Formation Java

ALI KOUDRI – ALI.KOUDRI@GMAIL.COM

Agenda

- Présentation générale
  - Caractéristiques
  - Historique
- Evolutions de la syntaxe
- Evolution des API
- Tests avec JUnit 5 et Mockito
- Concurrence
- Programmation Réactive
- IoC
- AOP
- JPA
- Pour aller plus loin
- Conclusion

Instructions

- Pour suivre la formation, il est nécessaire de disposer de ces outils:
  - Java 24 SDK
  - IntelliJ (Community)
  - Git
  - Une boîte de doliprane ;-)
- Veuillez clôner les dépôts suivants:
  - git clone https://github.com/akoudri/advanced-java-training.git
  - git clone https://github.com/akoudri/java-fx.git
  - Ils contiennent les exercices de la formation et les ressources qui seront utilisées

Présentation générale

- Langage de programmation logicielle orienté objet
  - Respectueux du paradigme objet
  - Avec depuis la version 8 le support des traits fonctionnels
- Langage compilé et interprété par une machine virtuelle
  - Compilé une fois, exécuté partout
  - Typage fort statique
  - Indépendance vis-à-vis de la plateforme d'exécution
  - Compilation native tout de même supportée (GCJ)

Présentation générale

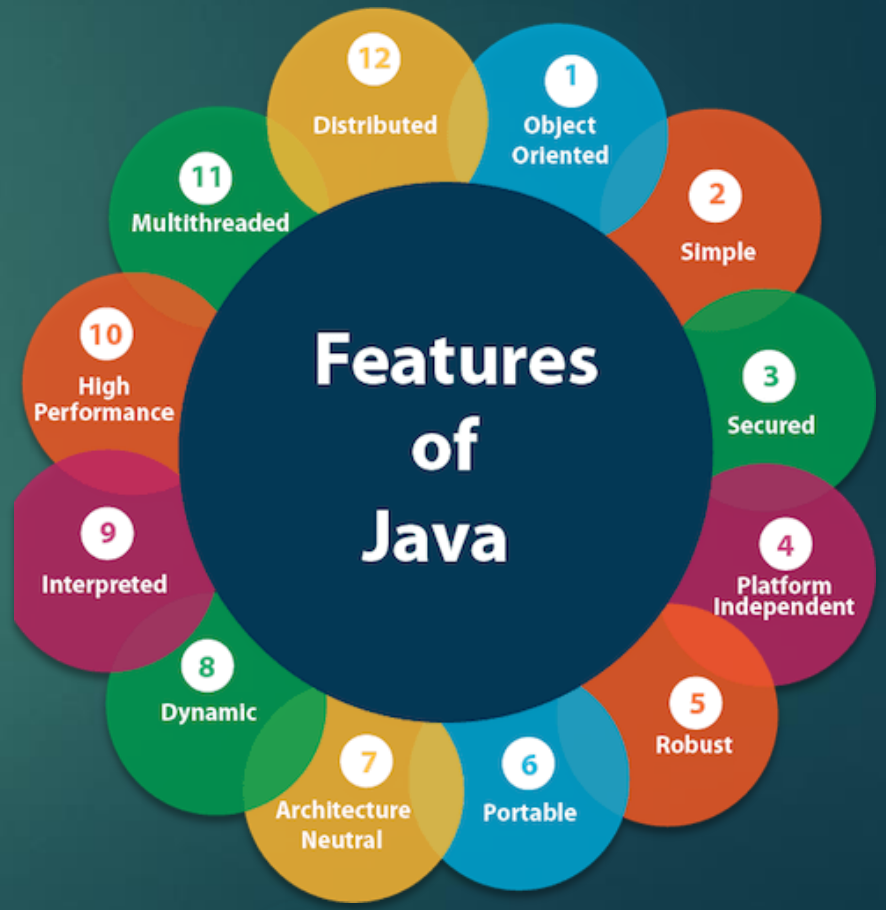
- Langage devenu performant grâce à différentes techniques
  - JIT: interprétation native @ runtime
  - Recompilation dynamique, prenant en compte les caractéristiques de la plateforme d'exécution
- Gestion de la mémoire optimisée
  - Allocation / désallocation de la mémoire automatique (GC)

Présentation générale

- Langage conçu pour les applications lourdes
  - Logiciels standalone, jeux
- Langage conçu pour le réseau et les applications réparties
  - Applets, Servlets, EJB
- Langage conçu pour les applications temps-réel embarquées
  - Gestion de la concurrence et du temps réel
  - Applications mobiles, smart cards, robotiques

Caractéristiques

- **Simple**: Java s'est inspiré de la syntaxe du C++ tout en la simplifiant, ce qui le rend accessible tout en restant puissant.
- **Orienté objet**: Java repose sur les principes de l'encapsulation, de l'héritage, du polymorphisme et de l'instanciation. Depuis Java 8, il intègre aussi des traits fonctionnels.
- **Indépendant de la plateforme / portable**: Grâce à la JVM, le bytecode Java est exécutable sur n'importe quel système d'exploitation : "compile une fois, exécute partout".
- **Sécurisé**: Java ne permet pas la manipulation directe des pointeurs, et les programmes s'exécutent dans une machine virtuelle qui gère les droits d'accès.
- **Robuste**: Le typage fort et la gestion automatique de la mémoire (garbage collector) limitent les erreurs courantes.
- **Performant / Dynamique**: La JVM, le compilateur JIT et le chargement dynamique des classes permettent des performances proches du code natif, avec une gestion fine des threads.
- **Distribué**: Java est conçu pour le développement d'applications distribuées, avec des technologies comme EJB et RMI, facilitant la communication entre composants sur différents systèmes.



Historique

- Langage proposé au début des années 90 à destination des systèmes embarqués par Sun Microsystems
  - Initialement nommé Oak mais changé en Java pour des raisons de droit
  - Java fait référence au café populaire ingurgité par les équipes de développement du langage
  - La signature des programmes Java est "cafe babe"
    - À observer avec *hexdump –C xxx.class | head -n1*
  - Le langage a été popularisé à la fin des années 90 avec les applets

Historique

1.0	Janvier 1996	Première version stable, langage orienté objet, JVM, API de base
1.1	Février 1997	Inner classes, JavaBeans, JDBC, RMI, Réflexion, JIT, Unicode
1.2	Décembre 1998	Collections, Swing, strictfp, JNI, IDL, Applets améliorées
1.3	Mai 2000	HotSpot, Corba, JNDI, JPDA, améliorations JVM
1.4	Février 2002	assert, NIO, expressions régulières, logs, XML, sécurité (JCE, JSSE, JAAS), Web Start
5	Septembre 2004	Génériques, annotations, autoboxing, énumérations, varargs, imports statiques, for-each, Scanner
6	Décembre 2006	Scripting (JSR 223), Console API, Java Compiler API, améliorations JVM
7	Juillet 2011	Switch sur String, try-with-resources, multi-catch, <> (diamond operator), Fork/Join Framework
8 (LTS)	Mars 2014	Lambdas, Stream API, Optional, Date/Time API (java.time), Nashorn, améliorations JVM

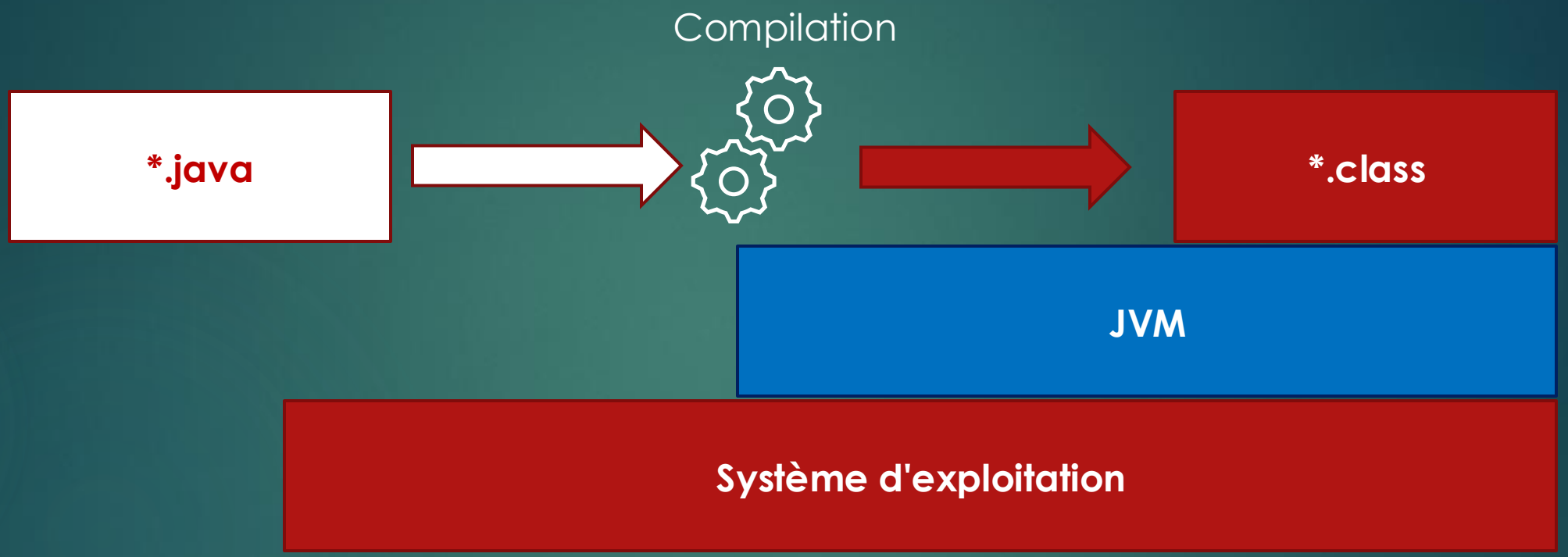
Historique

9	Septembre 2017	Modules (JPMS), JShell, HTTP/2 client, JLink, améliorations JVM
10	Mars 2018	Inférence de type local (var), améliorations JVM, G1 GC par défaut
11 (LTS)	Septembre 2018	var dans lambdas, HTTP client standard, suppression Nashorn, améliorations JVM
12	Mars 2019	Switch expressions (preview), teeing collector, compact numbers, améliorations JVM
13	Septembre 2019	Text blocks (preview), améliorations JVM, améliorations de la gestion mémoire
14	Mars 2020	Records (preview), pattern matching instanceof, améliorations JVM, G1 GC amélioré
15	Septembre 2020	Sealed classes (preview), text blocks finalisés, améliorations JVM
16	Mars 2021	Records, pattern matching instanceof, sealed classes (hors preview), améliorations JVM

Historique

17 (LTS)	Septembre 2021	Sealed classes, pattern matching switch (preview), améliorations JVM, Foreign Function API (incubator)
18	Mars 2022	UTF-8 par défaut, Simple Web Server, code snippets dans la doc, améliorations JVM
19	Septembre 2022	Virtual threads (preview), pattern matching switch, Foreign Function & Memory API (preview), Structured Concurrency (incubator)
20	Mars 2023	Virtual threads (preview), pattern matching switch, Foreign Function & Memory API (preview), vector API (incubator)
21 (LTS)	Septembre 2023	Virtual threads, pattern matching switch, Foreign Function & Memory API, vector API, améliorations JVM, String templates (preview)
22	Mars 2024	String templates (preview), améliorations du pattern matching, améliorations JVM
23	Septembre 2024	Améliorations du pattern matching, Foreign Function & Memory API, améliorations JVM
24	Mars 2025	String templates, améliorations JVM, nouvelles API incubator

Bases de java



- Le fichier source, représenté par "*.java", est le point de départ où le code est écrit.
- La compilation génère un fichier "*.class", qui est le bytecode exécutable par la Java Virtual Machine (JVM).
- La JVM joue un rôle crucial en permettant l'exécution du bytecode sur différents systèmes d'exploitation, ce qui confère à Java sa portabilité.

Syntaxe

Bases de java

- Syntaxe largement influencée par le C++
  - Langage fortement typé
  - Pas de surcharge des opérateurs
  - Héritage simple, réalisation de multiples interfaces
    - Permet de lever les ambiguïtés (problème du diamant)
- Types primitifs:
  - Entiers: **byte**, **short**, **int**, **long**
    - 1, 3L, 3_000, 0b1001, 0xCAFE_BABE (_ à partir de la version 7 pour la lisibilité)
  - Réels: **float**, **double**
    - 3.14_15F
  - Booléens: **boolean**
  - Caractères: **char**
  - Vide: **void**

Opérateurs

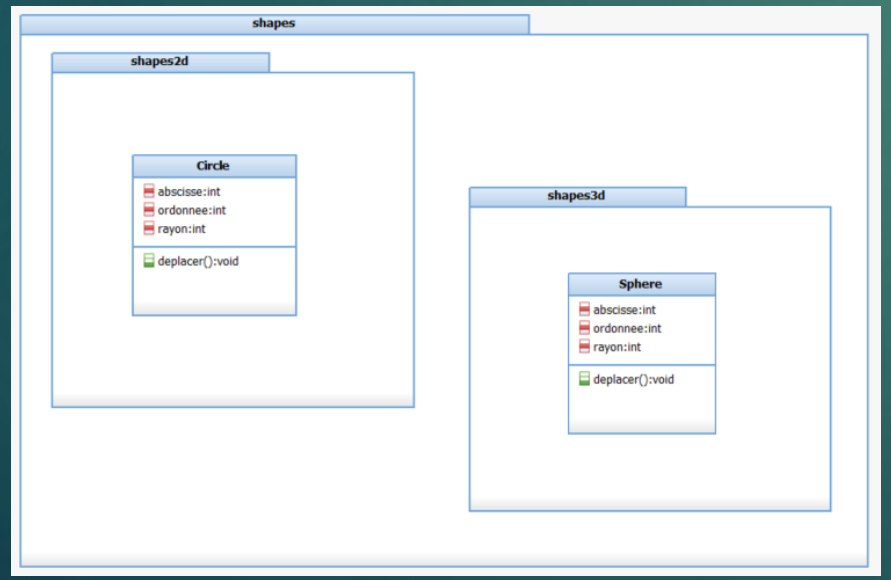
- Opérateur d'affectation: **= += -= *= /= %= &= ^= |= <<= >>= >>>=**
- Opérateurs arithmétiques: **+ - * / %**
- Opérateurs unaires: **++var, var++, --var, var--, +var, -var, !var**
- Opérateurs de comparaison: **< > <= >= == !=**
- Opérateurs binaires: **& ^ |**
- Opérateurs logiques: **&& || instanceof**
- Opérateur ternaire: (cond)**?**action_vrai**:**action_faux;

Contrôle

- Structures de contrôles
  - **if** (cond) { instructions } [**else** if (cond) { instructions } else { instructions }]
  - **for** (initialisation; cond d'arrêt; incrément) { instructions }
  - **while** (cond) { instructions }
  - **do** { cond } while (cond)
  - **switch**(var) { **case** val : instructions; **default**: instructions; }
    - Variable limitée aux entiers jusqu'à la version 15
  - **assert**: à partir de la version 1.4, permet de poser des conditions dans l'exécution d'opérations
- Contrôle du flôt d'exécution dans les boucles:
  - **continue**: passe à l'itération suivante
  - **break**: arrête la boucle

Structuration

- Les classes permettent de regrouper les caractéristiques communes à un ensemble d'objets
- Les classes encapsulent des données et des opérations
- Les classes sont organisées au sein de paquetages hiérarchisés




```
package shapes.shapes1d;

class Point {

    int abscisse = 0;
    int ordonnee = 0;

    void deplacer() {
        abscisse += 1;
        ordonnee += 1;
    }
}
```

propriétés

opérations

Visibilité

- Visibilité des classes
  - Les classes peuvent être privées - visibles uniquement au sein des paquetages dans lesquels elles sont définies
    - Aucun mot-clef n'est requis dans ce cas
  - Elles peuvent visibles en dehors des paquetages dans lesquels elles ont été définies
    - Ex: **public** class Cercle {}
  - Des classes peuvent être définies au sein d'une classe (à partir de la version 1.1)
    - Utile si elles ne sont instanciées dans le contexte d'un objet particulier
    - Dans ce cas, ces classes peuvent être cachées des autres classes
    - Ex: **private** class Point {}

Visibilité

- Les propriétés et les opérations possèdent 4 niveaux de visibilité
  - **Paquetage**: accessibles à toutes les autres classes définies au sein du même paquetage – aucun mot-clef requis
  - **Publique**: accessibles à toutes les classes
    - Ex: **public** int abscisse
  - **Privé**: accessible uniquement au sein de la classe
    - Ex: **private** int abscisse
  - **Protégé**: accessible au sein de la classe et aux classes descendantes dans le cas d'héritage
    - Ex: **protected** int abscisse

Accessibilité

- Les attributs, les opérations et les classes peuvent être marqués **final**
  - Dans le cas d'un attribut, indique que la valeur est constante
  - Dans le cas d'une opération, indique que la méthode ne peut être surchargée
  - Dans le cas d'une classe, indique que le classe ne peut être héritée

- Attributs / opérations d'objets vs Attributs / opérations de classes
  - Lorsqu'on déclare un attribut **static**, il n'a qu'une seule copie appartenant à la classe
  - Lorsqu'on déclare une opération **static**, elle ne peut être appelée que sur la classe
  - Il est possible d'importer les opérations statiques et les utiliser sans passer par la classe (à partir de la v5). Ex: **import static** java.lang.Math.*;

Accessibilité

- Les blocks statiques permettent d'initialiser des variables statiques
- Dans le cas de classes imbriquées, l'utilisation du mot-clef **static** permet de limiter l'accès aux champs de la classe parent qui sont statiques
  - Cf. Diapo suivante
- Les classes peuvent accéder aux définitions des autres classes / interfaces / énumération par des imports
  - Sous condition d'accessibilité
  - Ex: **import** java.utils.List;
  - Ex: **import** java.utils.*; //on importe tout

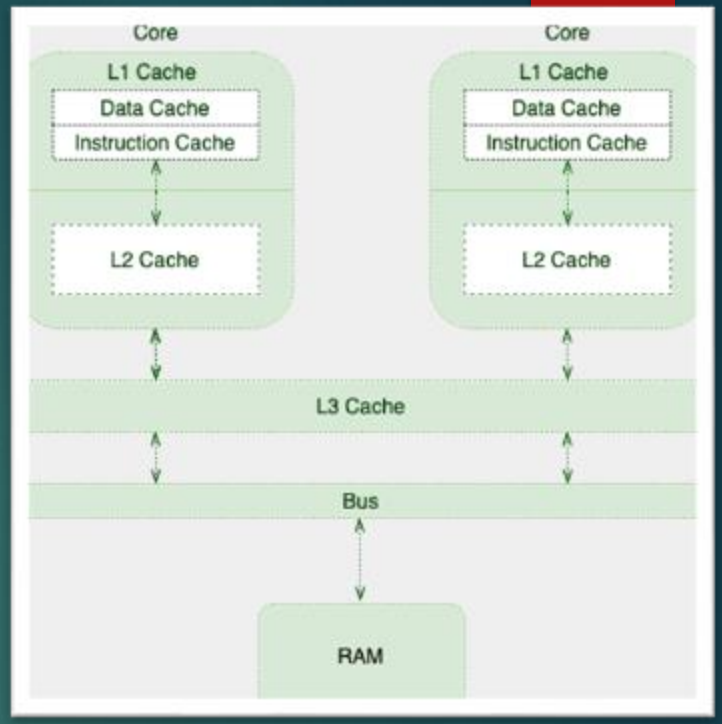
```
public static List<String> ranks = new LinkedList<>();
static {
    ranks.add("Lieutenant");
    ranks.add("Captain");
    ranks.add("Major");
}
```

Static nested classes vs inner classes

- Association avec une instance de classe englobante:
  - Static Nested Class : N'a pas d'association directe avec une instance de la classe englobante. Elle peut être créée sans instance de la classe englobante.
  - Inner Class : Est associée à une instance de la classe englobante. Elle nécessite une instance de la classe englobante pour être créée et peut accéder à ses membres (y compris les membres privés).
- Accessibilité :
  - Static Nested Class : Ne peut accéder qu'aux membres statiques de la classe englobante.
  - Inner Class : Peut accéder aux membres statiques et non statiques de la classe englobante.
- Utilisation :
  - Static Nested Class : Souvent utilisée pour regrouper logiquement des classes qui ne sont utilisées qu'à un seul endroit, afin d'améliorer l'encapsulation.
  - Inner Class : Utile pour gérer les événements ou construire des classes adaptatrices, car elles peuvent accéder directement aux membres de la classe englobante.

Autres modificateurs

- Un attribut peut être déclaré **transient** dans le cas d'une sérialisation
  - Cela empêche l'attribut d'être sérialisé
  - Ex: **int** age; **transient boolean** isAdult; //calculé à partir de age
- Dans le cas de programmation parallèle, la déclaration d'un attribut volatil permet d'assurer un ordre dans l'écriture et la lecture d'une variable
  - Cela évite qu'une variable soit mise en cache
  - Cela garantit qu'un thread n'accède pas à une valeur périmée
    - Mais cela n'est en aucun cas un mécanisme d'exclusion mutuelle
  - Ex: **static volatile int** num;




```
public class TaskRunner {
    private static volatil int number;
    private static  volatil boolean ready;
    private static class Reader extends Thread {
      @Override
       public void run() {
           while (!ready) {
               Thread.yield();
           }
           System.out.println(number);
       }
    }
     public static void main(String[] args) {
         new Reader().start();
         number = 42;
         ready = true;
     }
}
```

Autres modificateurs

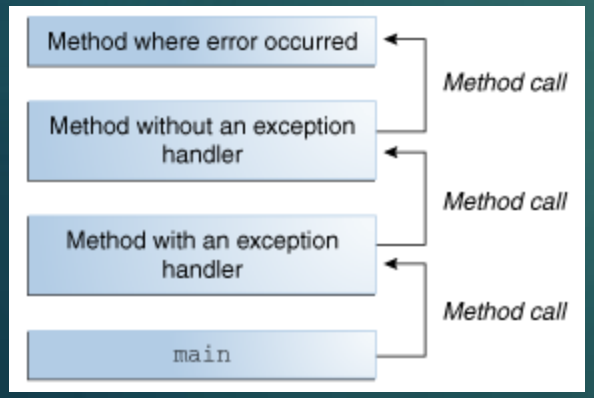
- Dans le cas d'une programmation parallèle, le mot-clef **synchronized** permet de poser un verrou en exclusion mutuelle
  - Sur une opération ou une portion de méthode
  - Les threads se synchronisent en posant un verrou sur un objet
    - L'instance courante (this) par défaut sur les opérations
      - Ex: **synchronized void** increment() { … }
    - N'importe quel instance explicite dans le cas de portions
      - Ex: **synchronized**(this) { … }
    - L'objet sert de clef garantissant un accès exclusif au code ou à la portion de code à protéger

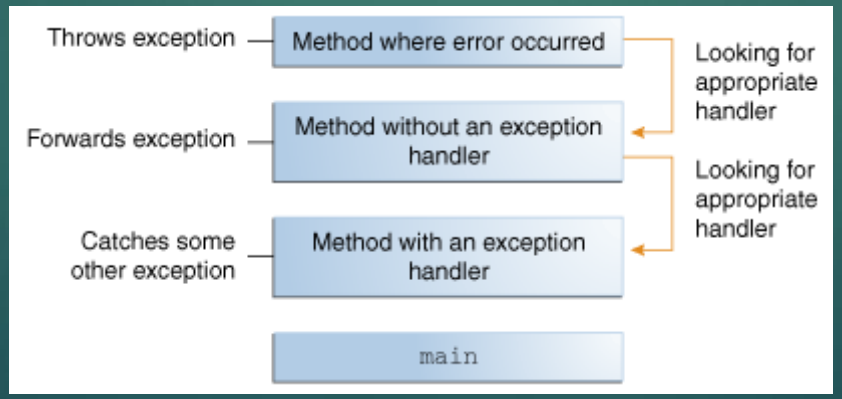
Opérations, Exceptions

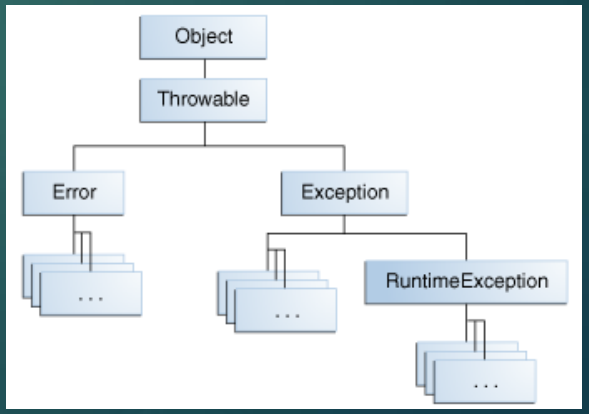
- Opérations typées
  - Paramètres, valeur de retour
    - Ex: **void** deplacer(**int** x, **int** y) { //méthode }
    - Ex: **int** carre(**int** x) { **return** x * x; }
- Java comprend des mécanismes de gestion des exceptions
  - Gestion de l'exception au sein d'une opération:
    - **try** { instructions; } **catch**(Exception ex) { instructions; } **finally** { instructions; }
    - Le bloc finally représente le code à exécuter quoi qu'il arrive
    - Les API standards de java définissent un grand nombre d'exceptions
    - On peut définir ses propres exceptions en étendant la classe Exception ou l'une de ses descendantes

Exceptions

- Evénement qui modifie l'exécution nominale d'un programme
- 3 types d'exception:
  - Les exceptions anticipées (prévisibles)
  - Les exceptions d'exécution (runtime exception, non prévisibles)
  - Les exceptions liées à l'environnement d'exécution (erreurs)







Exceptions anticipées

- **Définition** : Ce sont des situations exceptionnelles que le programmeur doit anticiper et gérer explicitement (par un bloc try/catch ou une clause throws).
- **Hiérarchie** : Hérite de java.lang.Exception (hors RuntimeException).
- **Obligation** : Le compilateur force le développeur à gérer ou déclarer ces exceptions.
- **Exemples courants** :
  - IOException (erreur d'entrée/sortie, fichier introuvable)
  - SQLException (erreur lors d'un accès à une base de données)
  - ClassNotFoundException (classe non trouvée lors du chargement dynamique)
- **But** : Permettre à l'appelant de réagir ou de récupérer la situation

Exceptions non vérifiées (Runtime)

- **Définition** : Ce sont des erreurs de programmation ou des situations inattendues à l'exécution, souvent dues à des bugs (mauvaise utilisation d'une API, logique erronée…).
- **Hiérarchie** : Hérite de java.lang.RuntimeException, elle-même sous-classe de Exception.
- **Obligation** : Le compilateur n'oblige pas à gérer ou déclarer ces exceptions ; elles peuvent survenir à tout moment et ne sont généralement pas prévues.
- **Exemples courants** :
  - NullPointerException (accès à une référence nulle)
  - ArrayIndexOutOfBoundsException (dépassement d'indice de tableau)
  - IllegalArgumentException (argument de méthode invalide)
  - ArithmeticException (division par zéro)
  - ClassCastException (mauvais cast)
- **But** : Signaler des erreurs de programmation que le développeur doit corriger, pas l'utilisateur final

Erreurs

- **Définition** : Ce sont des problèmes graves liés à l'environnement d'exécution ou à la JVM elle-même, sur lesquels le programme n'a généralement aucun contrôle.
- **Hiérarchie** : Hérite de java.lang.Error, elle-même sous-classe de Throwable.
- **Obligation** : Le compilateur n'oblige pas à les gérer, et il est déconseillé de les intercepter (catch).
- **Exemples courants** :
  - OutOfMemoryError (plus de mémoire disponible)
  - StackOverflowError (dépassement de pile, récursion infinie)
  - NoClassDefFoundError (classe manquante à l'exécution)
- **But** : Indiquer un état critique du système ou de la JVM, souvent irrécupérable

Exceptions personnalisées - Règles

- **Nom explicite**: Donner un nom précis à ton exception, décrivant clairement l'erreur (ex : InvalidAgeException, InsufficientFundsException).
- **Hériter de RuntimeException sauf besoin particulier**: Java ne distingue pas les exceptions vérifiées/non vérifiées : il est généralement recommandé d'hériter de RuntimeException pour les erreurs de logique ou de validation, sauf à forcer la gestion explicite de l'exception.
- **Inclure des informations utiles**: Ajouter des propriétés ou des messages détaillés pour faciliter le débogage (par exemple, la valeur fautive, le contexte, etc.).
- **Créer une hiérarchie si besoin**: Si le domaine applicatif comporte plusieurs types d'erreurs, crée une hiérarchie d'exceptions (classe de base abstraite ou sealed class, puis sous-classes spécifiques).
- **N'utiliser les exceptions personnalisées que si nécessaire**: Privilégier les exceptions standards si elles sont adaptées. Créer une exception personnalisée uniquement si cela apporte de la clarté ou une gestion plus fine des erreurs.
- **Documenter les exceptions** : dans quel cas sont-elles lancées, que signifient-elles, comment les gérer.

