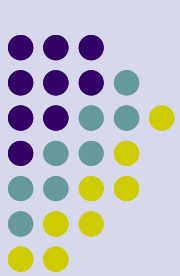
Les nouveautés de JAVA 8 à Java 11

m2iformation.fr

Les nouveautés de JAVA 8 à Java 11

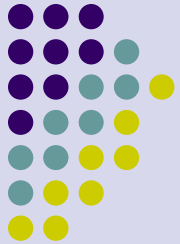


NGASSA

Hubert Landry

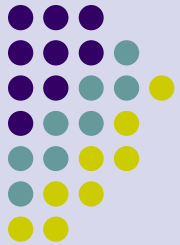
Senior Ing. chez Softeam Group

Introduction



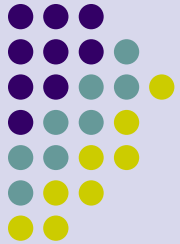
- Historique des versions java
  - 1996 : JDK 1.0
  - 1997 : JDK 1.1
  - 1998 : JDK 1.2, appelé Java 2
  - 2000 : JDK 1.3
  - 2002 : JDK 1.4
  - 2004 : JDK 1.5, appelé Java 5
  - 2006 : JDK 1.6, appelé Java 6
  - 2011 : JDK 1.7, appelé Java 7
  - 2014 : JDK 1.8, appelé Java 8.
  - septembre 2017 : JDK 9.
  - mars 2018 : JDK 10.
  - septembre 2018 : JDK 11.
  - mars 2019 : JDK 12

Introduction



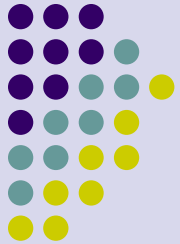
- 1996 : jdk 1.0
  - Nom de code OAK(Solaris Windows Mac os et linux)
  - Runtime Java
  - JVM
  - Et premières bibliothèques de classe
  - Outils(compilateur java ect…)
  - JRE(arrivé un peu plus tard)
- 1997: jdk 1.1
  - Reflection
  - RMI
  - Java bean
  - Inner Class
  - JDBC

Introduction



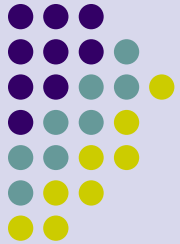
- 1998: jdk1.2
  - Collection Framework
  - Compilateur JIT
  - SWING 1.0
  - Drag & Drop
  - Java 2D
  - Amélioration JDBC

Introduction



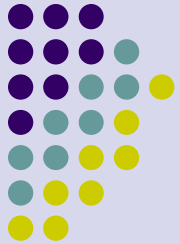
- 2000: jdk1.3
  - Amelioration dans tous les aspects de java
  - Indexation jar
  - Java sound
  - Jndi
- 2002: Jdk 1.4
  - Traitement XML
  - Java web start
  - JDBC 3.0 API
  - Exceptions chainnées
  - Expressions regulières
  - Image I/O API

Introduction



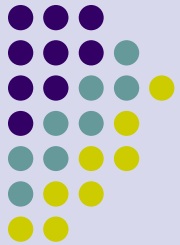
- 2004: J2SE 1.5 (java 5 Tiger)
  - Instrumentation
  - Annotations
  - Generics
  - Boucle for améliorée
  - Autoboxing / Unboxing
  - Imports statiques
- 2006 : Java SE 6
  - JDBC 4.0
  - Java compiler API
  - Support langage de scripts
  - Annotation pluggables
  - Web services intégrés

Introduction



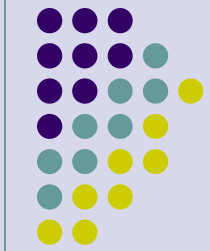
- 2011: Java SE 7
  - Syntaxe Diamant
  - NIO2
  - Try with ressources
  - Gestion des exceptions multiples
  - String dans les blocs switch

Introduction



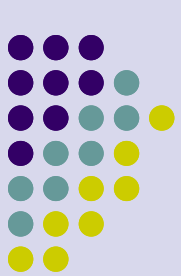
- 2014: Java SE 8
  - Lambda expression
  - Pipeline/Stream
  - Date / time API
  - Default Method
  - Ameliorations API Colection
  - Operations parallèles
- 2017 : Java SE 9
  - JPMS – Jigsaw project
  - REPL Jshell
  - Amélioration API Stream
  - HTTP Client 2

Introduction

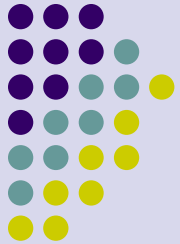


- Mars 2018 : Java SE 10
  - Inférence de type variables locales
  - Graal – JIT compiler
  - Parallèlisation complète de GC G1
- Septembre 2018 : Java SE 11
  - Epsilon GC
  - Constante dynamique dans les fichiers de classe
  - Suppression de modules Java EE et corba

I. Rappel des nouveautés en Java 8



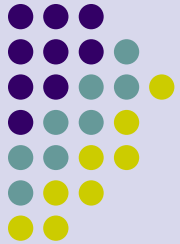
Introduction



- Java 8 est probablement la version la plus importante depuis la création du langage du fait de l'impact sur:
  - Le language
  - Le compilateur
  - Les libraries
  - Les API
  - Le runtime (JVM)

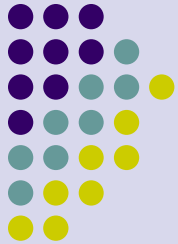
Introduction

Modifications du langage



- 1: Les interfaces
  - Méthodes par défaut
  - Methodes statiques
- 2: Référence de méthodes
- 3: Expressions lambda
- 4: Annotation de type

Modifications du langage (Interfaces)



Quelques situations rencontrées dans les versions antérieures à java 8 (1)

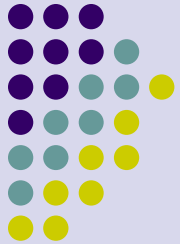
Lorsque plusieurs classes partagent la même implémentation

- Soit I une interface fournissant une méthode m1()
- Soient C1, C2,... CN des classes implantant I

a) Gestion du code commun : Si certains Ci ont la même implémentation de m1() quelle stratégie adopter?

b) Evolution : Si on ajoute une methode m2() à l'interface par la suite, il faut modifier toutes les Ci

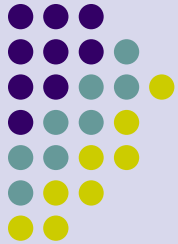
Modifications du langage (Interfaces : Default Methods)



- Solution apportée par java 8 : Permettre aux méthodes déclarées dans les interfaces d'avoir une implémentation !
- Syntaxe
  - La syntaxe est simple et sans surprises : il suffit de fournir un corps à la méthode, et de la qualifier avec le mot-clé default (mot-clé déjà utilisé pour les annotations, si vous vous rappelez).

```
public interface Foo {
            public default void foo() {
                        System.out.println("Default implementation of foo()");
             }
}
```

Modifications du langage (Interfaces : Default Methods)



- Implémentation

```
public interface Itf {
    /** Pas d'implémentation - comme en Java 7 et antérieur */
    public void foo();

   /** Implémentation par défaut, qu'on surchargera dans la classe fille */
   public default void bar() {
    System.out.println("Itf -> bar() [default]");
   }
  /** Implémentation par défaut, non surchargée dans la classe fille */
   public default void baz() {
    System.out.println("Itf -> baz() [default]");
   }
}
```

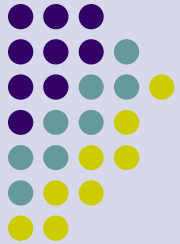
```
public class Cls implements Itf {
        @Override
        public void foo() {
         System.out.println("Cls -> foo()");
        }
        @Override
        public void bar() {
         System.out.println("Cls -> bar()");
         }
        /* NON SURCHARGE
        @Override
        public void baz() {
        System.out.println("Cls -> baz()");
        }*/
}
```

```
public class Test {
public static void main(String[] args) {
        Cls cls = new Cls();
        cls.foo();
        cls.bar();
        cls.baz();
}
}
```

Résultat :

```
Cls -> foo()
Cls -> bar()
Itf -> baz() [default]
```

Modifications du langage (Interfaces: Default Methods)



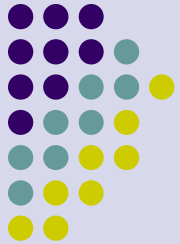
→ Nouveau concept induit : Les traits ou Extension

- Traits : Process permettant d'encapsuler un ensemble cohérent de méthodes, à caractère transverse et réutilisable et en général composé de :
  - une méthode abstraite qui fait le lien avec la classe sur laquelle il est appliqué
  - un certain nombre de méthodes additionnelles, dont l'implémentation est fournie par le trait lui-même car elles sont directement dérivables du comportement de la méthode abstraite.

- Exemple : Comparable et Ordorable
  - Comparable<T>
    - Interface du JDK (utilisé notamment dans les API)
    - Expose une seule méthode : public int compareTo(T o)
    - Ne pouvant modifier l'interface nous allons l'étendre

```
public interface Orderable<T> extends Comparable<T> {
 // La méthode compareTo() est définie
 // dans la super-interface Comparable
 public default boolean isAfter (T other) {
  return compareTo (other) > 0;
 }
 public default boolean isBefore (T other) {
  return compareTo (other) < 0;
 }
 public default boolean isSameAs (T other) {
  return compareTo (other) == 0;
 }
}
```

Modifications du langage (Interfaces: Default Methods)



- On peut l'appliquer à une classe...

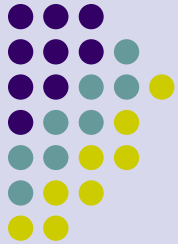
```
public class Person implements Orderable<Person> {
   private final String name;
   public Person (String name) {
      this.name = name;
   }
   @Override
   public int compareTo (Person other) {
      return name.compareTo (other.name);
   }
}
```

```
public class Test {
 public static void main (String[] args) {
  Person laurel = new Person ("Laurel");
  Person hardy = new Person ("Hardy");
  System.out.println ("Laurel compareto Hardy : " + laurel.compareTo (hardy));
  System.out.println ("Laurel > Hardy : " + laurel.isAfter (hardy));
  System.out.println ("Laurel < Hardy : " + laurel.isBefore (hardy));
  System.out.println ("Laurel == Hardy : " + laurel.isSameAs (hardy));
 }
}
```

- Qui beneficie des methodes isBefore() and isAfter()

```
Laurel compareto Hardy : 4
Laurel > Hardy : true
Laurel < Hardy : false
Laurel == Hardy : false
```

Modifications du langage (Interfaces : Default Methods)



- La problématique des diamants ou losanges

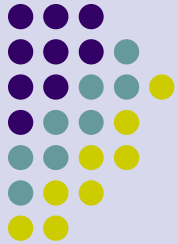
```
public interface InterfaceA {
  public default void foo() {
    System.out.println("A -> foo()");
  }
}

public interface InterfaceB {
   public default void foo() {
    System.out.println("B -> foo()");
   }
}
private class Test implements InterfaceA, InterfaceB {
 // Erreur de compilation : "class Test inherits unrelated defaults for foo() from types InterfaceA and InterfaceB"
}
```

- Pour résoudre le conflit, une seule solution : implémenter la méthode au niveau de la classe elle-même, car l'implémentation de la classe est toujours prioritaire.

```
public class Test implements InterfaceA, InterfaceB {
   public void foo() {
     System.out.println("Test -> foo()");
   }
}
```

- Inconvenient: Le code des méthodes par défaut n'est plus appelable directement. Par exemple super().foo()
- Pour y remedier Java 8 ajoute la syntaxe *<Interface>.super.<méthode>*
- *LE PROBLÈME DE L'HÉRITAGE EN DIAMANT EST DONC RÉSOLU PAR UNE VÉRIFICATION DE COMPATIBILITÉ AU NIVEAU DU COMPILATEUR, PLUS UNE SYNTAXE POUR ACCÉDER SÉLECTIVEMENT AUX IMPLÉMENTATIONS PAR DÉFAUT DES INTERFACES.*

Modifications du langage (Interfaces: Default Methods)



- Compilation ou Runtime ?

