

# Plan

- Modification du langage
  - Corps de méthodes dans les interfaces
  - Lambda-expressions et flux de traitement
- Modifications d'API
  - API date
  - Join sur les chaînes
  - Classe Optional
- Nouvelles commandes
  - Interpréteur javascript
  - Analyseur de dépendances

# Corps de méthodes dans les interfaces : méthodes par défaut (1)

- Problème
  - Données de base :
    - Soit  $I$  une interface fournissant une méthode  $m1()$
    - Soient  $C1, C2, \dots, C_N$  des classes implantant  $I$
  - Évolution
    - On rajoute à  $I$  une méthode  $m2()$
  - Conséquence
    - Plus aucune classe  $C_i$  n'est correcte ; elles doivent toutes être modifiées

# Corps de méthodes dans les interfaces : méthodes par défaut (2)

- Solution
  - Pouvoir définir dans les interfaces des implantation par défaut de méthodes
- Mise en œuvre
  - Rajouter le modificateur « default » devant la déclaration de la méthode, puis, après sa déclaration, donner sa définition
- Conséquence
  - Les classes qui ne définissent pas la méthode en question hériteront de l'implantation par défaut fournie par l'interface
- Contraintes
  - L'implantation par défaut ne peut pas utiliser de variables d'instance

# Corps de méthodes dans les interfaces : méthodes par défaut (3)

- Conséquence sur l'héritage entre interfaces
  - Soit I1 une interface définissant une méthode par défaut m1()
  - Soit I2 une interface héritant de I1
- 3 cas
  - Si I2 ne mentionne pas m1(), elle hérite du m1() de I1
  - Si I2 redéfinit m1(), cette nouvelle définition remplace celle de I1
  - Si I2 se contente de déclarer m1(), elle la rend de nouveau abstraite

# Corps de méthodes dans les interfaces : méthodes statiques

- Motivation
  - Regrouper des méthodes utilitaires susceptibles d'être partagées par toutes les classes implantant une interface donnée
  - Cas particulier : méthode utilitaire utilisée dans une méthode par défaut
- Mise en œuvre
  - Comme dans une classe Java

# Corps de méthodes dans les interfaces : héritage multiple

- Problème
  - Soit I1 une interface définissant une méthode par défaut m()
  - Soit I2 une interface définissant également une méthode m().
  - Soit C une classe implantant I1 et I2
- Résolution
  - Si I2 hérite de I1, c'est la définition de I2 qui est utilisée
  - Sinon, il faut redéfinir m() dans C pour éviter une erreur de compilation
- Redéfinition de m() dans C :
  - Soit de zéro
  - Soit en faisant appel à une des méthodes m() de I1 ou I2 ainsi : I1.super.m() ou I2.super.m()

# Lambda-expressions

## Exemple introductif : classe interne

```
public interface Calcul {  
    int calc (int a, int b) ;  
}  
public class Math {  
    private static int calculer(int a, int b, Calcul op) {  
        return op.calc(a,b) ;  
    }  
    public static void main (String[] args) {  
        int a = 2 ; int b = 3 ;  
        class Addition implements Calcul {  
            public int calc(int a, int b) {return a+b;}  
        }  
        class Produit implements Calcul {  
            public int calc(int a, int b) {return a*b;}  
        }  
        System.out.println(a + "+" + b + "=" + calculer(a,b, new Addition()));  
        System.out.println(a + "x" + b + "=" + calculer(a,b, new Produit()));  
    }  
}
```

# Lambda-expressions

## Exemple introductif : classe anonyme

```
public interface Calcul {  
    int calc (int a, int b) ;  
}  
public class Math {  
    private static int calculer(int a, int b, Calcul op) {  
        return op.calc(a,b) ;  
    }  
    public static void main (String[] args) {  
        int a = 2 ; int b = 3 ;  
  
        System.out.println(a + "+" + b + "=" + calculer(a,b, new Calcul() {  
            public int calc(int a, int b) {return a+b ;}}));  
        System.out.println(a + "x" + b + "=" + calculer(a,b, new Calcul() {  
            public int calc(int a, int b) {return a*b;}}));  
    }  
}
```