Erreurs – Précisions

- La classe Error représente des problèmes graves liés à l'environnement d'exécution ou à la JVM elle-même (par exemple, OutOfMemoryError, StackOverflowError). Ces erreurs signalent des situations anormales dont un programme ne peut généralement pas se remettre et qu'il ne devrait pas essayer d'attraper ou de gérer.
- Les exceptions personnalisées doivent hériter de Exception ou de ses sous-classes (RuntimeException), car elles sont faites pour signaler des erreurs applicatives ou métier, que le code peut raisonnablement gérer ou anticiper.
- Créer une classe héritant de Error reviendrait à signaler un problème fondamental du système, ce qui n'est pas le rôle d'une exception métier ou applicative. Par convention et par sécurité, il ne faut pas créer de nouvelles sous-classes de Error dans ses propres programmes.

Exceptions - Pratique

- Capture
- Propagation
  - Chaînage
- Génération
- Spécification
  - Spécialisation des interfaces standards
- A partir de la version 8, la construction "**try-with-resources**" permet de fermer automatiquement les ressources ouvertes

```
try { … }
catch (IOException | SQLException ex) { … }
finally { … }
```

```
try (BufferedReader r = new BufferedReader (new FileReader(path))) { … }
catch (IOException | SQLException ex) { … }
// Resource automatiquement fermée
```

```
void myOperation() throws IOException, SQLException { … }
```

```
void myOperation()  { … throw new IOException(); … }
```

Instanciation

- Utiliser le mot-clef new suivi d'un constructeur pour créer une instance de classe
  - Ex: Voiture v = **new** Voiture();
- Le constructeur est une opération spéciale, portant le même nom que la classe et ne renvoyant rien (même pas void)
- Toutes les classes ont un constructeur par défaut, tant qu'au moins un constructeur n'a pas explicitement été défini
- Le constructeur spécifie comment initialiser l'état d'un objet quand il est créé
  - Ex: Voiture v = **new** Voiture(age, kilometrage);

Classes abstraites

- Certaines classes ne servent qu'à regrouper des attributs / opérations communs et ne sont pas destinées à être instanciées
  - On les marque alors comme **abstract**
  - Ex: **abstract class** Vehicle { … }

- Les classes abstraites sont généralement partiellement définies
  - Certaines opérations n'ont pas de méthodes et sont également marquées **abstract**

```
public abstract class Shape {

    private int abscisse;
    private int ordonnee;

    public abstract int getPerimeter();

}
```

Constructeur

- Il est possible d'avoir plusieurs constructeurs avec différents paramètres pour initialiser l'état de l'objet
  - Dans le cas où les constructeurs ont tous des paramètres, le constructeur par défaut disparait
  - Pour éviter certaines ambiguïtés, on utilise le mot clef **this** pour faire référence à l'objet créé
    - Le paramètres des opérations sont prioritaires dans les méthodes
  - On peut également faire référence à un autre constructeur en utilisant la construction **this**(parameters...)

```
class Point {
    private int abscisse = 0;
    private int ordonnee = 0;

    public Point (int xy) {
        this(xy, xy);
    }

    public Point(int abscisse, int ordonnee) {
        this.abscisse = abscisse;
        this.ordonnee = ordonnee;
    }
}
```

Constructeur

- Dans certains cas, on peut vouloir interdire l'utilisation de constructeurs pour une classe
  - Technique très utilisée par les patrons de conception (singleton, fabrique, etc.)
  - Il suffit de rendre le constructeur par défaut privé
  - Et d'ajouter des méthodes statiques permettant une création contrôlée des objets

```
class Point {
    private int abscisse = 0;
    private int ordonnee = 0;

    private Point () {
    }

    public static Point createPoint(
                        int abscisse,
                        int ordonnee) {
        Point p = new Point():
        p.abscisse = abscisse;
        p.ordonnee = ordonnee;
        return p;
    }
}
```

Héritage

- Une classe ne peut hériter que d'une seule et unique classe parente
  - Par défaut, toutes les classes héritent de la classe Object
  - L'héritage est spécifié par le mot-clef **extends**
    - Ex: **class** Voiture **extends** Vehicle { … }
  - La classe qui hérite peut faire référence aux attributs, opérations, constructeurs de la classe parent par le mot clef **super**
  - Dans le cas d'une classe concrète qui hérite d'une classe abstraite, il faut fournir une implémentation de toutes les opérations abstraites – Utilisation de l'annotation **@Override**
  - On peut faire référence aux attributs / opérations de la classe parente par le mot clef **super**

```
public abstract class Shape {
    protected int abscisse;
    protected int ordonnee;

    public Shape(int abscisse, int ordonnee) {
        this.abscisse = abscisse;
        this.ordonnee = ordonnee;
    }

    public abstract int getPerimeter();
}

public class Square extends Shape {
    private int cote;

    public Square() {
        super(0, 0);
    }

    public Square(int abscisse, int ordonnee) {
        super(abscisse, ordonnee);
    }

    @Override
    public int getPerimeter() {
        return 4 * cote;
    }
}
```

Polymorphisme

- Le polymorphisme permet à un objet de prendre plusieurs formes.
- En Java, cela signifie qu'un objet peut être manipulé via une interface ou une classe parente, tout en utilisant son implémentation réelle.
- Types de polymorphisme en Java :
  - **Polymorphisme ad-hoc**: Surcharge (overload) : plusieurs méthodes avec le même nom mais des signatures différentes; Redéfinition (override) : une classe fille redéfinit une méthode de la classe parente.
  - **Polymorphisme paramétré**: Utilisation des génériques pour écrire du code adaptable à différents types.
  - **Polymorphisme par héritage**: Manipulation d'objets via leur classe parente ou une interface, permettant à chaque objet d'utiliser sa propre implémentation.

Interfaces

- Une interface définit un contrat que des classes implanteront
- Une interface représente un comportement visible de l'extérieur
- Une interface spécifie un ensemble d'opérations implicitement **abstract**
- Les attributs sont implicitement **static** et **final**
- Les attributs et les opérations sont implicitement **public**
- Les interfaces peuvent contenir des opérations statiques avec une méthode

```
interface Bicycle {

    int nbWheels = 2;

    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

Interfaces

- Une classe peut réaliser plusieurs interfaces
  - En utilisant le mot-clef **implements**
- Une classe qui réalise une interface doit implanter toutes les opérations définies par cette dernière
- Quand une interface évolue, par l'ajout des méthodes notamment, les classes qui l'implante doivent toutes êtres resynchronisées:
  - Implanter les méthodes ajoutées
  - Être déclarée abstract
- On peut éviter la situation où les classes qui implantent l'interface se retrouve dans un état incohérent
  - À partir de la version 8, on peut donner une implantation par défaut à la méthode ajoutée en la marquant **default**

```
interface Bicycle {

    int nbWheels = 2;

    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    default void applyBrakes(int decrement) { … }
}

class VTT implements Bicycle {
    //attributes
    //constructors
    //operations
    public void changeCadence(int newValue){ … }

    public void changeGear(int newValue){ … }

    public void speedUp(int increment){ … }

    public void applyBrakes(int decrement){ … }
}
```

Enumérations

- A partir de la version 5
- Correspond à une classe fermée
  - Dont toutes les instances sont connues et non-modifiables
  - Ex: Les jours de la semaine
- Les membres d'une énumération (litéraux) sont ordonnés et possèdent un nom
  - Ils sont accessibles par leur index ou leur nom
  - Ex: Jour.Lundi
- Comme toute classe, une énumération peut posséder des constructeurs et des méthodes
  - Le constructeur est privé

```
public enum Coefficient {
  UN(1), DEUX(2), QUATRE(4); //Litéraux

  private final int valeur;

  private Coefficient(int valeur) {
    this.valeur = valeur;
  }

  public int getValeur() {
    return this.valeur;
  }
}
```

API

Présentation

- Depuis Java 9, les API Java sont organisées en modules (JPMS, Java Platform Module System), permettant une meilleure encapsulation, une gestion fine des dépendances, et une réduction de la taille des applications.
  - Cette organisation reflète la logique métier ou technique des différents domaines couverts par Java.

- Les diapositives suivantes présentent les principaux modules du JDK, avec leur descriptif, l'organisation en packages, et les classes principales.

- https://docs.oracle.com/en/java/javase/24/docs/api/index.html

Modules essentiels

Module	Description	Packages principaux	Classes/Interfaces principales
java.base	Fondamentaux du langage et de la plateforme	java.lang, java.util, java.io, ...	Object, String, List, InputStream, ...
java.sql	Accès aux bases de données (JDBC)	java.sql, javax.sql	Connection, Statement, ResultSet, DataSource
java.xml	Manipulation de documents XML	javax.xml.parsers, org.w3c.dom, ...	SAXParser, Document, Transformer
java.desktop	Interfaces graphiques (AWT, Swing, JavaFX)	java.awt, javax.swing, javafx.*	Frame, JFrame, Stage, Button
java.net.http	Requêtes HTTP modernes	java.net.http	HttpClient, HttpRequest, HttpResponse

Modules essentiels

Module	Description	Packages principaux	Classes/Interfaces principales
java.management	Surveillance de la JVM (JMX)	javax.management	MBeanServer, ObjectName
java.logging	Journalisation (logs)	java.util.logging	Logger, Handler, Level
java.naming	Services de nommage (JNDI)	javax.naming	Context, InitialContext
java.rmi	Invocation de méthodes à distance (RMI)	java.rmi	Remote, Naming
java.compiler	Compilation dynamique	javax.tools	JavaCompiler, StandardJavaFileManager

Version 1.1 - JavaBean

- Composant réutilisable et manipulable visuellement par un outil de conception
- Techniquement, il s'agit d'une classe sérialisable avec juste un constructeur par défaut et des accesseurs
  - Les attributs sont privés
  - La classe ne peut être déclarée "final"
- Les librairies graphiques utilisent ces conventions pour leurs composants
- A ne pas confondre avec les EJB (Enterprise JavaBeans) qui sont des composants exécutés côté serveur

Version 1.1- JDBC

- Interface permettant d'accéder aux bases de données SQL
  - Implantations fournies par les fournisseurs de SGBD
  - Définie dans les paquetages java.sql et javax.sql
  - Le gestionnaire de pilotes a la responsabilité de chargé les bonnes implantations pour requêter les bases de données

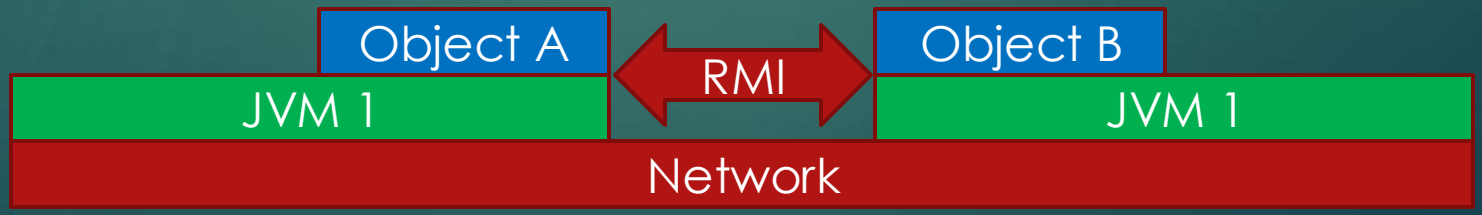
```
//Création d'une connection
Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost:5432/testdb",  "admin", "admin");
//Création d'une commande
Statement stmt = conn.createStatement();
//Exécution d'une requête
ResultSet rs = stmt.executeQuery("SELECT * from Person");
//Traitement de la réponse
while (rs.next()) {
        System.out.println(String.format("%s %s", rs.getString("firstName"), rs.getString("lastName")));
}
//Fermeture de la connection
conn.close();
```

Exercice

- Le fichier DataBaseAccess (db) dans le paquetage db contient du code faisant une requête sur le Web Sémantique afin de récupérer une liste de films
- Exécutez le fichier et observez les résultats
- Compléter le fichier afin de:
  - Déterminer le schéma de donnée
  - Stocker les résultats dans une base de donnée SQL
  - Enregistrer les informations concernant l'accès à la base dans un fichier .properties au préalable

Version 1.1 - RMI

- Remote Method Invocation
  - Facilite de développement d'applications distribuées en abstrayant la coucher réseau
  - Permet d'invoquer des méthodes sur des objets instanciés dans d'autres JVM, y compris sur des machines distantes
  - Basé sur l'implantation du protocole JRMP (Java Remote Method Protocol)
  - Généralement utilisé de pair avec l'interface JNDI (Annuaire)
  - Interopérable avec Corba à travers l'interface RMI-IIOP
  - Défini dans la paquetage java.rmi



Version 1.1 - RMI

1. Définition de l'interface distante
2. Implantation du serveur
3. Implantation du client
4. Exécution du serveur et du client

Exercice: implanter un service supportant les opérations élémentaires de l'arithmétique

```
public interface HelloRmi extends Remote {
    String sayHello() throws RemoteException;
}
```

```
public class HelloRmiServer extends UnicastRemoteObject implements HelloRmi {

    @Override
    public String sayHello() throws RemoteException {
        return "Hello !";
    }

    public static void main(String[] args) {
        LocateRegistry.createRegistry(1099);
         HelloRmiServer server = new HelloRmiServer();
         String url = "rmi://" + InetAddress.getLocalHost().getHostAddress()
                        + "/HelloRMI";
         Naming.rebind(url, server);
    }
}
```

```
public class HelloRmiClient {

    public static void main(String[] args) {
        Remote r = Naming.lookup("rmi://" + InetAddress.getLocalHost().getHostAddress() + "/HelloRMI");
        if (r instanceof HelloRmi) {
          String s = ((HelloRmi)r).sayHello();
          System.out.println(s);
        }
    }
}
```

Version 1.1: Réflexion et Introspection

- Accès à la définition d'une classe
  - En lecture à partir de la version 1.1
  - En écriture à partir de la version 1.2

```
Class c = Class.forName("fr.akfc.Personne");
Constructor constr = c.getConstructor(String.class, int.class);
Object o = constr.newInstance("Alain", 20);
```

- Le paquetage java.lang définit 3 classes pour l'utilisation dynamique des classes:
  - Class: accès aux caractéristiques d'une classe
  - ClassLoader: chargement des classes
  - Package: accès aux informations relatives aux paquetages
- La paquetage java.lang.reflect définit tous les objets permettant d'accéder de manière fine aux attributs / opérations d'une classe

Introspection: Exercice

- Écrivez un programme qui prend en entrée un nom de classe entièrement qualifié et affiche des informations sur la classe en utilisant l'API de réflexion de Java. Le programme doit fournir les informations suivantes :
  - Nom de la classe
  - Nom du paquetage
  - Nom de la superclasse (le cas échéant)
  - Interfaces implémentées (le cas échéant)
  - Constructeurs avec types de paramètres
  - Champs avec leurs noms et leurs types
  - Méthodes avec leurs noms, types de retour et types de paramètre
  - **Compléter la classe Introspection dans le paquetage misc**

Version 1.1: Internationalisation

- Adaptation aux caractéristiques locales des environnements d'exécution
  - Langue
  - Monnaie
  - OS
  - …

