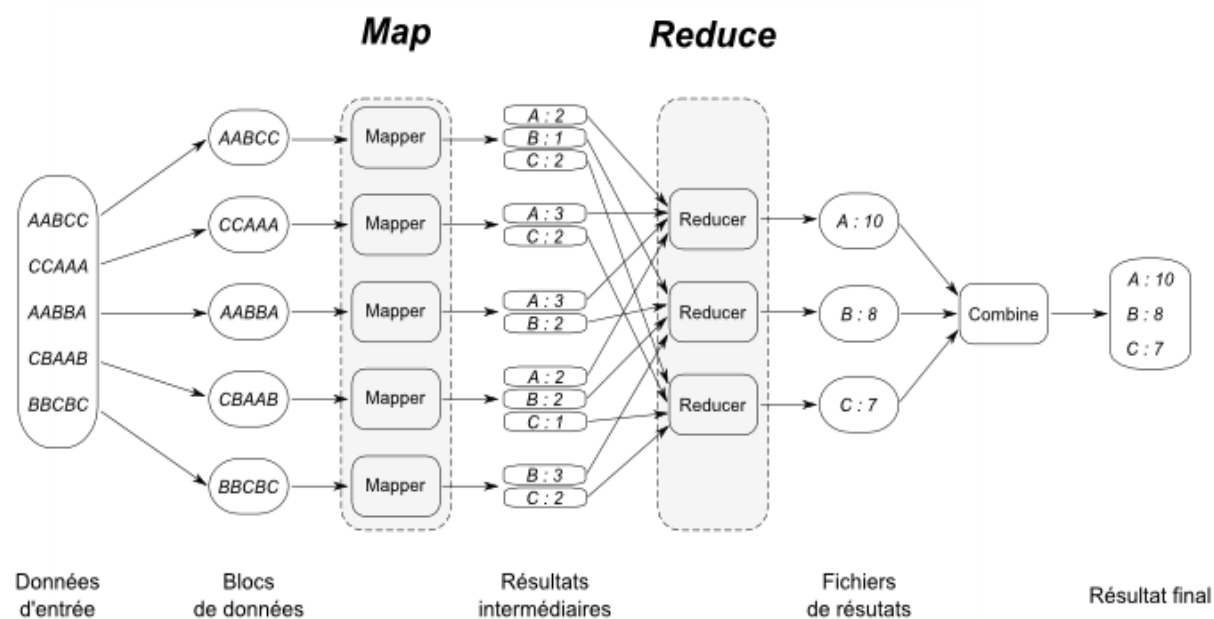
Deux opérations distinctes

Notes

Quatre étapes

- **Découper** (*split*) les données d'entrée en plusieurs fragments
- **Mapper** chacun de ces fragments pour obtenir des couples clef/valeur
- **Grouper** (*shuffle*) ces couples clef/valeur par clef
- **Réduire** (*reduce*) les groupes indexés par clef avec une valeur pour chacune des clefs distinctes

En modélisant le problème à résoudre de la sorte, on le rend parallélisable



Quatre étapes

Notes

Map

Architecture de type maître-esclave

- Un Noeud analyse un problème
- Le découpe en sous-problèmes
- Les délègue à d'autres noeuds (qui peuvent en faire de même récursivement)

```
Map(void * document){  
    int cles = 1;  
    for each mot m in document  
        calculIntermediaire(m,cles);  
}
```

- Les sous-problèmes sont traités par Reduce

Map

Notes

Reduce

- Associe à un couple (clé, valeur) un ensemble de nouveaux couples (clé, valeur)

```
Reduce(entier cles, Iterator values){  
    int result = 0;  
    for each v in values  
        result += v;  
}
```

- Les noeuds les plus bas font remonter leurs résultats à leur noeud parent
- Celui-ci calcule un résultat partiel
- Il remonte l'information à son tour

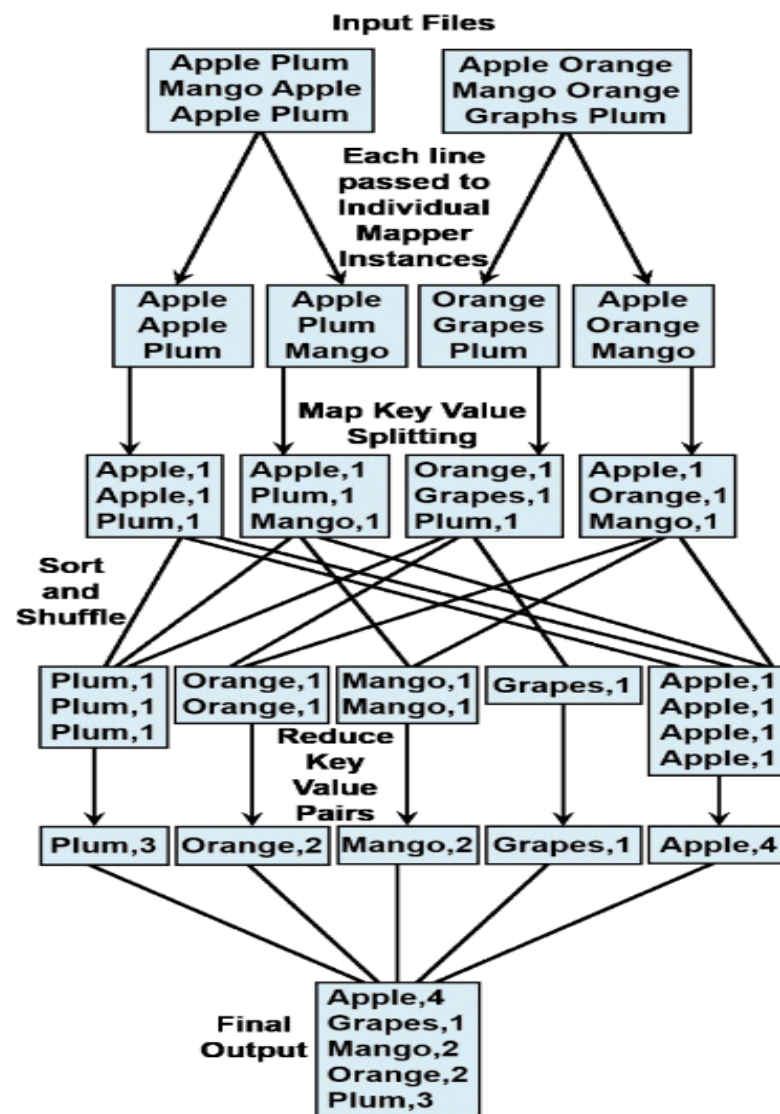
Reduce

Notes

Exemple

Compter le nombre d'occurrences de mots dans une liste

- Sous-ensembles traités simultanément par différents noeuds



Exemple

Notes

Nouveautés Java 8

API Stream

Version 3.0

- Définition et intérêt
- Opérations sur un stream

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Qu'est ce qu'un Stream

Séquence d'éléments provenant d'une source et qui

- Ne stocke pas les données.
- Ne modifie pas les données de la source sur laquelle il est construit.
- Source de données peut être infinie ou bornée.
- Qui réalise des opérations de manière Lazy (le plus tard possible, uniquement si le résultat est souhaité)

Objectifs

- Disposer d'un moyen élégant pour opérer des traitements sur un ensemble de données
- De façon séquentielle ou parallèle
- En faisant abstraction des détails d'implémentation

Qu'est ce qu'un Stream

Notes

- Ne stocke pas les données: Un stream ne contient pas d'objets, il se base sur une source de données (collections, socket, fichier...).
- Ne modifie pas les données de la source sur laquelle il est construit :
- N'est pas réutilisable. Lorsqu'on essaye de réutiliser un stream sur lequel on a appliqué une opération terminale on obtiendra une EXCEPTION : (**Exception in thread "main" [java.lang.IllegalStateException: stream has already been operated upon or closed](#)**).
- Réalise des opérations de manière Lazy : le traitement des opérations se fait sans stockage intermédiaire.

