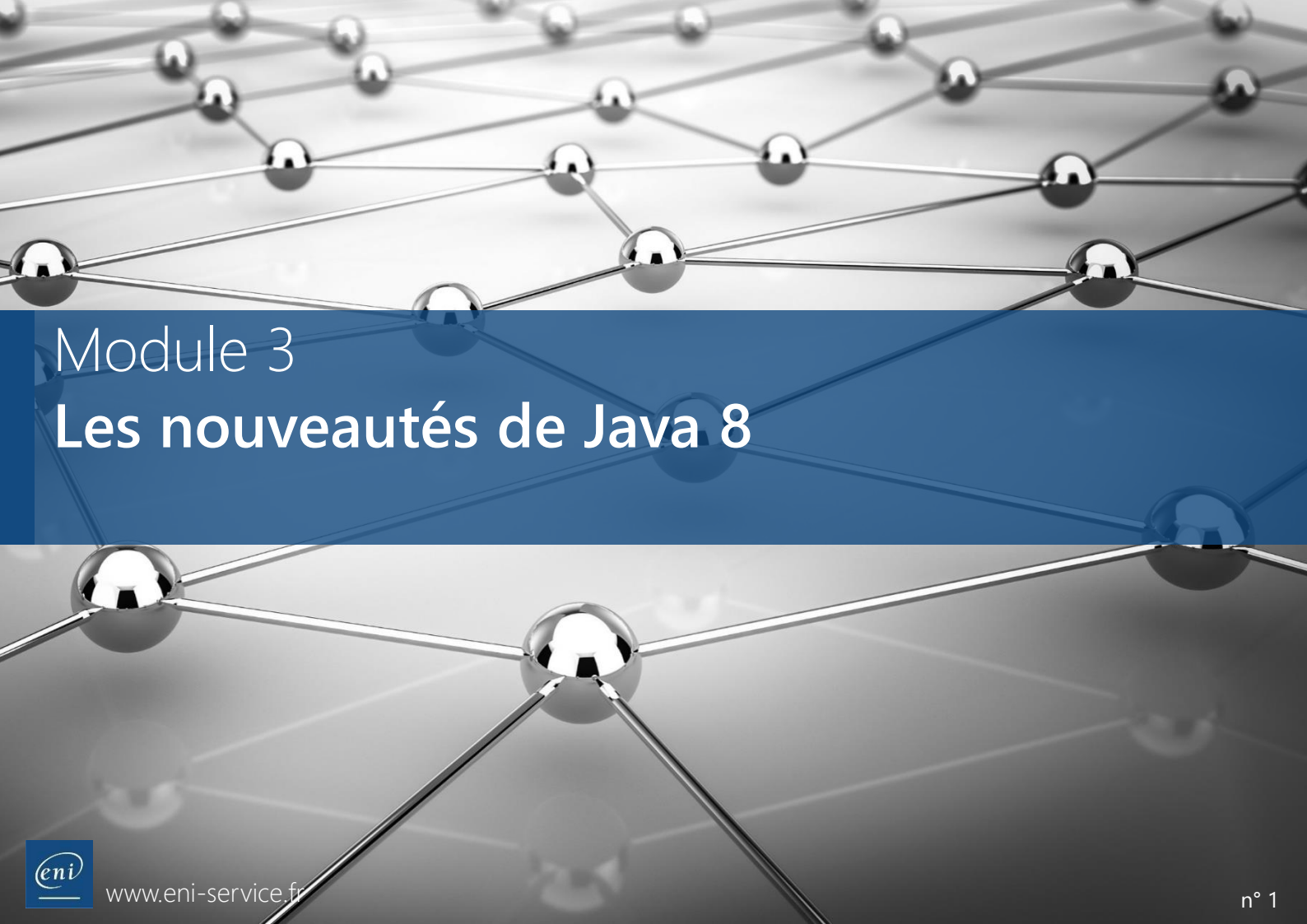

Module 3

Les nouveautés de Java 8

Contenu du module

- Nouveautés sur les interfaces
  - La notion d'interface fonctionnelle
  - Les méthodes par défaut
  - Les méthodes statiques
  - Les références de méthodes
- Le langage
  - Les expressions lambdas
- Une API de gestion de date simple !
  - Les classes fondamentales
  - Les possibilités de calcul
- Manipulation des collections avec les flux
  - La classe Stream et ses méthodes
  - Approche map/reduce

Nouveautés sur les interfaces

Les interfaces fonctionnelles

- Nouveauté de Java 8
- SAM Interfaces (Single Abstract Method Interfaces)
  - Une seule méthode abstraite
  - Runnable, Comparable, Comparator, …
- Identifiables par l'annotation `@FunctionalInterface`
  - Pour le compilateur !