# Lambda-expressions

## Exemple introductif : commentaires

- L'utilisation des classes anonymes est sensé alléger le code mais la syntaxe reste lourde :

```
new Calcul() {  
    public int calc(int a, int b) {return a+b ;}  
}
```
- Comment/Pourquoi simplifier ?
  - L'interface Calcul ne propose **qu'une méthode abstraite** ; on sait donc de quelle méthode on va donner le code => *interface fonctionnelle*
  - On sait que le troisième paramètre de la méthode calculer et de type Calcul
  - Les types des paramètres peuvent être déduits

# Lambda-expressions

## Exple introductif : nouvelle version

```
public interface Calcul {  
    int calc (int a, int b) ;  
}  
public class Math {  
    private static int calculer(int a, int b, Calcul op) {  
        return op.calc(a,b) ;  
    }  
    public static void main (String[] args) {  
        int x = 2 ; int y = 3 ;  
  
        System.out.println(x + "+" + y + "=" + calculer(x,y, (a, b) -> a+b)) ;  
        System.out.println(x + "x" + y + "=" + calculer(x,y, (a, b) -> a*b)) ;  
    }  
}
```

# Lambda-expressions

## Syntaxe

- Paramètres -> corps
  - Paramètres :
    - (listeParam) ou paramUnique ou ()
    - listeParam : paramUnique[,ListeParam]\*
    - paramUnique : nom ou type nom
  - Corps :
    - Expression ou bloc d'instructions
- Exemples
  - `a -> a-1`
  - `(int a) -> {return a+1;}`
  - `(int a, int b) -> {if (a > b) {return a;} {return b;}}`

# Lambda-expression

## Interface fonctionnelle

- Une interface fonctionnelle est une interface qui contient une et une seule méthode :
  - Non statique
  - Sans corps par défaut
  - Non présente dans `java.lang.Object`
- Exemple
  - L'interface `Comparator` de java 1.8 est une interface fonctionnelle

# Lambda-expressions

## Références de méthode (1)

- Problème
  - Soit l'interface fonctionnelle suivante

```
public interface Valuer<T> {  
    int valuer(T obj) ;  
}
```
  - On souhaite en dériver une implantation sur les chaînes où la valuation correspondrait à la longueur de la chaîne
  - On peut écrire : `(s -> s.size())`, ce qui n'est pas idéale
- Solution
  - Utiliser une référence à la méthode `size()` :
    - `String::size`

# Lambda-expressions

## Références de méthodes (2)

- Types de référence
  - Méthode de classe  
NomClasse::NomMéthode
  - Méthode d'instance  
NomClasse::NomMéthode
  - Méthode d'une instance particulière  
Instance::NomMéthode  
La méthode est appelé sur le premier paramètre
  - Constructeur  
NomClasse::new
- Liste de paramètres
  - Inférée automatiquement à partir de l'interface fonctionnelle

# Lambda-expressions

## Références de méthodes (3)

- Exemple

- Rappels

- Méthode de la classe `java.util.Arrays`  
`static <T> void sort(T[] a, Comparator<? super T> c)`
    - Interface `java.util.Comparator<T>` propose une seule méthode abstraite :  
`int compare(T o1, T o2)`
    - La classe `String` contient la méthode :  
`int compareToIgnoreCase(String str)`

- Application

```
String [] ani1 = {"chat", "COCHON", "Chien"} ;  
Arrays.sort(ani1, String::compareToIgnoreCase) ;
```

# Lambda-expressions

## Réf. à des méthodes d'instance

- Code introductif

```
class Personne {  
    String nom;  
    int age;  
    public int naissance(int anneeCourante) {  
        return anneeCourante - age;  
    }  
}
```

- Référence à partir d'une instance

```
Function<Integer,Integer> fonc = p1::naissance;  
System.out.println(fonc.apply(2014));
```

- Référence à partir de la classe

```
BiFunction<Personne,Integer,Integer> fonc2 = Personne::naissance;  
System.out.println(fonc2.apply(p1, 2016));
```



# Lambda-expressions

## Package `java.util.function` (1)

- Rôle
  - Fournir un certain nombre d'interfaces fonctionnelles standards
- Contenu (interfaces de base)
  - `Consumer<T>` (méthode `accept(T t)`)
  - `Function<T,R>` (méthode `R apply(T t)`)
  - `Predicate<T>` (méthode boolean `test(T t)`)
  - `Supplier<T>` (méthode `T get()`)
- Autres interfaces
  - Versions dédiées aux types simples ou prenant 2 paramètres