Intérêt

Simplifier l'écriture du code

- Exemple: comptage de tous les mots longs (plus de 12 lettres) d'une liste de chaîne de caractères (words)
- sans les streams

```
int count = 0;
for (String w : words) {
    if (w.length() > 12) count++;
}
```

- avec les streams

```
long count = words.stream().filter(w -> w.length() > 12).count();
```

- L'écriture bénéficie également des lambda-expressions et du chaînage des opérations de Java 8

Intérêt

Notes

- words est de type `List<String>`

Intérêt

Parallélisation des traitements

- Simple

```
long count = words.parallelStream().filter(w -> w.length() > 12).count();
```

- Respecte le principe "What, not how" de Java 8
 - on décrit ce que l'on veut faire: "obtenir les mots longs et les compter, de la façon la plus efficace possible"
 - pas le comment: l'ordre, la création des threads, ...

Mais

- Il peut quand même être utile de savoir combien de threads vont être mis à contribution par la JVM dans chaque cas ...

Intérêt

Notes

Principes d'utilisation d'un stream

Pipeline en trois étapes

- Création du stream à partir d'une source de données
- Définitions d'opérations intermédiaires de transformation
- Exécution d'une opération finale d'obtention d'un résultat, qui force l'exécution des opérations "lazy"
 - à l'issue de cette opération finale, le stream n'est plus utilisable

Sur l'exemple précédent

```
long count = words.stream().filter(w -> w.length() > 12).count();
```

- création = stream()
- transformations = filter()
- opération finale d'obtention du résultat = count()
 - cette opération déclenche l'exécution effective du pipeline

Principes d'utilisation d'un stream

Notes

Création d'un Stream

A partir d'une collection, d'un tableau, d'un fichier

- A partir d'une collection:

```
Collection<String> collection = ...;  
Stream<String> stream = collection.stream( );  
  
// Pour une collection de type List on a :  
List etudiants = ...;  
Stream str = etudiants.stream( );
```

- A partir d'un tableau :

```
Stream<String> stream = Array.stream(new String [ ] { "un", "deux", "trois" });
```

- A partir des méthodes factory de Stream

```
Stream<String> stream = Stream.of("un", "deux", "trois" );
```

Création d'un Stream

Notes

- De nombreuses classes de base ont été modifiées afin de permettre la création de Streams
- Dans cette exemple, la méthode **stream()** est une nouvelle méthode ajoutée à l'interface Collection.
- On peut également créer un stream avec la méthode `parallelStream()`. Il existe plusieurs autres méthodes permettant la création de Stream. Exemple :
 - `lines()` de la classe `BufferedReader` qui permet de créer un stream de chaîne de caractères à partir des lignes du fichier ouvert.
 - `ints()` : de la classe `Random` permet d'obtenir un stream d'entiers pseudo aléatoires.

Opérations sur un Stream

forEach()

- Prend un Consumer<T> en paramètre et ne retourne rien

```
List<Etudiant> etudiants = ...;
```

```
Stream<Etudiant> stream = etudiants.stream( );  
stream.forEach( e -> System.out.println(e));
```

//Permet d'afficher chaque étudiant à la console

peek()

- Prend Consumer<T> en paramètre et retourne le résultat

```
List<String> resultat = new ArrayList<>();  
List<Etudiant> etudiants = ...;
```

```
etudiants.stream( )  
    .peek(System.out :: println)  
    .filter(etudiant->etudiant.getAge()>20)  
    .peek(resultat :: add);
```

Opération sur un Stream

Notes

Opération sur un Stream

Opérations intermédiaires

- Opérations qui gardent le stream ouvert et permettent d'effectuer d'autres opérations.
- `filter()` et `map()`
- Exemple

```
List<Etudiant> etudiants = ...;  
  
Stream<Etudiant> stream = etudiants.stream( );  
Stream<Integer> mapAge = stream.map(x->x.getAge());  
  
Stream<Integer> filtreMajeur = mapAge.filter(age-> age>20);
```

Opération terminale

- Opération finale effectuée sur un stream.
- `reduce()`

```
Stream<Etudiant> stream = etudiants.stream( );  
Integer somme = filtreMajeur.reduce(0, (x,y)-> x+y);
```

Opération sur un Stream

Opérations intermédiaires

Exemple

- Pour la liste étudiants suivante: : (Déterminons la somme d'âge des étudiants ayant plus de 20 ans)
[nom=victorien, age=20], [nom=fabrice, age=30], [nom=laurent, age=25],
[nom=alain, age=18], [nom=jean, age=15], [nom=max, age=22]

map ()

- On crée un stream qui ne contient que l'âge des étudiants
- mapAge Contient : 20 30 25 18 15 22

filter()

- On ne conserve que les âges supérieurs à 20.
- filtreMajeur contient : 30 25 22

reduce()

- On applique la somme des âges.
- sum contient : 77

Nouveautés Java 8

Utiliser map-reduce avec Java 8

Version 3.0

- Quelques exemples

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Principes

Utilisation conjointe de plusieurs caractéristiques Java 8

- Lambda-expressions
 - simplifient les manipulations de collections de données
- Streams
 - fournissent un moyen simple de communiquer des flux de données entre algorithmes

Principes

Notes

Map

Exemple

- Appliquer une opération à un ensemble de données
- ici afficher un prix TTC à partir d'un prix HT extrait d'une liste

Sans Java 8

```
List prixHT = Arrays.asList(100, 200, 300, 400, 500);  
for (Integer prix : prixHT) {  
    double prixTTC = prixHT + .20 * prix;  
    System.out.println(prixTTC);  
}
```

Avec Java 8

```
List prixHT = Arrays.asList(100, 200, 300, 400, 500);  
prixHT.stream().map((prix) -> prix + .20*prix).forEach(System.out::println);
```

Map

Notes

Map Reduce

Exemple

- Appliquer une transformation à un ensemble de données
- Puis effectuer un second traitement de type "agrégat" (somme, max, min, ...),
- Les données entre les deux traitements sont communiquées par un stream

Sans Java 8

```
List prixHT = Arrays.asList(100, 200, 300, 400, 500);
double totalTTC = 0;
for (Integer prix : prixHT) {
    double prixTTC = prixHT + .20 * prix;
    totalTTC += prixTTC;
}
```

Avec Java 8

```
List prixHT = Arrays.asList(100, 200, 300, 400, 500);

double totalTTC
= prixHT.stream().map((prix) -> prix + .20*prix).reduce((sum, prix) -> sum + prix).get();
```

Map Reduce

Notes

Nouveautés Java 8

Mécanismes avancés des streams

Version 3.0

- Parallélisation
- Contrôle du nombre de threads utilisés par les streams parallèles
- Problèmes à éviter

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Parallélisation des traitements

parallelStream()

- Simple

```
long count = words.parallelStream().filter(w -> w.length() > 12).count();
```

- Respecte le principe "What, not how" de Java 8
 - on décrit ce que l'on veut faire: "obtenir les mots longs et les compter"
 - pas le comment: l'ordre, la création des threads, ...
- Parallélisation également disponible en utilisant `Stream::parallel()`

Mais

- Il peut quand même être utile de savoir combien de threads vont être mis à contribution par la JVM dans chaque cas ...
- Qui dit thread dit concurrences d'accès, ordonnancement, ...