```
Locale fr = new Locale("fr", "FR");
Locale us = new Locale("en", "US");
ResourceBundle bfr =
ResourceBundle.getBundle("fr.cenotelie.cours.Message", fr);
ResourceBundle bus =
ResourceBundle.getBundle("fr.cenotelie.cours.Message", us);
System.out.println(bfr.getString("greetings"));
System.out.println(bus.getString("greetings"));
```

- Utilisation des classes "Locale" et "ResourceBundle" pour internationaliser ses applications
  - La JVM est instanciée avec un objet Local par défaut, que l'on peut accéder par la méthode statique Local.getDefault()
  - La classe ResourceBundle s'appuie sur des fichiers dictionnaires postfixé par la langue et le pays

(**Message_fr_FR.properties**)
greetings=Salut

(**Message_en_US.properties**)
greetings=Hello

Internationalisation: Exercice

- Compléter la classe Internationalisation dans le paquetage misc
- Générer un fichier hello.txt à partir du template hello_template.txt
  - Vous devrez pour cela charger les fichiers d'internationalisation selon la locale de l'utilisateur
- Utiliser la classe MessageFormat pour formater la propriété greeting afin d'injecter le nom de l'utilisateur courant

Version 1.2: strictfp

- Restriction pour assurer la portabilité des calculs portant sur les nombres réels
- La précision des calculs diffère selon les capacités offertes par les plateformes matériels (co-processeurs)
  - Il peut y avoir des résultats différents pour un même calcul effectué sur deux machines différentes
  - Pour éviter ces situations, on peut forcer le compilateur à se conformer aux recommandations du standard IEEE 754 sur l'arithmétique des nombres flottants
- L'utilisation du mot-clef "**strictfp**" est recommandé dans ce cas
  - Sur la classe, sur une interface ou sur une méthode

Version 1.2: Swing

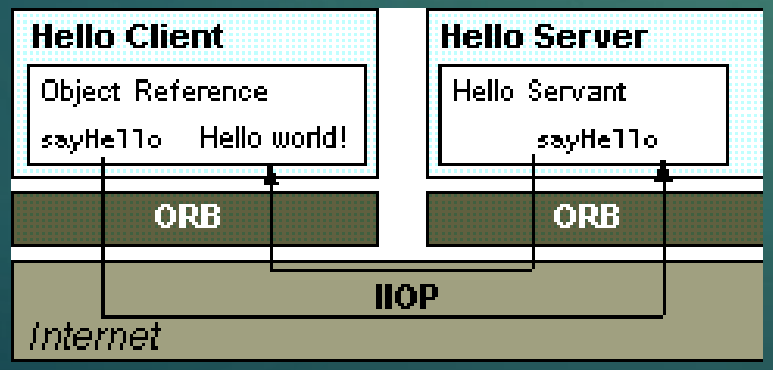
- Avant Swing, la librairie AWT permettait de spécifier des interfaces graphiques
  - En s'appuyant sur les widgets natifs du système d'exploitation
  - Avec pour résultat un manque de cohérence dans l'affichage des applications graphiques selon les plateformes (UNIX, Windows, iOS)
- La librairie Swing permet d'homogénéiser le rendu des interfaces graphiques, quel que soit le système d'exploitation
- La librairie Swing est
  - Indépendante de la plateforme
  - Modulaire
  - Extensible
  - Configurable
  - Calqué sur le pattern MVC, visant à rendre indépendante la représentation du modèle

Version 1.2: Applet

- Application embarquée dans les navigateurs
- Plus utilisé de nos jours, essentiellement pour des problèmes de
  - Dépendance (JRE)
  - Portabilité (Affichage sur les téléphones)
  - Sécurité
- **Déprécié à partir de Java 9**

Version 1.2: IDL

- Interface Definition Language
  - Implantation du standard OMG CORBA
  - Comme RMI, destiné à construire des applications réparties
  - A la différence de RMI, n'est pas propre à Java
    - CORBA permet d'interconnecter des objets implantés par différentes technologies (C++, python, etc.)
  - L'ORB (Object Resource Broker) agit comme l'annuaire RMI
  - Les ORB peuvent communiquer entre eux par le protocol IIOP (Internet Inter ORB Protocol)
- Evolution vers CCM (Corba Component Model) puis EJB (Enterprise JavaBean)



1. Déclaration des opérations
2. Mapping vers java avec l'outil idlj
3. Développement du serveur
4. Développement du client
5. Exécution de l'ORB
6. Exécution de l'application

Version 1.2: JNI

- Java Native Interface
  - Permet d'appeler du code natif C dans une classe java
  - Essentiellement pour des raisons de performances, au détriment de la portabilité
- Etapes:
  1. Déclaration de méthode native en java
  2. Compilation java
  3. Génération d'un fichier d'en-tête avec l'outil javah
     - java –jni MaClasse => MaClasse.h
  4. Ecriture du code natif implantant les opérations du fichier d'en-tête généré
  5. Compilation du code natif en une librairie (dll, so, …)

- Cf. Classe NativeCalculator (jni) et script build-native.sh à la racine du projet

JNI - Inconvénients et alternatives

- Inconvénients :
  - Processus laborieux.
  - Gestion manuelle de la mémoire et des conversions de types.
  - Risque d'erreurs de segmentation et de fuites mémoire.
  - Code natif spécifique à chaque plateforme.
- Alternatives:
  - **JNA**: Bibliothèque Java qui permet d'appeler du code natif sans écrire de code C/C++ spécifique ou générer des en-têtes.
  - **JNR**: Alternative à JNA, développée par la communauté JRuby, visant à améliorer les performances et la maintenabilité.

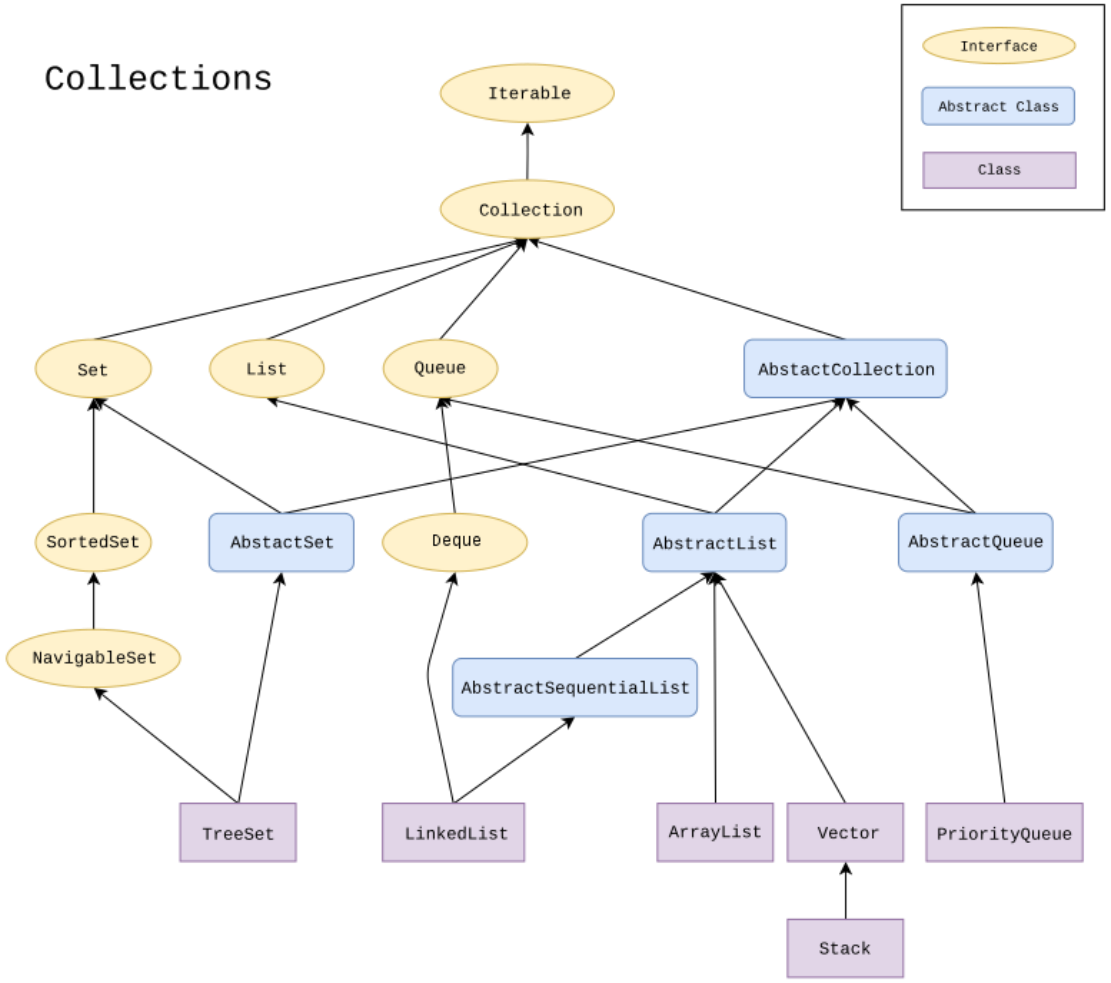
JNI - Comparatif

Aspect	JNI	JNA	JNR
Simplicité	Faible	Élevée	Élevée
Performances	Très élevées	Moyennes	Élevées
Portabilité	Faible (code natif spécifique)	Élevée (code Java uniquement)	Élevée (code Java uniquement)
Productivité	Faible	Élevée	Élevée
Maturité	Très élevée	Élevée	Moyenne

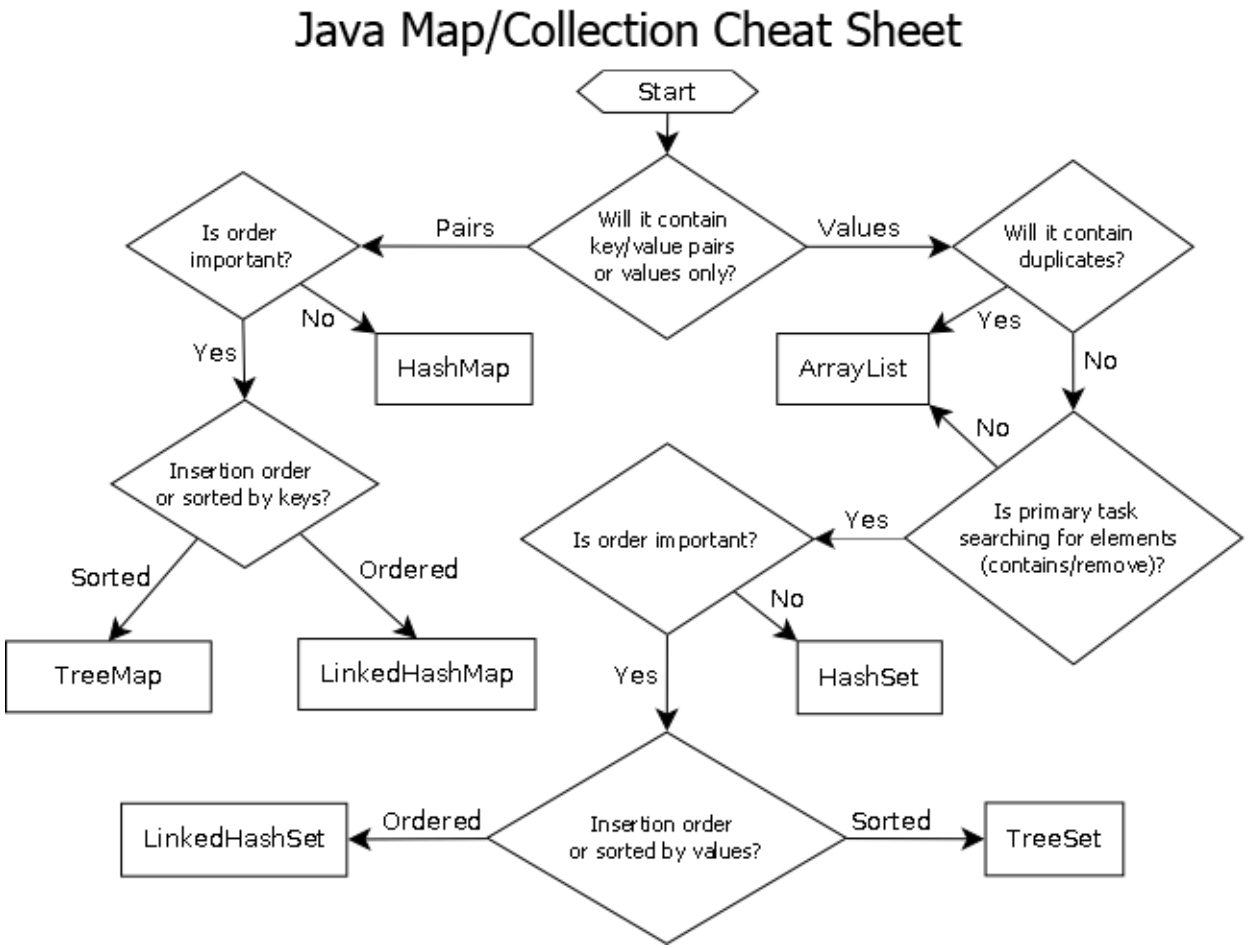
- JNI reste la solution de référence pour les besoins de performances extrêmes ou l'intégration de code natif complexe.
- JNA et JNR simplifient grandement l'intégration de bibliothèques natives en Java, avec un compromis sur les performances, mais un gain considérable en simplicité et en productivité.
- JNR se distingue par de meilleures performances et une API plus moderne que JNA, mais reste moins répandu.

Version 1.2: Collections

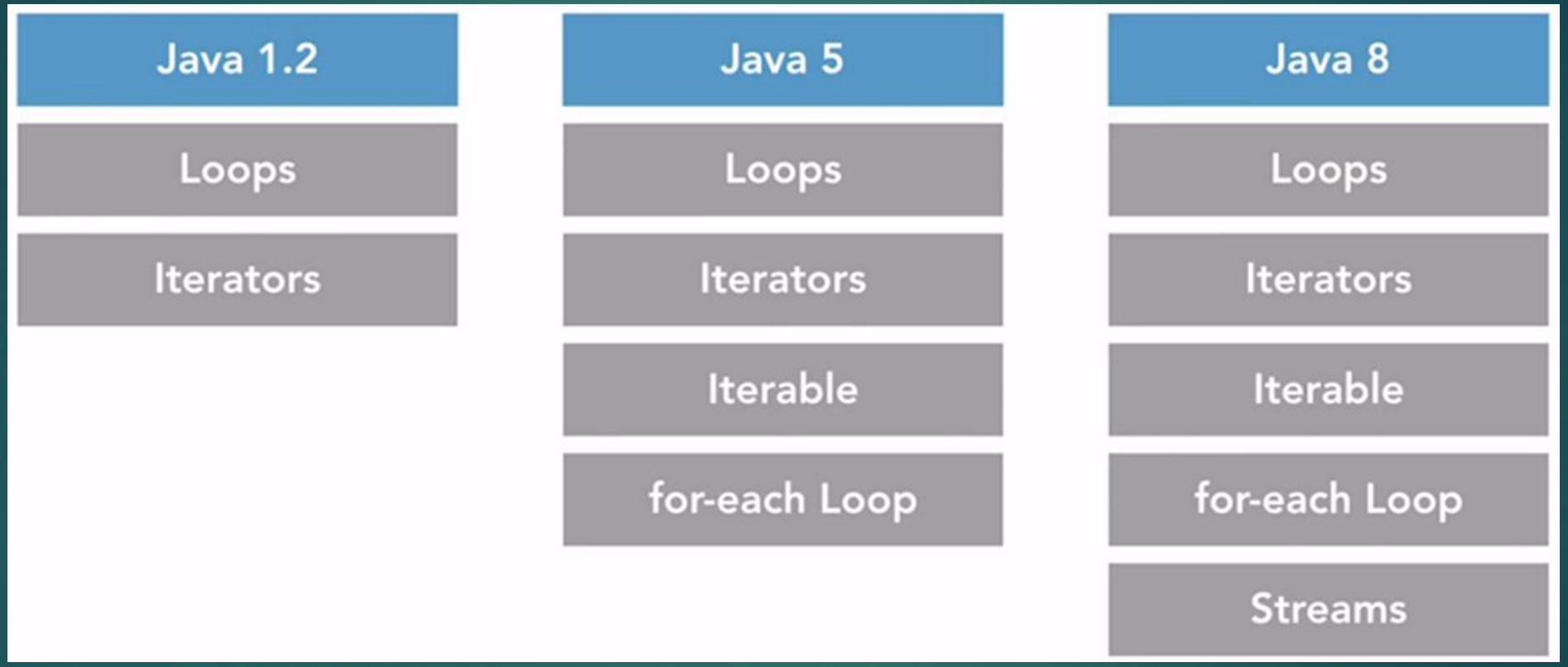
- Ensemble de classes et d'interfaces implantant les structures de données les plus communes
- Contrairement aux tableaux, pas de taille fixe à l'instanciation
- 3 grandes catégories de collection
  - **Liste** – collection ordonnée
  - **Ensemble** – collection non ordonnée, à éléments uniques
  - **Queue** – First-in First-out
    - Deque: Double-Ended Queue
- L'introduction du framework Collections est destiné à uniformiser et faciliter la manipulation de collections
- La classe Collections implante un certain nombre de méthodes utiles à la manipulation de collections
- Définis dans le paquetage java.util



Collections Cheat sheet



Collections: évolution des API



Exercice: Itération

```
record Student(String name) {}

public class Iteration {

    public static void main(String[] argv) {
        List<Student> students = new ArrayList<>(List.of(
                new Student("Pauline"),
                new Student("Alain"),
                new Student("Michel"),
                new Student("Marie")
        ));
    }

    //TODO: complete

}
```

1. Donnez les différentes manières d'itérer sur la collection d'étudiants pour afficher les noms
2. Supprimer l'élève dont le nom est Michel durant l'itération
3. Quels sont les avantages et inconvénients des implantations LinkedList et ArrayList ?

Compéter la classe Iterations dans le paquetage collections

Linked List

- **Avantages**
  - Insertion et suppression efficaces : La liste chaînée permet des opérations d'insertion et de suppression efficaces, en particulier lorsqu'il s'agit d'ajouter ou de supprimer des éléments au début ou au milieu de la liste. Il suffit de mettre à jour les références des nœuds voisins.
  - Taille dynamique : LinkedList ajuste automatiquement sa taille au fur et à mesure que des éléments sont ajoutés ou supprimés, ce qui la rend adaptée aux scénarios dans lesquels la taille de la liste change fréquemment.
  - Itération : LinkedList donne de bons résultats lors du parcours séquentiel de la liste, car elle permet une itération efficace du début à la fin.
- **Inconvénients**
  - Temps d'accès : Les performances de LinkedList en matière d'accès aléatoire (accès à un élément à un index spécifique) sont relativement plus lentes que celles de ArrayList. Il faut parcourir la liste depuis le début ou la fin jusqu'à ce que l'index souhaité soit atteint.
  - Surcharge de mémoire supplémentaire : LinkedList nécessite de la mémoire supplémentaire pour stocker les références de chaque élément, ce qui peut entraîner une consommation de mémoire supérieure à celle de ArrayList.
  - Vitesse d'itération : alors que LinkedList fonctionne bien pour l'itération séquentielle, l'itération inverse ou l'accès aux éléments à des positions arbitraires peuvent être plus lents en raison de la traversée de la liste nœud par nœud.

Array List

- **Avantages**
  - Accès aléatoire : Les listes de tableaux permettent un accès aléatoire efficace aux éléments, car elles autorisent l'indexation directe à l'aide d'une position d'index. L'extraction d'un élément à un index spécifique est rapide et a une complexité temporelle constante (O(1)).
  - Utilisation efficace de la mémoire : ArrayList utilise un tableau dynamique en interne, ce qui lui permet d'utiliser efficacement la mémoire. Elle consomme généralement moins de mémoire que LinkedList.
  - Vitesse d'itération : ArrayList fonctionne bien pour tout type d'itération, y compris séquentielle, inverse, ou pour accéder à des éléments à des positions arbitraires.
- **Inconvénients**
  - Redimensionnement dynamique : Lorsque la liste de tableaux dépasse sa capacité initiale, elle doit redimensionner le tableau interne, ce qui implique la création d'un nouveau tableau et la copie des éléments. Cette opération de redimensionnement peut être coûteuse pour les listes de grande taille.
  - Insertion et suppression : Les performances d'ArrayList pour les opérations d'insertion et de suppression, en particulier au début ou au milieu de la liste, peuvent être plus lentes que celles de LinkedList. Il est nécessaire de décaler les éléments pour tenir compte des changements.

forEach vs Stream

- **Syntaxe** : La boucle foreach est une construction simple fournie par Java pour l'itération sur des collections. Elle utilise la syntaxe améliorée de la boucle for introduite dans Java 5.
- **Mutabilité** : La boucle foreach opère sur la collection elle-même et permet de modifier les éléments de la collection pendant l'itération.
- **Séquentielle** : La boucle foreach itère sur la collection de manière séquentielle, en traitant chaque élément un par un dans l'ordre où il apparaît dans la collection.
- **Fonctionnalité limitée** : La boucle foreach offre une fonctionnalité de bouclage de base, mais ne dispose pas de fonctionnalités avancées telles que le filtrage, le mappage, la réduction et le traitement parallèle.

forEach vs **Stream**

- Fluent API: L'API de flux fournit une interface de programmation fluide et fonctionnelle pour traiter les collections et effectuer diverses opérations sur les éléments.
- Opérations fonctionnelles : Les flux prennent en charge un large éventail d'opérations telles que le filtrage, le mappage, le tri, la réduction et l'agrégation de données. Ces opérations peuvent être facilement combinées pour réaliser des transformations de données complexes.
- Évaluation paresseuse : Les flux sont évalués à la demande, ce qui signifie que les opérations intermédiaires sont exécutées paresseusement. Cela permet un traitement efficace, en particulier lorsque l'on travaille avec de grandes collections ou lorsque des courts-circuits sont possibles.
- Traitement parallèle : Les flux peuvent tirer parti des processeurs multicœurs en activant le traitement parallèle, ce qui permet de répartir automatiquement la charge de travail sur plusieurs threads pour une exécution plus rapide.
- Immutabilité : Les flux opèrent sur la collection source de manière non modifiable. La collection d'origine reste inchangée et chaque opération sur le flux renvoie un nouveau flux avec les données transformées.

Set

- 3 implantations:
  - HashSet: ordre basé sur le hash des objets
  - LinkedHashSet: ordre basé sur l'ordre d'insertions
  - TreeSet: Implanté par un arbre binaire

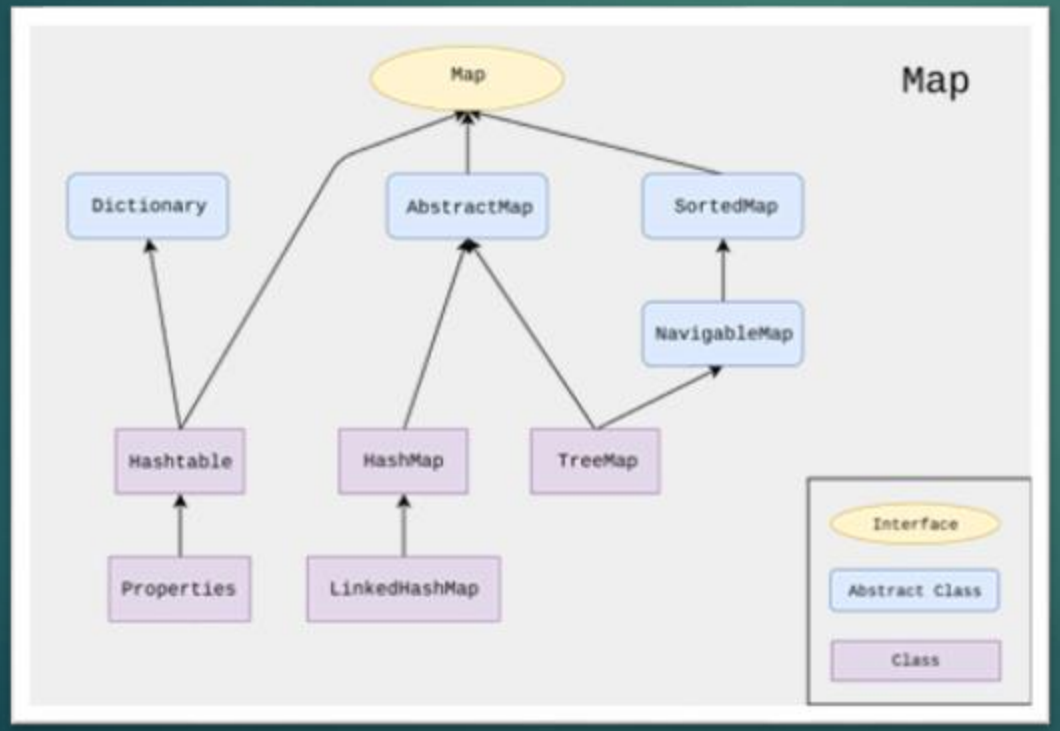
- **Exercice:** Compléter la classe BinarySearch pour:
  - Trier par âge et par nom en utilisant l'interface Comparator
  - Obtenir un ensemble ordonné d'hommes
  - Faire la recherche d'un homme ayant 41 ans en utilisant la méthode binarySearch de la classe Collections

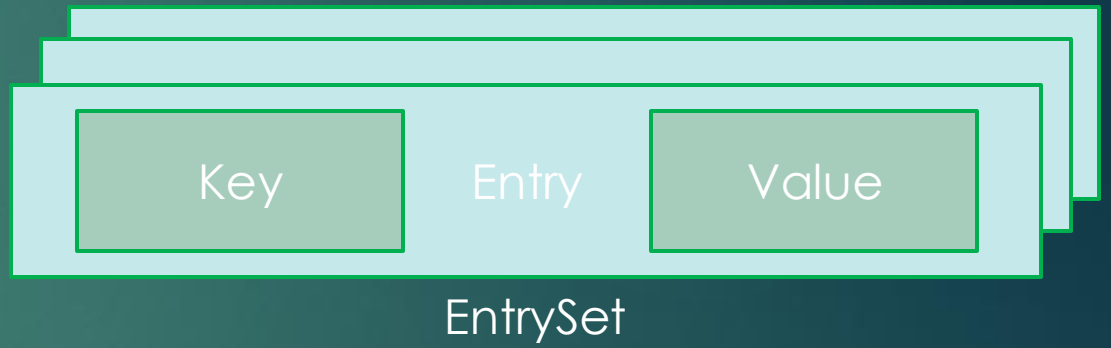
Listes non-modifiables

- Les méthodes d'usine statiques List.of et List.copyOf offrent un moyen pratique de créer des listes non modifiables. Les instances List créées par ces méthodes ont les caractéristiques suivantes :
  - Elles ne sont pas modifiables. Les éléments ne peuvent pas être ajoutés, supprimés ou remplacés.
  - Ils n'acceptent pas les éléments nuls.
  - Elles sont sérialisables si tous les éléments sont sérialisables.
  - L'ordre des éléments de la liste est le même que celui des arguments fournis, ou des éléments du tableau fourni.

Version 1.2 - Map

- N'étend pas l'interface Collection (non-itérable)
- Les valeurs sont accessibles par des clefs





Map non modifiables

- Les méthodes d'usine statiques Map.of, Map.ofEntries et Map.copyOf offrent un moyen pratique de créer des dictionnaires non modifiables. Les instances de Map créées par ces méthodes ont les caractéristiques suivantes :
  - Elles sont non modifiables. Les clés et les valeurs ne peuvent pas être ajoutées, supprimées ou mises à jour.
  - Ils n'acceptent pas les clés et les valeurs nulles.
  - Ils sont sérialisables si toutes les clés et valeurs sont sérialisables.
  - Ils rejettent les clés dupliquées au moment de la création.

Version 1.3: Améliorations notables

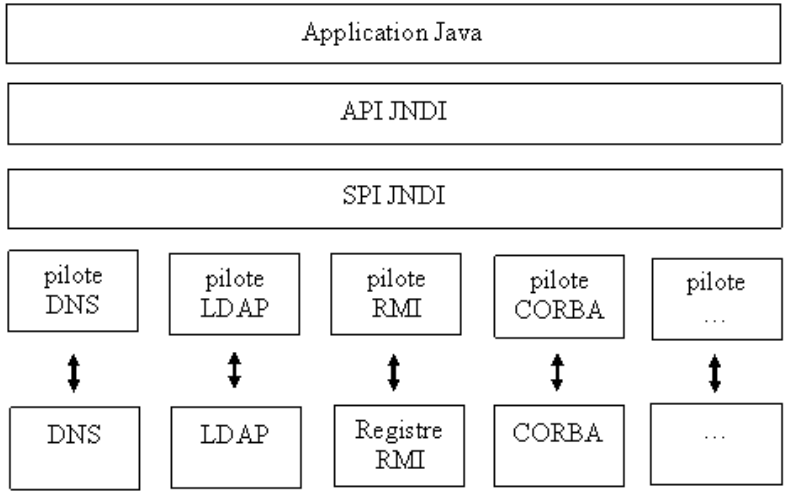
- JVM HotSpot
  - Pour bureau et serveur
  - Performance accrue
  - JIT et optimisation adaptative
- Amélioration de RMI
  - Interfaçage avec Corba
- Mot-clef assert(activé avec l'option –ea)
- Support des expressions régulières (java.util.regex)
- Synthetic Proxy Class
  - Classes générées à la volée par le compilateur
  - Notamment pour les classes anonymes qui implantent une ou plusieurs interfaces
- NIO: moyen plus efficace et plus souple d'effectuer des opérations d'entrée et de sortie

```
public class MyClass {
    public void someMethod() {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello World!");
            }
        }
        r.run();
    }
}
```

r => MyClass$1.class

Version 1.3 - JNDI

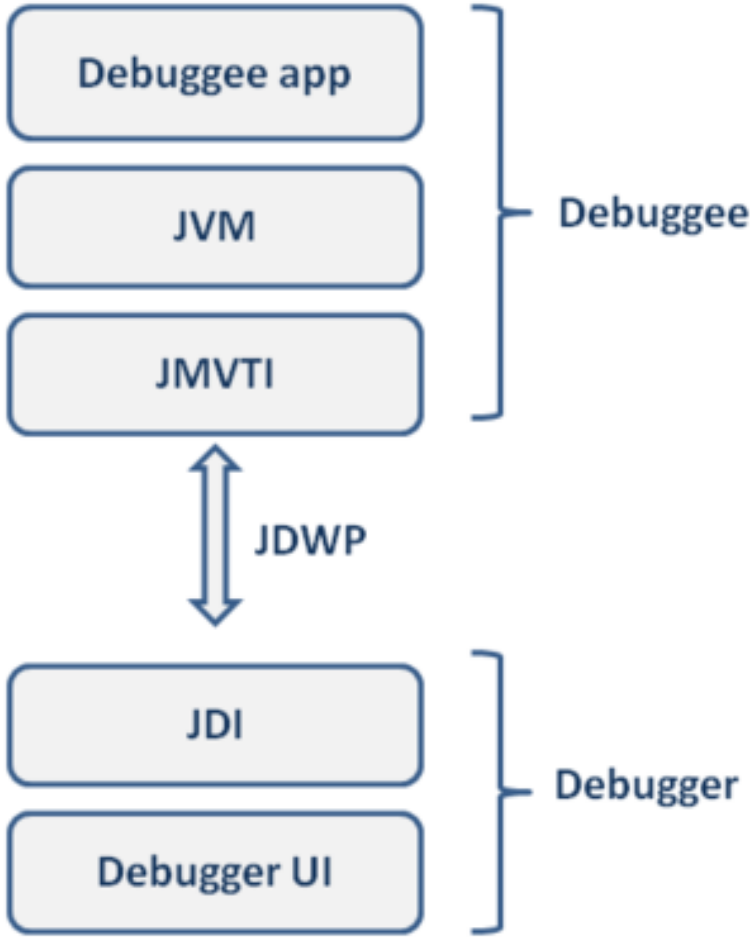
- **Java Naming and Directory Interface**
- Interface unique pour utiliser différents services de nommage
  - DNS, LDAP, NIS, COS, etc.
- Un service de nommage permet à partir d'un nom unique d'accéder à un objet qu'il se trouve
  - Utilisé par plusieurs API majeurs: JDBC, EJB, JMS, …
- Association nom / objet = **Binding**
- Ensemble d'associations nom / objet = **Context**
  - Contexte racine et sous-contexte
- Paquetages
  - javax.naming: utilisation de services de nommage
  - javax.naming.directory: utilisation de services d'annuaire
  - javax.naming.event: gestion des événements d'accès aux services
  - javax.naming.ldap: utilisation de la version 3 de ldap
  - javax.naming.spi: développement de pilotes d'interfaçage



Cf. Classes Jndi* (misc)

Version 1.3 - JPDA

- Java Platform Debugger Architecture
  - Java Debugger Interface (JDI) définit une interface permettant de développer des outils de debug distants
  - Java Debug Wire Protocol (JDWP) définit un protocol de communication entre le debugger et l'application à débugger
  - JVM Tools Interface (JVMTI) définit une interface native pour contrôler l'exécution d'un programme et inspecter son état



Version 1.4 - Améliorations notables

- Assertions
- Expressions régulières
- Chaînage des exceptions
- IPv6
- Entrées / Sorties non bloquantes (asynchrones)
- Logs (interface de standardisation des logs)
- Support des formats d'image JPEG et PNG
- Support de XML et XSLT
- Sécurité et cryptage: JCE (Crypto), JSSE (TLS/SSL) et JAAS (Authenfication)
- Java Web Start (jaws): Permet l'exécution de programmes java directement depuis internet - déprécié à partir de la version 9
- Support pour les préférences (java.util.prefs) => cf. Classe PreferencesExample (misc)

Exercices

- Compléter la classe NonBlockingFileRead pour lire de manière non-bloquante un fichier par l'API NIO (io)
- Compléter la classe GenerateHtml (misc) afin de générer un contenu HTML
  - À partir d'une source de données XML et d'une transformation XSLT
  - Le résultat sera à afficher sur la sortie standard
- Compléter la classe Security (misc) afin d'encrypter et de décrypter des données
  - Générer une clef de sécurité
  - Utiliser cette clef pour les opérations d'encryptage / décryptage

Version 5 - Types

- Enumérations