# Lambda-expressions

## Package `java.util.function` (2)

- Rôle des différentes interfaces de base
  - `Consumer<T>` (méthode `accept(T t)`)  
Appliquer une méthode (sans retour) à un objet
  - `Function<T,R>` (méthode `R apply(T t)`)
    - Appliquer une méthode (avec retour) à un objet
  - `Predicate<T>` (méthode boolean `test(T t)`)
    - Évaluer une propriété (vrai/faux) sur un objet
  - `Supplier<T>` (méthode `T get()`)
    - Obtenir un objet

# Les flux de traitement (Stream) (1)

- But
  - Permettre d'appliquer une succession de traitement à partir d'un paquet de données (tableau, collection)
- Mise en œuvre
  - Créer un flux à partir d'une source
  - Appliquer successivement des méthodes de traitement sur le flux (chaque méthode fournit en résultat un nouveau flux)
  - Appliquer au finale une opération terminale pour transformer le flux en une structure de données exploitable
- Intérêt
  - Facilité de mise en œuvre (boucle implicite, utilisation des lambda-expressions facilitée)
  - Efficacité (évaluation paresseuse, traitements parallèles aisément définissable)
- Attention !

**LES FLUX SONT A USAGE UNIQUE !**

# Les flux de traitement (Stream) (2)

## Création à partir d'une source

- Source = collection
  - Méthode `stream<E>()` de l'interface `Collection<E>`
  - `default Stream<E> parallelStream()`
- Source = tableau
  - Méthode static `<T> Stream<T> stream(T[] array)` de la classe `java.util.Arrays`
  - Méthode static `<T> Stream<T> of(T... valeurs)` de l'interface `java.util.stream.Stream`
- Source = flux d'entrée
  - Exemples
    - méthode `Stream<String> lines()` de la classe `BufferedReader`
    - méthode `Stream<String> lines(Path p)` de `java.nio.file.Files`
    - méthode `Stream<Path> list(Path p)` de `java.nio.file.Files`

# Les flux de traitement (Stream) (2)

## Méthodes intermédiaires

- Filtrage
  - `Stream<T> filter(Predicate< ? super T>)`
- Application avec résultat
  - `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
- Application sans résultat
  - `Stream<T> peek(Consumer<? super T> action)`

# Les flux de traitement (Stream) (3)

## Méthodes terminales

- Transformation en tableau
  - `Object[] toArray()`
- Min/max
  - `Optional<T> max(Comparator<? super T> comparator)`
  - `Optional<T> min(Comparator<? super T> comparator)`
- Traitement avec effet de bord
  - `void forEach(Consumer<? super T> action)`
- Dénombrement
  - `long count()`
- Any/All
  - `boolean allMatch(Predicate<? super T> predicate)`
  - `boolean anyMatch(Predicate<? super T> predicate)`