Parallélisation des traitements

Parallélisation des traitements

A discrétion de l'implémentation de la JVM utilisée

- `Collection::parallelStream()`

parallelStream

```
default Stream<E> parallelStream()
```

Returns a possibly parallel Stream with this collection as its source. It is allowable for this method to return a sequential stream.

This method should be overridden when the `spliterator()` method cannot return a spliterator that is IMMUTABLE, CONCURRENT, or *late-binding*. (See `spliterator()` for details.)

Implementation Requirements:

The default implementation creates a parallel Stream from the collection's Spliterator.

Returns:

a possibly parallel Stream over the elements in this collection

Since:

1.8

Parallélisation des traitements

Contraintes

- Les opérations doivent être stateless
- Les opérations doivent être thread-safe
- Le résultat que l'on souhaite obtenir doit être le même, qu'il soit calculé séquentiellement ou en parallèle

Exemple à ne pas faire

```
int[] shortWords = new int[12];
words.parallel().forEach(
    s -> { if (s.length() < 12) shortWords[s.length()]++; });
    // Error-race condition!
System.out.println(Arrays.toString(shortWords));
```

- la variable shortWords est modifiée simultanément par plusieurs threads
- le résultat obtenu va dépendre des conditions d'exécution et de l'ordonnancement des threads par la JVM

Contraintes

Liens avec les threads

JDK Oracle

- Un Stream parallèle exploite:
 - le thread courant
 - et si nécessaire les threads du pool par défaut obtenu par:
- Taille de ce pool
 - par défaut = nombre de coeurs logiques - 1
 - modifiable au lancement de la JVM
 - A adapter aux besoins de l'application !

```
ForkJoinPool.common()
```

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=8
```

Liens avec les threads

Liens avec les threads

JDK Oracle

- Attention à la durée des traitements intermédiaires réalisés à partir d'un stream parallèle
- Exemple

```
long a = IntStream.range(0, 100).mapToLong(x -> {  
    for (int i = 0; i < 100_000_000; i++) {  
        System.out.println("X:" + i);  
    }  
    return x;  
}).sum();
```

- chaque exécution de la lambda-expression monopolise un thread du pool
- au détriment du reste de l'application ...

Liens avec les threads

Utiliser un pool de threads spécifique

JDK Oracle

- Instancier ForkJoinPool

```
ForkJoinPool myPool = new ForkJoinPool(8);  
myPool.submit(() ->  
    list.parallelStream().forEach(/* Do Something */);  
).get();
```

Utiliser un pool de threads spécifique

Problèmes à éviter

Réutiliser un stream

```
1 | IntStream stream = IntStream.of(1, 2);  
2 | stream.forEach(System.out::println);  
3 |  
4 | // That was fun! Let's do it again!  
5 | stream.forEach(System.out::println);
```

Modifier la collection sur laquelle le stream est construit

```
1 | list.stream()  
2 |     // remove(Object), not remove(int)!  
3 |     .peek(list::remove)  
4 |     .forEach(System.out::println);
```

Problèmes à éviter

Problèmes à éviter

Utiliser un traitement qui n'a pas de fin

- Exemple 1

```
1 // Will run indefinitely
2 IntStream.iterate(0, i -> i + 1)
3     .forEach(System.out::println);
```

- Exemple 2

```
1 IntStream.iterate(0, i -> ( i + 1 ) % 2)
2     .distinct()
3     .limit(10)
4     .forEach(System.out::println);
```

Problèmes à éviter

Nouveautés Java 8

Les améliorations JDBC

Version 3.0

- API JDBC 4.2

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

API JDBC

Présentation

- JDBC : Java Database Connectivity
- API pour permettre un accès aux bases de données avec Java
- Se compose de deux Packages : `java.sql` et `javax.sql`
- Pour l'utiliser il faut un pilote qui est spécifique à la base de données à laquelle on veut accéder.
- Version 4.1 intégrée à Java 7

Modification dans Java 8

- Suppression du pont JDBC-ODBC
- Ajout de l'interface : `java.sql.DriverAction`
- Prise en compte de la nouvelle API Date Time
- Support de REF_CURSOR (procédures stockées)
- Remplacement de Types par JDBCType

API JDBC

Modification dans java 8

Suppression du pont JDBC-ODBC

- Oracle recommande d' utiliser des pilotes JDBC fournis par le fournisseur de votre base de données à la place du pont JDBC - ODBC .

interface `java.sql.DriverAction`

- Interface à implémenter lorsque le DriverManager veut notifier le Driver.

Impacts de Date Time

Conversion de et vers les classes de java.time

- java.sql.Date de/vers LocalDate
- java.sql.Time de/vers LocalTime
- java.sql.Timestamp de/vers LocalDateTime

Impacts de Date Time

Statement::executeLargeUpdate

Pour exécuter un update massif

- Nombre de lignes dépassant Integer.MAX_VALUE
- La méthode retourne un long

Statement::executeLargeUpdate

Nouveautés Java 8

Les nouveaux outils

Version 3.0

- Jdeps
- javapackager
- Java Mission Control

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Jdeps

Présentation

- Outils permettant d'analyser les dépendances de classes liées à un package ou une classe.

Syntaxe

- `jdeps [options] classes ...`
- `[options]` : une ou plusieurs options peuvent être utilisées
- exemples : `-doutput`, `-s`, `-v`, `-cp`, `-p` ...

Exemple

- Analyser les dépendances d'un jar nommé Exemple.jar

```
$ jdeps Exemple.jar
Exemple.jar -> /usr/java/jre/lib/rt.jar
<unnamed> (Exemple.jar)
  -> java.awt
  -> java.awt.event
  -> java.beans
  -> java.io
  -> java.lang
  -> java.net
  -> java.util
```

Jdeps

Jdeps

- Utilisation de l'option -P (ou -profile) pour analyser les profils dont le jar dépend :

```
$ jdeps -P Exemple.jar
```

```
Exemple.jar -> /usr/java/jre/lib/rt.jar (Full JRE)
```

```
<unnamed> (Exemple.jar)
```

-> java.awt	Full JRE
-> java.awt.event	Full JRE
-> java.beans	Full JRE
-> java.io	compact1
-> java.lang	compact1
-> java.net	compact1
-> java.util	compact1

Jdeps

javapackager

Présentation

- Outils qui effectue les tâches relatives au packaging et la signature des applications Java et JavaFX.
- Anciennement appelé javafxpackager.

Syntaxe

- `javapackager command [options]`
- `command` : La tâche qui doit être effectuée.
- `[options]` : Une ou plusieurs options pour la commande séparés par des espaces.
- exemples de commandes : `-createbss`, `-createjar`, `-deploy`, `-makeall`, `-sinjar` ...
- exemples d'options : `-outdir dir`, `-srcdir dir`, `-srcfiles files`, -

Exemples

- 1 - Création d'un jar:

```
javapackager -createjar -appclass package.ClassName -srcdir classes -outdir out -outfile outjar -v
```

javapackager

javapackager

- 2- Signer un jar:

```
javapackager -signJar --outdir dist -keyStore sampleKeystore.jks -storePass ****  
-alias duke -keypass **** -srcdir dist
```

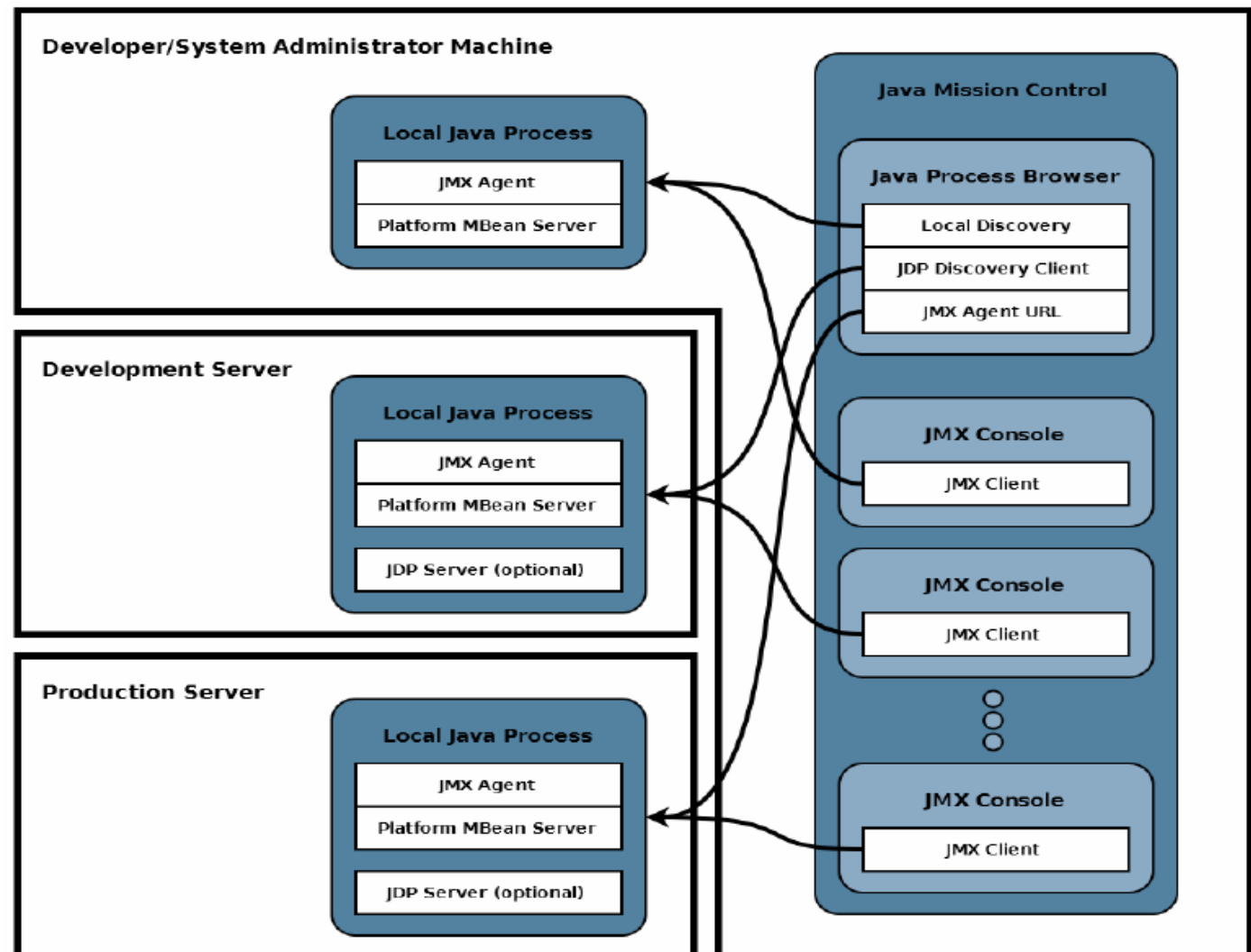
- Signe tous les fichiers JAR dans le répertoire dist , utilisant un certificat avec l' alias spécifié , keyStore et Storepass , et met les fichiers JAR signés dans le répertoire dist .

javapackager

Java Mission Control

Suite d'outils

- Monitoring JVM
- Fourni par Oracle
- Impact minimisé
- Console graphique



Java Mission Control

- L'outil est présent dans le répertoire bin du JDK (jmc.exe)

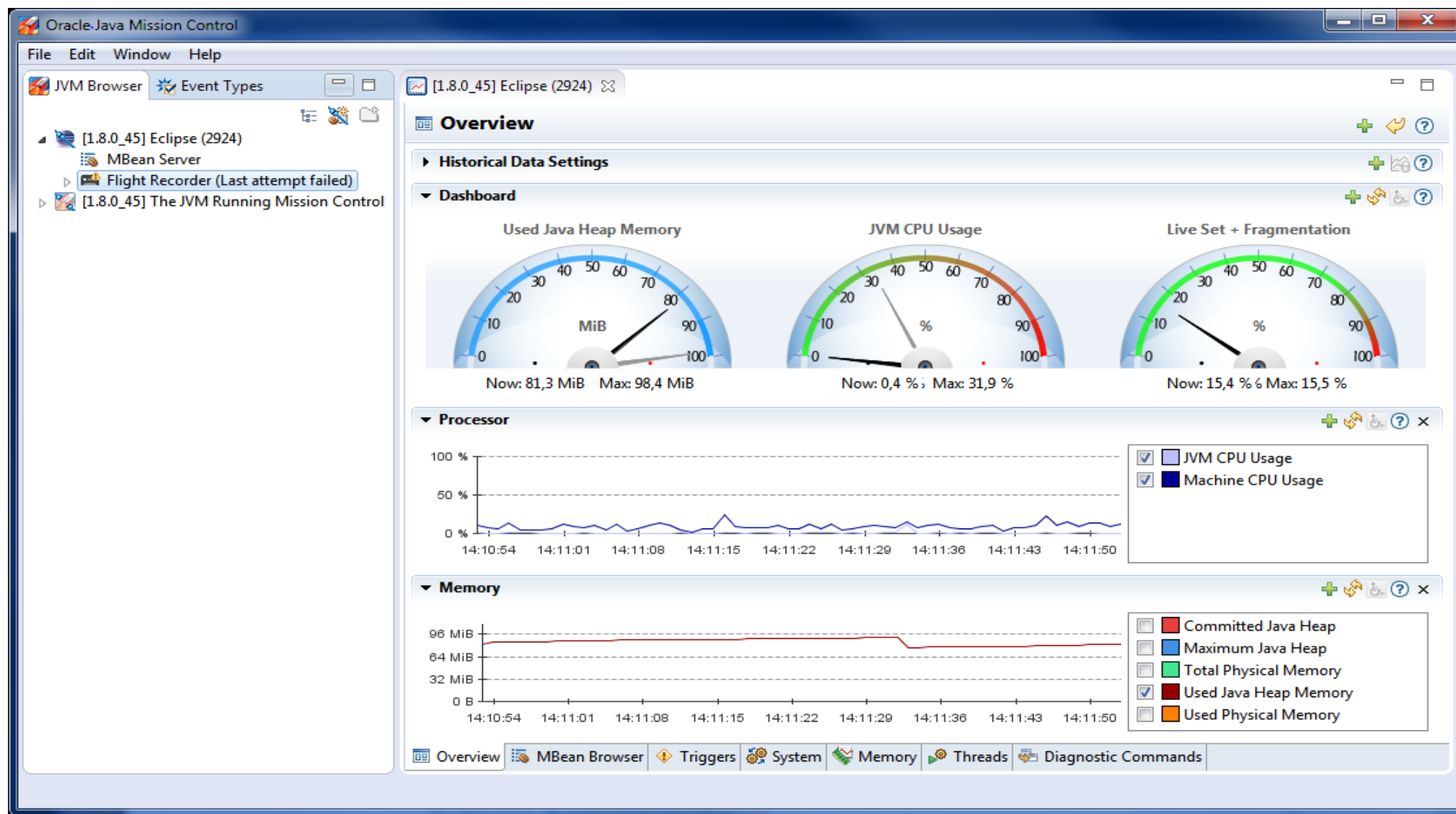
Java Mission Control

Fonctionnalités principales

- Connexion au MBean Server
- Représentation graphique des principaux indicateurs (mémoire, cpu, threads, ...)
- Enregistrement d'informations avec Flight Recorder, avec surcoût estimé à 2%
 - temps de latence des threads
 - entrées-sorties
 - execution du GC
 - profiling d'appels de méthodes
 -

Java Mission Control

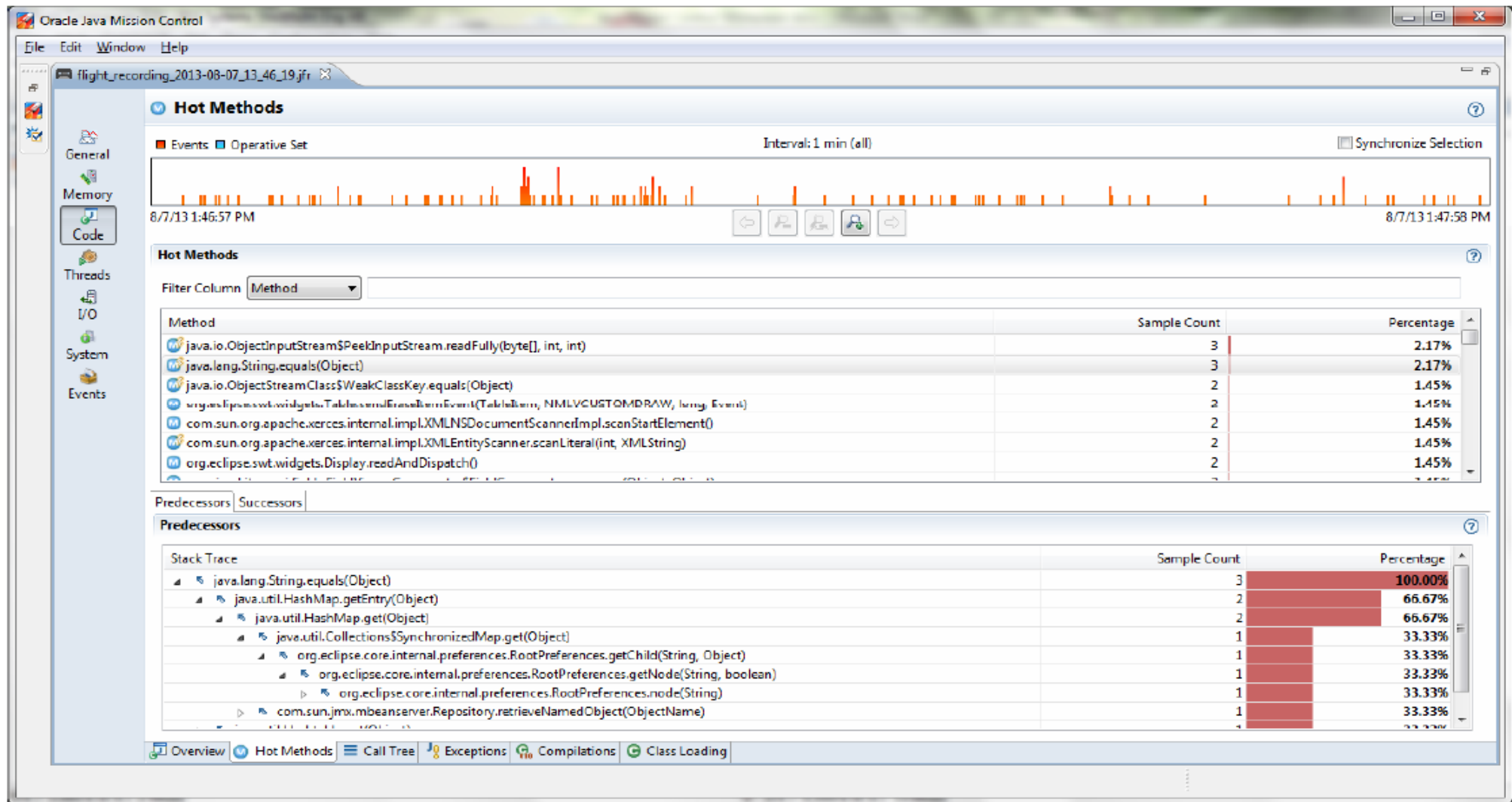
Java Mission Control



Java Mission Control

Java Mission Control

Profiling



Java Mission Control

Nouveautés Java 8

Annexes

Version 3.0

- Date et Heure selon ISO 8601
- Classes anonymes
- Javascript

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Nouveautés Java 8

Dates et heures avec ISO 8601

Version 3.0

- Vue d'ensemble de la norme

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Introduction

Norme internationale de représentation de la date et de l'heure

- Basée sur le calendrier grégorien et le système horaire de 24 heures
- Créée en 1988, mise-à-jour ensuite
- Permet de limiter le risque de confusion dans les communications internationales
- Conçue en vue d'une utilisation informatique
- Classement par ordre lexicographique correspondant à l'ordre chronologique

<http://www.iso.org/iso/fr/home/standards/iso8601.htm>

La date ISO courante :

2015-11-10

L'heure ISO courante :

07:57:10Z

La date et l'heure ISO courantes :

2015-11-10T07:57:10Z

ou encore :

2015-W46-2T07:57:10Z

Introduction

Norme

Contenu de la norme

- La norme traite de la représentation des éléments suivants :
 - Date
 - Heure du jour
 - Temps Universel Coordonné (UTC)
 - Heure locale avec décalage horaire UTC
 - Date et heure
 - Intervalles de temps
 - Intervalles de temps périodiques

Norme

Hypothèses de base

Numérotation

- La semaine commence le lundi
- Les jours de chaque semaine sont numérotés de 1 (un) pour le lundi, à 7 (sept) pour le dimanche
- La semaine 1 est celle qui contient le premier jeudi de l'année
- Une année ISO comporte 52 ou 53 semaines entières
- Une éventuelle semaine additionnelle (intercalaire) peut être ajoutée
- Ce système a un cycle de 400 années, soit 146 097 jours (20 871 semaines), avec une année de longueur moyenne d'exactly 365,2425 jours (identique au calendrier grégorien)

Hypothèses de base

Conséquences

La semaine 1 est la première à contenir la majorité de ses jours (au moins 4 jours) dans l'année. Elle contient systématiquement le 4 janvier. Elle contient systématiquement le premier jour ouvré de l'année en considérant que le 1er janvier, les samedis et les dimanches sont chômés. C'est aussi celle dont le lundi est le plus proche du jour de l'an. Elle commence au plus tôt le 29 décembre ou au plus tard le 4 janvier ;

La dernière semaine de l'année (52 ou 53) est celle qui contient le dernier jeudi de l'année. Elle est la dernière à avoir la majorité de ses jours (au moins 4) dans l'année et contient systématiquement le 28 décembre. C'est aussi celle dont le dimanche est le plus proche du 31 décembre. Elle termine au plus tôt le 28 décembre ou au plus tard le 3 janvier ;

Il n'y a pas de semaine zéro (la semaine 1 d'une année succède immédiatement à la dernière semaine de l'année précédente) ;

Les premiers jours de la semaine 1 peuvent éventuellement être situés fin décembre de l'année précédente. De même, la dernière semaine (52 ou 53) d'une année peut avoir ses derniers jours au tout début de l'année suivante. Une année comporte toujours soit 52, soit 53 semaines au total ;

Les années ont 52 semaines en général ($365 \text{ jours} = 52 \times 7 + 1$). Donc, si une année a son 1er janvier un lundi, son 31 décembre tombe aussi un lundi (un mardi si bissextile). Seules les années dont le jour de l'an est un jeudi auront 53 semaines (un mercredi ou un jeudi pour les années bissextiles).

Formats - dates

- aaaa représente l'année grégorienne sur 4 chiffres fixes .
- - est écrit littéralement (signe moins -, ASCII ou mathématique -) dans les formats étendus (séparateur de date standard ISO), absent des formats de base.