```
public enum Day {
    MONDAY(false), TUESDAY(false), WEDNESDAY(false),
    THURSDAY(false), FRIDAY(false),
    SATURDAY(true), SUNDAY(true);

    private boolean isWeekend;

    Day(boolean isWeekend) {
        this.isWeekend = isWeekend;
    }
}
```

- Autoboxing

```
Integer i = new Integer(9);
Integer i = 9; // erreur avant la version 5
//Unboxing
Integer k = new Integer(4);
int l = k.intValue(); // correct dans toutes les versions
int m = k;            // erreur avant la version 5
```

Version 5 - Génériques

- Amélioration du système de types de java
  - Permettant d'opérer sur des objets de différents types de manière sûre et reproductible
  - Préoccupations de capitalisation et de réutilisation
  - Vérification de la sûreté du code à la compilation
  - Très utilisé dans les collections par exemple
    - List<**String**> mylist = new ArrayList<**String**>();
    - Evite des opérations de cast ou de vérification
  - S'appliquent sur les classes ou les opérations

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}

void drawAll(List<? extends Shape> shapes) { … }

<T> void fromArrayToCollection(T[] a, Collection<T> c)
{
    for (T o : a) {
        c.add(o);
    }
}
```

```
public interface List <E> {
    void add(E x);
    Iterator<E> iterator();
}
```

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; //Error!!!
lo.add(new Object());
```

Version 5 - Génériques

- Vous pouvez restreindre les types qui peuvent être utilisés comme arguments de type dans un type paramétré.
- Les génériques prennent également en charge les caractères génériques (?), qui représentent un type inconnu.
  - Les caractères génériques peuvent être utiles dans les paramètres, les champs et les variables locales et peuvent être délimités ou non.
  - Caractère générique non délimité: Le joker non borné ? représente n'importe quel type. Il est utilisé lorsque vous ne connaissez pas ou ne vous souciez pas du type spécifique du paramètre.
  - Caractère générique délimité : Les caractères génériques délimités limitent le type inconnu à une gamme spécifique de types
- **Exercice**: Compléter la classe Generics (misc)

Version 5 - Annotations

- Métadonnées sur le code source
  - Introduites en alternative aux fichiers XML
  - Accessibles grâce à la réflexion (java.lang.reflect.AnnotatedElement)
- Les annotations peuvent être utilisées lors de la compilation ou à l'exécution
  - En utilisant la réflexion
- Il existe un certain nombre d'annotations standards
  - Appliqués au code java: @Override, @Deprecated, @SuppressWarnings
  - Appliqués aux déclarations d'annotations: @Retention, @Documented, @Target, @Inherited
- La paquetage javax.validation.constraints standardise un certain nombre de contraintes de validation des données
  - @NotNull, @Past, @Size

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
      String first();
      String last();
 }

@Author(first = "Oompah", last = "Loompah")
  Book book = new Book();

Class<Book> clObj = Book.class
AnnotatedElement e = (AnnotatedElement) clObj;
if (e.isAnnotationPresent(Author.class)) {
  Annotation a = e.getAnnotation(Author.class);
  Author auth = (Author)a;
  System.out.println(auth.first());
}
```

Version 5: Améliorations notables

```
void printReport(String header, int... numbers) {
//numbers represents varargs
    System.out.println(header);
    for (int num : numbers) {
        System.out.println(num);
    }
}
```

- Imports statiques
- Varargs
  - Le dernier argument d'une méthode peut être un paramètre d'arité variable
- Syntaxe de la boucle for étendue pour supporter les itérations sur n'importe quelle collection itérable
- Sémantique d'exécution des programmes multi-threadés améliorée par un modèle de mémoire plus clair
  - Dans le paquetage **java.util.concurrent**
- Génération automatique de stub pour les objets RMI
- Classe Scanner pour le parsing de données provenant de différentes entrées / buffers
- Swing: Look & Feel customisable

Version 6 - Améliorations notables

- Performances
- Services Web (JAX-WS)
- Support des langages de scripts
- Invocation du compilateur de manière programmatique
- JAXB (Java XML Binding)
- JDBC 4.0
- Garbage collector

Version 6 - Exemples

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");
String script = "var message = 'Hello, Java 6 scripting!'; message;";
Object result = engine.eval(script);
```

```
Console console = System.console();
String name = console.readLine("Enter your name: ");
char[] password = console.readPassword("Enter your password: ");
console.printf("Hello, %s! Your password is %s\n", name, new String(password));
```

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
int result = compiler.run(null, null, null, "HelloWorld.java");
if (result == 0) { System.out.println("Compilation succeeded"); }
else { System.out.println("Compilation failed"); }
```

Version 7: Améliorations notables

- JVM:
  - Support des langages dynamiques (instruction invokedynamic)
    - Utilisation de invokedynamic également pour les lambdas
  - Amélioration de la concurrence
- Langage:
  - Switch: Extension de la structure de contrôle aux chaînes de caractères
  - Opérator <> pour l'inférence de type sur les génériques
  - Exceptions: multi-catch
  - Concurrence: RecursiveTask

```
try { … }
catch (IOException | SQLException e) { … }
```

Exercice : Performance

- Dans la classe Fibo, écrire une méthode récursive qui calcule le terme de rang n de la suite de Fibonacci
  - Exécuter et mesurer le temps d'exécution pour fibo(50)
  - Qu'observez-vous ?
- Améliorer la méthode en utilisant un accumulateur
  - Exécuter et mesurer le temps d'exécution pour fibo(50)
  - Qu'observez-vous ?
  - Comment peut-on améliorer l'algorithme ?
- Ecrire une méthode qui affiche tous les nombres pairs de la suite de Fibonacci inférieurs ou égaux à un nombre donné en paramètre

Version 8: lambdas

- Inspiré par les langages fonctionnels où les fonctions sont des citoyens de premier ordre
- Une fonction peut être passée en paramètre d'une autre fonction (high order function)
- Une fonction peut être anonyme
  - On appelle lambda ou closure de telle fonction
- Les traits fonctionnels ont été introduits dans la plupart des langages modernes
  - Ils peuvent être utilisés en remplacement des classes anonymes

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

Version 8: Lambdas

- Une simple interface annotée @FunctionalInterface
- Interface qui ne peut avoir qu'une seule opération
- Le paquetage java.util.function définit un certain nombre d'interfaces fonctionnelles utilisées par les autres paquetages
- Parmi les interfaces fonctionnelles, l'interface **Predicate** représente une fonction à un argument et qui renvoie une valeur booléenne

Version 8: Lambdas

- Les lambdas peuvent utilisés pour faire référence à des méthodes existantes
- Java autorise le référencement des opérations existantes
- 4 types de références
  - Les méthodes statiques: MaClasse::maMethodeStatique
  - Les méthodes d'instances: monObjet::maMethodeDInstance
  - Les méthodes d'instances au niveau classe: MaClass::maMethodeDInstance
  - Les constructeurs: MaClass::new

Lambdas - Exemples

```
for (int i = 0; i < 10; i++) {
        l.add(r.nextInt());
}
//utilisation de référence de méthode
l.forEach(System.out::println);
```

```
monBouton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("clic");
} });
```

Vs

```
monBouton.addActionListener(event -> System.out.println("clic"));
```

```
Thread monThread = new Thread(() -> { System.out.println("Mon traitement"); });
monThread.start();
```

Exercice : lambdas

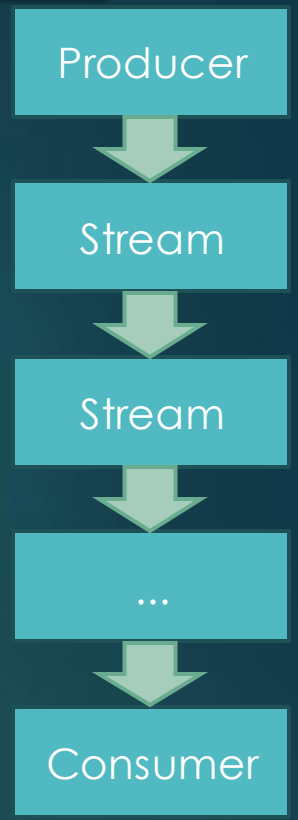
- Compléter la classe Lambdas (streams)
- Ecrire une fonction f: x -> x^2
- Ecrire une fonction g: x -> x + 5
- Composer les fonctions f et g pour calculer:
  - $(2 + 5)^2$
  - $2^2 + 5$

Version 8: API Stream

- Facilite l'exécution de traitements séquentielles ou parallèles sur des collections de données
  - Sans se préoccuper des détails techniques bas niveau (itérations, …)
  - Par une approche fonctionnelle (utilisation de lambdas)
  - Permet une plus grande concision du code dans un style déclaratif et une évaluation tardive (paradigme fonctionnel)
  - Fait un usage intensif de l'API Collectors (java.util.stream.Collectors)
  - A ne pas confondre avec les streams du paquetage java.io

```
double tailleMoyenneDesHommes = employes.stream() .filter(e -> e.getGenre() == Genre.HOMME)
                                                  .mapToInt(e -> e.getTaille()) .average() .orElse(0);
```

```
double tailleMoyenneDesHommes = employes.parallelStream() .filter(e -> e.getGenre() == Genre.HOMME)
                                                  .mapToInt(e -> e.getTaille()) .average() .orElse(0);
```



Version 8: Optional

- La gestion des pointeurs null peut parfois amener à écrire du code compliqué
  - Par exemple:
    - String version = computer.getSoundCard().getUsb().getVersion();
    - Possible NPE si l'ordi n'a pas de carte son
  - Solution 1: programmation défensive
  - Solution 2: utilisation de la classe Optional
    - Avantage: on explicite la possibilité d'une valeur nulle

```
String version = "UNKNOWN";
if(computer != null){
  Soundcard soundcard =
computer.getSoundcard();
  if(soundcard != null){
    USB usb = soundcard.getUSB();
    if(usb != null){
      version = usb.getVersion();
    }
  }
}
```

```
public class Computer {
  private Optional<Soundcard> soundcard;
  public Optional<Soundcard> getSoundcard() { ... }
  ...
}
public class Soundcard {
  private Optional<USB> usb;
  public Optional<USB> getUSB() { ... }

}
public class USB{
  public String getVersion(){ ... }
}
```

```
String name = computer.flatMap(Computer::getSoundcard)
                      .flatMap(Soundcard::getUSB)
                      .map(USB::getVersion)
                      .orElse("UNKNOWN");
```

Version 8: API Collectors

- L'API Collectors fait partie du package java.util.stream et a été introduite avec Java 8 pour faciliter la collecte des résultats de traitements effectués sur des Streams.

- Elle permet de transformer un Stream en une structure de données finale (List, Set, Map, etc.) ou d'effectuer des opérations d'agrégation (somme, moyenne, min, max, etc.) de manière concise et lisible.

- Exemples:
  - List<String> names = stream.collect(Collectors.toList());
  - Set<String> uniqueNames = stream.collect(Collectors.toSet());
  - Collection<String> collection = stream.collect(Collectors.toCollection(LinkedList::new));

API Collectors - Agrégation

- Somme, Moyenne, Min, Max
  - double average = stream.collect(Collectors.averagingInt(Integer::intValue));
  - int sum = stream.collect(Collectors.summingInt(Integer::intValue));
  - Optional<String> max = stream.collect(Collectors.maxBy(Comparator.naturalOrder()));
  - Optional<String> min = stream.collect(Collectors.minBy(Comparator.naturalOrder()));

- Regroupement, Partitionnement
  - Map<String, List<Employee>> byDept = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment));
  - Map<Boolean, List<Employee>> partition = employees.stream().collect(Collectors.partitioningBy(e -> e.getSalary() > 1000));

API Collectors: Teeing

- Méthode statique permet de collecter à l'aide de deux collecteurs indépendants, puis de fusionner leurs résultats à l'aide de la BiFunction fournie, introduite à partir de la version 12

- Chaque élément transmis au collecteur résultant est traité par les deux collecteurs en aval, puis leurs résultats sont fusionnés à l'aide de la fonction de fusion spécifiée dans le résultat final.

- **Exercice**: Compléter la classe Teeing (collections) afin de calculer la moyenne des nombres pairs et des nombres impairs d'une liste de nombres

```
List<Employee> employeeList = Arrays.asList(
                new Employee(1, "A", 100),
                new Employee(2, "B", 200),
                new Employee(3, "C", 300),
                new Employee(4, "D", 400));

    HashMap<String, Employee> result = employeeList.stream().collect(
            Collectors.teeing(
                Collectors.maxBy(Comparator.comparing(Employee::getSalary)),
                Collectors.minBy(Comparator.comparing(Employee::getSalary)),
                (e1, e2) -> {
                  HashMap<String, Employee> map = new HashMap();
                  map.put("MAX", e1.get());
                  map.put("MIN", e2.get());
                  return map;
                }
            ));

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

I/O : Scanner

- 2 types de Flux:
  - Entrants : InputStream (byte), Reader (char): Fichier, audio, Objet, Pipe, Buffer, etc
  - Sortants: OutputStream (byte), Writer (char): Fichier, Filtre, Objet, Pipe, etc.
- Les versions bufferisées de ces flux permettent d'éviter les appels directs à l'OS
  - Améliore les performances (temps d'accès, accès concurrents, etc.)
- L'API Files facilite énormément la manipulation des fichiers
- Pour faciliter la lecture des fichiers, la version 8 de java a introduit l'objet **Scanner**
  - Se charge de convertir le contenu d'un fichier en tokens (séparés par des espaces par défaut).

Exercice - Streams

- A partir du fichier titanic.csv
- Compléter la classe Titanic (streams) avec les méthodes suivantes
  - loadData(): charge les données comme un tableau de records
    - Utiliser la classe Scanner avec les contraintes suivantes:
      - ";" utilisé comme séparateur de champs
      - Le deuxième champs doit être interprété comme un booléen
      - Le quatrième champs doit être interprété comme une énumération (homme | femme)
      - Seul les 5 premiers champs sont nécessaires
  - head(n: int) et tail(n: int) qui renvoient respectivement les n premières / dernières lignes du tableau dans un format compréhensible
  - getCustomers(): renvoie la liste de l'intégralité des entrées de la table

Exercice - Streams (Suite)

- Dans la méthode main, utiliser les streams séquentiels pour faire les requêtes suivantes:
  - Âge moyen des hommes
  - Civilité de chaque personne dans le format suivant: titre | prénom | nom
  - Moyenne des âges par sexe
  - Nombre de personnes par catégories d'âge
    - Moins de 20 ans, 20 /40 ans, 40 / 60 ans, plus de 60 ans
  - Afficher les entrées correspondantes aux femmes célibataires voyageant en première classe trié par âge et ayant survécu
- Modifier avec des streams parallèles (si possible)
- Pour cet exercice, vous devrez écrire une énumération Sex et un Record Customer

Exercice: I/O

- Écrire un programme qui fait une recherche récursive sur:
  - Une extension de fichier
  - Une expression régulière contenue dans le fichier
  - Code java à modifier: SearchFiles (io)

- Fournir 3 implantations:
  - Une fonction récursive
  - Utilisation de la fonction statique Files::walkFileTree
  - Utilisation de la fonction statique Files::find

Version 8: Nashorn

- Nashorn est un moteur JavaScript développé dans le langage de programmation Java par Oracle disponible à partir de la version 8

- Avec la sortie de Java 11, Nashorn est déclaré déprécié. Il est supprimé du JDK avec sa version 15.

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
Object result = engine.eval( "var greeting='hello world'; print(greeting); greeting");
```

Version 9: Modularisation (JIGSAW)

- Nouveau niveau d'abstraction introduit au-dessus du paquetage
  - Java Platform Module System (JPMS) ou simplement Module
  - Pour alléger la JVM (avec l'utilisation de JLink)
- Un module est formé d'un groupe cohérent de paquetages décrit par un fichier de description
  - En quelque sorte un paquetage de paquetages
- Conformément à ce que son fichier de description indique, un module est caractérisé par:
  - Son nom
  - Ses dépendances
  - Ses paquetages publiques
  - Les services offerts / requis
  - Les permissions de réflexion

Version 9: modularisation

- Il existe 4 types de modules:
  - Les modules "système", incluant java se et java jdk
  - Les modules "application", ceux assemblés par les utilisateurs
  - Les modules "automatique", constitués par les fichiers jar ajoutés au chemin des modules (module path)
  - Les modules "non nommé", constitués par les fichiers jar ajoutés au classpath et non au chemin des modules
- Règle à respecter: un module par fichier jar
- Lister les modules: java --list-modules

Version 9: Modularisation - Exemple

- Dépendances - mot clef "requires"
  - A la compilation uniquement "requires static"
  - Transitive "requires transitive"
- Exports – mot clef "exports"
  - Permet de déclarer les paquetages visibles
  - Avec des restrictions possible "exports xxx to yyy"
    - Le paquetage xxx n'est visible que du paquetage yyy

```
module monmodule {
    requires transitive java.logging;
    exports fr.cenotelie.monmodule;
    uses java.util.LinkedList;
    provides fr.cenotelie.monmodule.MyInterface
                    with fr.cenotelie.monmodule.MyInterfaceImpl;
    opens fr.cenotelie.monmodule to java.logging;
}
```

Version 9: Modularisation - Exemple

- Services requis / offerts
  - Le mot clef "uses" permet de spécifier une dépendance directe à une classe sans avoir à charger tous les modules (par transitivité) nécessaires à l'exécution de cette classe – chargement dynamique
  - Le mot clef "provides" permet de spécifier les classes réalisant certaines interfaces qui sont offertes aux autres modules
- Permission de réflexion
  - Le mot clef "open" permet d'ouvrir des paquetages à la réflexion
  - La permission peut être restreinte à certains modules avec le mot clef "to"

```
module monmodule {
    requires transitive java.logging;
    exports fr.cenotelie.monmodule;
    uses java.util.LinkedList;
    provides fr.cenotelie.monmodule.MyInterface
                    with fr.cenotelie.monmodule.MyInterfaceImpl;
    opens fr.cenotelie.monmodule to java.logging;
}
```

Version 9: Modularisation - JLink

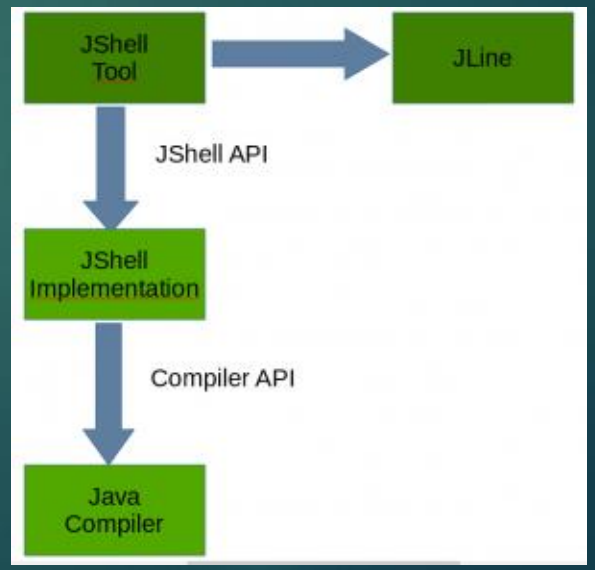
- JLink est un outil permettant de produire une "JVM light" n'embarquant que les modules utilisés par l'application
- Fusionne l'ensemble des modules utilisés en une archive exécutable
- L'image produite est auto-suffisante et peut être déployée sur des postes clients sans nécessiter l'installation d'une quelconque JVM
- Possibilité de créer des images pour différentes plateformes cibles

Version 9: Modularisation - JLink

- Création d'une image
  - $ jlink –module-path . --add-modules fr.akfc.monmodule --output monimage
  - Lancement de l'image: $ monimage/bin/java en précisant la classe "main"
- Création d'une image en précisant la classe "main"
  - $ jlink –module-path . --add-modules fr.akfc.monmodule --output monimage –launcher start=fr.akfc.monmodule/fr.akfc.Main
  - Lancement de l'image: $ monimage/bin/java -m fr.akfc.monmodule/fr.akfc.Main

Java 9: JShell (KULLA)

- REPL: Read Evaluation Print Loop
  - Interpréteur de code en ligne
  - Pour apprendre ou évaluer
- Chaque ligne est encapsulée dans une classe anonyme avant d'être exécutée
  - Les imports sont placés en haut de cette classe
  - Les méthodes, les variables et les classes en deviennent des membres statiques
- Lancé en ligne de commande
  - $ jshell
- JShell interpréte deux types d'expression
  - Les commandes JShell, qui commencent pas un "/"
  - Du code java



Java 9: JShell

- Pour faciliter l'exécution des commandes, un certain nombre de librairies sont automatiquement chargées
  - La commande "/list -start" permet de les lister
  - La commande "/set feedback verbose" permet d'avoir plus d'information sur les opérations exécutées
  - La commande "/vars" permet de lister toutes les variables
  - La commande "/types" permet de lister l'ensemble des types
  - La commande "/imports" permet de lister l'ensemble des imports
- La déclaration de classes, de variables et de méthodes se fait comme dans un fichier java
- Les imports se font comme dans un fichier java

Java 9: Ahead Of Time Compilation

- Le code généré par le compilateur java (javac) est portable
  - La compilation est de ce fait finalisée à l'exécution (JIT)
  - Coût sur la performance et la consommation
  - Prohibé dans certaines situations (iOS)

- Pour toutes ces raisons, java fournit un compilateur qui évite ces écueils - jaotc
  - Génération de librairies natives
  - Perte de la portabilité
  - `$ jaotc --output ../libSimpleSample.so -J-cp -J./bin SimpleSample`
  - `$ java -XX:+UnlockExperimentalVMOptions -XX:AOTLibrary=./libSimpleSample.so SimpleSample`

Java 9: Gargabe Collector G1

- Le collecteur Garbage-First (G1) est un collecteur de déchets de type serveur, destiné aux machines multiprocesseurs dotées de grandes mémoires.
- Il atteint les objectifs de temps de pause du ramasseur de déchets (GC) avec une forte probabilité, tout en obtenant un débit élevé. Il est conçu pour les applications qui :
  - Peuvent fonctionner simultanément avec des threads d'applications comme le collecteur CMS.
  - Compactent l'espace libre sans temps de pause prolongés induits par le GC.
  - Ont besoin de durées de pause GC plus prévisibles.
  - Ne veulent pas sacrifier beaucoup de performances en termes de débit.
  - Ne pas avoir besoin d'un tas Java beaucoup plus grand.
- G1 est prévu comme le remplacement à long terme du collecteur de balayage de marques concurrentes (CMS).
- Déjà présent à partir de la version 7, mais arrivé à maturité dans la version 9
- https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html

Java 10, 11 et 12 - Améliorations notables

- Les améliorations portent essentiellement sur la JVM
  - Inférence de type des variables locales
  - Concurrence
  - Unicode
  - Sécurité (certificats)
  - Cryptographie
  - Switch dans les expressions

```
int dayNumber = switch (day) {
  case "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY" -> 1;
  case "SATURDAY", "SUNDAY" -> 2;
  default -> throw new IllegalArgumentException("Invalid day: " + day);
};
```

- Manipulation des chaînes de caractères et des fichiers simplifiée
- Possibilité d'exécuter des fichiers java sans passer par la phase de compilation (shebang)

Java 10, 11 et 12 Améliorations notables

- Utilisation du mot-clef **var**
  - Repose sur l'inférence de type des variables locales pour alléger le code
  - Utilisé dans les lambdas pour faciliter les annotations
  - À utiliser avec précaution
- Client Http pour faire des requêtes sur le web
- JavaFX avec la version 11
  - Conçu pour remplacer Swing et Awt afin de prendre en compte les nouveaux usages

Mot-clef var - Précautions

- Utilisation uniquement pour les variables locales (pas pour les attributs ou paramètres de méthodes)
- Initialisation obligatoire (pour l'inférence)
- Garder un nommage explicite (pour compenser l'absence de type)
- Ne fonctionne pas avec des lambdas
- Inférence pas toujours évidente