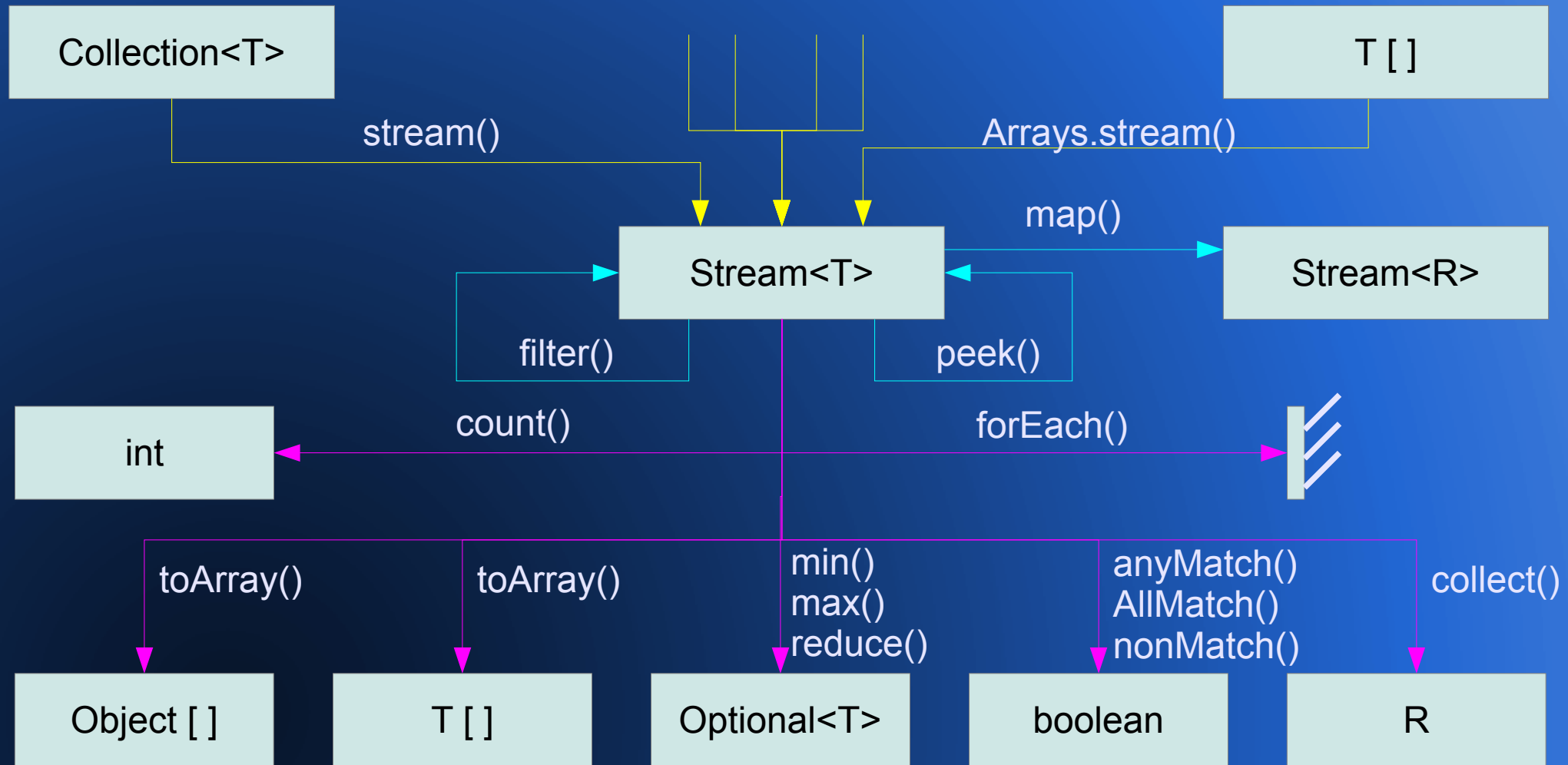
# Les flux de traitement (Stream) (4)

## Réductions

- Principe générale
  - Méthode permettant d'appliquer une méthode associative successivement sur les éléments d'un flux (somme ou moyenne des éléments par exemple)
- Méthodes associées
  - `Optional<T> reduce(BinaryOperator<T> accumulator)`
  - `<R,A> R collect(Collector<? super T,A,R> collector)`

À étudier

# Les flux de traitement (Stream) Résumé





# Les flux de traitement (Stream)

## Liste des transformeurs de flux

- Filtrage
  - `Stream<T> distinct()`
  - `Stream<T> filter(Predicate< ? super T>)`
  - `Stream<T> limit(long)`
  - `Stream<T> skip(long)`
- Tri
  - `Stream<T> sorted()`
  - `Stream<T> sorted(Comparator < ? super T>)`
- Transformation
  - `Stream<R> map(Function< ? super T, ? extends R>)`
  - `Stream<R> flatMap(Function< ? super T, ? extends Stream< ? extends R>>)`
- Modification
  - `Stream<T> peek(Consumer< ? Super T>)`

# Les flux de traitement (Stream)

## Liste des générateurs de flux

- `static <T> Stream<T> empty()`
  - Flux vide
- `static <T> Stream<T> of(T)`
  - Flux d'un élément
- `static <T> Stream<T> of(T...)`
  - Flux à partir d'un tableau ou de quelques éléments
- `static <T> Stream<T> generate(Supplier<T>)`
  - Flux infini à partir d'un constructeur
- `static <T> Stream<T> iterate(T depart, UnaryOperator<T> fonction)`
  - Flux infini par itération d'une fonction à partir d'un point de départ
- `static <T> Stream<T> concat(Stream< ? extends T>, Stream < ? extends T>)`
  - Concaténation de flux