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

Interfaces : méthodes par défaut

- Depuis Java 8
- Implémentation « par défaut » de méthodes dans les interfaces
  - Les interfaces peuvent avoir du code d'implémentation !
  - Ces méthodes n'ont pas obligatoirement à être redéfinies
- Mot clé `default`
- Permettent de faire évoluer les APIs en conservant une compatibilité ascendante
- L'API standard en profite grandement en enrichissant certaines de ses interfaces
  - Dans l'API de Collections dont les interfaces s'enrichissent de plusieurs méthodes

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

Interfaces : méthodes statiques

- Les interfaces peuvent posséder des méthodes statiques
  - Déclarées avec le mot-clé `static`
  - Fournissant une implémentation
- Ce sont des méthodes agrégée à l'interface
  - Elle ne peuvent être invoquée qu'a partir du nom de l'interface
  - En aucun cas à partir du nom d'une classe implémentant cette interface !
    - Les méthodes statiques ne sont jamais « héritées »

Interfaces fonctionnelles de l'API

- Package `java.util.function`
- Fournit des interfaces fonctionnelles de base
  - Pour une utilisation avec les lambdas
- `Consumer<T>` : méthode qui accepte un unique argument (type T) et ne retourne pas de résultat.
  - `void accept(T);`
- `Function<T,R>` : méthode qui accepte un argument (type T) et retourne un résultat (type R).
  - `R apply(T);`
- `Supplier<T>` : méthode qui ne prend pas d'argument et qui retourne un résultat (type T).
  - `T get();`

Les références de méthodes

- Utilisées pour définir une méthode en tant qu'implémentation de la méthode abstraite d'une interface fonctionnelle
- Syntaxe :

```
Classe::methode
```

  - Ou bien :

```
instance::methode
```

- Référence vers une méthode static :

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

- Référence vers une méthode d'instance spécifique :

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

- Référence vers un constructeur

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

Le langage

Les expressions lambdas

- Plus grande nouveauté de Java 8
  - JSR-335
- Ajout de la programmation fonctionnelle en Java
- Peuvent être assimilées à des fonctions anonymes
  - Peuvent être affectées dans des interfaces fonctionnelles
  - Le code de l'expression sera l'implémentation de **LA** méthode
- Syntaxe :
  - `(<paramètres>) -> implémentation`
    - S'il n'y a qu'une seule instruction dans l'implémentation
  - `(<paramètres>) -> { implémentation }`
    - S'il y a plusieurs instructions

Les expressions lambdas : exemple

- Prenons la méthode `sort()` de la classe `Collections`
  - `static <T> void sort(List<T> list, Comparator<? super T> c)`

- Sans les expressions lambdas avant Java 8, mise en œuvre possible (via une classe anonyme) :

```
Collections.sort(liste, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

Les expressions lambdas : exemple (suite)

- Depuis Java 8, avec les lambdas, on peut écrire :

- Forme longue avec bloc et typage :

```
Collections.sort(liste, (String s1, String s2) -> { return s1.length() - s2.length(); } );
```

- Forme courte sans bloc (1 seule instruction) :
  - Mot clé `return` implicite !

```
Collections.sort(liste, (String s1, String s2) -> s1.length() - s2.length() );
```

- Forme courte sans bloc et sans types :
  - Inférence de type sur les paramètres

```
Collections.sort(liste, (s1, s2) -> s1.length() - s2.length() );
```

Les expressions lambdas : autres exemples

- Avec `Runnable` :

```
Runnable r1 = () -> System.out.println("My Runnable");
Thread t = new Thread(r1);
t.start();
```

- Avec `ActionListener` :