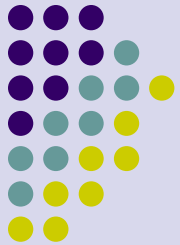
```
Object proxy = Proxy.newProxyInstance(
     Test.class.getClassLoader(),
     new Class[]{InterfaceA.class, InterfaceB.class},
    (tagetProxy, targetMethod, targetMethodArgs) -> {
       System.out.println("Calling " + targetMethod.toGenericString());
       return null;
});

((InterfaceA) proxy).foo();
((InterfaceB) proxy).foo()
```

- Quel résultat ?

```
Calling public default void InterfaceA.foo()
Calling public default void InterfaceA.foo()
```

- Étrange non?

Modifications du langage (Interfaces : static Methods)

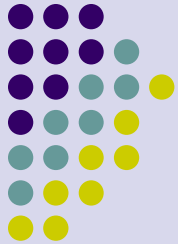


- Les methodes statiques introduites en java 8 sont d'un point de vue fonctionnel, similaire au méthodes par défaut, à la seule différence qu'on ne puisse pas les surcharger.

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

- Elles sont généralement utilisées en tant qu'utilitaires (test de nullité, tri etc…)
- Permettent de sécuriser le code qu'on ne souhaite pas surcharger

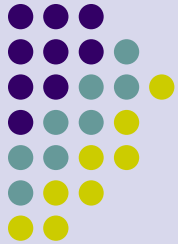
Modifications du langage (Interfaces : interface fonctionnelle)



- Une interface fonctionnelle est une interface qui déclare une seule méthode abstraite.
- Cette méthode abstraite est appelée la méthode fonctionnelle de l'interface fonctionnelle.
- Une interface fonctionnelle peut cependant déclarer d'autres méthodes non abstraites mais doit fournir une implementation par defaut (default) à toutes ces méthodes
- Une interface fonctionnelle peut aussi définir des méthodes statiques.
- Exemple : java.util.function.Predicate

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef) ? Objects::isNull : object -> targetRef.equals(object);
    }
}
```

Modifications du langage (Interfaces : interface fonctionnelle)



- Revenons un peu en arrière
  - Avant Java 8 comment passait-t-on une fonction en paramètre

```
public class Name {
 private String firstName;
 private String lastName;
 public Name(String firstName, String lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
}

 public String getFirstName() {
   return firstName;
 }
 public String getLastName() {
   return lastName;
 }
}
```

```
public class NameParser {

  public T parse(String name, Creator creator) {
     String[] tokens = name.split(" ");
     String firstName = tokens[0];
     String lastName = tokens[1];
     return creator.create(firstName, lastName);
  }
}



public interface Creator {
  T create(String firstName, String lastName);
}
```

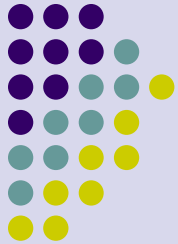
  - Pour utiliser notre NameParser, nous devons l'appeler en lui passant une instance d'une classe implémentant l'interface Creator. Nous avons donc recours à une classe anonyme :

```
NameParser parser = new NameParser();
Name res = parser.parse("Eric Clapton", new Creator<name>() {
      @Override
      public Name create(String firstName, String lastName) {
       return new Name(firstName, lastName);
      }
    });
```

- Responsabilités sont clairement dissociées

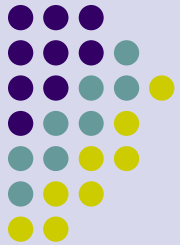
/!\ syntaxe résultante est très verbeuse et la lisibilité du code est rendue difficile…

Modifications du langage (Interfaces : interface fonctionnelle)



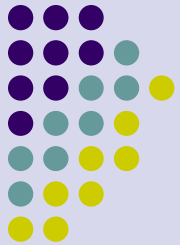
- Java 8 apporte une réponse au problème précédent grâce aux “interfaces fonctionnelles”. Mais pour cela nous devons évoquer des notions que nous verrons dans la suite .
- Le but d’une interface fonctionnelle est de définir la signature d’une méthode qui pourra être utilisée pour passer en paramètre :
  - une référence vers une méthode statique
  - une référence vers une méthode d’instance
  - une référence vers un constructeur
  - une expression lamba.
- Nous y reviendrons

Modifications du langage (Interfaces : interface fonctionnelle du jdk)



- Package *java.util.function*
  - Rôle : Fournir un certain nombre d'interfaces fonctionnelles standards
  - Contenu (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
- Autres interfaces : Versions dédiées aux types simples ou prenant 2 paramètres

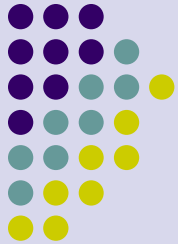
Modifications du langage (Les expressions Lambda: historique)



- L'ajout des expressions lambda dans le langage Java a été un processus long qui a nécessité plus de huit années de travail.
  - 2009 : lancement du projet Lambda
  - 2006 : propositions pour ajouter le concept de « clôture »/« fermeture » (closure)
- 1997 : ajout du concept de « classe anonyme » (anonymous inner class)
- 1941 : travaux d'Alonzo Church sur la théorie du calcul, d'où vient la terminologie

- Quelques contraintes qui ont amené la réticences à plusieurs propositions qui ont été faites entretemps :
  - ne pas ajouter un nouveau type fonction au lagage pour éviter les écueils des generics
  - s'appuyer sur les interfaces qui existent déjà et permettent donc une transition en douceur
  - les expressions lambda ne sont pas transformées en classes par le compilateur : elles n'utilisent donc pas les classes anonymes internes
- Ce qui a impliqué quelques déconvenues à leur intégration au langage (bien que ceux-ci constituent un véritable atout au langage Java).
- Ce sont les premiers pas vers la programmation orientée fonction, appelée aussi programmation fonctionnelle

Modifications du langage (Les expressions Lambda: historique)



Pourquoi des lambda expressions JAVA ?

- Commençons avec un code qui itère sur une collection d'objets modifiables

```
List<Point> pointList = Arrays.asList(new Point(1,2), new Point(2,4));
for(Point p : pointList) {
  p.translate(1,1);
}
```

  - La ligne 2 de ce code est traduit par le compilateur en le code qui suit :

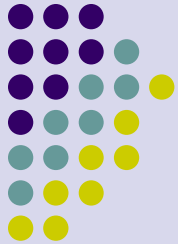
```
Iterator pointItr = pointList.iterator();
while (pointItr.hasNext()) { ((Point) pointItr.next()).translate(1,1); }
```

/!\ Problème : le compilateur génère du qui sera toujours séquentiel

  - Nous souhaiterions une exécution qui puisse être parallèle à la demande, c'est-`a-dire selon la mise en œuvre de la méthode forEach
  - D'où, nous souhaitons quelque chose comme ceci :

```
List<Point> pointList = Arrays.asList(new Point(1,2), new Point(2,4));
pointList.forEach(/*translation d'un point selon le vecteur (1,1)*/);
```

Modifications du langage (Les expressions Lambda: Definition)



→ Une expression lambda est conceptuellement une fonction anonyme : sa définition se fait sans déclaration explicite du type de retour, ni de modificateurs d'accès ni de nom. C'est un raccourci syntaxique qui permet de définir une méthode directement à l'endroit où elle est utilisée.

- Sa syntaxe :
  - un ensemble de paramètres, d'aucun à plusieurs
  - l'opérateur ->
  - le corps de la fonction

Elle peut prendre deux formes principales :

- (paramètres) -> expression;
- (paramètres) -> { traitements; }

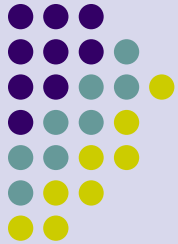
→Une expression lambda est aussi un objet (une instance d'une interface (fonctionnelle))

```
Runnable r = () -> {}; // crée une lambda expression
// et affecte une référence vers cette lambda expression à r
Object o = r; // transtypage vers le haut, comme pour une référence/un objet
```

→ Une lambda peut être utilisée là où une interface fonctionnelle est déclarée

- Interface fonctionnelle => une méthode => pas d'ambiguïté
- P.ex., « public void forEach(Consumer<? super T> consumer); »
  permet d'écrire « pointList.forEach(p -> p.translate(1, 1)); »

Modifications du langage (Les expressions Lambda: Definition)



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

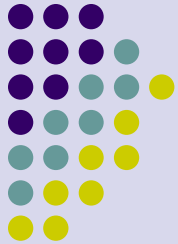
Modifications du langage (Les expressions Lambda: Definition)

- **Exemples de *lambda expressions***