# Les flux de traitement (Stream)

## Liste des traitements terminaux

- Réductions booléennes
  - boolean allMatch(Predicate< ? super T>)
  - boolean anyMatch(Predicate< ? super T>)
  - boolean noneMatch(Predicate< ? super T>)
- Recherche d'éléments
  - Optional<T> findAny()
  - Optional<T> findFirst()
  - Optional<T> max(Comparator< ? super T>)
  - Optional<T> min(Comparator< ? super T>)
- Dénombrement
  - long count()
- Récupération des éléments
  - Object[] toArray()
  - T[] toArray(IntFunction<T[]> generator)

# Les flux de traitement (Stream)

## Réductions (1)

- `Optional<T> reduce(BinaryOperator<T> accumulateur)`
  - L'accumulateur prend 2 paramètres : la valeur accumulée et la nouvelle valeur ; il doit donc prévoir le cas d'initialisation où la valeur accumulée est à null
  - Exemple

```
Stream<String> exemple1 = Stream.of("Tata", "Titi", "Toto");
Optional<String> resultat1 = exemple1.reduce((x,y) -> {if (x == null) return y; else return x + y;});
```
- `T reduce(T elementNeutre, BinaryOperator<T> accumulateur)`
  - Similaire à précédemment, mais on passe l'élément de départ, qui doit être élément neutre
  - Exemple

```
Stream<Integer> exemple4 = Stream.iterate(1, x -> x+1).limit(10);
int produit4 = exemple4.reduce(1, (x, y) -> x*y);
```

# Les flux de traitement (Stream)

## Réductions (2)

- `<U> U reduce(U elementNeutre, BiFunction<U, ? super T, U> accumulateur, BinaryOperator<U> combineur)`
  - Permet de ne pas forcément utiliser un traitement séquentiel ; le « combineur » permet de combiner 2 structures accumulatrices intermédiaires
  - Intérêt : permettre un traitement parallèle, plus efficace
  - Remarque : l'accumulation peut avoir lieu dans un type différent que le type des éléments du flux
  - Exemple

```
ArrayList<String> liste = ...  
Stream<String> exemple = liste.parallelStream();  
StringBuilder resultat = exemple.reduce(new StringBuilder(),  
    StringBuilder::append, StringBuilder::append);
```

Ajout d'un String à un StringBuilder

Ajout d'un StringBuilder à un StringBuilder

# Les flux de traitement (Stream) Collectes (1)

- Définition (d'après javadoc)
  - Un opérateur de collecte effectue une *réduction mutable* sur les éléments d'un flux.
  - Une *réduction mutable* est une réduction dans laquelle la valeur réduite est un conteneur de résultat mutable (i.e. donnée modifiable), tel qu'une ArrayList ; les éléments y sont donc incorporés en mettant à jour l'état du résultat plus qu'en le remplaçant.
- Méthode de base

```
<R> R collect( Supplier<R> supplier,  
              BiConsumer<R,? super T> accumulator,  
              BiConsumer<R,R> combiner)
```

# Les flux de traitement (Stream) Collectes (2)

- Exemple d'utilisation

```
ArrayList<String> liste8 = ...
```

```
Stream<String> exemple8 = liste8.parallelStream();
```

```
StringJoiner resultat8 = exemple8.collect(
```

```
    () -> new StringJoiner(", "),
```

Supplier<StringJoiner>

```
    StringJoiner::add,
```

BiConsumer<StringJoiner, ? Super String>

```
    StringJoiner::merge);
```

BiConsumer<StringJoiner, StringJoiner>

Rappel de la classe StringJoiner :

```
StringJoiner add(CharSequence texte)
```

```
StringJoiner merge(StringJoiner autre)
```



# Les flux de traitement (Stream) Collectes (3)

- Interface `Collector<T, A, R>`
  - Rôle
    - Regrouper tout les paramètres nécessaires à la méthode `collect()` au sein d'un seul objet
    - Permettre une éventuelle transformation finale du résultat accumulé
  - Méthodes principales
    - `Supplier<A> supplier()`
    - `BiConsumer<A, T> accumulator()`
    - `BinaryOperator<A> combiner()`
    - `Function<A,R> finisher()`
  - Exemple

```
Collector<String,StringJoiner,String> monCollecteur =
    Collector.of() -> new StringJoiner(","),
    StringJoiner::add, StringJoiner::merge,
    StringJoiner::toString);
String resultat9 = exemple9.collect(monCollecteur);
```



# Les flux de traitement (Stream) Collectes (4)

- Classe Collectors
  - Fournit un certain nombre de collecteurs standards
- Collecteurs statistiques (définis pour les différents types numériques)
  - `static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)`
  - ...
  - `static <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper)`
  - `static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)`