- En cas de notation de fractions (sur les heures, les minutes ou les secondes), le séparateur décimal est une virgule ou un point, la virgule étant le signe préféré.
- mm représente le numéro de mois dans l'année sur deux chiffres (01 à 12).
- qq représente le quantième dans le mois sur deux chiffres (01 à 31).
- qqg représente le quantième dans l'année sur 3 chiffres (001 à 366).
- AAAA représente l'année ISO sur 4 chiffres.
- W est toujours écrit littéralement.
- ww représente la semaine ISO sur deux chiffres (01 à 53).
- j représente le jour ISO de la semaine sur un chiffre (1 à 7).

Formats - dates

Notes

- aaaa représente l'année grégorienne sur 4 chiffres fixes (1583 à 9999, les années 0001 à 1582 nécessitent un accord mutuel, de même que l'extension aux années à plus de 4 chiffres, il n'y pas de représentation standard des années avant Jésus-Christ, mais certains le permettent avec un accord mutuel en utilisant le préfixe littéral B suivi de 4 chiffres à partir de 0001 pour la désignation classique des années sans 0000, ou le préfixe U suivi d'un signe et de 4 chiffres à partir de 0000 pour l'extension grégorienne proleptique du calendrier UTC ; pour les années au-delà de 9999 = U+9999 ou avant B9999 ou avant U-9999, un accord mutuel est aussi nécessaire pour utiliser 5 chiffres fixes ou plus car cela entraîne un conflit avec les formats standards de base des dates incomplètes).

Formats - dates

- Les formats suivants peuvent alors être utilisés pour indiquer :
 - l'année seule :
format de base : "aaaa" (ex : 1997)
pas de format étendu
 - l'année et le mois :
format de base : "aaaamm" (ex : 199707)
format étendu : "aaaa-mm" (ex : 1997-07)
 - la date complète calendaire (année, mois, quantième) :
format de base : "aaaammqq" (ex : 19970716)
format étendu : "aaaa-mm-qq" (ex : 1997-07-16)
 - la date complète ordinale (année, jour de l'année) :
format de base : "aaaaqqq" (ex : 1997206)
format étendu : "aaaa-qqq" (ex : 1997-206)
 - la date complète hebdomadaire (année ISO, semaine ISO, jour de la semaine ISO) selon le numéro de semaine :
format de base : "AAAWwwj" (ex : 2004W453)
format étendu : "AAAA-Www-j" (ex : 2004-W45-3)

Formats - dates

Formats - heures

- T est toujours écrit littéralement (peut être omis lorsqu'il n'y a pas de risque de confusion⁵).
- hh représente l'heure du jour sur deux chiffres (00 à 23).
- : est écrit littéralement dans les formats étendus (séparateur horaire standard ISO).
- mi représente la minute sur deux chiffres (00 à 59).
- ss représente la seconde sur deux chiffres (00 à 60, en tenant compte des secondes ajoutées à la dernière minute de certains jours, selon la norme UTC)
- , est écrit littéralement dans les formats étendus (séparateur décimal ISO préféré⁶ ; le point étant acceptable).
- n représente la fraction de seconde sur un ou plusieurs chiffres.
- zzzzz représente le fuseau horaire dans le format de base (Z ou +hhmi ou -hhmi).
- zzzzzz représente le fuseau horaire dans le format étendu (Z ou +hh:mi ou -hh:mi).

Formats - heures

Formats - heures

- la date complète calendaire avec heures et minutes :
format de base : "aaaammqqThhmizxxxx" (ex : 19970716T1920+0100)
format étendu : "aaaa-mm-qqThh:mizxxxx" (ex : 1997-07-16T19:20+01:00)
- la date complète calendaire plus l'heure, les minutes et les secondes :
format de base : "aaaammqqThhmissxxxx" (ex : 19970716T192030+0100)
format étendu : "aaaa-mm-qqThh:mi:ssxxxx" (ex : 1997-07-16T19:20:30+01:00)
- la date complète calendaire et l'heure avec des fractions de seconde :
format de base : "aaaammqqThhmissnxxxx" (ex : 19970716T1920304+0100)
format étendu : "aaaa-mm-qqThh:mi:ss,nxxxx" (ex : 1997-07-16T19:20:30,4+01:00)
- l'heure seulement :
format de base : "Thhxxxx" (ex : T19+0100)
format étendu : "Thhxxxx" (ex : T19+01:00)
- l'heure et les minutes :
format de base : "Thhmizxxxx" (ex : T1920+0100)
format étendu : "Thh:mizxxxx" (ex : T19:20+01:00)
- l'heure, les minutes et les secondes :
format de base : "Thhmissxxxx" (ex : T192030+0100)
format étendu : "Thh:mi:ssxxxx" (ex : T19:20:30+01:00)
- l'heure complète avec des fractions de seconde :
format de base : "Thhmissnxxxx" (ex : T1920304+0100)
format étendu : "Thh:mi:ss,nxxxx" (ex : T19:20:30,4+01:00)

Formats - heures

Nouveautés Java 8

Les classes imbriquées et anonymes en Java

Version 3.0

- Comprendre les classes imbriquées et anonymes

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Classe imbriquée

Caractéristiques

- Définie à l'intérieur d'une autre classe
 - soit au niveau des attributs ou des méthodes,
 - soit à l'intérieur d'un bloc: classe locale
- Peut accéder à son contexte englobant
 - variables et méthodes d'instances si elle n'est pas statique
 - variables et méthodes statiques si elle est statique
 - variables locales constantes si elle est locale