```
1- (int x, int y) -> x + y // retourne la somme des deux arguments
2- (x, y) -> x - y // retourne la diff´erence
3- () -> 42 // pas d'argument, retourne 42
4- (String s) -> System.out.println(s) // affiche l'argument et ne retourne
rien
5-  x -> 2 * x // un argument (sans parenth`eses)
6- c -> f int s = c.size(); c.clear(); return s; g // type de l'argument
poss´edant les m´ethodes
size() et clear(), p.ex. une collection
```

- *Types des arguments explicites (1, 4) ou inférés (2, 5, 6)*
  - ***Pas de mélange entre « explicite » et « inféré »***
- *Le corps peut ˆetre un bloc (6) ou une expression (1–5)*
  - ***Bloc retournant une valeur (dit value-compatible) ou rien (dit void-compatible)***
  - ***Idem pour l'expression : une valeur (1, 2, 3, 5) ou rien (4)***

Modifications du langage (Les expressions Lambda: Définition)

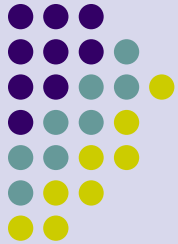


- **Contexte d'exécution d'une *lambda expression***
  - Une lambda expression s'exécute dans un contexte
    - Comme attribut de classe : class Bar { Foo foo = () → 42; }
    - Comme variable locale : void bar() f Foo foo = () → 42; g;
  - Elle peut utiliser les variables de son contexte, y compris « this » lorsqu'elle est imbriquée dans une instance de classe
  - Règles classiques d'utilisation et de nommage des éléments du contexte
    - Un exemple légal : l'argument « i » de la lambda cache l'attribut « i »

      *class Bar { int i; Foo foo = i → i * 2; };*
  - Un exemple illégal : « i » est d´ej`a une variable locale de « bar »

    *void bar() {int i; Foo foo = i → i * 2; };*

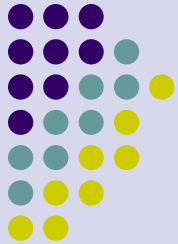
Modifications du langage (référence de methodes: Définition)



- Cas particulier d'une lambda expression avec un seul argument
  - l'expression est un unique appel à une méthode avec un seul argument
    - Par exemple : *str* → *Integer.parseInt(str)*
    - Simplification de l'écriture avec la forme « référence de méthode »
      - Pour le même exemple : *Integer::parseInt*
- Plus généralement, une lambda expression peut être représentée par une méthode concrète d'une classe

➔ *Une référence de méthode est un raccourci d'écriture d'une lambda expression* avec un argument et l'expression formée de l'appel unique de la méthode référencée

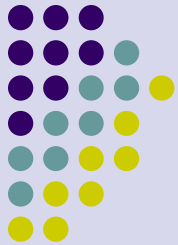
Modifications du langage (référence de méthodes: Syntaxe)



- Syntaxe
  - ReferenceType::Identifier pour méthode de classe,
    - *p.ex. « Integer::parseInt »*
  - ObjectReference::Identifier pour méthode d'instance
    - p.ex. « System.out::println »
  - • ReferenceType::new pour constructeur,
    - p.ex. « ArrayList::new »

Forme «l*ambda expression* »	Forme «référence de méthode »	Type de méthode de référence
str → Integer.parseInt(str)	Integer::parseInt	Statique
Instant then = Instant.now(); t → then.isAfter(t)	Instant.now()::isAfter	Lié (*bound*)
str → str.toLowerCase()	String::toLowerCase	Libre (*unbound*)
() → new TreeMap<K,V>	TreeMap<K,V>::new	Constructeur de classe
len → int[len]	int[]::new	Constructeur de tableau

Modifications du langage (Type annotation: définition)



- De manière générale, Les annotations ont été spécifiées dans la JSR 175 (implémenté dans java SE 5): leur but est d'intégrer au langage Java des métadonnées.
- Avant java 8, les annotations sont essentiellement utilisées dans des déclarations.
- Le renforcement de cette notion d'annotation dans Java est donc de Java - *un langage fortement typé comme tout le monde le sait* - encore plus typé.
- Les annotation de type (comme le nom indique) sont donc des annotations qui peuvent être placée partout où on utilise un type.
  - Operateur *new*
  - Un type forcé (*cast*)
  - À coté des clauses *implements* *ou throws*

Librairie: Focus sur queslques APIs

- 1: L’API Stream
- 2: Nouvelle API Date et Time(JSR 310) de Java8
- 3: Optional
- DateTimeAPI
- JavaFX (eventuellement)

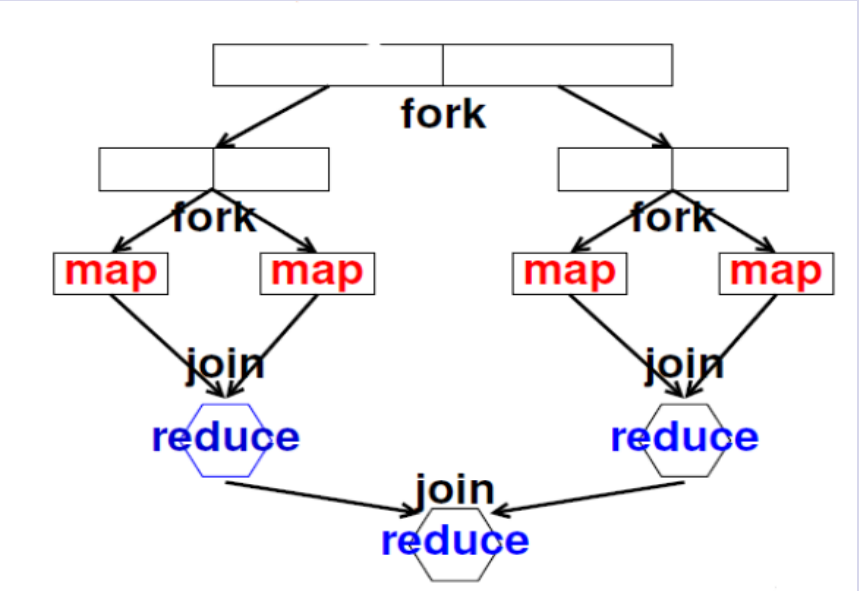
Librairie: L'API Stream

- Rappel des motivations générales qui ont induit l'utilisation des lambda
  - Motivation 1 : introduire des éléments de programmation orientés fonction
  - Motivation 2 : raccourcir l'écriture avec le nouveau sucre syntaxique « -> »
  - Motivation 3 : permettre des exécutions parallèles
    - C'est cette motivation qui donne la bibliothèque des Streams

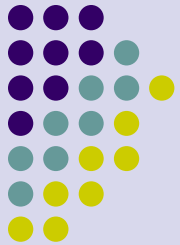
  Exemple : Imaginons une collection de grande taille ET un ordinateur avec plusieurs coeurs

  → Mise en oeuvre du paradigme de programmation « Map/Reduce »

```
double maxDistance =
pointList.parallelStream()
.map(p -> p.distance(0, 0))
.reduce(Double::max)
.orElse(0.0);
```

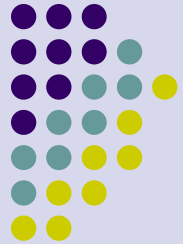


Librairie
L'api Stream (Définition)



- java.util.stream.Stream : *Abstraction d'un flux d'élément sur lequel on veut faire des calculs*
- Du coup un stream n'est pas:
  - Une Collection car un Stream ne contient pas d'élément
  - Un Iterator car un Stream correspond à un calcul complet et pas à une étape du calcul. Un Stream à une vision plus globale du traitement.
- Comme un Stream ne stocke pas les données, elles proviennent d'une source
  - A partir de valeurs
    - Stream.empty(), Stream.of(E… element), Stream.ofNullable(E element)
  - A partir d'une collection
    - collection.stream()
  - A partir d'un fichier
    - Files.lines(Path path)
  - Ect…

Librairie
L'api Stream (Les opérations)



- L'API des Streams est basé sur des opérations sans état (stateless)
  - filter, map/flatMap, reduce
  - Dites « *Short circuit* », capable d'arrêter le calcul si résultat est trouvé (*limit(), findFirst()/findAny(), takeWhile()*)
  - Les opérations intermédiaire qui modifie le Stream
  - les opérations terminales qui lancent le calcul

Librairie :
L'api Stream (Les opérations)

- Opérations intermédiaires(1/2)
  - Sélectionne si un élément reste dans Stream *filter(Predicate)*
  - Transforme un élément Stream

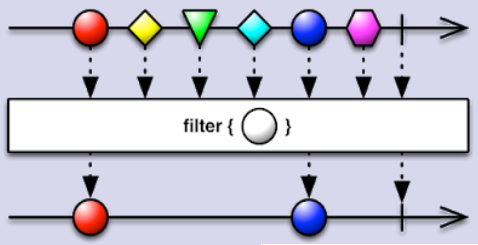
    <R> Stream <R> map(

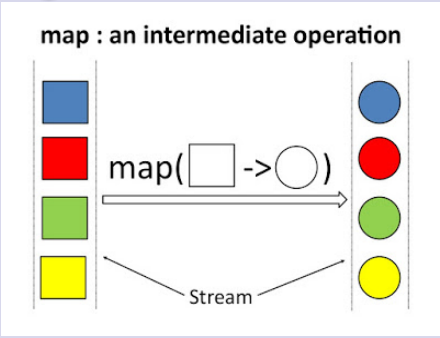
    Function<? super E, ? extends R>)

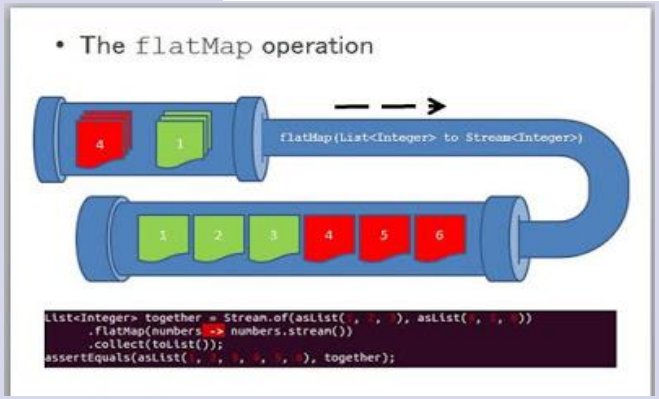
  - Tranforme un élement en une série d'élements

    <R> Stream<R> flatMap(

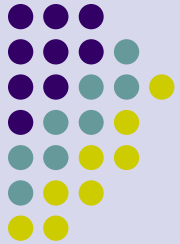
    Function<? super E, ? extends Stream<R>>)
  - *Saute des éléments*
    - *Stream<E> skip(int length)*
  - *Sélectionne les premiers éléments*
    - *Stream<E> limit(int maxSize)*





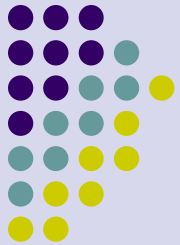


Librairie :
L’api Stream (Les opérations)



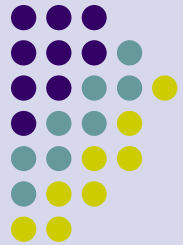
- Opérations intermédiaires(2/2)
  - Supprime les doublons
    - *Stream<E> distinct()*
  - Trie les élements
    - *Stream<E> sorted(Comparator<? super E>)*
  - Obtenir les éléments au milieu du Stream (pour débugger)
    - *Stream<E> peek(Consumer<? super E>)*
  - Sélectionner/supprimer des élements (stop après)
    - *Stream<E> takeWhile(Predicate<? super E>)*
    - *Stream<E> dropWhile(Predicate<? super E>)*

Librairie : L’api Stream (Les opérations)



- Opérations Terminales
  - Compte les éléments
    - *long count()*
  - Appel le consumer pour chaque élement
    - *Stream<E> forEach(Consumer<? super E>)*
    - Stream<E> forEachOrdered(Consumer<? superE>)
  - Vrai si tout les/au moins un élement(s) qui match
    - *allMatch(Predicate<? super E>)*
    - *anyMatch(Predicate<? super E>)*
  - Trouver le/un premier élement
    - Stream<E> findFirst()
    - Stream<E> findAny()
  - Créer un tableau
    - *E[ ] toArray(IntFunction<E[ ]>)*
  - aggrège les données (sans mutation)
    - *reduce(T seed, BinaryOperator<T> reducer)*
  - aggrège les données (mutable)
    - *T collect(Collector<E,A,T>)*

Librairie :
L'api Stream (Les opérations)



- Des Streams de type primitif
  - IntStream, LongStream et DoubleStream
    - Évite le boxing
    - Possède des méthodes spécifiques (sum, average, etc.)

    Sur un Stream, il existe plusieurs version de map(), mapToInt, mapToDouble, mapToLong qui renvoie des Stream de type primitif (même chose pour flatMap)

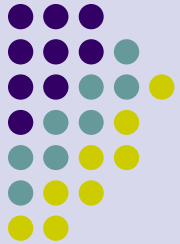
    Exemple d'utilisation : Afficher les valeurs inférieur à 100 de la suite:

    – U0 = 1

    – Un = 2 * Un-1 + 1

    Remarque: Un stream peut être infinie

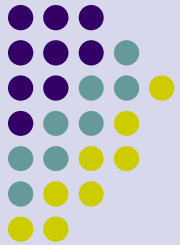
Librairie :
L’api Stream (Les streams parallèles)



- Généralités:

- Les streams parallèles induisent le réelle notion de calcul simultanés, avec chaque élément dans un thread séparé.
- L’existence de thread ne garanti pas forcément la performance. Il faut tenir compte:
  - De l’OS
  - Du matériel (nombre de cœur, vitesse du processeur ect…)
- Du fait de l’utilisation de thread, la charge est plus importante

Librairie :
L'api Stream (Les streams parallèles)



- Création d'une stream parallèle :
  - À partir d'une stream :
    - Exemple:

```
Stream stream = Stream.of("John", "Mike", "Ryan","Donald", "Matthew");
Stream parallelStream = stream.parallel();
```

  - À partir d'une source :
    - Exemple :

```
Stream parallelStream = Arrays.asList("John", "Mike", "Ryan","Donald", "Matthew").parallelStream();
```

- Mise en œuvre de stream Parallèle
  - Exemple: voir Algo nombre premier
- Mise en œuvre de l'aléa du fait des threads
  - Exemple: voir affichage aléatoire
- Mise en œuvre de la performance
  - Exemple : voir performance avec les thread

Librairie : L'api Optional

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

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

- Optional permet de le rendre explicite ;
- ça ne résoud que partiellement le problème (il est au niveau du langage lui-même) ;
- ***Optional*** est un monad comme Stream, en plus de représenter une valeur (ou non) d'un calcul, il est possible d'effectuer des opérations directement sur un Optional
  - Exemple : on demande à l'Optional de faire le calcul

```
Optional<String> opt = …
if (opt.isPresent()) {
   System.out.println(opt.get());
}
```

```
Optional<String> opt = …
opt.ifPresent(v -> System.out.println(v));
```

Librairie : L'api Optional

- Creer un Optional
  - Optional possède des méthodes statiques de
    - création (static factory methods)
      - Créer un Optional sans valeur

        Optional.empty()
    - Créer un Optional avec une valeur non null

      Optional.of(E element)
    - Créer un Optional qui peut être null

      Optional.ofNullable(E elementOrNull)

Exemple:

```
OptionalInt result = ...
if (result.isPresent()) {
System.out.println("result is "
+ result.get());
} else {
System.out.println("result not
found");
}
```

```
OptionalInt result = …
System.out.println(result
.map(value - > "result is " + value)
.orElse("result not found"));
```

Librairie : L'api Optional

- Utilisation d'Optional
  - *Il ne faut pas :*
    - Stocker un Optional dans un champ !

      → Double dé-référencement inutile

```
public class Foo {
 private final Optional<Bar> bar;
 public Foo(Optional<Bar> bar) {
    this.bar = bar;
 }
 public Optional<Bar> getBar() {

    return bar;
 }
}
```

```
public class Foo {
 private final Bar bar; // maybe nul
 public Foo(Optional<Bar> bar) {
   this.bar = bar.orElse(null);
 }
 public Optional<Bar> getBar() {
   return Optional.ofNullable(bar);
 }
}
```

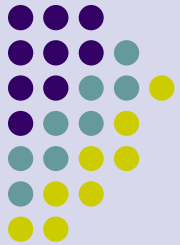
    - Avoir une Collection ou Map de Optional

      → Autant ne pas mettre les trucs qui existe pas dans la collection

```
List<Foo> foos = …
List<Optional<Bar>> bars =
foos.stream()
.map(foo - > foo.getBar())
.collect(Collectors.toList())
```

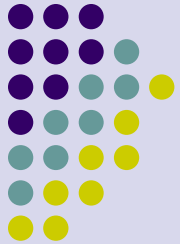
```
List<Foo> foos = …
List<Bar> bars =
foos.stream()
.flatMap(foo - >
foo.getBar().stream())
.collect(Collectors.toList())
```

Librairie : L’api Time(Définition)



- Principe général
  - Les dates et heures sont maintenant représentées par des classes du package java.time dont les instances sont immuables.
- Problématiques évoquées dans la JSR 310 :
  - Remplacer *java.util.Date*, *Calendar*, *TimeZone*, *DateFormat*
  - *Une Api :*
    - *simple,*
    - *flexible,*
    - *sécurisé,*
    - *fortement typée (Units, Fields, and Chronologies)*
    - *adaptée aux opérations sur les dates(formattage, periodicité, ect…)*
    - *Rétrocompatible avec les librairie existantes sur les dates*
    - *Supportant la régionalisation*
    - *Compatible avec avec JDBC, java.sql.Date/Time/Timestamp*
    - *ISO 8601 Calendar for global business*

Librairie : L'api Time(date/time type)

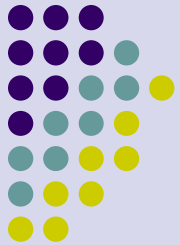


Local Date

- Stores year-month-day
  - 12th March 2017
- Use cases: birthdays, start/end dates, holiday dates

```
LocalDate current = LocalDate.now();
LocalDate date = LocalDate.of(2013,Month.SEPTEMBER,12);
if (current.isAfter(date)) …

String str = date.toString();  // 2013-09-12

boolean leap = date.isLeapYear();
int monthLen = date.lengthOfMonth();
```

Librairie : L'api Time(date/time type)



Local Date-Time

- Stores **`LocalDate`** and **`LocalTime`**
  - 12th September 2013 at 13:30
- Use case: local date-time a flight takes off

```
dt1 = LocalDateTime.now();
dt2 = LocalDateTime.of(2013,SEPTEMBER,12, 13,30);

dt1 = dt1.plusDays(2).minusHours(1);
dt1 = dt1.with(next(TUESDAY));

dt2.toString();  // 2013-09-12T13:30

dt2 = dt2.truncatedTo(MINUTES);
```

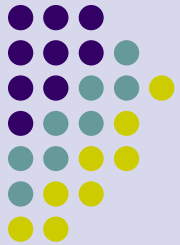
Librairie : L'api Time(date/time type)

Instant

- Stores nanoseconds from 1970-01-01Z
- Closest equivalent to java.util.Date
- Use case: timestamp in logging

```
instant1 = Instant.now();
instant2 = Instant.now();

if (instant1.isAfter(instant2)) { … }
```

Librairie : L'api Time(Time Zone)

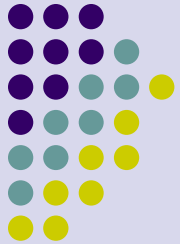


Time zone design

- Four classes manage time-zone complexity

- **`ZoneId`** - "Europe/Paris", as per **`java.util.TimeZone`**
- **`ZoneOffset`** - "-05:00", offset from UTC/Greenwich
- **`ZoneRules`** - behind the scenes class defining the rules
- **`ZonedDateTime`** - main date/time class with time-zones

Librairie : L'api Time(Duration)

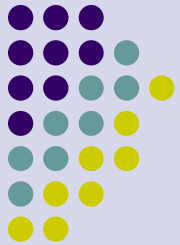


Duration

- Time-based amount
  - hours, minutes, seconds and nanoseconds
  - some support for 24-hour days
- Use cases: sport stopwatch, timeout

```
duration = Duration.ofHours(6);          // PT6H
duration = duration.multipliedBy(3);     // PT18H
duration = duration.plusMinutes(30);     // PT18H30M

dt = LocalDateTime.now();
dt = dt.plus(duration);
```

Librairie : L'api Time(Period)



Period

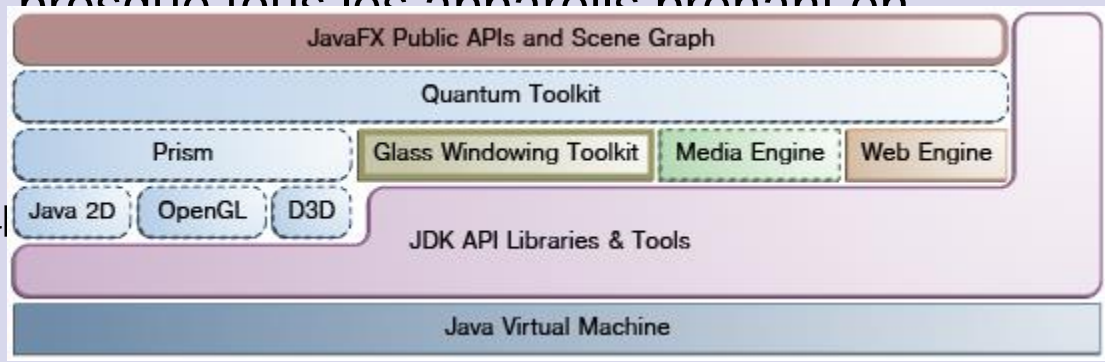
- Date-based amount
  - years, months and days
- Use cases: length of pregnancy, length of holiday

```
period = Period.ofMonths(9);    // P9M
period = period.plusDays(6);    // P9M6D

dt = LocalDateTime.now();
dt = dt.plus(period);