Pour obtenir en un seul parcours :

- moyenne
- nombre
- min
- max
- somme

# Les flux de traitement (Stream) Collectes (5)

- Méthodes pour fusionner des chaînes de caractères

- `static Collector<CharSequence,?,String>Joining()`

Pour concaténer les chaînes d'un flux

- `joining(CharSequence delimiter)`

Idem, mais avec un séparateur

- `joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`

Idem, mais on précise en plus un texte de début et un texte de fin

- Exemple

```
String texte = classe.stream()  
    .map(Object::toString)  
    .collect(Collectors.joining(",","[",""]));
```

# Les flux de traitement (Stream) Collectes (6)

- Regroupements
  - `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`
  - `static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)`
- Génération de collections
  - `ToList()`  
pour obtenir une liste (sans plus d'info)
  - `toCollection(Supplier<C> collectionFactory)`  
pour obtenir une collection spécifique
  - `toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
  - `toSet()`

# Traitement des dates (bases)

## Introduction

- Principe général

Les dates et heures sont maintenant représentées par des classes du package `java.time` dont les instances sont immuables
- Principales classes de `java.time`
  - Instant
    - Instant : marqueur en millisecondes
    - LocalDate : date seule
    - LocalTime : heure seule
    - LocalDateTime : date+heure
    - ZonedDateTime : date+heure avec zone (Europe/Paris)
  - Durée
    - Duration : en nanosecondes
    - Period : en unités « lisibles »

# Traitement des dates

## Création d'une date/heure

- Représentation de l'instant courant

Méthode statique `now()` des différentes classes

- `Instant.now()`
- `LocalDate.now()`
- `LocalTime.now()`
- `LocalDateTime.now()`

- Représentation d'un instant donné

Méthodes statiques `of(...)` des différentes classes

- `Instant.ofEpochMilli(long val)`
- `LocalDate.of(année, mois, jour)`
- `LocalTime.of(heure, minute, seconde)`
- `LocalDateTime.of(année, mois, jour, heure, minute, seconde)`

# Traitement des dates

## Dériver une date/heure

- Par addition/soustraction d'une unité
  - `LocalDate date3 = date2.minusDays(30);`
  - `LocalDate date4 = date3.plus(365, ChronoUnit.DAYS);`
- Par addition/soustraction d'une période
  - `Period delai = Period.ofDays(200);`
  - `LocalDate date5 = date4.plus(delai);`
- Par modification d'un champ
  - `LocalDate date6 = date5.withDayOfMonth(23);`
- Remarque :
  - type énuméré `java.time.temporal.ChronoUnit` donne la liste des différents « champs » représentant des dates/heures

# Traitement des dates

## Calculs sur les dates/heures

- Comparaison
  - `date5.isBefore(date4)`
  - `date5.isAfter(date4)`
- Intervalle
  - `long intervalleJ = date6.until(date3, ChronoUnit.DAYS);`
  - `long intervalleW = date6.until(date3, ChronoUnit.WEEKS);`

# Traitement des dates

## affichage des dates/heures

- Formats « de base »

- « locaux »

```
DateTimeFormatter formateur  
DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);  
String dateFormatee = date6.format(formateur)  
DateTimeFormatter formateurF =  
DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);  
String dateFormateeF = date6.format(formateurF);
```

23 oct. 2013

mercredi 23 octobre 2013

- « spécifiques »

```
DateTimeFormatter formateurAnglais = formateur.withLocale(new  
Locale("en","UK"));  
String dateFormateeUK = date6.format(formateurAnglais);
```

Oct 23, 2013

- Formats personnels (voir javadoc DateTimeFormatter)

```
DateTimeFormatter formateurPerso =  
DateTimeFormatter.ofPattern("eeee/MMMM (G)");  
String dateFormateePerso = date6.format(formateurPerso);
```

mercredi/octobre (ap. J.-C.)