```
JButton bouton = new JButton("Click me !");

bouton.addActionListener((e) -> bouton.setText("Thank You !"));
```

Une API de gestion de date simple !

La gestion des dates en Java

- La gestion de date et d'heure a longtemps été la bête noire des développeurs Java.
  - La classe `java.util.GregorianCalendar` et `java.util.Date` étaient disponibles pour répondre aux problèmes de manipulation de date et d'heure.
  - De nombreuses fonctionnalités étaient prévues mais leur utilisation relevait parfois du casse-tête.
- Le problème est complexe.
  - Travailler en base 60 pour les secondes et les minutes puis en base 24 pour les heures n'est pas très simple.
  - Mais la palme revient à la gestion des mois qui n'ont pas tous le même nombre de jours, voire pire puisque certains mois ont un nombre de jours variable suivant les années.
- Les ordinateurs utilisent une technique différente, en ne travaillant pas directement avec des dates et heures mais en nombre de secondes ou de millisecondes depuis une date de référence (généralement le 1er janvier 1970 à 0 heure).
- Ce mode de représentation n'est cependant pas très pratique pour un humain. La valeur 61380284400000 n'est pas très évocatrice, par contre 25/12/2014 est beaucoup plus parlant. C'est pourquoi de nombreuses fonctions permettent le passage d'un format à l'autre.

L'API de date de Java 8 : `java.time`

- Dans la version 8 de Java, la gestion des dates et des heures a été complètement repensée.
  - Au lieu de n'avoir qu'une ou deux classes dédiées à cette gestion et avec lesquelles il fallait jongler, de nombreuses classes spécialisées ont fait leur apparition.

`LocalDate`	Représente une date (jour mois année) sans heure.
`LocalDateTime`	Représente une date et une heure sans prise en compte du fuseau horaire.
`LocalTime`	Représente une heure sans prise en compte du fuseau horaire.
`OffsetDateTime`	Représente une date et une heure avec le décalage UTC.
`OffsetTime`	Représente une heure avec le décalage UTC.
`ZonedDateTime`	Représente une date et une heure avec le fuseau horaire correspondant.
`Duration`	Représente une durée exprimée en heures minutes secondes.
`Period`	Représente une durée exprimée en jours mois années.
`MonthDay`	Représente un jour et un mois sans année.
`YearMonth`	Représente un mois et une année sans jour.

Création d'une date / d'une 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)`
- ATTENTION :
  - Les objets créés sont immutables !!!

Traitement des dates - Dériver une date / une 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);`
- L'enum `java.time.temporal.ChronoUnit` donne la liste des unités utilisables représentant des dates/heures

Traitement des dates - Calculs sur les dates / les heures

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

Manipulation des dates

```java
final LocalDate localDate = LocalDate.of(2013, 05, 26);

// Year
Assert.assertEquals(2013, localDate.getYear());

// Month
Assert.assertEquals(Month.MAY, localDate.getMonth());
Assert.assertEquals(5, localDate.getMonthValue());

// Day
Assert.assertEquals(26, localDate.getDayOfMonth());
Assert.assertEquals(DayOfWeek.SUNDAY, localDate.getDayOfWeek());
Assert.assertEquals(146, localDate.getDayOfYear());

// Leap Year
Assert.assertFalse(localDate.isLeapYear());
Assert.assertTrue(LocalDate.of(2004, 05, 26).isLeapYear());

//---- Operations
final LocalDate localDate2 = LocalDate.of(2013, 04, 26);

// Before, After, Equal, equals
Assert.assertTrue(localDate.isAfter(localDate2));
Assert.assertFalse(localDate.isBefore(localDate2));
Assert.assertTrue(localDate.isEqual(LocalDate.of(2013, 05, 26)));
Assert.assertTrue(localDate.equals(LocalDate.of(2013, 05, 26)));

// plus & minus
Assert.assertEquals("2013-04-26", localDate.minusMonths(1).toString());
Assert.assertEquals("2013-06-05", localDate.plusDays(10).toString());
```

Manipulation des heures

```
final LocalTime localTime = LocalTime.of(12, 35, 25, 452_367_943);

// Hour, Minute, Second, Nanosecond
Assert.assertEquals(12, localTime.getHour());
Assert.assertEquals(35, localTime.getMinute());
Assert.assertEquals(25, localTime.getSecond());
Assert.assertEquals(452_367_943, localTime.getNano());

//---- Operations
// Before, After, Equal, equals
final LocalTime localTime2 = LocalTime.of(12, 35, 25, 452_367_942);
Assert.assertTrue(localTime.isAfter(localTime2));
Assert.assertFalse(localTime.isBefore(localTime2));
Assert.assertTrue(localTime.equals(LocalTime.of(12, 35, 25, 452_367_943)));

/// plus & minus
Assert.assertEquals("12:25:25.452367943", localTime.minusMinutes(10).toString());
Assert.assertEquals("17:35:25.452367943", localTime.plusHours(5).toString());