period = Period.between(startDate, endDate);
```

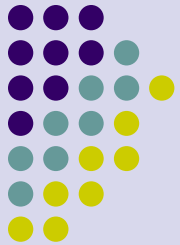
Librairie : JavaFX(eventuelle)

- Sortie en 2008, avec pour objectif:
  - Remplacer Swing
  - Repondre à la concurrence de Silverlight de Microsoft et Flex d'Adobe

- JavaFX est une bibliothèque permettant de créer des applications "rich client" avec Java.
- Il fournit une API pour la conception d'applications à interface graphique fonctionnant sur presque tous les appareils prenant en charge Java.
- Architecture :
  - JavaFX utilise un pipeline gra



Les nouveautés en Java 9



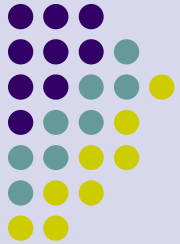
- Objectifs
  - Qu'est-ce qu'un Module?
  - Dépendance (de Module)
  - Les Directives (require, export, open, uses, …)
  - Graphes
  - TP: création d'un jeu
    - Sans Module
    - Avec Module
    - Avec Service
  - Jar modulaire
  - Process de migration du code existant
  - Maven
  - Custom runtime

Les nouveautés en Java 9

- Objectifs (1/2)
  - Depuis toujours : (Classes, enum, interfaces, …) -> package -> jar [ ] → classpath

  - Problèmes soujacents :
    - Comment limiter la portée d'éléments declarés en tant que public dans une classe?
    - En general dans des applis, on se perd rapidement dans les dépendances du fait de leurs tailles
    - Les jar ne définissent pas de dépendance entre eux.
    - Problématique de la transitivité de dépendance des jars.
    - NoClassDefFoundError – au runtime
      - Dépendance de jar non détecté (absent du classpath - Shadowing)
      - Multiples Versions de jar déclarées simultanément dans le classpath → à l'execution la première reference trouvée est exécutée;

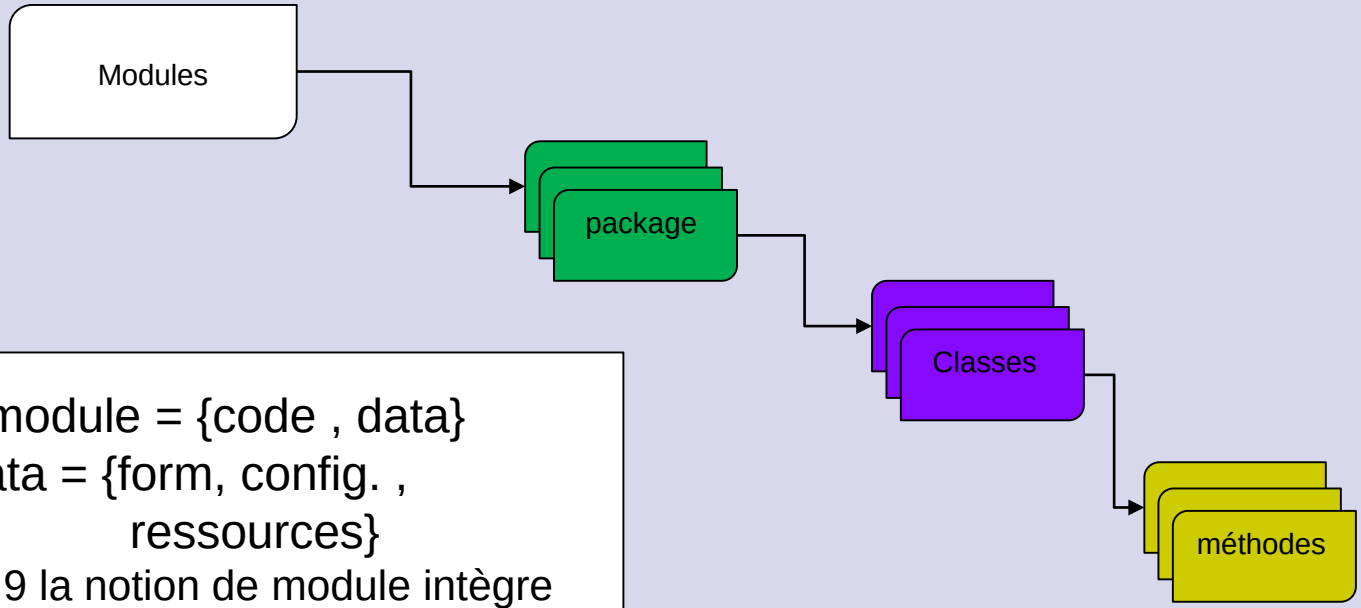
Les nouveautés en Java 9



- Objectifs (2/2)
  - Configuration fiable
  - Une encapsulation plus solide
  - Anticiper l'évolutivité des plateforme java
    - Anticipation sur le future prometteur des IoT
    - Alléger la JDK lorsqu'on est contrain en terme ressources
  - Assurer une intégrité et une Maintenabilité grâce à la modularité

Les nouveautés en Java 9 : Modules

- Modules
  - Définition

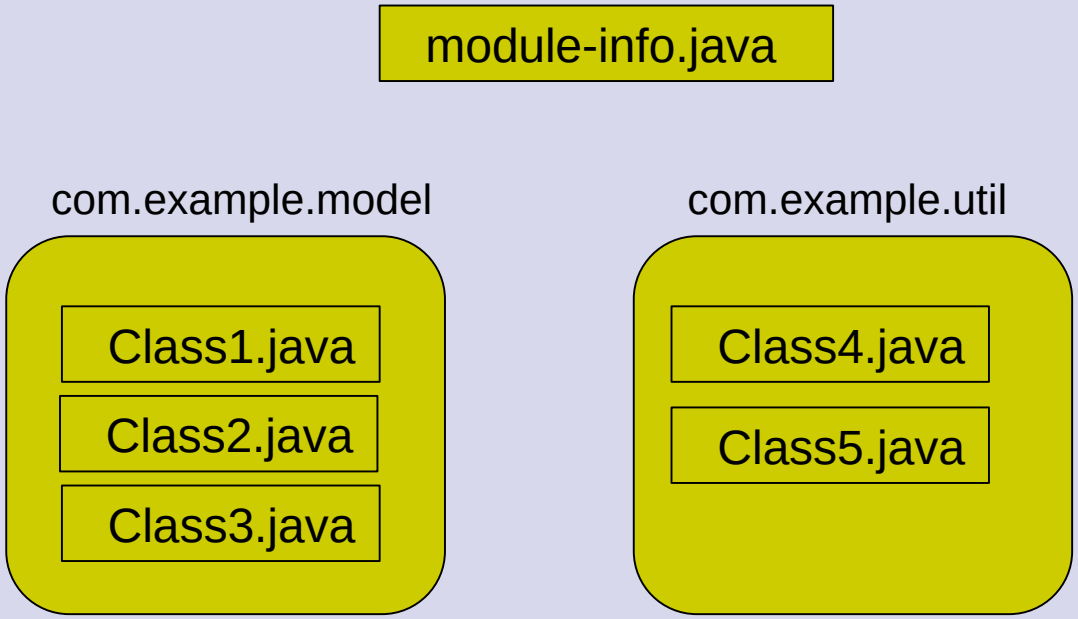


- Un module = {code , data}
  -Data = {form, config. , ressources}

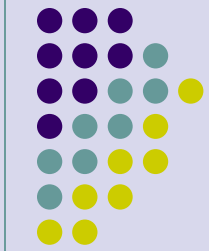
En java 9 la notion de module intègre aussi l'idée qu'un module devrait définit exactement ce dont il a besoin et ce qu'il fourni.

Les nouveautés en Java 9 : Modules

- Modules
  - Déclaration (1/3)



Les nouveautés en Java 9 : Modules



- Modules
  - Déclaration (2/3)

Sans Module

```
src
└── com
    └── example
        ├── model
        │   ├── Class1.java
        │   ├── Class2.java
        │   └── Class3.java
        └── util
            ├── Class4.java
            └── Class5.java
```

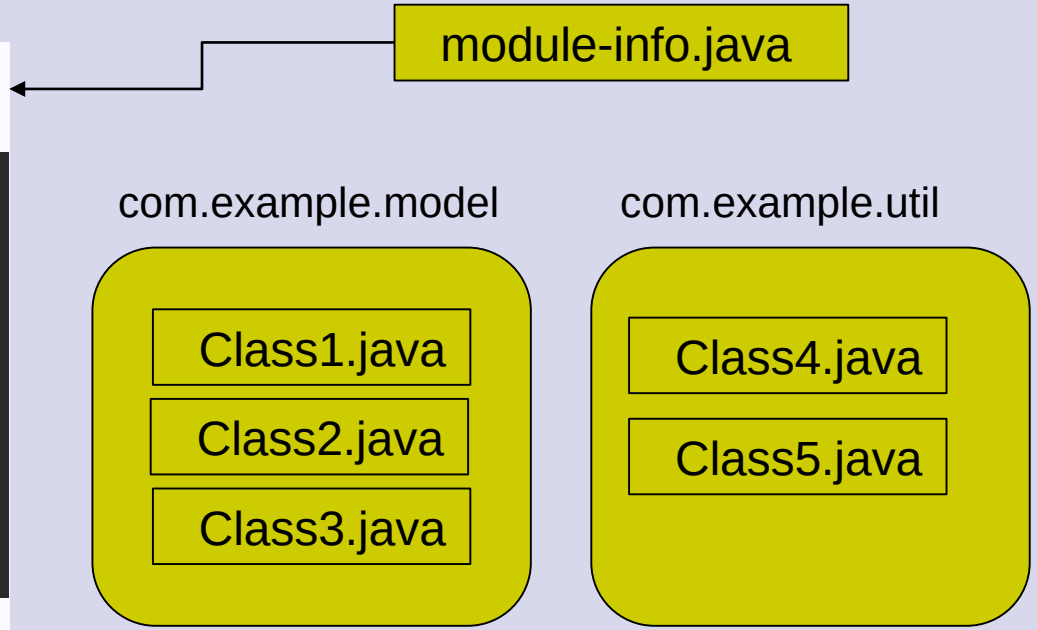
Avec Module

```
src
└── com.example.model
    ├── com
    │   └── example
    │       ├── model
    │       │   ├── Class1.java
    │       │   ├── Class2.java
    │       │   └── Class3.java
    │       └── util
    │           ├── Class4.java
    │           └── Class5.java
    └── module-info.java
```

Les nouveautés en Java 9 : Modules

- Modules
  - Déclaration (3/3)

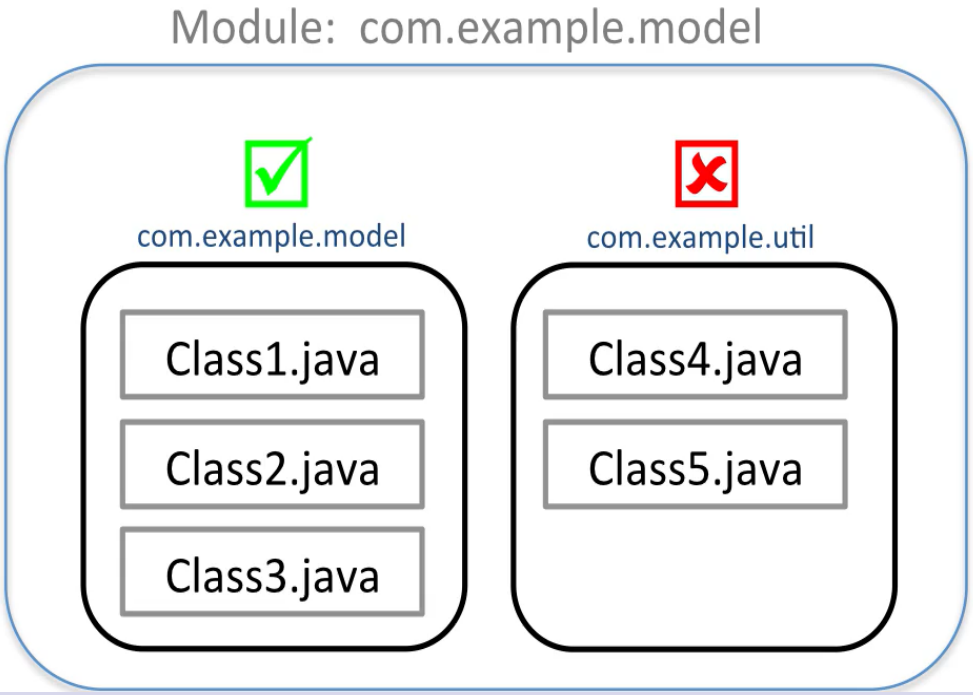
```
Module com.example.model{
/*
* De quels autres modules j'ai
besoin?
* Quels packages sont exposés à
d'autres modules?
* Quels services sont exposés à
d'autres modules?
* De Quels services ai-je besoin?
* Mon package est-il destiné à la
réflexion?
* */
}
```



Les nouveautés en Java 9 : Modules

- Modules
  - Les directives(1/4)

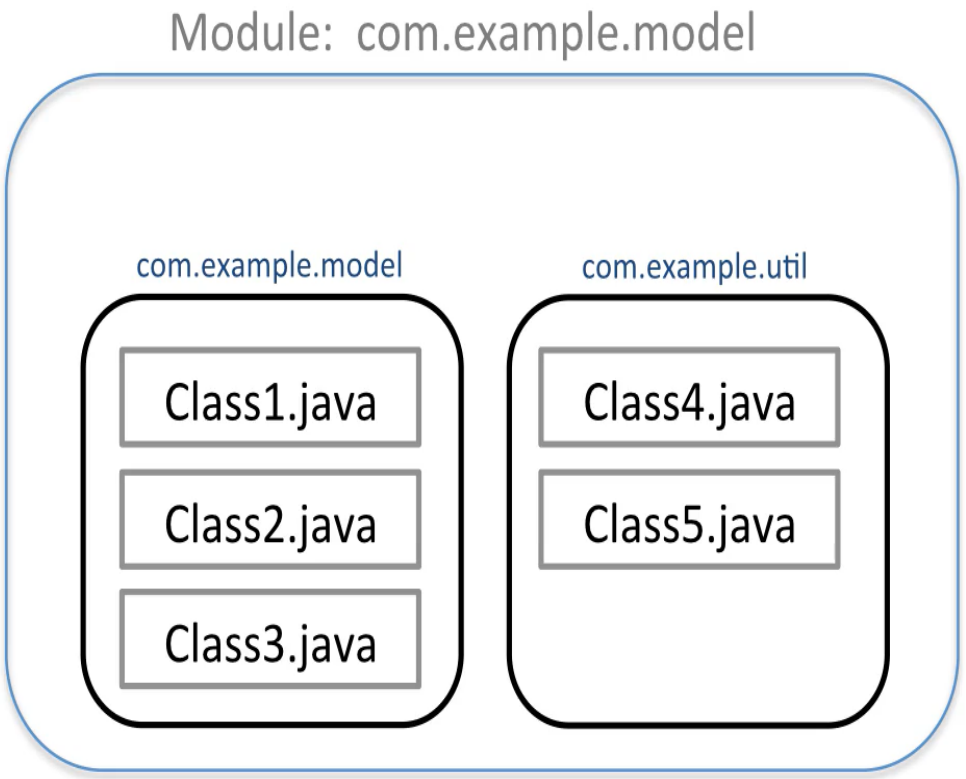
```
module com.example.cli {
  requires com.example.model;