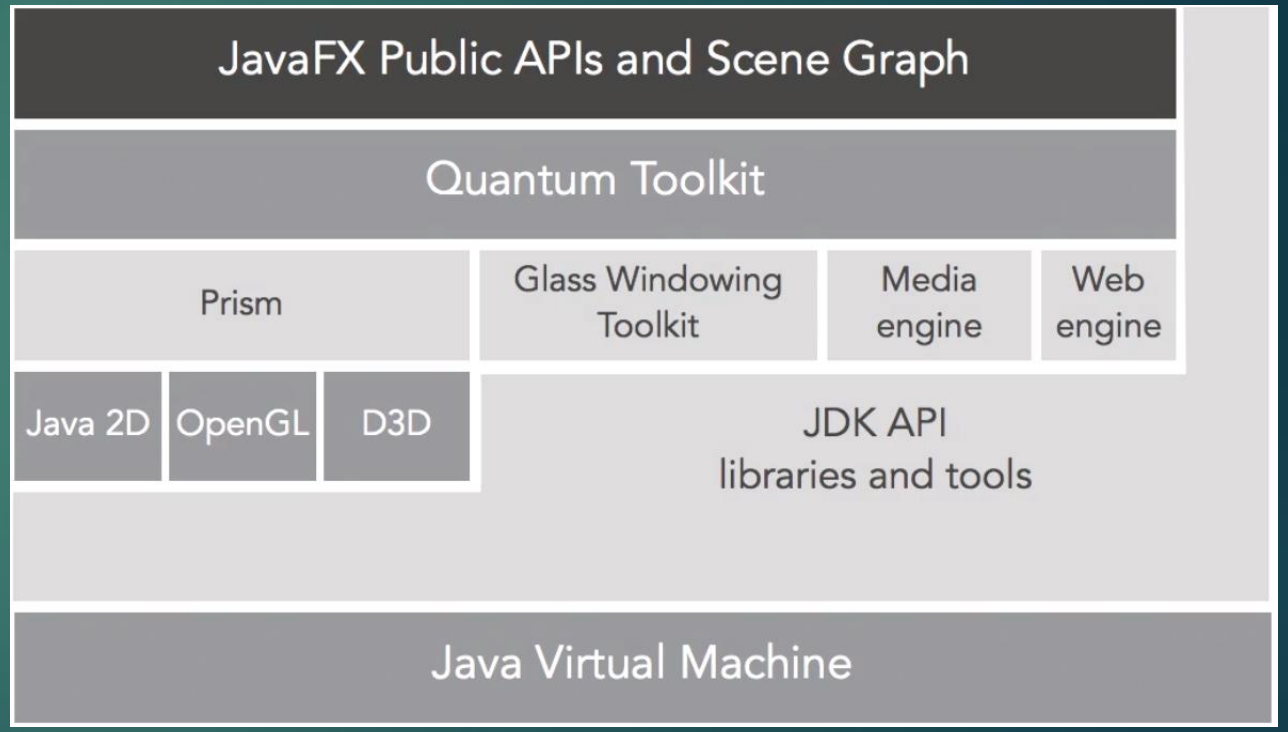
Compact numbers

- Avec la version 12, il est possible de représenter les nombres dans un format compact
  - Par exemple: 1000000 → 1M
- La représentation compact dépend évidemment des spécificités locales
- Possibilité de créer ses propres formats compacts

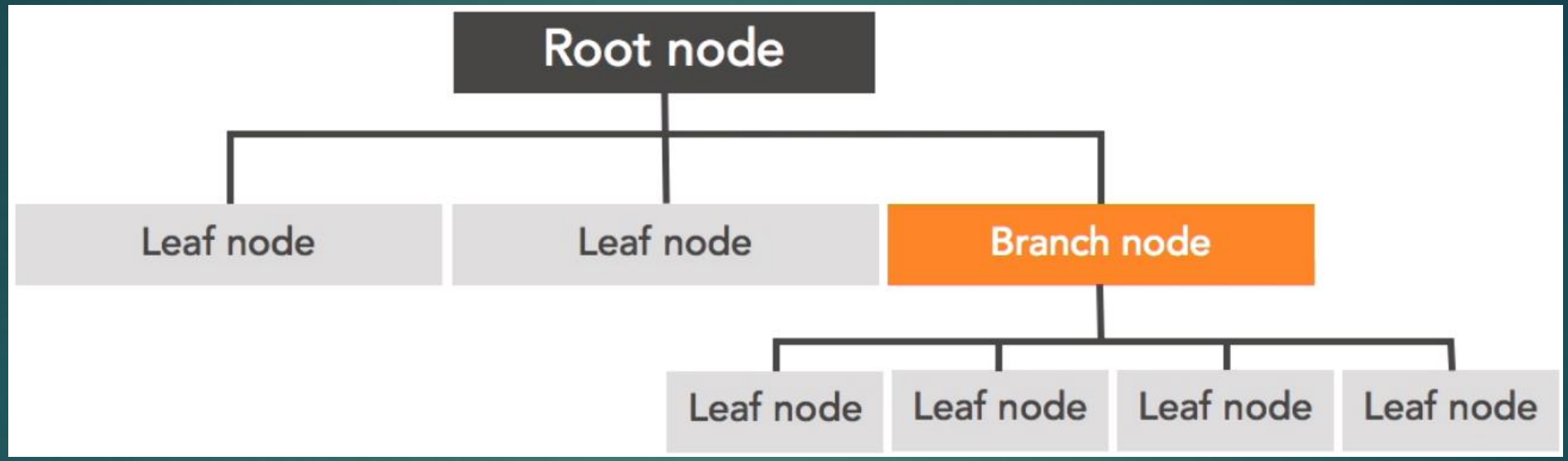
```
NumberFormat fmt = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.LONG);
fmt.setMinimumFractionDigits(3);
System.out.println(fmt.format(1_200_000));
System.out.println(fmt.parse("10 thousand"));
```

JavaFX (OpenJFX)

- API pour la conception de GUI
- Basé sur le pattern MVC
- Comprend une partie déclarative (FXML + CSS) et une partie impérative
  - Séparation de la présentation et de la logique applicative
- Compatibilité avec Swing
- Utilise au mieux les performances du matériel

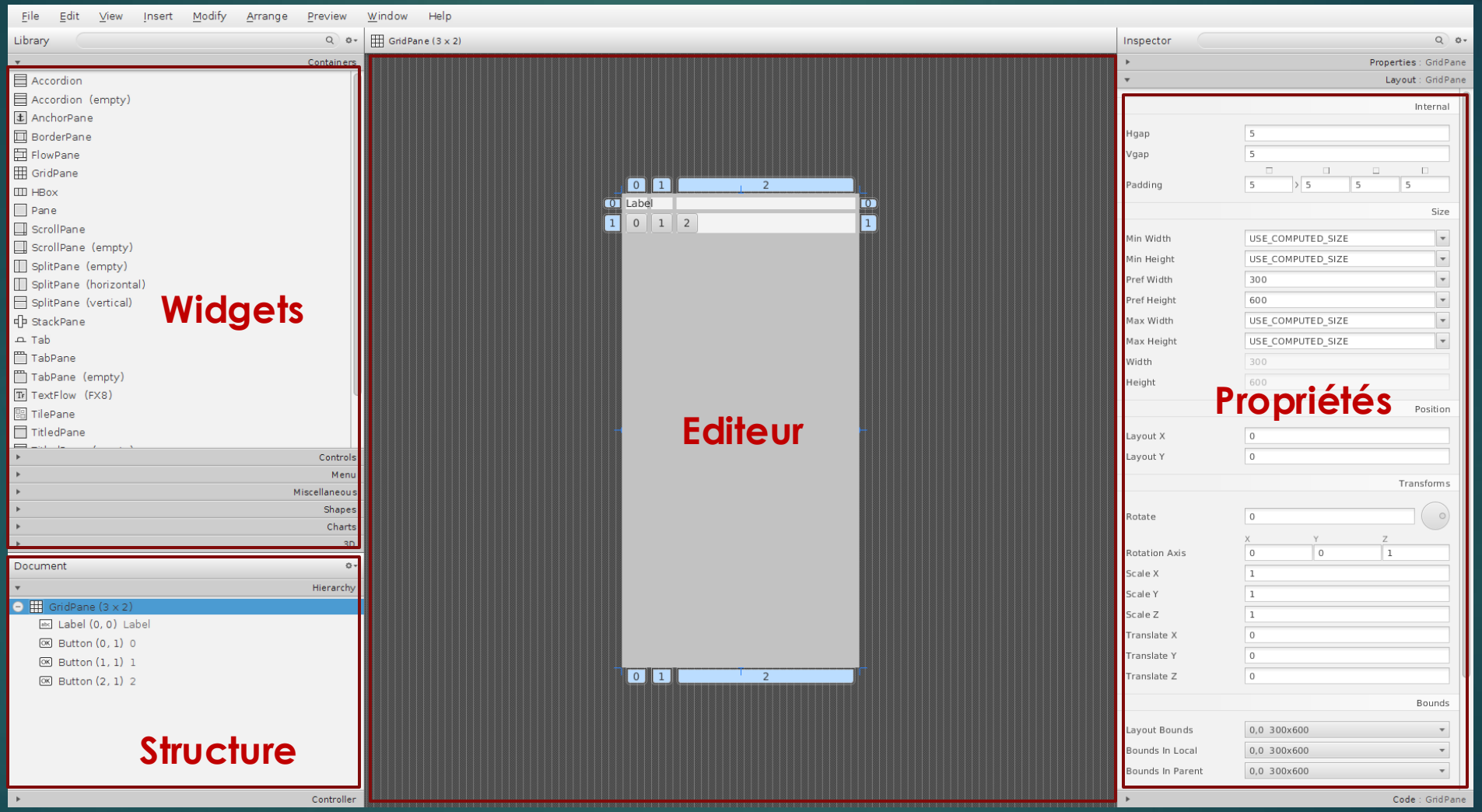


OpenJFX: Scene Graph



- Une application graphique est structurée en arbre
  - Chaque nœud a un identifiant
  - Avec l'avantage de mieux gérer les effets, transformation, événements ou états de l'application

JavaFx Scene Builder



- https://www.oracle.com/java/technologies/javafxscenebuilder-1x-archive-downloads.html

OpenJFX: Liens entre parties déclarative et impérative

Partie déclarative

```
<GridPane xmlns="http://javafx.com/javafx/8"
          xmlns:fx="http://javafx.com/fxml/1"
          fx:controller="fr.cenotelie.training.gui.CalcController">
  <stylesheets>
    <URL value="@calc.css" />
  </stylesheets>
  <padding>
    <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
  </padding>
  <fx:define>
    <Double fx:id="width" fx:value="80.0"/>
    <Double fx:id="height" fx:value="80.0"/>
  </fx:define>
  <children>
    <Label fx:id="display" prefHeight="$height" GridPane.columnSpan="4" styleClass="disp" minWidth="310"/>
    <Button onAction="#onNumberClick" prefHeight="$height" prefWidth="$width" text="1" GridPane.rowIndex="1" />
  </children>
</GridPane>
```

Partie impérative

```
public class CalcController {

    private StringBuffer number = new StringBuffer();
    @FXML
    private Double width;
    @FXML
    private Double height;
    @FXML
    private Label display;

    @FXML
    protected void onNumberClick(ActionEvent event) {
        Button button = (Button) event.getSource();
        number.append(button.getText());
        display.setText(number.toString());
    }
}
```

```
.disp {
    -fx-background-color: white;
    -fx-border-color: black;
}
```

OpenJFX: Exercice

- Ouvrir le projet openjfx-training
- Implanter toutes les opérations de la calculatrice
- Donner une implantation pour chacune de ces touches
- Faire du projet un module exécutable
  - $ mvn javafx:jlink

Java 13

- Text Block
- Améliorations JVM
  - Sockets
  - Gestion de la mémoire

```
String html = """
              <HTML lang="en">
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

Exercice : Client Http et text block

- Compléter la classe Beatles (network)
- Récupérer le titre et l'année de parution de tous les albums des Beatles
- Enregistrer les résultats dans un fichier XML
- Générer une page HTML contenant un tableau
  - Une ligne par album
  - Une colonne par propriété
- Fichier à modifier: Beatles.java
  - Utiliser les API de haut niveau pour la manipulation de fichiers et de chaînes de caractères

```
HttpClient client = HttpClient.newHttpClient();
var url = new URL("https://www.google.fr");
var request = HttpRequest.newBuilder().GET().uri(url.toURI()).build();
var response = client.send(request, HttpResponse.BodyHandlers.ofString());
System.out.println(response.body() + " " + response.statusCode());
```

Java 14 - Record

- Les records ont été introduits dans Java 14 (en preview) et finalisés en Java 16.
- Ils offrent un moyen simple et concis de définir des classes immuables destinées à stocker des données, sans avoir à écrire le code répétitif
des accesseurs, equals, hashCode et toString.
- Avantages:
  - **Réduire le code boilerplate** : Les records génèrent automatiquement les méthodes d'accès, equals, hashCode et toString.
  - **Clarté et lisibilité** : Leur syntaxe exprime clairement l'intention : une simple agrégation de données.
  - **Immutabilité** : Par défaut, les champs d'un record sont final et ne peuvent être modifiés après la création de l'instance.
  - **Sécurité et robustesse** : Moins de risques d'erreurs liées à la gestion manuelle des méthodes de base.

```
record Point(int x, int y) { }
Point p = new Point(3,4);
System.out.println( p.x() );
```

Java 14

- Pattern Matching pour l'instruction "instanceof"

```
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println( s.length() );
}

Vs

if (obj instanceof String s) {
    System.out.println( s.length() );
}
```

- Option ShowCodeDetailsInExceptionMessages pour la JVM
  - Permet d'avoir des détails plus pertinents dans les exceptions
- Switch: plusieurs cas sur une même ligne
  - case 1, 3, 5 -> …;
  - La notation fléchée sort de la fonction (return) et évite l'utilisation du mot clef break à chaque cas

```
boolean b = switch(v) {
   case 3, 5, 7 -> true;
   default -> false;
}
```

Java 15 – Classes scellées

- Restreignent les classes pouvant les étendre
- Utiles pour contrôler les hiérarchies de classes / interface de manière sécurisée
- Peut être vu comme le modificateur "final" avec quelques généralisations autorisées, ou encore comme une énumération de classes filles
- L'ensemble constitué par l'union des classes filles constituent l'ensemble exhaustif de tous les sous-types possibles (partitions)

```
sealed interface Celestial permits Planet, Star, Comet {
    record Planet(String name, double mass, double radius)
        implements Celestial {}
    record Star(String name, double mass, double temperature)
        implements Celestial {}
    record Comet(String name, double period, LocalDateTime lastSeen)
        implements Celestial {}
}
```

Java 16 & 17

- Améliorations des classes scellées et des records
- Améliorations au niveau de la JVM
  - Support Linux / Windows
  - Sockets
  - Outils de packaging
  - Pattern matching pour l'instruction "instanceof"

Java 18

- Jeu de caractères par défaut: UTF-8
- Simple Web Server
- Code Snippets dans la documentation

```
HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
server.createContext("/", new MyHandler());
server.setExecutor(null);
server.start();
```

```
/**
* @param a the first integer
* @param b the second integer
* @return the sum of {@code a} and {@code b}
* @example
* <pre>{@code
* int result = add(1, 2);
* System.out.println(result); // prints "3" *
}</pre>
*/
public static int add(int a, int b) { return a + b; }
```

- **Exercice**: Compléter la classe ArithmeticServer (network) afin d'implanter un service prenant en entrée une opération (add, mult, div, …) et deux opérandes et renvoyant le résultat de l'éxecution.

Java 19

- Structured Concurrency
  - Destiné à simplifier la programmation concurrente
- Switch pattern matching
- Foreign function & Memory API
  - Permet d'intéropérer avec du code hors java – En remplacement de JNI

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> { ... });
```

```
try (MemorySegment segment = MemorySegment.allocateNative(1024)) {
   CLinker linker = CLinker.getInstance();
   FunctionDescriptor descriptor =
                FunctionDescriptor.of(CLinker.C_LONG, CLinker.C_POINTER);
   MethodHandle function = linker.downcallHandle(
   LibraryLoader.loadLibrary("mylib"), "myfunction", descriptor);
   long result = (long) function.invokeExact(segment.baseAddress(), 42L);
   System.out.println(result);
}
```

Foreign Function & Memory API

- Les Foreign Function & Memory API (API d'appel de fonctions étrangères et de gestion de mémoire native), ont été introduites progressivement dans les versions récentes de Java (à partir de Java 16 en preview, et stabilisées en Java 21).

- Elles apportent une modernisation et une simplification majeure à l'intégration de code natif par rapport à l'approche traditionnelle JNI.

- Les Foreign Function & Memory API permettent d'intégrer du code natif en Java de façon plus simple, plus sûre et plus moderne, en évitant la complexité et les risques de JNI. Elles sont particulièrement adaptées aux applications nécessitant des performances natives ou l'interopérabilité avec des bibliothèques C/C++ existantes

Foreign Function& Memory API - Apports

- Abstraction de la gestion de la mémoire native
  - L'API permet d'allouer, d'accéder et de libérer de la mémoire native de façon sûre, sans avoir à manipuler directement les pointeurs ou à écrire du code natif pour la gestion mémoire.
  - Les objets MemorySegment et MemoryAddress offrent une interface Java pour manipuler des zones mémoire natives, avec vérification des accès et gestion automatique de la durée de vie si nécessaire.

- Appel direct de fonctions natives
  - Il est possible d'appeler des fonctions C ou d'autres langages natifs directement depuis Java, sans avoir à écrire de code JNI (pas besoin de générer de fichiers .h ni d'implémenter de fonctions natives dans une classe Java).
  - L'API fournit des outils pour construire des descripteurs de fonctions (FunctionDescriptor) et les appeler via des MethodHandle.

Foreign Function & Memory API - Apports

- Gestion des signatures de fonctions et des conversions de types
  - L'API gère automatiquement la conversion entre types Java et types natifs (entiers, flottants, pointeurs, structures, etc.).
  - Plus besoin de gérer manuellement les conversions ni les appels via JNIEnv.

- Sécurité et Robustesse
  - Contrôle des accès mémoire: Les accès à la mémoire native sont vérifiés à l'exécution, ce qui limite les risques de corruption mémoire ou de fuites.
  - Gestion automatique de la durée de vie: Les segments mémoire peuvent être liés à la durée de vie d'objets Java (via le garbage collector ou le try-with-resources), ce qui réduit les risques de fuites de mémoire.
  - Moins de code natif à écrire: L'essentiel du code peut rester en Java, ce qui réduit la surface d'erreurs potentielles et facilite la maintenance.

Foreign Function & Memory API vs JNI

- **Réduction du code natif** : plus besoin d'écrire du code C/JNI pour chaque fonction appelée.
- **Gestion mémoire simplifiée** : allocation, accès et libération sécurisés depuis Java.
- **Appel direct de fonctions** : invocation de fonctions natives sans boilerplate.
- **Sécurité accrue** : vérification des accès mémoire et gestion automatique de la durée de vie.
- **Meilleure intégration avec le reste du code Java** : utilisation de types Java modernes (records, streams, etc.) pour manipuler des données natives.

Java 20

- Structured concurrency: améliorations
- Virtual threads
  - Threads légers pouvant être utilisés pour améliorer les performances, ils sont similaires aux threads ordinaires, en plus efficaces
  - Géré et planifié par une machine virtuelle (JVM), alors qu'un thread normal est géré et planifié par le système d'exploitation.
  - Multiplexé sur un ou plusieurs threads du système d'exploitation, ils peuvent ainsi être créés et détruits beaucoup plus rapidement que les threads ordinaires
  - Utiles lorsque de nombreuses tâches concurrentes passent la majeure partie de leur temps d'exécution en attente
- Vector API: Traite les opérations sur les tableaux de manière plus efficace

```
ExecutorService executor = Executors.newVirtualThreadExecutor();
```

Java 21à 24

- Templates
- Pattern matching dans les switch
- Amélioration des librairies
  - Threads virtuels
  - API vectoriel
  - StringBuilder::repeat
- Amélioration des performances
- Amélioration des outils (Runtime.exec, ProcessBuilder)

```
int age = 30;
String info = STR."J'ai \{age} ans.";
System.out.println(info);
```

```
Object obj = ...;
switch(obj) {
    case Integer i when i > 0 -> System.out.println("Entier positif");
    case String s when s.length() > 10 -> System.out.println("Longue chaîne");
    default -> System.out.println("Autre");
}
```

Templates

- Les templates sont gérés via des processeurs de templates (template processors), qui transforment un template (une chaîne avec des interpolations) en une chaîne ou un autre objet.
- Les principaux processeurs proposés sont :
  - **STR** : Le processeur standard pour l'interpolation simple.
  - **FMT** : Le processeur pour l'interpolation avec formatage, similaire à String.format().
  - **RAW** : Le processeur qui ne fait aucune interpolation, il retourne le template tel quel (utile pour manipuler ou analyser le template avant traitement).

```
int age = 30;
double height = 1.75;
String info = FMT."Nom : \{name}, Âge : %d ans, Taille : %.2f m".formatted(age, height);
```

Templates : Custom Processors

- Bien que Java propose des processeurs standards comme STR, FMT et RAW, il est possible d'écrire ses propres processeurs pour des besoins spécifiques.

- Un processeur personnalisé doit implémenter l'interface StringTemplate.Processor<R, E extends Exception>, où :
  - R est le type de résultat (ici, String).
  - E est le type d'exception éventuellement levée (ici, IllegalArgumentException).

- Cf. Classes StringTemplateExamples et PhoneProcessor (misc)

Vector API

- Les applications modernes manipulent souvent de grands volumes de données numériques (calculs scientifiques, traitement d'images, intelligence artificielle, etc.).

- Traditionnellement, Java traite ces données via des boucles sur des tableaux ou des collections, ce qui limite l'exploitation des capacités matérielles avancées des processeurs :
  - Instructions SIMD (Single Instruction, Multiple Data) : Les processeurs modernes proposent des instructions permettant d'appliquer une même opération sur plusieurs valeurs simultanément (vectorisation).
  - Sous-utilisation du matériel : Le code Java classique n'exploite pas ces instructions, ce qui réduit la performance des applications gourmandes en calculs.

Vector API - Motivations

- **Exploiter les instructions SIMD** du processeur sans avoir à écrire de code natif (C, assembleur).
- **Améliorer la performance** des applications nécessitant des calculs intensifs sur des tableaux numériques (comme Matlab ou Numpy).
- **Proposer une API portable** : le code reste lisible, maintenable et fonctionne sur toutes les plateformes supportant Java, tout en tirant parti des optimisations matérielles disponibles.

Vector API - Principes

- **Abstraction des opérations vectorielles** : La Vector API permet de manipuler des vecteurs (tableaux de valeurs numériques) comme des entités uniques, sur lesquelles on applique des opérations (addition, multiplication, etc.) de façon optimisée.

- **Types vectoriels** : Les vecteurs sont typés (par exemple, int, float, double) et leur taille dépend du matériel (128, 256, 512 bits, etc.).

- **Instructions SIMD transparentes** : Le compilateur Java et la JVM traduisent les opérations de l'API en instructions SIMD si le matériel et le runtime le permettent.

- **Portabilité** : Si le matériel ne supporte pas les instructions SIMD, la JVM utilise une implémentation logicielle, garantissant que le code fonctionne partout.

Vector API - Exemple

```
private int[] a = new int[1000];
private int[] b = new int[1000];
private int[] c = new int[1000];
```

```
for (int i = 0; i < a.length; i++) {
    c[i] = a[i] * b[i];
}
```

Multiplication classique

```
VectorSpecies<Integer> species =
IntVector.SPECIES_PREFERRED;
IntVector va = IntVector.fromArray(species, a, 0);
IntVector vb = IntVector.fromArray(species, b, 0);
IntVector vc = va.mul(vb);
vc.intoArray(c, 0);
```

Vector API

Vector API – Principales limitations

- L'API Vector est **conçue pour exploiter les instructions SIMD du CPU, pas du GPU**. Pour utiliser la puissance du GPU, il faut passer par d'autres solutions (OpenCL, CUDA, frameworks comme TornadoVM ou Aparapi via JNI).

- **Exclusivité CPU** : Les calculs restent limités au processeur, ce qui peut être un frein pour certains cas d'usage nécessitant une accélération massive.

- **Support variable selon l'architecture** : Les performances dépendent fortement du matériel sous-jacent (support des instructions SIMD, taille des registres vectoriels, etc.).

- **Fallback logiciel** : Si le matériel ne supporte pas les instructions SIMD, la JVM utilise une implémentation logicielle, ce qui réduit fortement la performance.

Vector API : Principales limitations

- **Courbe d'apprentissage** : L'API Vector introduit de nouveaux concepts (Species, Masks, Shuffles, etc.) et une syntaxe spécifique, ce qui la rend plus complexe à utiliser que les boucles ou l'API Stream classique.
- **Abstraction limitée** : Les développeurs doivent gérer explicitement la taille des vecteurs, les types, et parfois l'alignement mémoire.
- **Types supportés** : L'API Vector fonctionne principalement avec les types primitifs (int, float, double, long). Elle ne permet pas de vectoriser des opérations sur des objets ou des structures complexes.
- **Pas de support générique** : Il n'est pas possible d'utiliser des génériques ou des types personnalisés.

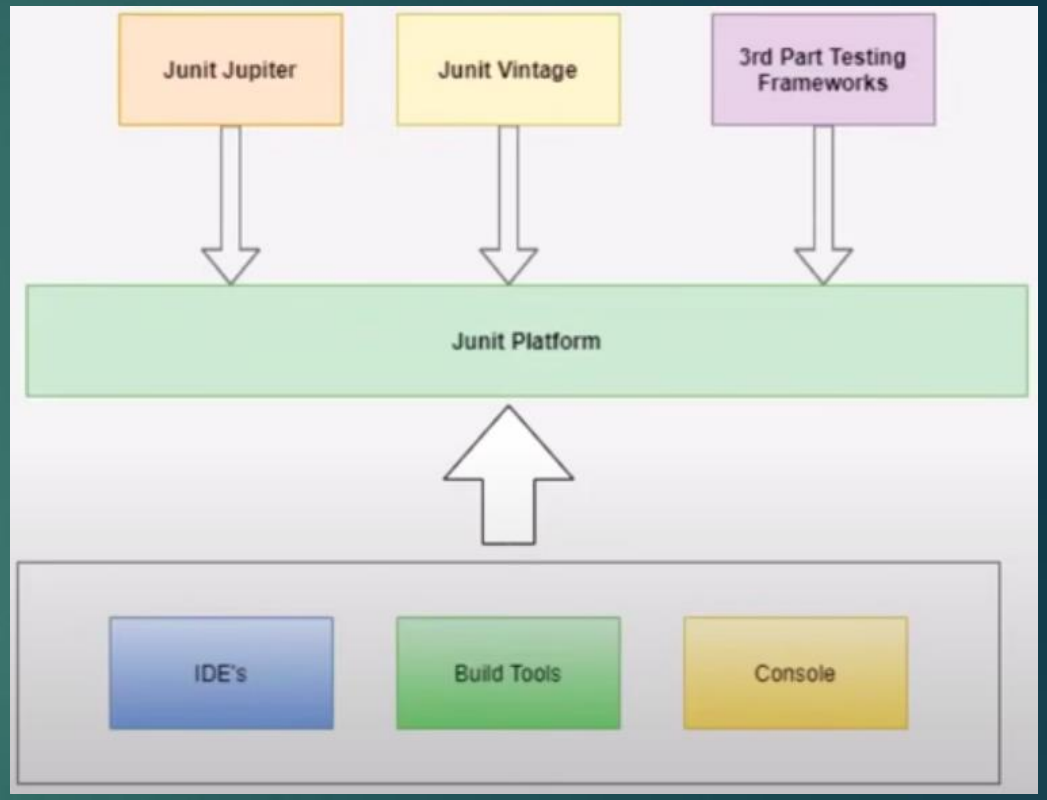
Test

Présentation

- Les tests sont un aspect crucial du développement de logiciels qui garantit la qualité et la fiabilité du code.
- Dans l'écosystème Java, JUnit et Mockito sont deux des cadres les plus populaires utilisés pour écrire et exécuter des tests.
  - JUnit est principalement utilisé pour les tests unitaires, qui consistent à tester des composants individuels de l'application de manière isolée.
  - Mockito, quant à lui, est utilisé pour simuler l'environnement des objets testés, ce qui permet de simuler le comportement d'objets complexes de manière contrôlée.
- Cette combinaison permet aux développeurs d'écrire des tests complets qui couvrent différents scénarios et interactions entre les objets.