```
public class A {  
    int x;  
    private class B {  
        int y;  
        public void m()  
        {  
            System.out.println(y + " " + A.this.x);  
        }  
    };  
    public void methode() {  
        final int i = 12;  
        class Locale {  
            public void m() {  
                System.out.println(i + " " + A.this.x);  
            }  
        }  
    }  
}
```

Classe imbriquée

Caractéristiques

La différence essentielle entre des classes locales et les fonctions membres est la possibilité de faire référence aux variables locales dans la portée de leur définition. Une restriction cruciale repose sur le fait que ces variables locales et paramètres doivent être final.

Une classe locale peut utiliser des variables locales car le compilateur donne automatiquement à la classe un champ privé pour maintenir une copie de chaque variable locale. Le constructeur ajoute aussi des arguments cachés à tous les constructeurs de classes locales pour initialiser automatiquement ces champs private à une valeur particulière.

La seule façon de travailler correctement avec des variables locales est de ne manipuler que des variables locales déclarées en final ce qui interdit tout changement. Avec cette garantie, une classe locale peut être assurée que les copies internes des variables sont en mode "sync" avec les variables locales réelles de cette classe.

Les classes locales ne peuvent contenir de champs, méthodes ou classes déclarées en static. Les membres static doivent être déclarés au niveau le plus haut. Aussi parce que les interfaces imbriquées sont implicitement static, les classes locales ne peuvent être imbriquées dans une définition d'interface.

Une autre restriction des classes locales réside dans le fait qu'elles ne peuvent être déclarées en public, protected, private ou static. Ces modificateurs sont tous utilisés pour les membres d'une classe et ne sont pas permis pour les déclarations de variables locales. Pour la même raison cela n'est pas permis avec les déclarations de classes locales.

La classe imbriquée renforce l'encapsulation

Exemple typique: structure de données et itérateur