}


module com.example.model {

  exports com.example.model;
}
```



Les nouveautés en Java 9 : Modules

- Modules
  - Les directives(2/4)

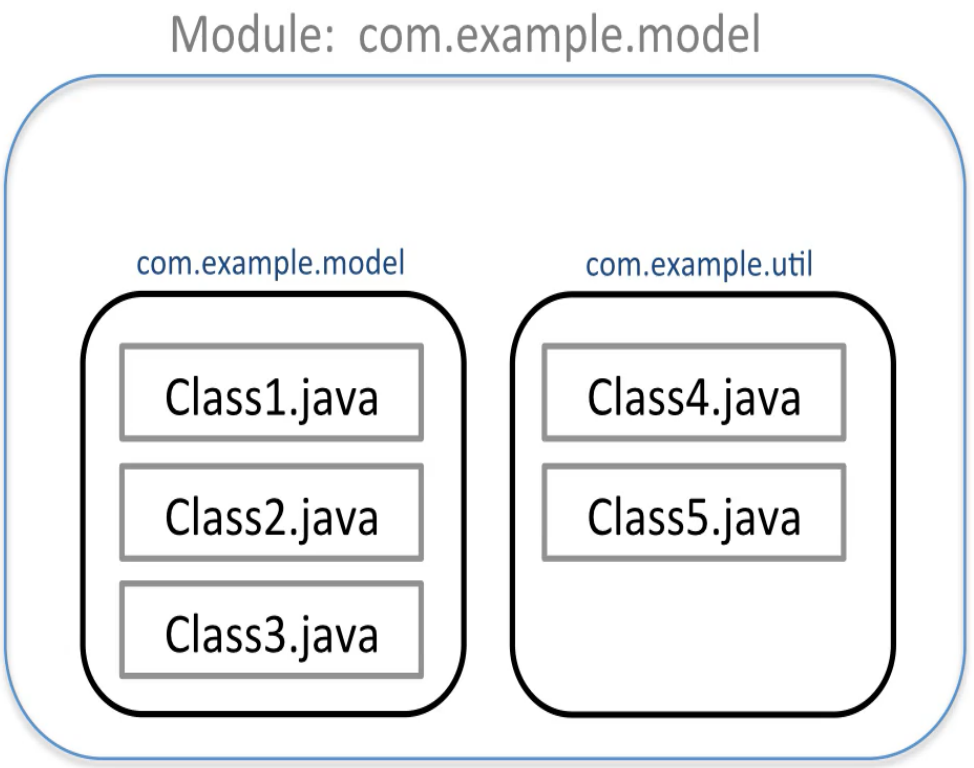
```
module com.example.cli {
  requires com.example.model;

}


module com.example.model {

  exports com.example.model
    to com.example.cli;
}
```



Les nouveautés en Java 9 : Modules

- Modules
  - Les directives(3/4)

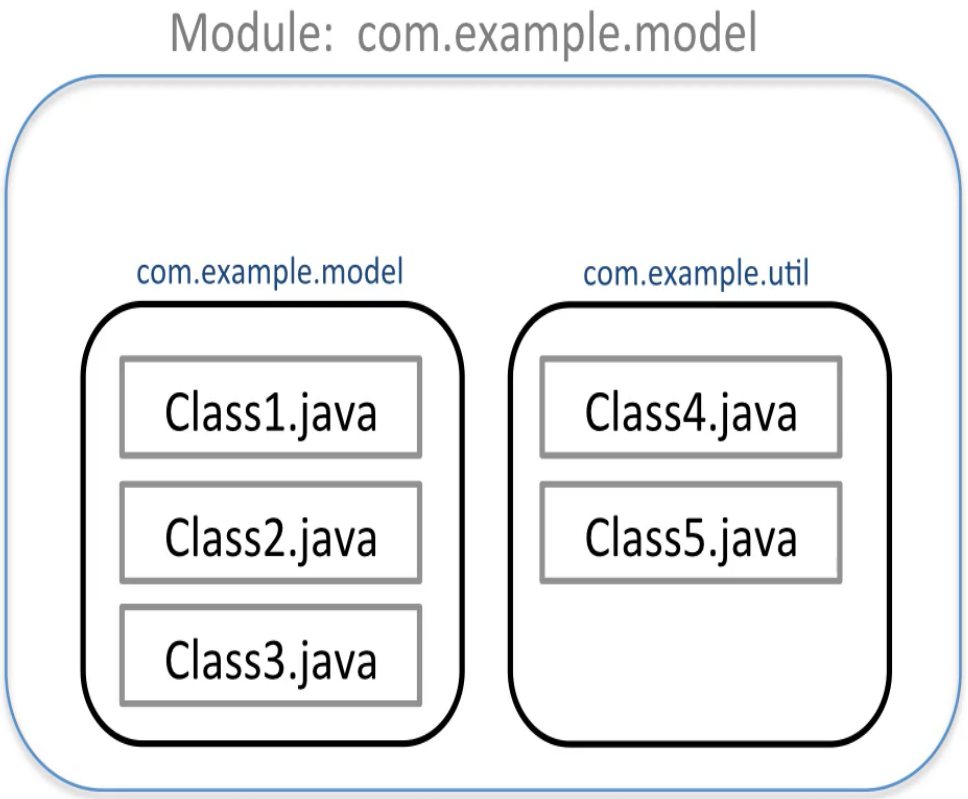
```
module com.example.cli {
  requires transitive
       com.example.model;

}

module com.example.model {
  requires com.example.another;
  exports com.example.model;
}
```



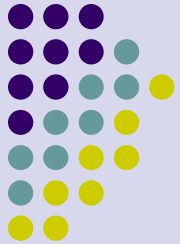
Les nouveautés en Java 9 : Modules

- Modules
  - Les directives(4/4)

```
module com.example.cli {
    requires com.example.model;
    requires java.sql;
    requires java.base;
}

module com.example.model {
    requires java.logging;
    requires java.base;
    exports com.example.model;
}
```



Les nouveautés en Java 9 : Modules

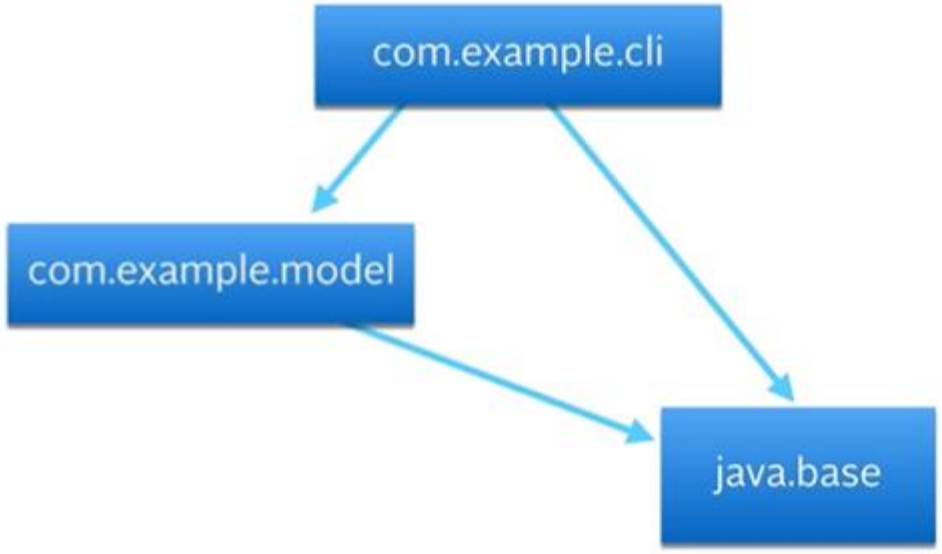


- Modules
  - Le JDK
    - Compètement modularisé
    - Java SE 9 – 94 Modules (java –list-modules
    - JEP 200
    - Java 8
      - http://openjdk.java.net/projects/jigsaw/doc/jdk-modularization.html
    - Java 9
    - Principaux changements
      - Modularité
      - Changement

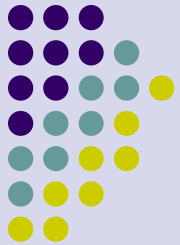
Les nouveautés en Java 9 : Modules

- Modules : Les dépendances

```
module com.example.cli {
    requires com.example.model;
}

module com.example.model {
    exports com.example.model;
}
```

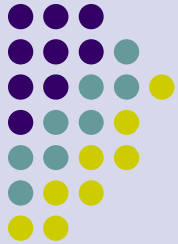


Les nouveautés en Java 9 : Modules(Services)



- Service Provider Interface (SPI)
  - Mécanisme de découverte et de chargement dynamique de classes répondant à une interface donnée.
  - Introduit en java 1.6
  - composé de 3 éléments :
    - Un service, représenté par une classe abstraite.
    - Un fichier de configuration placé dans META-INF/services, déclarant une ou plusieurs implémentations de ce service
    - Le ServiceLoader, responsable de la découverte et du chargement des implémentations du service.
- Les directives uses & provides-with
  - Démo – présentation d'une combinaison de SPI et modularité
  - TP : Service Loarder et java 9

Les nouveautés en Java 9 : Modules(reflexivité)



- La **réflexivité**, aussi appelée introspection, consiste à découvrir de façon dynamique des informations relatives à une classe ou à un objet.
- Les directives :
  - **open** – permet d'exposer tous les packages d'un module pour la réflexivité (tous les eléments *public, protcted, private and default* sont accessibles uniquement au runtime)

```
open module com.m2i.command{

}
```

  - **opens** – permet d'exposer spécifiquement des packages d'un module pour le réflexivité.