JUnit

- Cadre pour spécifier et exécuter des tests reproductibles
- Permet de tester des classes ou des méthodes (SUT: Subject Under Test)
- Constitué de 4 composants:
  - Plateform: Interface pour lancer des tests depuis un client
  - Jupiter: Fournit les constructions pour spécifier des tests et des extensions
  - Vintage: Moteur de test supportant une compatibilité ascendante
  - 3rd Party: Permet de créer son propre cadre de tests en réutilisant les briques de JUnit



JUnit

- Nouveauté de la version 5:
  - les tests imbriqués
  - les tests dynamiques
  - les tests paramétrés qui offrent différentes sources de données
  - un nouveau modèle d'extension
  - l'injection d'instances en paramètres des méthodes de tests
- Cycle de vie
  - BeforeAll: appelée à l'instanciation de chaque classe de test
  - BeforeEach: appelée avant l'exécution de chaque test
  - Test: exécution du test
  - AfterEach: appelée après l'exécution de chaque test
  - AfterAll: appelée quand la classe est déchargée

JUnit

Annotation	Rôle
@Test	La méthode annotée est un cas de test. Contrairement à l'annotation @Test de JUnit, celle-ci ne possède aucun attribut
@ParameterizedTest	La méthode annotée est un cas de test paramétré
@RepeatedTest	La méthode annotée est un cas de test répété
@TestFactory	La méthode annotée est une fabrique pour des tests dynamiques
@TestInstance	Configurer le cycle de vie des instances de tests
@TestTemplate	La méthode est un modèle pour des cas de tests à exécution multiple
@DisplayName	Définir un libellé pour la classe ou la méthode de test annotée
@BeforeEach	La méthode annotée sera invoquée avant l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @Before de JUnit 4
@AfterEach	La méthode annotée sera invoquée après l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @After de JUnit 4

[Source: https://www.jmdoudoux.fr/java/dej/chap-junit5.htm]

JUnit

Annotation	Rôle
@BeforeAll	La méthode annotée sera invoquée avant l'exécution de la première méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @BeforeClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class
@AfterAll	La méthode annotée sera invoquée après l'exécution de toutes les méthodes de la classe annotées avec @Test, @RepeatedTest, @ParameterizedTest et @Testfactory. Cette annotation est équivalente à @AfterClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class.
@Nested	Indiquer que la classe annotée correspond à un test imbriqué
@Tag	Définir une balise sur une classe ou une méthode qui permettra de filtrer les tests exécutés. Cette annotation est équivalente aux Categories de JUnit 4 ou aux groups de TestNG
@Disabled	Désactiver les tests de la classe ou la méthode annotée. Cette annotation est similaire à @Ignore de JUnit 4
@ExtendWith	Enregistrer une extension

[Source: https://www.jmdoudoux.fr/java/dej/chap-junit5.htm]

TDD – JUnit Assertions

Egalité	Nullité	Exceptions
assertEquals()	assertNull()	assertThrows()
assertNotEquals()	assertNotNull()	
assertTrue()		
assertFalse()		
assertSame()		
assertNotSame()		

Et bien d'autres méthodes encore...
https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

TDD – JUnit Exemples

```
class CalcTest {

    private Calc calc;

    @BeforeAll
    public static void setupAll(){
        System.out.println("Launching tests for Calc");
    }

    @BeforeEach
    public void setup() {
        calc = new Calc();
    }

    @Test
    @DisplayName("Test Case for sqrt method")
    void sqrt() {
        List<Double> entries = Arrays.asList(1.0, 2.0);
        List<Double> actual =

                entries.stream().map(calc::sqrt).collect(toList());
        List<Double> expected =

                 Arrays.asList(0.5, 1.0);
        assertIterableEquals(actual, expected);
    }
}
```

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class CalcTest {

    private static Calc calc;

    @BeforeAll
    public static void setupAll(){
        System.out.println("Launching tests for Calc");
        calc = new Calc();
    }

    //Intéressant sur des valeurs générées aléatoirement
    @RepeatedTest(value=5)
    @DisplayName("Test Case for sqrt method")
    void sqrt() {
        assertEquals(calc.sqrt(2.0), 1.0);
    }
}
```

TDD – JUnit Test de performance

```
@Test
void checkTimeout() {
   Assertions.assertTimeout(
     Duration.ofMillis(200), () -> { return ""; }
   );
}
```

Exercices:

- Écrire des tests pour la classe Calculator (misc)
- Écrire des tests pour la classe Array (misc)
- Créer deux tests pour la méthode Fibo::getEvenNumber(int num) (misc)
  - Le premier pour s'assurer que la méthode renvoie le bon tableau pour un num donné
  - Le second pour s'assurer que la méthode met moins de 2 ms pour s'exécuter

Mockito

- Mockito est un cadre de simulation qui vous permet de créer et de configurer des objets simulés.
  - Il est particulièrement utile lorsque vous devez tester un composant qui interagit avec d'autres composants complexes.

- En utilisant des objets fictifs, vous pouvez simuler le comportement de ces dépendances sans avoir à les instancier.
  - Cela est particulièrement utile dans les tests unitaires, lorsque vous souhaitez isoler le composant testé.

- JUnit et Mockito fonctionnent bien ensemble, ce qui vous permet d'écrire des tests complets qui couvrent à la fois la logique des composants individuels et leurs interactions avec les dépendances.
  - Mockito s'intègre parfaitement à JUnit et fournit des annotations telles que @Mock pour créer des objets fictifs et @InjectMocks pour les injecter automatiquement dans la classe testée.

Mockito - Exemple

```java
public class ServiceTest {

    @Test
    public void testServiceMethod() {
        // Create a mock object of the repository
        Repository mockRepository = mock(Repository.class);

        // Configure the mock to return a specific value when a method is called
        when(mockRepository.getData()).thenReturn("Expected Result");

        // Create an instance of the service, injecting the mock repository
        Service service = new Service(mockRepository);

        // Call the method under test
        String result = service.performAction();

        // Verify the result
        assertEquals("Expected Result", result);

        // Verify interactions with the mock
        verify(mockRepository).getData();
    }
}
```

Mockito: Exercice

- Compléter la classe Service Test
  - Tester l'appel à la méthode performAction
  - Tester l'appel à la méthod square sur une série de valeur
- Pensez à injecter un ojbet fictif de type Repository afin de mimer l'environnement de la classe Service
- **Question:** Quels sont les avantages du mock ?

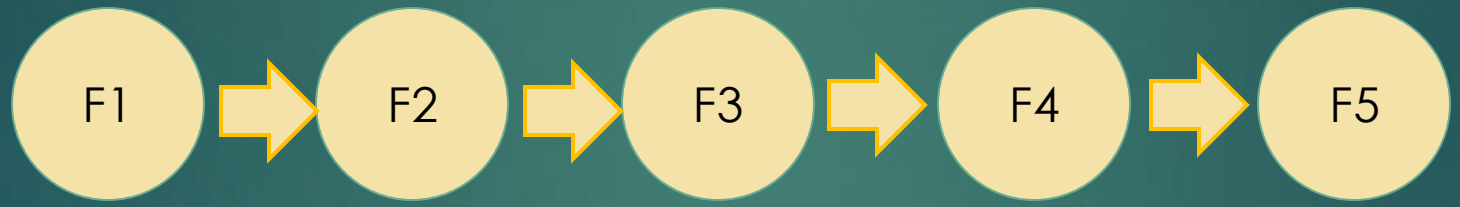
Concurrence

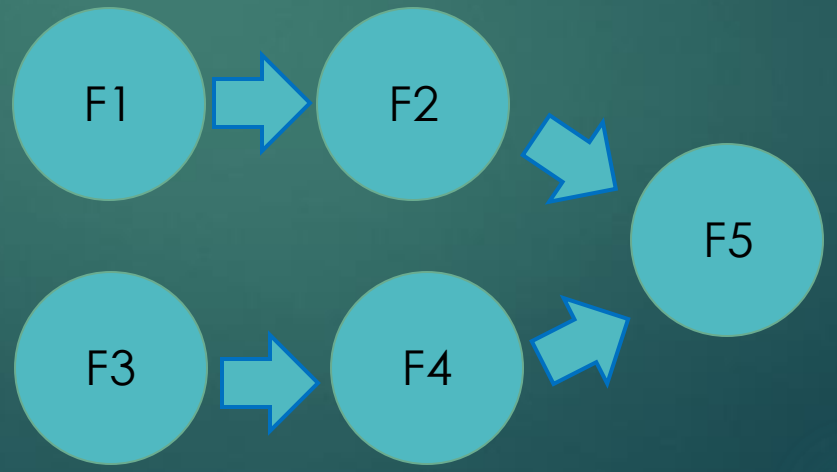
Concurrence

- Concepts fondamentaux: Process (processus lourd) et Thread (processus léger)
- En java, l'exécution de la JVM est représenté par un processus
  - De fait, on ne manipule que des threads en java, qui nécessitent des mécanismes de communication / synchronisation plus simples
  - Les threads partagent le même espace mémoire, celui de leur processus
  - Depuis la version 9, il est possible d'avoir plus d'information sur le processus qui exécute la JVM (pid, ppid, état, etc.)
- Parallèle ou concurrent ?
  - Concurrence: parallélisme potentielle, i.e. possibilité d'exécuter du code en parallèle
  - Parallèle: Relatif à l'architecture de la machine (SISD, SIMD, MISD, MIMD)
- Pourquoi paralléliser ?
  - Essentiellement pour des raisons de performances
  - Pour tirer pleinement des avantages offerts par la machine

Concurrence

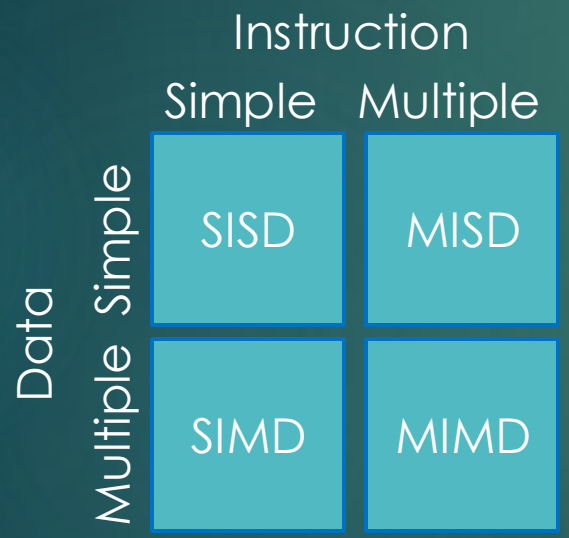
- Basé sur l'analyse du séquencement et des dépendances
- La difficulté vient de l'accès exclusif aux ressources partagées
  - Surtout quand ils sont exécutés sur des architectures parallèles

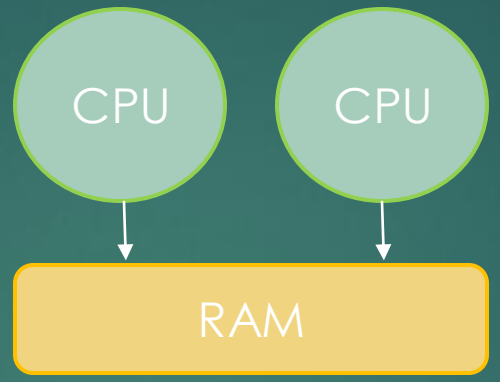




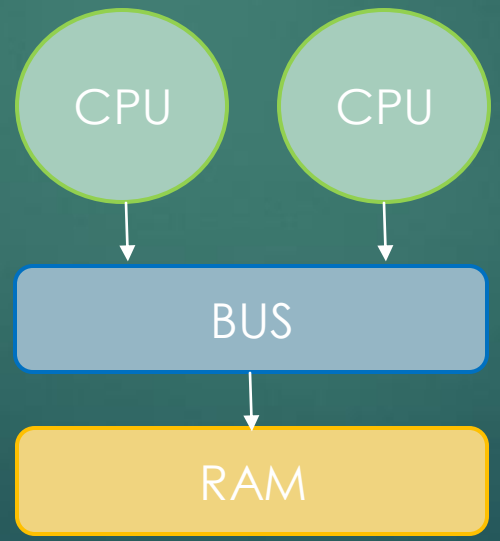
Parallélisme

- Taxinomie de Flynn





UMA: Uniform Memory Access

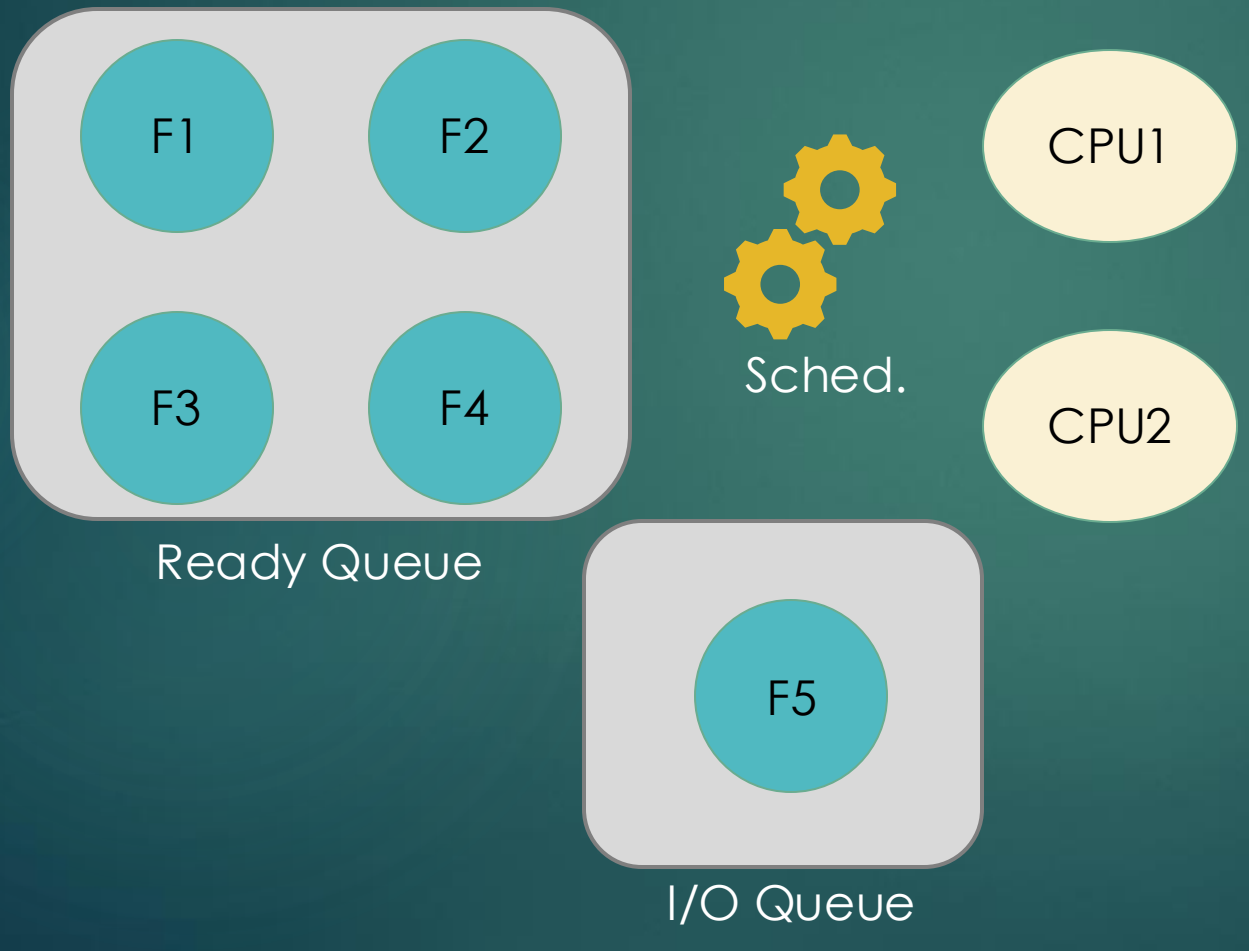


SMP: Symmetric Multiprocessing

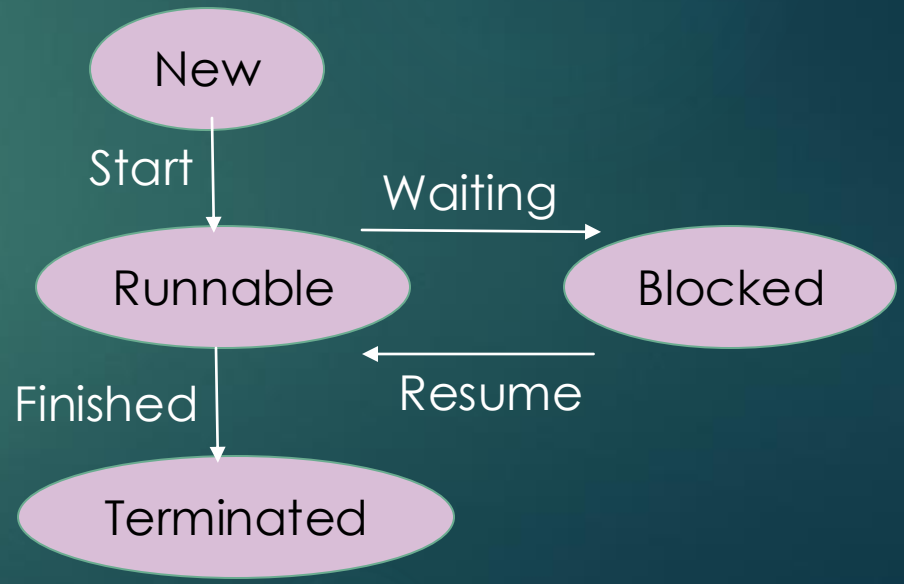
Problème de la mémoire partagée:
cohérence des caches
Dans le cas d'architectures distribuées (NUMA, Distributed Memory Access), la cohérence ne peut être assurée qu'au prix de gros efforts.

Concurrence

- Ordonnancement assuré par l'OS



- Différentes stratégies d'ordonnancement
  - Pour maximiser les débits
  - Pour maximiser l'égalité d'accès aux ressources
  - Pour minimiser les temps d'attente
  - Pour minimiser la latence



Etats d'un thread

Concurrence

- Lors de l'exécution d'un programme
  - La JVM exécute un thread principal (main)
  - Le thread principal peut exécuter d'autres threads (enfants)
  - Un thread peut exécuter d'autres threads (arborescence)
  - Un thread doit attendre que ses enfants soient terminés avant de se terminer (join)

- En plus des 4 états vus dans la diapositive précédente, java ajoute deux états qui sont des variants de l'état Blocked
  - Waiting – un thread attend qu'un autre thread ait fini une action
  - Timed_Waiting - Même chose que waiting mais dans un temps borné
  - La méthode statique Thread.currentThread() renvoie une référence vers le thread en cours d'exécution

Concurrence

- Un thread est caractérisé par
  - Un identifiant unique (long)
  - Un nom (généré si non spécifié - Thread-N)
  - Une priorité, de 1 à 10, de valeur 5 par défaut
- 2 manières de spécifier un thread
  - Implanter l'interface Runnable
    - Pratique si la classe doit étendre une autre classe
  - Etendre la classe Thread
    - Pratique si plusieurs threads partagent le même code
- Un thread peut être détaché (daemon)
  - Le parent qui l'a créé n'a pas attendre qu'il soit terminé avant de se terminer
    - setDaemon(true)
  - A utiliser avec attention car cela peut laisser des ressources dans un état incohérent

Exercice: Data race

- Ecrivez un thread qui incrémente un compteur statique sur 10 tours de boucle
- Instanciez deux threads et affichez l'état du compteur à la fin de l'exécution
  - Que constatez-vous ?
- Modifier le nombre de boucle à 100.000
  - Que constatez-vous ?
  - Quel est le problème ?
- Classe à modifier: DataRace

Mutex

- Dans les cas simples, on peut faire usage des types de données atomiques
  - Définis dans le paquetage java.util.concurrent.atomic
  - Accès automiquement protégé par des verrous
- On peut utiliser le mot clef **synchronized** sur une méthode ou un bloc pour déclarer une section critique
  - Dans le cas d'une méthode, c'est
    - la classe qui sert de verrou dans le cas d'une méthode statique
    - L'objet si la méthode n'est pas statique (à utiliser avec attention)
  - Dans le cas d'un bloc, il faut préciser sur quel objet poser le verrou
    - Sur un objet et non un type primitif
- Lock ou synchronized ?
  - Synchronized est sûr et facile à mettre en œuvre
  - Lock apporte beaucoup plus de flexibilité et améliore les performances

Exercice – Data race

- Apportez une première modification en utilisant un type de donnée atomique.

- Apportez une autre modification en utilisant le mot-clef synchronized
  - Sur une méthode
  - Sur un block
    - En se synchronisant sur l'objet courant **this**
    - En se synchronisant sur le compteur lui-même
    - Dans les deux cas, que constatez-vous ?
    - Que faut-il modifier dans le second cas ?

Mutex

- Reentrant lock (ReentrantLock)
  - Peut être verrouillé plusieurs fois
    - Doit être alors déverrouillé autant de fois qu'il a été verrouillé
    - Utile notamment dans le cas des fonctions récursives
  - Fournit une méthode non-bloquante du lock
    - Permet à un thread d'éviter une attente active avant l'obtention du verrou
    - L'opération tryLock() retourne vrai si le thread a pu prendre le verrou
      - Elle renvoie faux sinon et le thread continue son exécution, rendant le programme plus performant