```
public class LinkedList
{
    public interface Linkable { ... }

    // la tete de liste
    Linkable head;

    public void addToHead(Linkable node) { ... }
    public Linkable removeHead() { ... }

    // cette methode cree et retourne la liste chainee
    public Enumeration enumerate()
    {
        // definition d'une classe locale
        class Enumerator implements Enumeration {
            Linkable current;
            public Enumerator() { this.current = LinkedList.this.head; }
            public Object nextElement() {
                if (current == null)
                    throw new NoSuchElementException("LinkedList");
                Object value = current;
                current = current.getNext();
                return value;
            }
        }
        // creation et retour d'une instance de la classe Enumerator
        return new Enumerator();
    }
}
```

La classe imbriquée renforce l'encapsulation

Exemple typique: structure de données et itérateur

Une structure de données est fréquemment accompagnée d'autres classes qui en facilitent l'utilisation telles que les itérateurs.

Le mécanisme des classes imbriquées permet de définir les classes et interfaces d'itérateurs à l'intérieur de la structure de données. Les avantages d'une telle construction sont intéressants:

- rattachement des itérateurs à leur structure de définition,
- simplification de la programmation des itérateurs.

Classe imbriquée et visibilité

Champs et variables accessibles depuis une classe locale

```
class A { protected char a = 'a'; }
class B { protected char b = 'b'; }

public class C extends A {
    private char c = 'c';
    public static char d = 'd';
    public void createLocalObject(final char e) {
        final char f = 'f';
        int i = 0; // i n'est pas final donc non utilisable par une classe locale
        class Local extends B {
            char g = 'g';
            public void printVars() {
                // tous ces champs et ces variables sont accessibles
                // par cette classe
                System.out.println(g); // this.g : champ de cette classe
                System.out.println(f); // f : variable local final
                System.out.println(e); // e : argument local final
                System.out.println(C.d); // C.this.d : champ de la classe englobante
                System.out.println(C.this.c); // C.this.c : champ de la classe englobante
                System.out.println(b); // b : hérité par cette classe
                System.out.println(C.this.a); // a : hérité par la classe englobante
            }
        }
        Local l = this.new Local();
        l.printVars();
    }
}
```

Classe imbriquée et visibilité

Notes

Classe imbriquée et visibilité

Champs et variables accessibles depuis une classe locale

Le main se présente de la façon suivante:

```
public static void main(String[] args) {  
    // cree une instance de la classe englobante et invoque la methode  
    // qui definit et cree la classe local  
    C c = new C();  
    c.createLocalObject('e');// passe une valeur au param. final e  
}
```

Une classe locale peut utiliser:

- des champs définis dans la classe locale elle-même
- des champs hérités par la classe locale
- des paramètres final dans la portée de la définition de la classe locale,
- des champs de la classe englobante,
- des champs hérités par la classe englobante.

Classe imbriquée et visibilité

Notes

Classe anonyme

Caractéristiques

- Classe sans nom qui ne peut pas être réutilisée
- Confond les opérations de définition de classe et d'instanciation
- Possède les mêmes facultés qu'une classe locale
- Le nom de classe qui suit l'opérateur new peut être:
 - une classe: la classe anonyme en est alors une sous-classe
 - une interface: la classe anonyme est alors une sous-classe de Object

```
public interface Description {  
    public String decrisTOI() ;  
}  
  
Itineraire iti = new Itineraire();  
Capitale paris = new Capitale("Paris");  
iti.ajouter(paris);  
iti.ajouter(new Description() {  
    public String decrisTOI() {  
        return "La Tour Eiffel";  
    }  
});
```

Classe anonyme

Caractéristiques

Une classe anonyme est une extension du concept de classe locale. Au lieu de déclarer une classe locale avec un traitement Java et ensuite l'instancier avec un autre traitement Java, une classe anonyme combine les deux activités dans une seule expression Java. C'est une extension de l'utilisation de l'opérateur new.

Bien entendu, une classe anonyme ne porte pas de nom et parce qu'elle est définie dans la même expression qui l'instancie, elle ne peut être instanciée qu'une seule fois.

A l'exception de ces différences, une classe anonyme est grandement similaire à une classe locale dans son comportement.

Lorsque l'on écrit une classe adaptateur, le problème de savoir si on nomme la classe ou pas ne dépend que d'un point de vue de clarté. Cela entraîne un style de programmation plutôt qu'un style de fonctionnement.

Sur l'exemple de la page précédente si le nom suivant l'opérateur new est un nom de classe, la classe anonyme est une sous classe de la classe nommée, mais si le nom est un nom d'interface (comme dans l'exemple) la classe anonyme est simplement une implémentation de l'interface. Dans ce cas, la classe anonyme est toujours une sous-classe de Object.

Il n'y a aucune possibilité de spécifier une clause extends ou implements

Classe anonyme

Classes anonymes contre classes locales

```
import java.util.*;
public class LinkedList
{
    public interface Linkable { ... }

    // la tete de liste
    Linkable head;

    public void addToHead(Linkable node) { ... }
    public Linkable removeHead() { ... }