// Adjusters
Assert.assertEquals("05:35:25.452367943", localTime.withHour(5).toString());
```

Traitement des dates - Affichage des dates / des heures

- Formats « de base »
  - « locaux »
    - `DateTimeFormatter format1 = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);`
    - `String dateFormatee1 = date6.format(format1);` // 23 oct. 2013
    - `DateTimeFormatter format2 = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);`
    - `String dateFormatee2 = date6.format(format2);` // mercredi 23 octobre 2013
  - « spécifiques »
    - `DateTimeFormatter formatUK = DateTimeFormatter.withLocale(new Locale("en","UK"));`
    - `String dateFormateeUK = date6.format(formatUK);` // Oct 23, 2013
- Formats personnels (Grâce à la classe DateTimeFormatter)
  - `DateTimeFormatter formatPerso = DateTimeFormatter.ofPattern("eeee/MMMM (G)");`
  - `String dateFormateePerso = date6.format(formatPerso);`

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

Manipulation des collections avec les flux

Les flux - java.util.stream

- Il s'agit de réaliser des opérations fonctionnelles sur les collections
  - Potentiellement en parallèle !
- Toutes les classes implémentant l'interface `Collection` sont capables de fournir un objet de type `Stream`.
  - C'est cet objet qui est responsable de l'itération sur les éléments présents dans la collection.
- Intérêt
  - Facilité de mise en œuvre (boucle implicite, utilisation des lambdas facilitée)
  - Efficacité (évaluation paresseuse, traitements parallèles aisément définissable)
- Attention !
  - LES FLUX SONT A USAGE UNIQUE !

Principes de fonctionnement

- Les Streams peuvent être infinis et avec état.
- Ils peuvent être séquentiels ou parallèles.
  - Ils sont séquentiels par défaut
    - Inutile d'appliquer la méthode `sequential()`
  - Peuvent être transformés en flux parallèles
    - En appliquant la méthode `parallel()` sur l'objet Stream
- Principe :
  - Obtenir tout d'abord un Stream depuis une source
  - Réaliser une ou plusieurs opérations intermédiaires
  - Enfin une unique opération finale de terminaison.

Les sources

- Avec les collections :
  - Méthode `stream<E>()` de l'interface `Collection<E>`
  - `default Stream<E> parallelStream()`
- Avec les tableaux
  - 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`
- Avec un flux d'entrée (fichier)
  - Méthode `Stream<String> lines()` de la classe `java.io.BufferedReader`
  - Méthode `Stream<String> lines(Path p)` de `java.nio.file.Files`
  - Méthode `Stream<Path> list(Path p)` de `java.nio.file.Files`

Les opérations

- Les différents types d'opérations disponibles sur l'interface Stream sont soit des opérations intermédiaires permettant de faire du filtrage ou du tri, soit des opérations terminales qui vont appliquer un traitement sur les éléments éventuellement traités par les opérations intermédiaires.
  - Les opérations intermédiaires sont LAZY !!!
  - Leur traitement ne s'applique qu'a l'exécution de l'opération terminale !
- Opérations intermédiaires
  - `filter`, `map`, `flatMap`, `peel`, `distinct`, `sorted`, `limit`.
- Opérations terminales
  - `forEach`, `toArray`, `reduce`, `collect`, `min`, `max`, `count`, `anyMatch`, `allMatch`, `noneMatch`, `findFirst`, et `findAny`.

Exemples : Traitements (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)`

Exemples : Traitements (terminaisons)

- 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)`

Retour des opérations terminales

- Les opérations terminale peuvent retourner un objet apparu également avec Java 8
  - Optional
- C'est un conteneur pour une valeur pouvant être `null`.
- Il possède deux états : contenant une valeur ou ne contenant rien.
  - La méthode `isPresent()` permet de savoir si un résultat est présent

Les réductions

- Méthode terminale `reduce()`
- Objectif :
  - Agréger le contenu d'un Stream pour fournir un résultat unique.
    - Application d'une opération

```
Stream<Personne> sp = maCollection.stream();
sp = maCollection.stream();

Integer sum = sp.filter(x -> x.getAge() > 18)
                .map(x -> x.getAge())
                .reduce(1, (x,y) -> x+y);

System.out.println(sum);
```

Création de collections

- Méthode terminale `collect()`
- Permet de récupérer le résultat des opérations successives sous une certaines forme.
- Cette forme est définie par un 'Collector' (implémentant l'interface `Collector`).
- On pourra donc obtenir un résultat sous forme de `Set`, de `Map`, de `List`, ...
- Une classe utilitaire très pratique :
  - `java.util.stream.Collectors`
  - Implémente des opérations de variées, telles que convertir des streams en collections, ou l'agrégation d'éléments, ...