```
module com.m2i.command{
    opens com.m2i.command.spi;
}
```

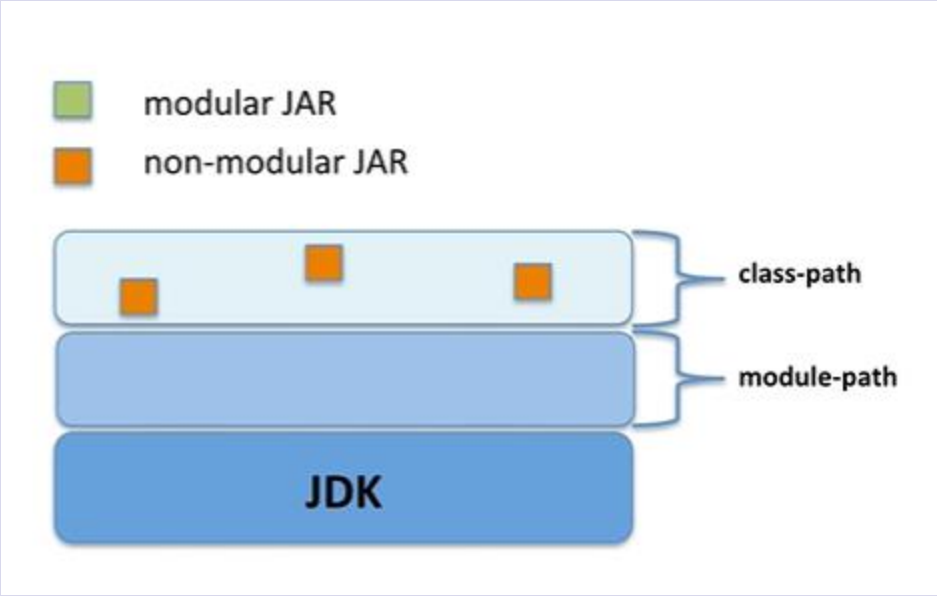
  - **opens-to** permet d'exposer spécifiquement des packages d'un module pour le réflexivité à un autre module.

```
module com.m2i.command{
opens com.m2i.command.spi
to javafx.fxml, javafx.graphics;
}
```

Les nouveautés en Java 9 : Migration

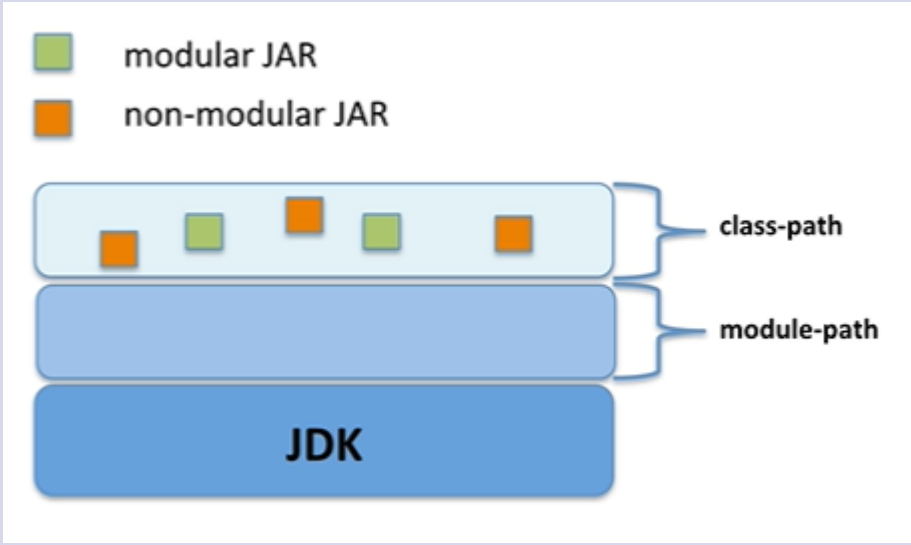
- Problématique générale:
  - Incompatibilité
  - Code non modulaire
  - Outils de dev innadaptés
  - La plupart des API interne du jdk disponibles jusqu'en java 8 ont été encapsulés (JEP260)
  - 6 API supprimés (voir JEP-162)

  - Cas d'une Migration :
    - Notion de modularité inexistante
      → Posiibilé de « jar hell »
    - Pas d'encapsulation et liens entre les jar

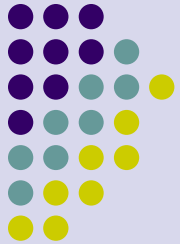


Les nouveautés en Java 9 : Migration

- Tout devant être modularisé en java 9 , on considère tout ce qu'il y a dans le classpath comme étant un module anonyme.
- Ce module anonyme:
  - Accède à tous les autres modules
  - Expose tous ses packages aux autres modules
- Il est aussi possible de convertir des Jars non modulaire en jar modulaire via des lignes de commande:
  - Exemple : my-library-1.24.jar → my.library
  - Les jar ainsi convertis, tout comme le module anonyme, accèdent à tous les modules et exposent tous leurs packages



Les nouveautés en Java 9 : Migration



- Quelques commandes utiles pour la migration

jdeps: Java dependency analysis tool

- Find uses of JDK-internal APIS

```
jdeps --jdk-internals jar/class
```

- Find dependencies

```
jdeps jar/class
```

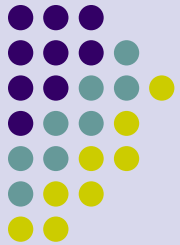
- Generate module-info.java file

```
jdeps --generate-module-info jar
```

Command-line options

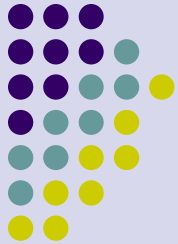
- --add-exports
  use an inaccessible internal API – breaks encapsulation
  --add-exports module/package=target-module
- --add-opens
  opens a module or package for deep reflection
  --add-opens module/package=target-module
- --add-reads
  allows a module to be read without a requires statement
- --permit-illegal-access
  allows illegal access operations by code on classpath to any named module via reflection

Les nouveautés en Java 9 : Migration



- Demo Migration :
  - Part-1: Seul un classpath existe
  - Part-2: Creation de Modules partir de jars existants
  - Part-3: Elimination du classpath
  - Part 4: Modularisation d’un code existant.
- TP reprendre un des projets Java8 et appliquer ce que nous venons de voir

Les nouveautés en Java 9 : Diamond operator



- <> : Il permet au compilateur d'inférer le type de classe générique

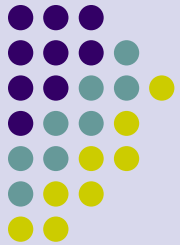
```
List<String> l1 = new ArrayList<String>();
List<String> l2 = new ArrayList<>();
```

Anonymous inner classes

```
MyClass<String> c = new MyClass<String>("James") {
    @Override
    void processData() {
        System.out.println("processing "+ getData());
    }
};
c.processData();
```

```
MyClass<String> c = new MyClass<>("James") {
    @Override
    void processData() {
        System.out.println("processing "+ getData());
    }
};
c.processData();
```

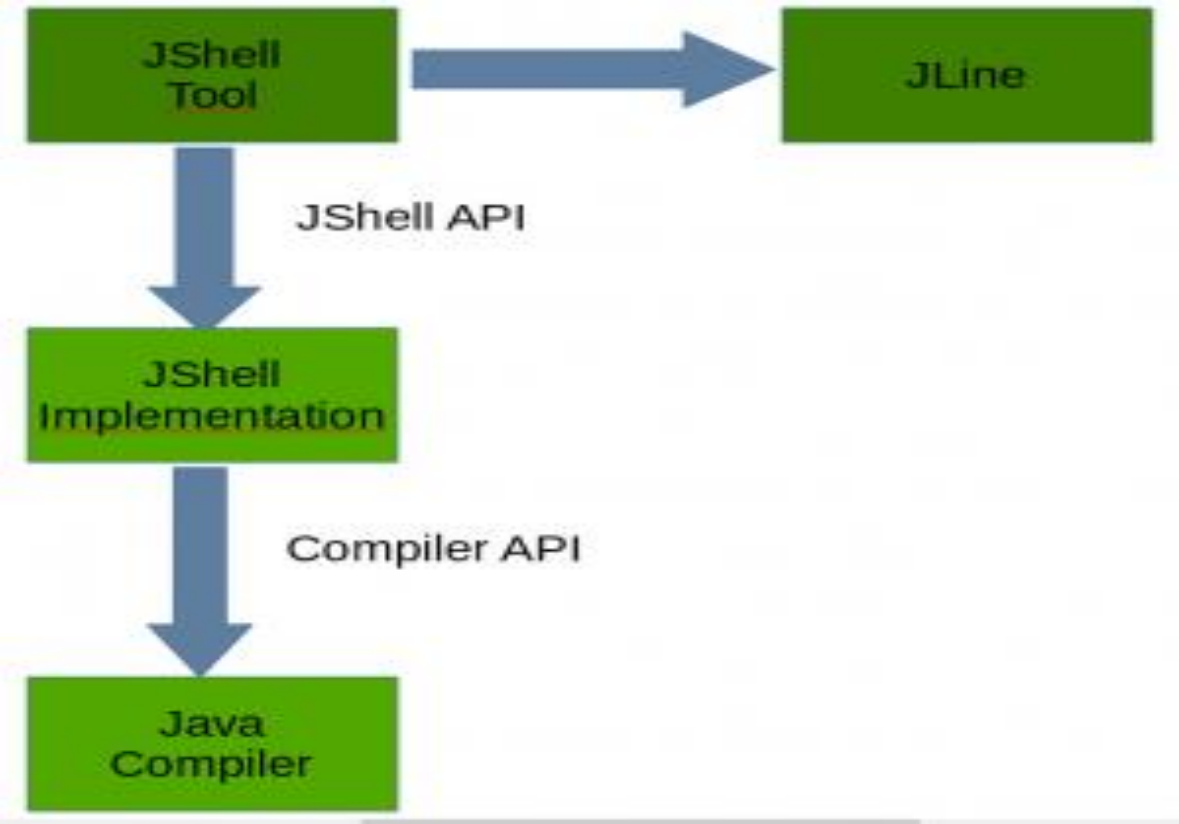
Les nouveautés en Java 9 : Jshell



- Qu’est-ce ?
  - un interpréteur de code en ligne
- Pourquoi l’avoir intégré à Java ?
  - pour s’alligner sur le standard d’autres langage qui fourni un tel outils (python, Lisp, Ruby ect…)
- Comment était testé une simple ligne de code avant java 9?
  - IDE ou ligne de commande

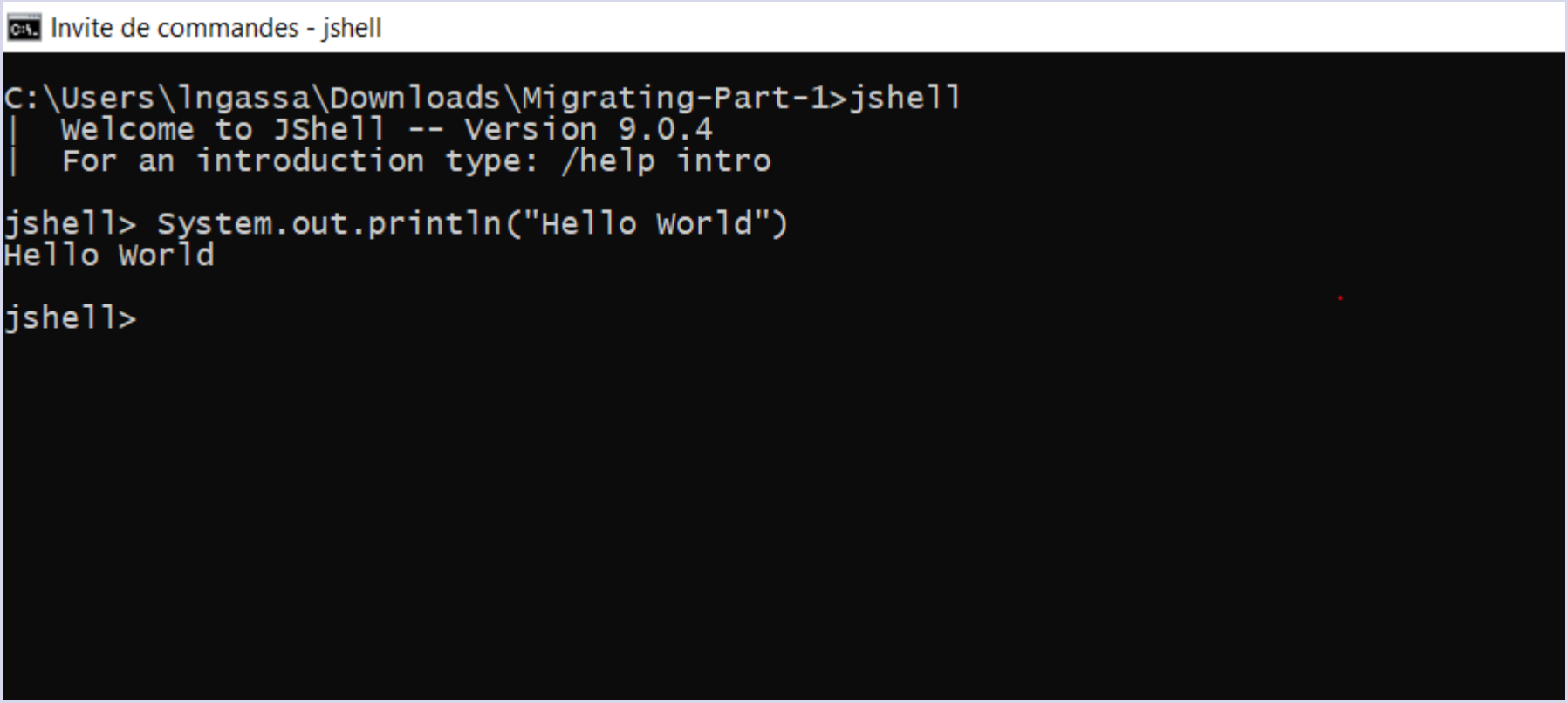
Les nouveautés en Java 9 : Jshell

- Architecture

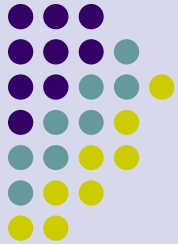


Les nouveautés en Java 9 : Jshell

- Il existe deux types de commandes que l'on peut saisir :
  - Les commandes internes à JShell, celles-ci commencent par un "/"
  - Du code Java



Les nouveautés en Java 9 : Jshell

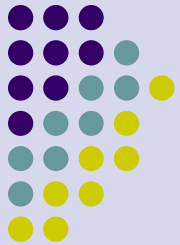


- Certaines librairies sont pré chargées pour savoir lesquels : /list –start

Invite de commandes - jshell