    // cette methode cree et retourne la liste chainee
    public Enumeration enumerate()
    {
        // instancie et retourne cette implémentation
        return new Enumeration() {
            Linkable current = head; // obligatoire car il n'y a pas
                                     // de constructeur pour une classe anonyme
            public boolean hasMoreElements() { return current != null; }
            public Object nextElement() {
                if (current == null)
                    throw new NoSuchElementException("LinkedList");
                Object value = current;
                current = current.getNext();
                return value;
            }
        }; // le ; est obligatoire
    }
}
```

Classe anonyme

Classes anonymes contre classes locales

Parce que cette classe n'a pas de nom, il n'est pas possible de définir de constructeur dans le corps de cette classe. Ainsi, tous les arguments spécifiés entre les parenthèses suivant le nom de la super-classe dans une définition de classe anonyme sont implicitement passés aux constructeurs de la super-classe. Les classes anonymes sont couramment utilisées comme des sous-classes simples dont les potentiels constructeurs ne prendraient aucun argument.

Les classes anonymes ne portant pas de nom, la compilation de l'exemple de la page précédente entraîne la création de deux fichiers: un pour la classe englobante et un autre pour la classe anonyme. Le nom qui est donné au fichier correspondant à la classe anonyme porte le nom de la classe englobante suivi du rang de la classe anonyme. Pour notre exemple, cela donne: Liste.class et Lister\$1.class.

Dans votre code, lorsque vous avez le choix entre utiliser une classe anonyme et employer une classe locale, la décision découle souvent de choix de style de programmation. En général, vous devrez considérer l'usage de classe anonyme à la place de classe locale dans les cas suivants:

- la classe a un corps très réduit,
- seule une instance de la classe est nécessaire,
- la classe est utilisé juste après avoir été définie,
- nommer la classe ne fournira aucune aide supplémentaire,
- aucun constructeur n'est utile pour la définition de l'instance utile dans la suite du code.

Nouveautés Java 8 JavaScript

Version 3.0

- Présentation
- Syntaxe
- Objets du navigateur
- DOM

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Présentation

Standard ECMA

- JavaScript est lié à HTML.
 - Langage interprété par les navigateurs Web.
 - Code encapsulé dans des pages HTML et exécuté coté client.
 - Permet de rendre dynamique les pages affichées par le navigateur.
- JavaScript est un langage de programmation orienté objet.
 - chaque objet possède des attributs et des méthodes.
- JavaScript est un langage sensible à la casse.
- Cependant :
 - JavaScript n'est pas implémenté de la même façon sur chaque navigateur.
 - Il est quasi-impossible de produire du code JavaScript compatible avec tous les navigateurs.

Présentation

Notes

Instructions - Conditions - Boucles

- Les instructions se terminent par ';'.
- Les instructions JavaScript se trouvent généralement au sein de fonctions.

Définition d'une fonction

- Utilisation du mot clé function

```
function maFonction() {  
    ...  
}
```

Instructions - Conditions - Boucles

Notes

Instructions - Conditions - Boucles

Conditions

- Egal : ==
- Différent : !=
- Inférieur ou égal : <=
- Supérieur ou égal : >=
- Inférieur strict : <
- Supérieur strict : >
- ET logique : &&
- OU logique : ||
- Identité : ===
- Non identité : !==
- ET bit à bit : &
- OU bit à bit : |

```
if (var1==var2)
{
    .....
}
else
{
    .....
}
```

Instructions - Conditions - Boucles

Notes

Instructions - Conditions - Boucles

Boucles

```
for (i=0; i<5; i++)  
{  
    .....  
}
```

```
while (a<b)  
{  
    .....  
}  
  
do  
{  
    .....  
}while (a<b)
```

Instructions - Conditions - Boucles

Notes

Intégration de JavaScript dans une page HTML

- Utilisation de la balise `<script>` dans l'en-tête du fichier

```
<html>
  <head>
    <title></title>
    <script language="javascript" type="text/javascript">
      ...
    </script>
  </head>
  ...
</html>
```

- Référence à un fichier JavaScript externe

```
<html>
  <head>
    <title></title>
    <script language="javascript" type="text/javascript" src="script.js"></script>
  </head>
  ...
</html>
```

Intégration de JavaScript dans une page HTML

Notes

Variables

- Les variables ne sont pas typées.
- Les variables sont déclarées avec le mot clé **var**.

```
var i, chaine, bool; //Déclaration de 3 variables  
i = 2;  
chaine = "bonjour";  
bool = true;
```

Variables

Notes

Evénements

- Permet de réagir à une action coté client.
- Spécifié en ajoutant un attribut onXxx à certains éléments HTML

```
<body onload="maFonction()">  
<input type="button" onclick="maFonction()">
```

- Peut être basé sur l'usage de pseudo-URL

```
<a href="javascript:alert('Coucou !!')">Mon Lien</a>
```

Evénement	Survient	Evènement	Survient
onload	après le chargement de la page	onmouseup	quand on relâche le bouton de la souris
onunload	lors de la fermeture de la page	onkeydown	quand on enfonce une touche du clavier
onbeforeunload	juste avant la fermeture de la fenêtre	onkeyup	quand on relâche la touche
onclick	lors d'un clic	onsubmit	juste avant l'envoi d'un formulaire
ondblclick	lors d'un double clic	onreset	lors de la réinitialisation d'un formulaire
onmousedown	quand on enfonce le bouton de la souris	onselect	quand le contenu d'un élément est sélectionné

Evénements

Notes

Les objets du navigateur

- `window` : créé lors de l'ouverture du navigateur. Cet objet est implicite.
- `window.navigator` : représente le navigateur (type, version, ...)
- `window.document` : représente le document HTML courant

Les objets du navigateur

Notes

DOM en JavaScript

- Modification de la structure d'un document HTML suivant l'API W3C DOM
- `window.document` et chaque élément HTML sont des noeuds (Node)
- Chaque Node possède des propriétés et des méthodes

Propriétés	Commentaires
<code>childNodes</code>	noeuds enfants
<code>firstChild</code>	premier noeud enfant
<code>lastChild</code>	dernier noeud enfant
<code>nextSibling</code>	prochain noeud d'un type (noeud de même niveau)
<code>parentNode</code>	noeud parent
<code>previousSibling</code>	noeud précédent d'un type (noeud de même niveau)
<code>nodeName</code>	nom du noeud
<code>nodeValue</code>	valeur / contenu du noeud
<code>nodeType</code>	type du noeud (cf. ci-dessous)

Types de noeuds :

1 - Noeud élément
2 - Noeud attribut
3 - Noeud texte
4 - Noeud pour CDATA
5 - Noeud pour référence d'entité
6 - Noeud pour entité

7 - Noeud pour instruction de traitement
8 - Noeud pour commentaire
9 - Noeud document
10 - Noeud type de document
11 - Noeud de fragment de document
12 - Noeud pour notation

DOM en JavaScript

Notes

DOM en JavaScript

Méthodes	Commentaires
<code>createElement()</code>	Méthode pour créer un nouvel élément HTML dans le document (div, p, span, a, form, input, etc...).
<code>createTextNode()</code>	Méthode pour créer un nœud texte.
<code>appendChild()</code>	Pour ajouter l'élément créé dans le document. L'élément sera ajouté comme étant le dernier nœud enfant d'un élément parent.
<code>insertBefore()</code>	Pour ajouter l'élément créé avant un autre nœud.
<code>removeChild()</code>	Pour supprimer un nœud.

DOM en JavaScript

Notes