```
counter = 0

fa() {
  lock();
  …
  unlock();
}

fb() {
  lock();
  …
  unlock();
}

f() {
   lock();
   fa();
    fb();
   unlock();
}
```

Exercice – Data Race

- Apportez une nouvelle modification utilisant un lock réentrant
  - Qu'observez-vous ?
  - Quel est le problème ?
  - Comment améliorer la situation ?
  - Comment évitez une attente active ?

Mutex - ReentrantReadWriteLock

- Reader-Writer Lock (ReentrantReadWriteLock) permet de poser des verrous:
  - En lecture partagée - readLock()
  - En écriture exclusive – writeLock()
  - Intéressant quand le nombre de lecteurs est largement supérieur au nombre d'écrivains
  - Ex: accès à une base de données
  - Mécanismes de synchronisation plus complexes, à utiliser avec parcimonie

Exercice – Data Race

- Modifiez le code de manière à avoir:
  - 8 threads ayant accès en lecture au compteur
  - 2 threads ayant accès en écriture au compteur
  - À chaque accès, les threads s'endorment pour un temps aléatoire variant de 0 à 2 secondes et incrémentent le compteur

Mutex

- Problème: Accès aux données partagées
  - Notamment dans le cas de la cohérence des caches
  - Très difficile à détecter et à debugger
  - Il faut s'en prémunir de manière active
    - Par un verrou (mutex) - Son acquisition est atomique
    - Attention toutefois à relâcher le verrou rapidement pour ne pas bloquer les autres threads
    - On utilise un objet de type Lock, défini dans le paquetage java.util.concurrent.locks
      - Reentrant Lock
      - Read/Write Lock
- L'utilisation des verrous peut générer différents problèmes
  - Deadlocks
  - Livelocks
  - Starvation

Mutex: Problèmes

- **Deadlock**
  - Interblocage mutuel
    - Ex: Le diner des philosophes
  - Un thread ayant acquis des verrous se termine avant d'avoir relâcher ces verrous
- **Starvation** (famine)
  - Un thread pouvant avoir une priorité moindre n'arrive jamais à acquérir les verrous
  - Un grand nombre de threads accède à des ressources limitées
- **Livelock**
  - Chaque thread, en contribuant à résoudre des problèmes de deadlock finit par bloquer les autres threads
  - Difficile à détecter et à débugger
- Solutions possibles:
  - Veiller à bien ordonnancer l'acquisition des verrous
  - Mettre un timeout sur l'acquisition des verrous et réessayer après un temps d'attente aléatoire

Exercice: Diner des philosophes

- Implanter le diner des philosophes
  - Chaque philosophe est représenté par un thread
  - La quantité de nourriture restante est représentée par un entier
  - Les fourchettes sont représentées par des locks
  - Pour pouvoir manger, un philosophe doit avoir 2 fourchettes
    - A chaque fois qu'il mange, il enlève une certaine quantité à la quantité restante
    - Quand il s'est servi, il repose ses fourchettes
    - Il recommence tant qu'il reste de la nourriture sur la table
  - Illustrer les différents cas de deadlocks
    - Dead lock
    - Live lock
    - Starvation
  - Comment éviter ces différents cas de figure ?

Mutex: Performance

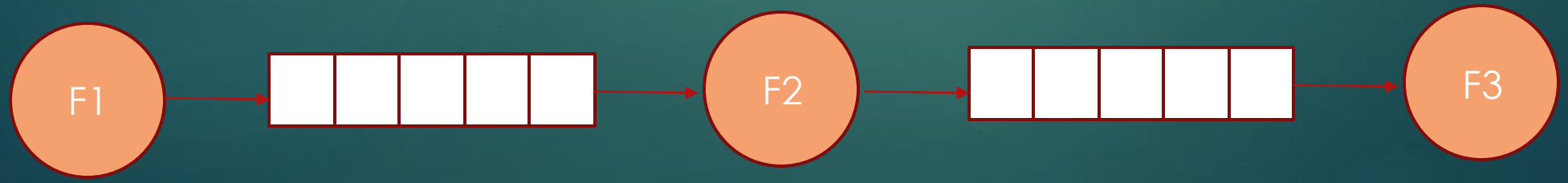
- L'utilisation seule de mutex a ses limitations en terme de performance
  - Les threads peuvent rester dans des attentes actives
    - Ils acquièrent le mutex de manière intempestive pour vérifier une condition
    - Il est nécessaire d'utiliser d'autres mécanismes (notifications) pour réveiller les threads quand les conditions nécessaires à leur exécution sont réunies
- Les moniteurs associent mutex et variables de condition
  - Les mutex s'acquièrent sur les conditions
  - Ils permettent de relâcher le verrou quand une condition n'est pas vérifiée
    - Et de **notifier** les autres threads que le mutex a été relaché
    - Cf. La classe ConditionDemo pour l'exemple

Exercice: Monitor

- Modifier la classe MonitorDining et utiliser un moniteur afin de mieux gérer les attentes
  - Obtenir une instance de la classe Condition sur le Lock
    - Condition c = lock.newCondition()
    - c.await() -> mise en attente sur une condition
    - c.signalAll() -> notification des autres threads que le mutex est relâché
  - L'utiliser pour gérer les mécanismes d'attente et de notification

Producteur / Consommateur

- Modèle de synchronisation / communication asynchrone
  - Une queue est utilisée entre producteurs et consommateurs
    - Il est nécessaire que les producteurs et les consommateurs posent un verrou sur la queue
    - Il faut faire attention aux rythmes de production / consommation de part et d'autre de la queue
    - La queue peut être bornée / non bornée
      - Mais limitée quoiqu'il en soit par la taille de la mémoire
    - La queue peut être généralisée en pipeline
    - L'interface BlockingQueue possède plusieurs implantations "thread-safe"

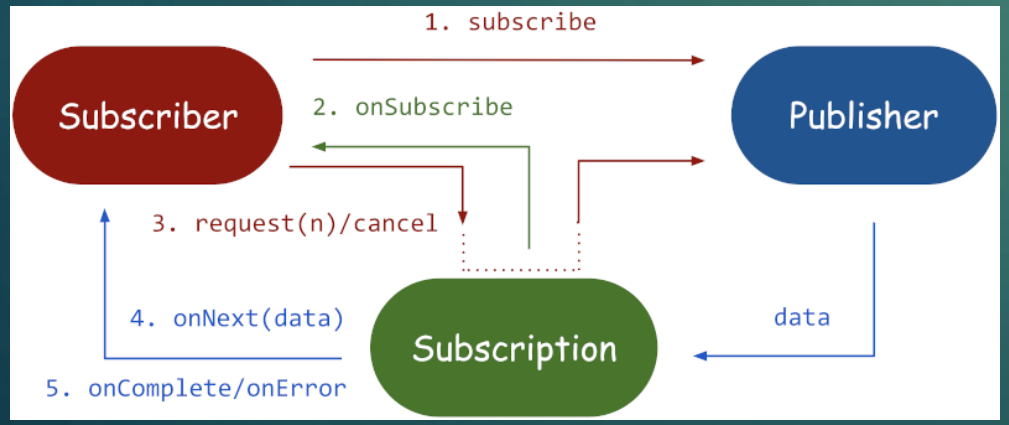


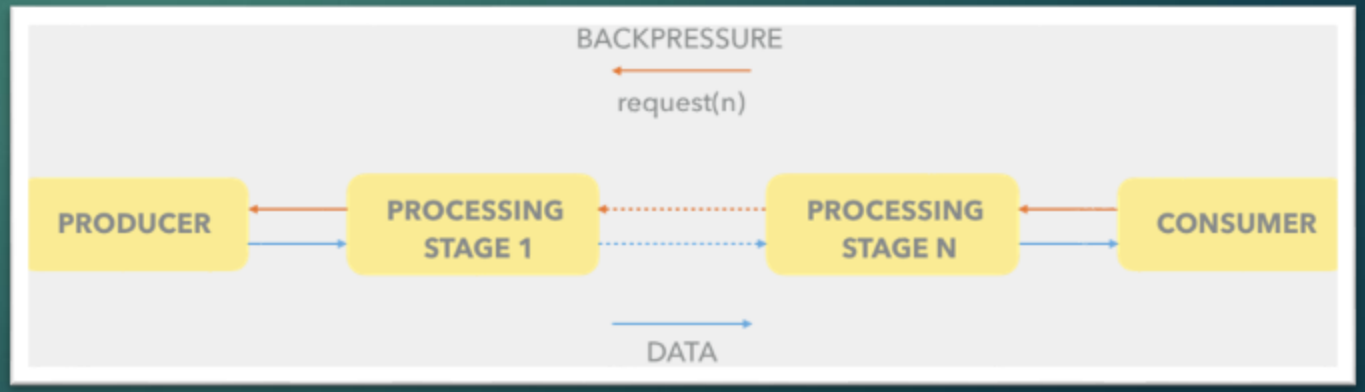
Exercice: Blocking Queue

- Instancier une queue d'entiers d'une capacité 5
  - BlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(5);
- Instancier 1 thread producteur et 2 threads consommateurs
  - Que se passe t'il si le producteur produit plus vite que les consommateurs ne peuvent consommer ?
- Classe à compléter: QueueExercise (concurrency)

Reactive Streams

- Initiative visant à établir un standard pour le traitement de flux asynchrones non-bloquants
  - Introduit dans la version 9
- Interfaces définies dans le paquetage java.utils.concurrent.Flow
  - Permettent de créer des programmes asynchrones basés sur des séquences observables d'événements
  - Etendent le patron de conception "Observer"





Exercice: Reactive Streams

- Compléter la classe ReactiveDemo afin d'implanter la chaîne représentée par ce schéma:
  - Un pubisher publier une série d'entiers
  - Un transformer (publisher / suscriber) met ces entiers au carré
  - Un suscriber stocke les résultats dans un tableau

Sémaphore

- Contrairement au mutex, il peut être utilisé par plusieurs threads en même temps
- Davantage utilisé comme un mécanisme de synchronisation
- Un compteur associé au sémaphore permet de limiter l'accès simultané à un certain nombre de threads
  - Les threads n'ayant pas pu accéder au sémaphore sont notifiés dès que le compteur est incrémenté
  - Un sémaphore binaire (compteur limité à 1) ressemble à un mutex
    - Mais le compteur peut être modifié par n'importe quel thread
  - L'acquisition d'un sémaphore par le thread est bloquante
    - Cela endort le thread jusqu'à ce que le sémaphore soit disponible

Sémaphore: Exemple

```
class SemThread extends Thread {

    private static Semaphore sem = new Semaphore(1);

    public SemThread(String name) {
        this.setName(name);
    }

    @Override
    public void run() {
        try {
            sem.acquire();
            System.out.format("%s acquiring the semaphore\n", getName());
            Thread.sleep(ThreadLocalRandom.current().nextInt(1000, 2000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.format("%s releasing the semaphore\n", getName());
            sem.release();
        }
    }
}

public class TestSemaphore {

    public static void main(String[] argv) {
        for (int i = 0; i < 10; i++) {
            new SemThread("Thread-"+i).start();
        }
    }
}
```

Barrières

- Les verrous protègent l'accès concurrent aux données
  - Mais ils ne garantissent pas les séquencements critiques (race conditions)
  - L'ordre d'exécution des threads peut avoir un impact sur l'exécution globale
    - Et provoquer des bugs très difficiles à détecter et corriger
- Pour protéger les séquencements critiques, on utilise des barrières

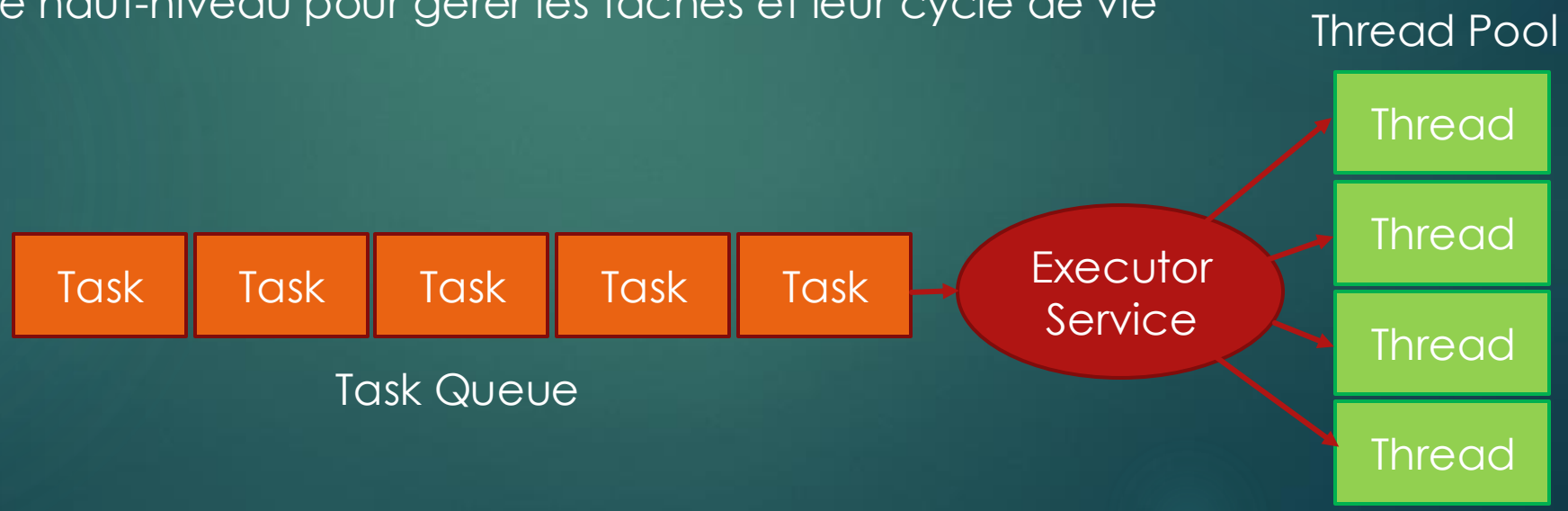
- Il existe plusieurs 2 types de barrière:
  - Les barrières réutilisables (CyclicBarrier)
  - Les barrières à usage unique (CountDownLatch)
    - Mécanisme différent (compteur)

Exercice: Barrières

- Implanter deux threads
  - Un premier qui implante la fonction f: x -> x^2
  - Un second qui implante la fonction g: x -> x+5
- Instancier 5 threads de chaque
  - S'assurer qu'ils se coordonnent afin qu'ils se combinent et produisent toujours le même résultat: x^2 + 5.
  - Utiliser pour cela une barrière cyclique
- Classe à modifier: BarriereExercise (concurrency)

Encore plus de performance...

- La création de nouveaux threads consomment de la ressource
  - Effet de bord possible: certains threads passent leur temps à attendre
  - Solution: Utiliser des pools de threads
- Java fournit l'interface ExecutorService
  - Défini dans le paquetage java.util.concurrent
  - Interface haut-niveau pour gérer les tâches et leur cycle de vie



Threads Pool: Exemple

```
class Pthread extends Thread {
    @Override
    public void run() {
        System.out.println(getName() + " executed");
    }
}

public class ThreadPoolDemo {

    public static void main(String[] argv) {
        int numProcs = Runtime.getRuntime().availableProcessors();
        System.out.println(numProcs);
        ExecutorService pool = Executors.newFixedThreadPool(numProcs);
        for (int i = 0; i < 100; i++) {
            pool.submit(new Pthread());
        }
        pool.shutdown();
    }
}
```

Les threads virtuels

- Les threads virtuels, introduits dans Java dans le cadre du projet Loom, sont des threads légers qui visent à simplifier le développement d'applications concurrentes à haut débit.
  - Ils sont conçus comme une alternative aux threads traditionnels de la plate-forme, qui sont directement mappés sur les threads du système d'exploitation (OS).

- Lorsqu'un thread virtuel effectue une opération d'E/S bloquante, le système d'exécution Java suspend le thread virtuel et libère le thread du système d'exploitation sous-jacent pour qu'il puisse effectuer des opérations pour d'autres threads virtuels.
  - Cela permet de créer un grand nombre de threads virtuels avec un petit nombre de threads du système d'exploitation, à l'instar du fonctionnement de la mémoire virtuelle

Les threads - Problèmes

- Chaque thread Java traditionnel est directement mappé sur un thread du système d'exploitation (OS thread).
- Les OS threads sont coûteux à créer, à détruire et à gérer (mémoire, temps de création, gestion du contexte).
- Le nombre maximal de threads natifs est limité par la configuration du système (mémoire, paramètres du noyau).

Les threads - Problèmes

- Les applications modernes, notamment les serveurs web ou les microservices, doivent gérer un grand nombre de connexions concurrentes.
- L'utilisation massive de threads natifs pour chaque tâche concurrente conduit rapidement à la saturation des ressources.
- Les développeurs doivent souvent choisir entre simplicité (code synchrone bloquant) et performance (code asynchrone non-bloquant, réactif), ce qui complexifie le développement et la maintenance.

Threads virtuels

- Les threads virtuels sont des entités légères, gérées par la JVM et non par le système d'exploitation.
- Ils sont beaucoup moins coûteux à créer et à détruire que les threads natifs.
- Les threads virtuels sont exécutés sur un pool réduit de threads natifs (appelés "carriers").
- La JVM s'occupe de la planification et du multiplexage des threads virtuels sur ces carriers.

Threads virtuels

- Le code écrit avec des threads virtuels reste synchrone et bloquant, comme avec les threads classiques.
- La JVM suspend automatiquement un thread virtuel lors d'une opération bloquante (I/O, attente, etc.), libérant le thread natif pour d'autres tâches.
- Pas de changement de paradigme : les API existantes (Runnable, ExecutorService, etc.) fonctionnent sans modification.
- Les threads virtuels sont compatibles avec les outils de debug, de monitoring et de profilage existants.

Les threads virtuels - Exemples

```
Thread.startVirtualThread(() -> {
    System.out.println("Thread virtuel unique : " + Thread.currentThread());
});
```

```
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 10; i++) {
        executor.submit(() -> {
            System.out.println("Thread virtuel depuis le pool : " + Thread.currentThread());
        });
    }
}
```

Communication asynchrone: Les "Future"

- Mécanisme permettant d'accéder au résultat d'une opération asynchrone
  - Qui n'est pas nécessairement disponible de suite
- Un thread asynchrone implante l'interface Callable
  - Au lieu de l'interface Runnable
  - L'unique opération "call" renvoie un résultat

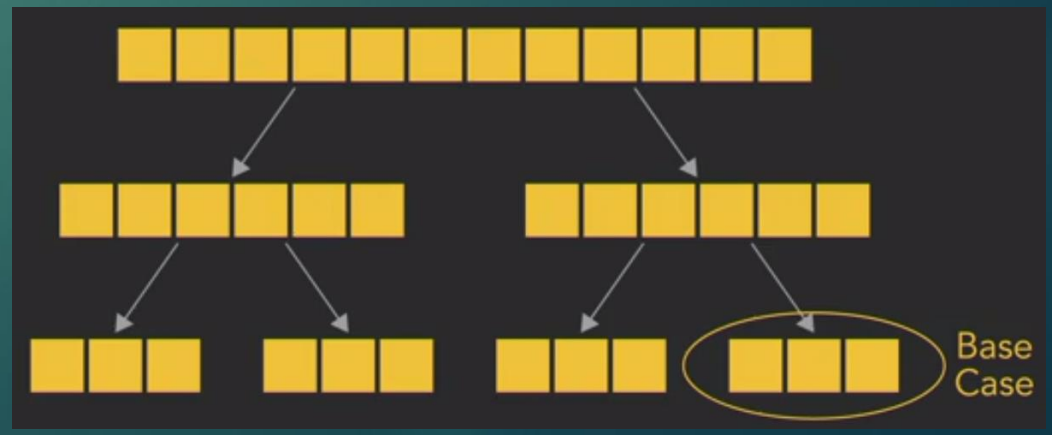
Futures: Exemple

```
class FCallable implements Callable<Integer> {
    public Integer call() throws Exception {
        System.out.println("Callable performing computation...");
        Thread.sleep(3000);
        return 42;
    }
}

public class FutureDemo {
    public static void main(String args[]) throws ExecutionException, InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Integer> result = executor.submit(new FCallable());
        System.out.println("Requesting a computation...");
        System.out.println("Computation response = " + result.get());
        executor.shutdown();
    }
}
```

ThreadPool: Divide & Conquer

- Java fournit des API pour exécuter des algorithmes récursifs de manière optimisée
  1. On divise un problème en sous-problèmes
  2. On résout les sous-problèmes de manière récursive
  3. On combine les différentes solutions
- Le problème est représenté sous la forme d'un arbre binaire
  - Une branche gauche et une branche droite
- Il faut instancier la classe ForkJoinPool dans ce cas
  - La méthode fork() pour diviser le problème
  - La méthode join() pour combiner les solutions
- 2 types de tâches
  - Avec retour: RecursiveTask<V>
  - Sans retour: RecursiveAction



Recursive Task: Exemple (1/2)

```
class RecursiveSum extends RecursiveTask<Long> {

    private long lo, hi;

    public RecursiveSum(long lo, long hi) {
        this.lo = lo;
        this.hi = hi;
    }

    @Override
    protected Long compute() {
        if (hi - lo <= 100_000) {
            long total = 0;
            for (long i = lo; i <= hi; i++) {
                total += i;
            }
            return total;
        } else {
            long mid = (hi + lo) / 2;
            RecursiveSum left = new RecursiveSum(lo, mid);
            RecursiveSum right = new RecursiveSum(mid+1, hi);
            left.fork();
            return right.compute() + left.join();
        }
    }
}
```

Recursive Task: Exemple (2/2)

```
public class DivideConquerDemo {

    public static void main(String[] argv) {
        long total = 0;
        long start = System.currentTimeMillis();
        for (long i = 0; i <= 1_000_000_000; i++) {
            total += i;
        }
        long duration = System.currentTimeMillis() - start;
        System.out.println(total + " computed in " + duration + " ms");
        ForkJoinPool pool = ForkJoinPool.commonPool();
        start = System.currentTimeMillis();
        total = pool.invoke(new RecursiveSum(0, 1_000_000_000));
        duration = System.currentTimeMillis() - start;
        pool.shutdown();
        System.out.println(total + " computed in " + duration + " ms");
    }
}
```

Exercice:

- Exécutez la classe DivideConquerDemo (concurrency) et observer le résultat
- Changez le nombre d'itérations (x1000) et observez de nouveau le résultat
- Que remarquez-vous ? Quelle explication pouvez-vous apporter ?

Exercices

- Faire les 3 exercices proposés dans le dossier "exercises"
- Toutes les ressources nécessaires sont contenues dans le projet

Programmation réactive avec Reactor

Présentation

- Reactor est né de la nécessité de réaliser des applications logiciels pouvant facilement passer à l'échelle et pouvant gérer de grandes quantités de données
  - Dans le cadre du projet Spring XD (https://docs.spring.io/spring-xd/docs/current-SNAPSHOT/reference/html/)
  - En optimisant l'utilisation des ressources de mémoire et de calcul
  - En optimisant les temps de réponse
  - En offrant un cadre cohérent aux développeurs

- Reactor a été conçu autour d'un pattern comportemental, le "**Reactor pattern**"
  - Événements asynchrones / traitements synchrones
  - https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

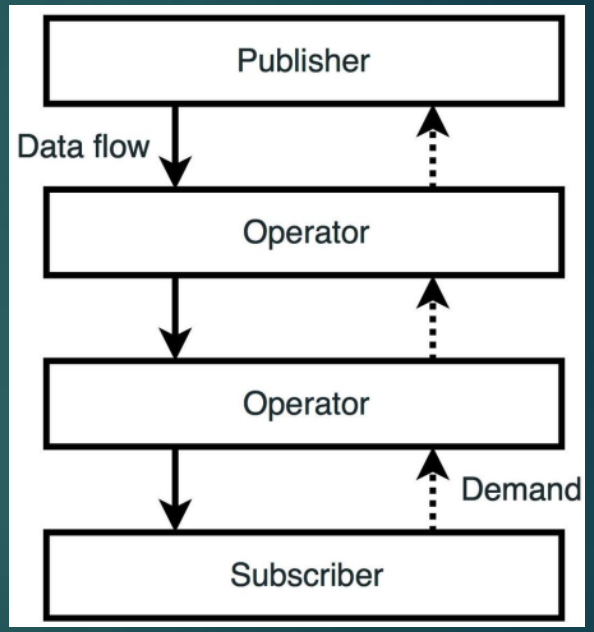
Objectifs

Une grande partie de la complexité de la création d'applications Big Data réelles est liée à l'intégration de nombreux systèmes disparates en une solution cohérente pour toute une série de cas d'utilisation. Les cas d'utilisation courants rencontrés lors de la création d'une solution big data complète sont les suivants:

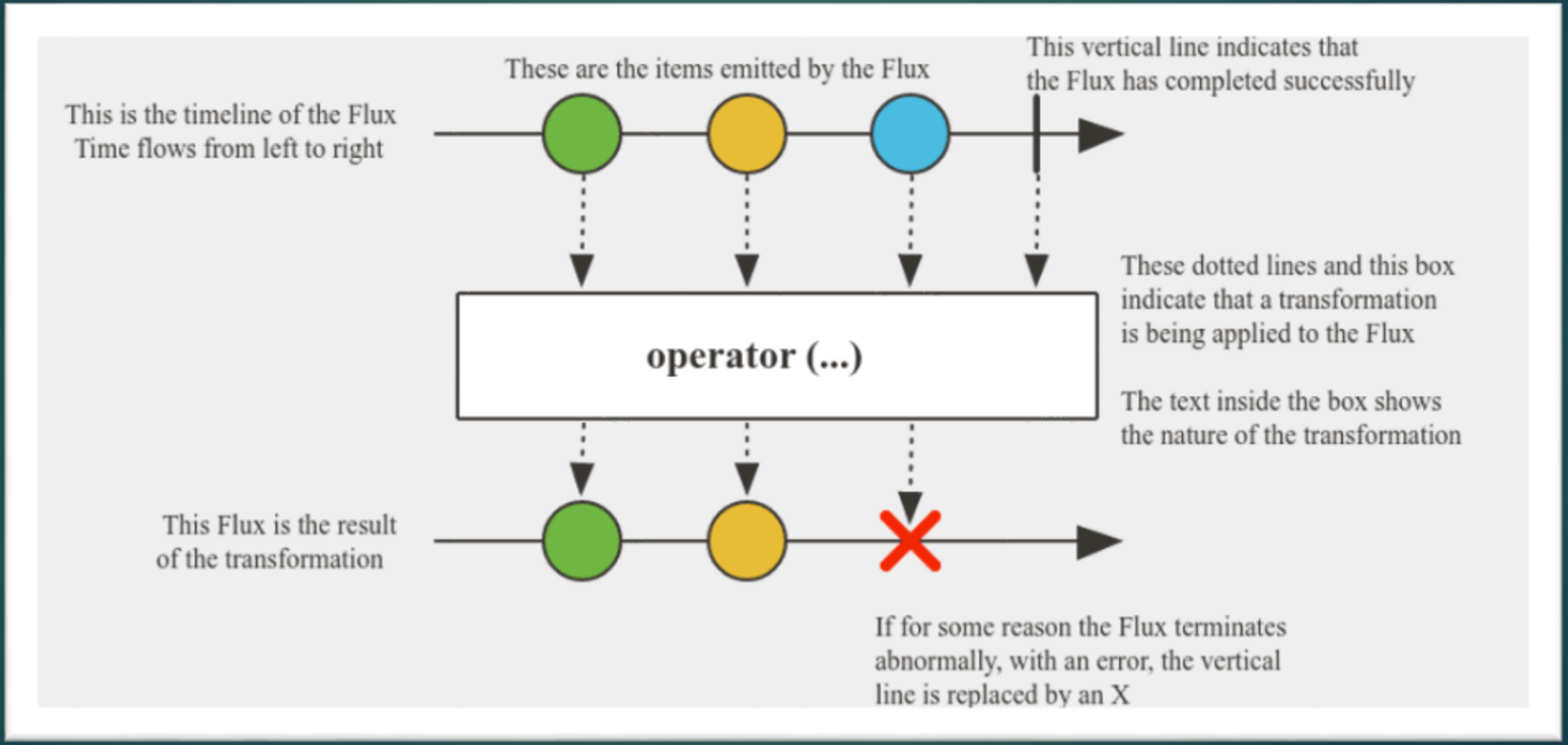
- **L'ingestion de données distribuées à haut débit** à partir d'une variété de sources d'entrée dans un store big data tel que HDFS ou Splunk.
- **Analyse en temps réel** au moment de l'ingestion, par exemple, collecte de métriques et comptage de valeurs.
- **Gestion du flux** via des batchs combinant des interactions avec des systèmes d'entreprise standard (par exemple, SGBDR) ainsi que des opérations Hadoop (par exemple, MapReduce, HDFS, Pig, Hive ou HBase).
- **Exportation de données à haut débit**, par exemple de HDFS vers un SGBDR ou une base de données NoSQL.
- **Composition** La création de valeur repose la composition de services, comprenant leur agrégation et les échanges de données basés sur un couplage faible.

Concepts

- Nous allons retrouver dans Reactor les même concepts que ceux proposés par RxJava: Publisher, Processor, Subscriber
- Avec un cadre de développement plus rigoureux
- Tout en supportant différents paradigmes d'exécution
  - Push → subscription.request(Long.MAX_VALUE)
  - Pull → subscription.request(1)
  - Push-pull → adaptation des rythmes de production / consommation
- Reactor fournit 2 types de publishers
  - **Flux** → 0..N éléments produits, tout comme un **Stream**
  - **Mono** → 0..1 éléments produits, tout comme un **Optional**
  - Avec des opérations de conversion possibles entre les deux



Concepts


These are the items emitted by the Flux
This vertical line indicates that
the Flux has completed successfully
This is the timeline of the Flux
Time flows from left to right
These dotted lines and this box
indicate that a transformation
is being applied to the Flux
operator (...)
The text inside the box shows
the nature of the transformation
This Flux is the result
of the transformation
If for some reason the Flux terminates
abnormally, with an error, the vertical
line is replaced by an X

Mono

- Création
  - Mono.just(element): à utiliser pour des données existantes
  - Mono.fromSupplier(lambda): à utiliser pour des données à calculer
  - Mono.fromRunnable(runnable): à utiliser pour des données asynchrones
  - Mono.fromCallable(callable): à utiliser pour des données à calculer
  - Mono.fromFuture(completableFuture): à utiliser pour des données asynchrones
  - Mono.empty()
  - Mono.error(err)
- Exercices :
  - Compléter la classe MonoCreate
  - Tester les différentes méthodes de création et observer les différences

Mono

- Exécuter le code de MonoSupplier
  - Qu'observez-vous ?
  - Comment y remédier ?

Mono

- Dans le code précédent, on peut observer que les 3 pipelines sont exécutées dans le thread Main
  - On observe en conséquence un comportement non bloquant
- Reactor permet d'exécuter les publishers de manière asynchrone selon différentes stratégies
  - Il faut dans ce cas veiller à bloquer le thread main jusqu'à ce que les threads créés se terminent
  - Il est possible également dans ce cas de bloquer le thread main jusqu'à la complétion de threads créés (Mono::block)
    - Cela créer implicitement un souscripteur qui se bloque en attente du résultat du publisher
    - À éviter autant que possible car cela va à l'encontre de la philosophie réactive

Flux

- Création
  - Flux.just(elements...)
  - Flux.from(publisher)
  - Flux.fromIterable(it)
  - Flux.fromStream(stream)
  - Flux.empty()
  - Flux.error(throwable)
- Génération
  - Flux.range(start, count)
  - Flux.interval(duration)

Exercice:
Tester les différentes méthodes

Flux - Debug

- Il est possible d'afficher les détails des événements dans la production / consommation de données

- Flux::log
  - Ex: Flux.range(1, 10).log().subscribe(System.out::println)

Flux - Subscription

- L'utilisation de la méthode **subscribe()** génére un subscriber avec une implémentation par défaut

- Il est possible de fournir une implémentation custom avec la méthode **subscribeWith()**
  - Il faut fournir une implémentation pour toutes les méthodes de l'interface Subscriber
  - On peut alors manipuler de manière explicite l'objet Subscription créé lors de la souscription

Flux / Mono conversion

- Il peut être utile dans certaines situations de conversion un Mono en Flux, et inversement
  - Flux.from(mono)

- La conversion Flux vers Mono pose la question de la cardinalité des éléments
  - 0..N vers 0..1
  - Plusieurs stratégies sont possibles alors:
    - On sélectionne un seul élément du Stream: next(), à combiner avec un filtre pour récupérer l'élément voulu
    - Générer un mono d'un élément représentant le tableau des éléments contenus dans le flux, possible que dans certaines situations

- **Exercice**: compléter la classe FluxToMono

Génération d'un flux (potentiellement) infini

- Jusqu'à présent, nous avons étudié des flux finis

- Reactor fournit la possibilité de créer des flux infinis que l'on peut arrêter selon certaines conditions, de différentes manières:
  - Flux::create(sink) - cf. FluxGenerate
  - Flux::generate(synchronousSink), stateless, une seule émission autorisée dans la définition, mais qui est exécutée à l'infini (loop implicite)
  - Flux::generate(stateSupplier, biFunction), idem mais statefull
  - Flux::push, pareil au generateur mais un seul thread par émission

- **Exercice**: compléter la classe FluxGenerate

Hooks

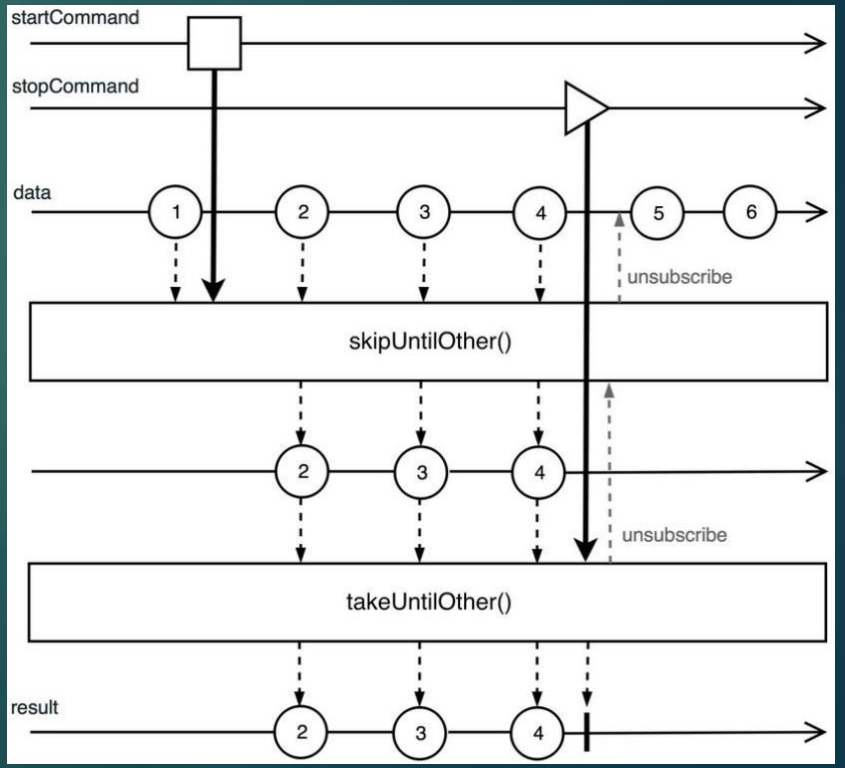
- Reactor permet d'ajouter des hooks aux événements associés à la publication
- Ces méthodes sont généralement de la forme doOnXXXX
  - XXXX représentant l'événement sur lequel on désire ajouter un hook
- **Exercice**: Modifier la classe FluxGenerate afin de tester ces différents hooks

Opérateurs

- Avec Reactor, nous retrouvons à peu de choses près les mêmes **opérateurs** que RxJava
  - Map
  - Filtre (take, takeLast, takeUntil, …)
  - Collection (just, collect, collectSortedList,…)
  - Réduction (any, all, count, …)
  - Combinaison (concat, merge, zip, …)
  - Batch (buffer et variants)
  - Cf. https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

Pour savoir quel opérateur choisir:
https://projectreactor.io/docs/core/release/reference/#which-operator