```
C:\Users\lngassa\Downloads\Migrating-Part-1>jshell
|  Welcome to JShell -- Version 9.0.4
|  For an introduction type: /help intro

jshell> System.out.println("Hello World")
Hello World

jshell> /list -start

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;

jshell>
```

Les nouveautés en Java 9 : Jshell



- Quels codes peut-on saisir dans l'invite de commande ? Il s'agit du code suivant :
  - Importation de librairies
    - *Jshell> 3+2*

      $2 ==> 5
  - Déclaration de variables
  - Déclaration de Classes
  - Déclaration d'Interfaces
  - Déclaration de méthodes
  - Déclaration de constantes
  - Déclaration d'expressions

Les nouveautés en Java 9 : Jshell

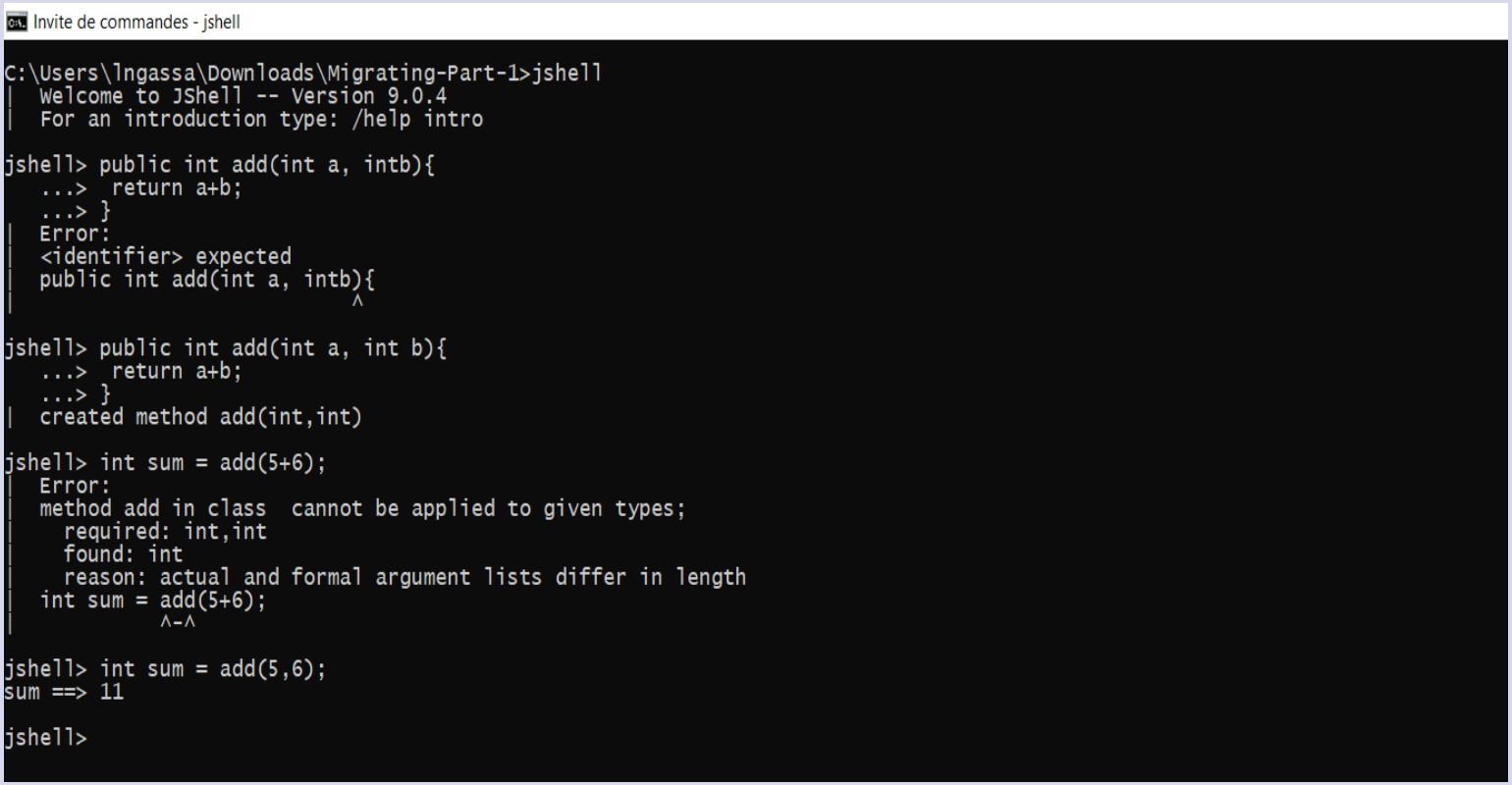
- La déclaration de variables

Invite de commandes - jshell

```
C:\Users\lngassa\Downloads\Migrating-Part-1>jshell
|  Welcome to JShell -- Version 9.0.4
|  For an introduction type: /help intro

jshell> int x
x ==> 0

jshell> String str = "Hello"
str ==> "Hello"

jshell> String str = "wrong declaration"
str ==> "wrong declaration"

jshell> static int j = 3+8
|  Warning:
|  Modifier 'static'  not permitted in top-level declarations, ignored
|  static int j = 3+8;
|  ^----^
j ==> 11

jshell> String str1 = str + "!"
str1 ==> "wrong declaration!"

jshell> final String str = new String();
|  Warning:
|  Modifier 'final'  not permitted in top-level declarations, ignored
|  final String str = new String();
|  ^---^
str ==> ""

jshell> /drop x
|  dropped variable x

jshell> x
|  Error:
|  cannot find symbol
|    symbol:   variable x
|  x
|  ^

jshell>
```

Les nouveautés en Java 9 : Jshell

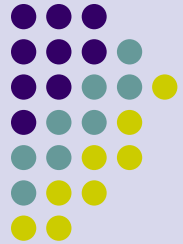
- Déclaration des méthodes : Dans JShell, il est possible de déclarer et d'appeler des méthodes sans avoir à les déclarer dans une classe.



Les nouveautés en Java 9 : Jshell

- Autres Commandes utiles(1/2):
  - /type
  - /import
  - /list
    - /list : affiche l'ensemble du code actif entré dans la session en cours (hors code chargé au démarrage de JShell).
    - /list [ID] : affiche le code source correspondant à l'ID renseigné.
    - /list [Nom_Code] : affiche le code source correspondant au nom renseigné.
    - /list -start : affiche l'ensemble du code chargé au démarrage d'une session de JShell.
    - /list -all : liste l'ensemble du code actif, inactif, en erreur et préchargé, entré au cours d'une session JShell.
    - La capture ci-dessous vous montre l'usage de cette commande :
  - /edit
    - /edit : utilisé sans argument, la commande /edit affiche l'ensemble du code actif dans l'éditeur de texte.
    - /edit [ID] : affiche dans l'éditeur de texte le code correspondant à l'ID saisi.
    - /edit [Nom_Code] : affiche dans l'éditeur de texte le code correspondant au nom renseigné.

Les nouveautés en Java 9 : Jshell



- Autres Commandes utiles(1/2):
  - /save
    - */save [file-path]* : sans argument, cette commande enregistre l'ensemble du code actif saisi au cours de la session, dans le fichier renseigné en second argument. Notez que le code saisi en erreur ainsi que les commandes internes ne sont pas enregistrées dans le fichier.
    - */save -all [file-path]* : enregistre l'ensemble du code actif, en erreur et au démarrage, saisi au cours de la session, dans le fichier renseigné en second argument. Les commandes internes saisies ne sont quant à elles pas enregistrées.
    - */save -history [file-path]* : enregistre l'ensemble du code et des commandes saisis au cours de la session, dans le fichier renseigné en second argument. Même les commandes internes sont enregistrées dans le fichier.
    - */save -start* : enregistre l'ensemble du code lancé au démarrage de JShell.
  - /open
  - /reload
  - /reset