# Join sur les chaînes de caractères

- Méthode `join()` de la classe `String`

- Avec un tableau

```
String[] tableau = {"abc", "def", "ghi"};  
String tabJoin = String.join(";", tableau);
```

- Avec un « `Iterable` »

```
ArrayList<String> liste = new  
    ArrayList<>(Arrays.asList(tableau));  
String listeJoin = String.join("-", liste);
```

- Classe `StringJoiner`

```
StringJoiner joint = new StringJoiner("/");  
for (String s : tableau) {joint.add(s);}  
String resultat = joint.toString();
```

# Classe `java.util.Optional<T>`

## Introduction

- Présentation générale
  - Classe permettant d'éviter des problèmes de « `NullPointerException` »
  - Analogue au type `Maybe` de Haskell
- En pratique
  - Encapsule une donnée d'un type quelconque
  - La donnée peut être « `null` » ou avoir une valeur réelle

# Classe `java.util.Optional<T>`

## Méthodes principales

- `static <T> Optional<T> ofNullable(T valeur)`
  - Crée un « `Optional<T>` » à partir d'une donnée de type `T`, éventuellement « `null` »
- `boolean isPresent()`
  - Renvoie vrai si la donnée encapsulée n'est pas « `null` »
- `T orElse(T autre)`
  - Si la donnée encapsulée n'est pas « `null` », la renvoie, sinon renvoie « `autre` »
- `void ifPresent(Consumer<? Super T> consumer)`
  - Applique « `consumer` » si la donnée n'est pas « `null` »
- `T get()`
  - Renvoie la donnée encapsulée (lève une `NoSuchElementException` si `null`)
- `<U> optional<U> flatMap(Function< ? Super T, Optional<U> fonc)`
  - Applique une fonction renvoyant un `Optional<U>` à une donnée encapsulée dans un `Optional<T>`

# Classe `java.util.Optional<T>`

## `Optional<T>` est une monade

- **Monade : définition**  
Une monade est un type `M`
  - Paramétré par un autre type `T`
  - Avec une « fabrique » pour créer une monade à partir d'une simple donnée de type `T`
  - Avec une fonction permettant d'appliquer une fonction de `T` vers `M<U>` à une monade `M<T>`
- `Optional<T>` est une monade analogue au type `Maybe` de Haskell avec
  - `return` → `ofNullable(T value)`
  - `>>=` → `flatMap(Function<? super T,Optional<U>> mapper)`

# Modifications d'API

## Quelques exemples

# Package java.nio.file

## Classe Files ; extension java 1.8

- Static Stream<Path> find(Path start, int maxDepth, BiPredicate<Path, BasicFileAttributes> matcher, FileVisitOption... options)
- Static Stream<String> lines(Path path)
- Static Stream<String> lines(Path path, Charset cs)
- Static Stream<Path> list(Path dir)
- Static BufferedReader newBufferedReader(Path path)
- Static BufferedWriter newBufferedWriter(Path path, OpenOption... options)
- Static List<String> readAllLines(Path path)
- Static Stream<Path> walk(Path start, FileVisitOption... options)
- Static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options)
- Static Path write(Path path, Iterable<? extends CharSequence> lines, OpenOption... options)

# Extension de l'interface Iterable<T>

- Avant
  - `Iterator<T> iterator()`
- Après
  - `Default void forEach(Consumer<? super T> action)`
  - `Iterator<T> iterator()`
  - `Default Splititerator<T> spliterator()`

# Extension de l'interface Collection<E>

- Ajouts
  - default Stream<E>parallelStream()
  - default boolean removeIf(Predicate<? super E> filter)
  - default Spliterator<E> spliterator()
  - default Stream<E>stream()



# Extension de l'interface Comparator<T>

Interface φονχτιοννελλ  
ε ↙

- Avant

- `int compare(T o1, T o2)`
- `Boolean equals(Object obj)`

- Après

- `int compare(T o1, T o2)`
- `Static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)`
- `Static <T,U> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)`
- `Static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor)`
- `Static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)`
- `Static <T> Comparator<T> comparingLong(ToLongFunction<? super T> keyExtractor)`
- `Boolean equals(Object obj)`
- `Static <T extends Comparable<? super T>> Comparator<T> naturalOrder()`
- `Static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)`
- `Static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)`
- `Default Comparator<T> reversed()`
- `Static <T extends Comparable<? super T>> Comparator<T> reverseOrder()`
- `Default Comparator<T> thenComparing(Comparator<? super T> other)`
- `default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor)`
- `Default <U> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)`
- `Default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)`
- `Default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)`
- `Default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor)`

# Extension de l'interface Iterator<E>

- Avant

- boolean hasNext()
- E next()
- void remove()

- Après

- default void forEachRemaining(Consumer<? super E> action)
- boolean hasNext()
- E next()
- default void remove()