```
Flux.range(0,100)
        .flatMap(v -> Mono.fromCallable(() -> {
                Thread.sleep(500);
                return v;
        }))
        .skipUntilOther(Mono.delay(Duration.of(5, ChronoUnit.SECONDS)))
        .takeUntilOther(Mono.delay(Duration.of(10, ChronoUnit.SECONDS)))
        .subscribe(System.out::println);
```



Opérateurs

- **Take**(n): retourne un flux contenant n éléments
  - Attention à l'utilisation avec un flux infini
- **Map**(function): applique une fonction de transformation
- **Filter**(predicate): filtre selon le prédicat passé en paramètre
- **Handle**(biConsumer): map + filter
- **LimiteRate**(num, pourcentage): permet de limiter la taille du pipeline entre le publisher et le subscriber
- **DelayElements**(duration): permet de produire avec une certaine latence

Opérateurs

- **Timeout**(duration, fallback): permet de limiter l'attente sur le publisher et de fournir un autre publisher le cas échéant
- **DefaultIfEmpty**: valeur à retourner si absence de donnée(s)
- **SwitchIfEmpty**: remplace le publisher si absence de donnée(s)
- **Transform**: permet de capitaliser / réutiliser des opérateurs complexes
- **SwitchOnFirst**: permet de remplacer un flux sur la base du premier élément transmit

Opérateurs

- **FlatMap**: permet de "mettre à plat" des publishers
  - Ex:

```
UserService.getUsers()
  .flatMap(user -> OrderService.getOrders(user.getUserId())) //Flux
  .subscribe(Util.subscriber());
```

- **ConcatMap**: Similaire à FlatMap à la différence que l'ordre dans la transformation est préservée

- **Exercice**: exécuter la classe FluxFlatMap et observer le résultat

Gestion des erreurs

- La réalisation de systèmes réactifs repose un paradigme de concurrence / communication qui favorise un couplage faible entre composants
  - Les communications reposent sur la notion de signal
  - Les erreurs sont des signaux particuliers qui sont bien gérés par ce paradigme
- De fait, et conformément à la logique métier, il est possible lorsque une erreur se produit de:
  - Retourner une valeur par défaut (**onErrorReturn**)
  - Arrêter le flux (**onErrorComplete**)
  - Changer le flux nominal (**onErrorResume**)
  - Dans les 2 cas précédents, la souscription est annulée quand une erreur se produit
    - On peut utiliser la méthode onErrorContinue pour éviter d'interrompre le flux
  - Retenter l'exécution de l'opération qui a échouée (**retry**, **retryBackoff**)
  - De déférer l'exécution à d'autres flux selon les conditions d'exécution (**defer**)
- **Exercice**: compléter la classe FluxErrors afin de tester les différentes stratégies

Gestion du temps

- Comme nous l'avons vu dans les slides précédents, il est possible de définir la durée de vie des événements dans le cache: buffer (duration)
  - Il est également possible d'utiliser la méthode window et ses variantes pour jouer sur la durée et la taille des fifos

- Il est possible de générer des événements à intervalles régulier avec la méthode delayElements(duration)
  - Et même calculer des intervalles entre deux événements avec la méthode elapsed()

```
Logger log = Loggers.getLogger("app-logger");
Flux.range(0,5)
        .delayElements(Duration.ofMillis(100))
        .elapsed()
        .subscribe(e -> log.info("Elapsed {} ms: {}", e.getT1(), e.getT2()));
Thread.sleep(1000);
```

Cold vs Hot publishers

- Jusqu'à présent, nous n'avons utilisé des publishers que de type "cold"
  - Les publishers ne sont pas instanciés tant qu'il n'y a pas de souscripteur
  - Une instance par souscripteur
- Les hot pubishers sont instanciés une seule fois et émettent pour tous les souscripteurs
  - VoD vs direct stream
- La méthode **share** permet de générer un hot publisher à partir d'un cold publisher
- La méthode **publish** permet de préciser davantage les propriétés du partage (ex: nombre de souscripteur min pour lancer l'émission)
  - La méthode **autoconnect** permet de lancer l'émission même en l'absence de souscripteurs
  - La méthode **cache** permet de mettre en cache les données émises pour un replay
- **Exercice:** Modifier la classe FluxDelay afin d'illustrer le concept de Hot Publisher

Combinaisons

- Reactor supporte différentes manières de combiner des publishers
- La méthode **startWith** ajoute au début d'un flux des éléments venant d'un autre flux / collection
- La méthode **concatWith** fait l'inverse, elle ajoute à la fin d'un flux des éléments venant d'un autre flux / collection
  - Cf. Les méthodes concat* pour aller plus loin
- La méthode **merge** permet de fusionner plusieurs publishers de manière non séquentielle, au contraire des méthodes précédentes
- La méthode **zip** permet de fusionner les données (une à une) provenant de différents publishers
- La méthode **combineLatest** permet de fusionner les dernières données créées par chaque publisher à combiner

Batching

- Reactor supporte différentes stratégies pour le traitement par lots
- La méthode **buffer**(n) collecte les données produites par lots de n éléments (liste)
  - La méthode buffer(duration) collecte les données sur une durée
  - La méthode bufferTimeout combine les deux (capacité et durée)
- La méthode **window** diffère de la méthode buffer dans la mesure où elle renvoie un flux pour chaque lot, au lieu d'une liste
- La méthode **groupBy** diffère de la méthode window dans la mesure où elle renvoie différentes flux créés selon certains critères

(Dé)Matérialisation

- Il peut être parfois utile de considérer un flux de données comme un flux de signaux, et inversement

```
Flux.range(1, 3)
        .doOnNext(e -> log.info("data : {}", e))
        .materialize()
        .doOnNext(e -> log.info("signal: {}", e))
        .dematerialize()
        .collectList()
        .subscribe(r-> log.info("result: {}", r));
```

Ordonnanceurs

- Comme pour RxJava, Reactor fournit un certain nombre d'ordonnanceurs utilisables à travers les opérateurs publishOn et subscribeOn
  - **single**() : optimisé pour les exécutions à faible latence
  - **parallel**(): optimisé pour les exécutions non-bloquantes rapides
  - **elastic**(): optimisé pour les exécutions bloquantes / non-bloquantes plus longues
  - **boundedElastic**(): équivalent à elastic() avec un nombre de tâches actives borné
  - **immediate**(): pour une exécution immédiate (pas d'ordonnancement)
  - **fromExecutorService**(): pour une exécution basée sur l'utilisation d'une instance de ExecutorService (API Java)

```
Flux.range(1,10)
        .map(i -> i + 1)
        .map(i -> i * 2)
        .map(i -> i + 1)
        .subscribe(System.out::println);
```

```
Flux.range(1,10)
        .parallel(2)
        .runOn(Schedulers.parallel())
        .map(i -> i + 1)
        .map(i -> i * 2)
        .map(i -> i + 1)
        .subscribe(System.out::println);
```

Manipulation des I/O

- Qu'il s'agisse d'accéder à des fichiers ou à des bases de données
  - À des ressources qu'il est nécessaire d'ouvrir ou de fermer
  - Reactor offre l'équivalent de la construction try-with-resources en Java

- **Exercice**: Modifier la classe Titanic, remplacer les streams par des Flux.

```
Flux<String> ioRequestResults = Flux.using(
        Connection::newConnection,
        connection -> Flux.fromIterable(connection.getData()),
        Connection::close
);
```

Gestion du back-pressure

Malgré le mécanisme de back-pressure permettant une meilleure résilience du système, il est possible de surcharger un consommateur

- Les traitements par batch peuvent améliorer la situation, mais pas toujours
- Il existe d'autres mécanismes permettant de gérer de manière plus fine le back-pressure
- Le fifotage dans une queue non-bornée des données envoyées (onBackPressureBuffer)
- La suppression des données les plus récentes ne pouvant être traitées (rythme de production > rythme de consommation → onBackPressureDrop)
- La suppression des données les plus âgées ne pouvant être traitées (rythme de production > rythme de consommation → onBackPressureLast)

```
Flux.range(1, 10000000)
        .subscribeOn(Schedulers.parallel())
        .onBackpressureLatest()
        .publishOn(Schedulers.single())
        .concatMap(Mono::just, 1)
        .subscribe(ts);
```

Transformations entre Streams

- Il est possible de définir avec Reactor des fonctions permettant de générer un flux à partir d'un autre
  - En spécifiant une fonction binaire
  - Deux types de transformation: statique ou dynamique

```
Function<Flux<Integer>, Flux<Object>> transfo = stream -> stream
        .index()
        .map(Tuple2::getT1);
Flux.range(100,5)
        .map(i -> i * 2)
        .transform(transfo)
        .subscribe(System.out::println);
```

```
Random random = new Random();
Function<Flux<Integer>, Flux<Integer>> composition = stream -> {
        if (random.nextBoolean()) {
                return stream.doOnNext(e -> System.out.println(e * 2));
        } else {
                return stream.doOnNext(e -> System.out.println(e * 3));
        }
};
Flux<Integer> flux = Flux.range(100, 5).transform(composition);
flux.subscribe();
```

Tests

- Reactor fournit un objet permettant de tester des flux
  - Step Verifier
- StepVerifier.create(flux).expectNext(i).verifyComplete()
- StepVerifier.create(flux).expectNext(i).verifyError(err)
- Cf. Exemples dans les tests

IoC

Présentation

- L'IoC permet de décharger les développeurs de la gestion du cycle de vie des objets
  - Chargement, dépendances, suppression
- Les objets et leurs dépendances sont gérées par un conteneur
- Les objets peuvent être injectés au démarrage ou en cours d'exécution
- Le conteneur est chargé et configuré au démarrage de l'application
  - Il se charge ensuite du cycle de vie de tous les objets constituants l'application, et de leurs dépendances
  - Il repose sur l'utilisation d'une factory chargée de la gestion des Beans, la "Bean Factory"
  - La Bean Factory est configurée avec le contexte applicatif "Application Context"
- Le conteneur utilise les annotations afin de gérer le cycle de vie des objets
  - Les annotations sont supportées de manière native par Java
  - Ils peuvent être utilisées à la compilation ou en cours d'exécution

Concepts fondamentaux

- Application Context
  - Élément central d'une application Spring
  - Implante le principe de IoC
  - Encapsule la Bean Factory
  - Fournit les métadonnées servant à la création de Beans
  - Une application peut avoir plus d'une instance de Application Context

- La Bean Factory assure la création de Beans
  - Comprenant les singletons
  - Assure l'ordre de création des Beans (important pour la gestion des dépendances)

Configuration

- La configuration du contexte d'une application Spring peut se faire de deux manières
  - Fichiers XML (plus vraiment utilisés)
  - Annotations

- L'utilisation d'annotations présentent plusieurs avantages
  - Un seul langage, le java
  - Vérification à la compilation
  - Meilleure intégration dans les IDE

Configuration

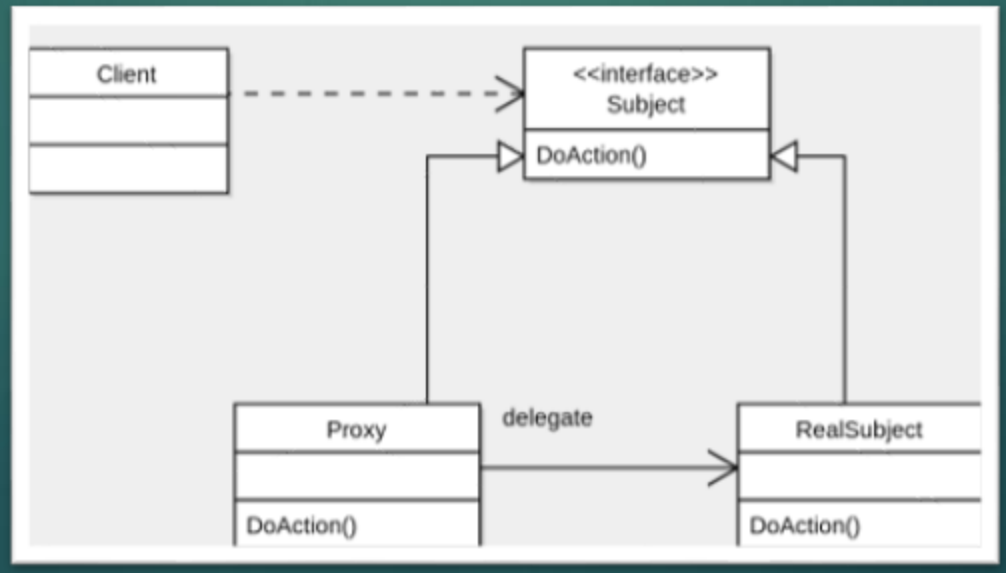
- Spring a accès aux variables d'environnement du système
- D'autres variables, propres à l'application, peuvent être déclarées via des fichiers de propriétés
  - Les variables d'environnement ont la priorité sur les variables issues des fichiers de propriétés
- Il est possible de configurer différents profiles
  - Utilisation de l'annotation "Profile('name')" sur les Beans
  - Et le paramétrage de la JVM: -Dspring.profiles.active=name

Langage d'expression SPEL

- Pour des configurations plus complexes, Spring offre un langage d'expression
  - Apporte plus de contrôle et de flexibilité
  - Exemple: @Value("#{new Boolean(environment['spring.profiles.active'] != 'dev')}")
- Une expression SPEL débute par un #
  - Elle se base sur la syntaxe de Java et dispose de variables prédéfinies
- Documentation:
  - https://docs.spring.io/spring-framework/docs/3.0.x/reference/expressions.html

Beans: Principes de fonctionnement

- En Spring, tous les objets sont des proxies
  - Pour des raisons de performances (virtual proxies)
- Permet d'ajouter du comportement dynamiquement
  - Notamment par la Programmation par Aspect (AoP)
- Les méthodes privées et appels internes ne sont pas accessibles via les proxies
  - Source de bugs dans l'utilisation de Spring si on n'y prête pas attention



Chargement automatique des Beans

- La configuration du contexte applicatif permet, en partie, de configurer la Bean Factory
  - En fournissant notamment des méthodes permettant d'instancier des Beans
- La configuration des composants peut être automatisée par le scanning
  - Il faut alors autoriser la configuration du contexte applicatif à charger certains Beans automatiquement
  - La chargement automatique de Beans repose sur le scanning

Chargement automatique des Beans

- Le chargement automatique des composants passe par 2 annotations
  - Une annotation "@ComponentScan" avec en paramètres les packagages où chercher les composants à charger
  - Une annotation "@Component" sur les Beans à charger

- L'annotation "@Component" est spécialisée par d'autres annotations
  - L'annotation "@Service" sert à indiquer que le Bean à charger implante une logique métier
  - L'annotation "@Repository" sert à indiquer que le Bean à charger sert à gérer la persistance d'objets métiers

Chargement automatique des Beans

- Le contexte de l'application scanne les composants dans les paquetages passés et dans tous leurs sous-paquetages
  - La définition de ces composants est automatiquement chargée
  - L'injection de leurs dépendances est réalisée essentiellement sur la base de l'annotation @Autowired
  - L'injection de valeurs est réalisée sur la base de l'annotation @Value
  - Le scanning doit être explicitement appelé au démarrage de l'application
  - Dans le cas d'une application Spring Boot, le scanning est implicite

- Lorsqu'il existe plusieurs implémentations pour un même composant, il est possible d'en sélectionner une en particulier grâce à l'annotation @Qualifier

Cycle de vie des Beans

- **Instanciation** de la Bean Factory
  - Chargement de la définition des beans: Annotations ou XML, proxies uniquement
  - Post-traitement de la définition des beans
    - Il est possible à ce niveau de customiser le comportement de la Bean Factory par l'interface BeanFactoryPostProcessor (non recommandé si vous n'êtes pas expert)
- Pour chaque Bean
  - Instanciation: dans l'ordre pour respecter les dépendances
    - Il est possible de préciser que l'instanciation est "lazy", mais le contexte peut l'ignorer
  - Exécution des setters: injection des données et des dépendances
  - Initialisation
    - Pré-initialisation: Appel de @PostConstruct
    - Post-initialisation: Il est possible d'injection des comportements aux Beans par l'interface BeanPostProcessor

Cycle de vie des Beans (suite)

- **Utilisation** des Beans
- **Destruction**
  - La méthode annotée @PreDestroy est appelée juste avant que l'objet soit nettoyé par le GC
  - Intéressant pour fermer des fichiers / connexions

TP: IoC

- Créer un paquetage et ajouter deux classes comme indiqué dans les encarts
  - A compléter
- Créer une classe "Main" avec une méthode main comme indiqué
  - Exécuter la classe Main
- Passer par une interface et fournir plusieurs implantations

```
public class Hello {
  private String message;
  public String getMessage()
{
    return message;
  }
}
```

```
public class Display {
  private Hello hello;
  public void displayMessage() {
    System.out.println(hello.getMessage());
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Hello hello = new Hello("Hello World");
    Display display = new Display(hello);
    display.displayMessage();
  }
}
```

TP: IoC

- Il s'agit de mettre en œuvre le principe de l'IoC
- Il faut tout d'abord importer deux librairies Spring
  - Spring-core
  - Spring-context

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
</dependency>
  <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
</dependency>
```

- Ensuite, il s'agit de définir la configuration du contexte
  - Cela passe par la spécification d'une classe annotée @Configuration
  - La classe de configuration possède deux méthodes permettant d'instancier
    - Un bean Hello et un bean Display
    - Les deux méthodes sont annotées @Bean

TP: IoC (Suite)

- Il faut enfin modifier la classe Main
  - Pour instancier un objet ApplicationContext
  - Nous choisissons une implantation basée sur les annotations: AnnotationConfigApplicationContext
  - Nous pouvons alors demander le chargement du bean Display et appeler la méthode displayMessage

TP: Configuration

- Créer dans le dossier "resources" un fichier hello.properties
  - Avec une entrée "app.greeting=Hello World"
- Ajouter à la configuration du contexte ce fichier comme source de propriétés
  - Par l'ajout de l'annotation @PropertySource("classpath:hello.properties")
  - Les propriétés définies dans ce fichier sont maintenant disponibles pour l'application
- Supprimer le constructeur de la classe Hello et injecter la valeur de la propriété app.greeting
  - Par l'ajout de l'annotation @Value("${app.greeting}") sur un attribut de la classe

TP: Configuration

- Dans la configuration du Main au niveau de l'outil, ajouter une variable d'environnement app.greeting avec la valeur "Hello SPRING"
  - Exécuter et observer le résultat

- Créer dans l'outil une deuxième configuration "dev"
  - Utiliser SPEL pour configurer le message à afficher selon les profils; tester
    - "Hello Dev" si profil dev, "Hello Prod" sinon
    - Paramétrage de la JVM: -Dspring.profiles.active=dev

TP: Configuration

- Nous allons maintenant configurer le chargement automatique des beans
- Il faut tout d'abord indiquer à la configuration de scanner les beans à charger automatiquement au démarrage
  - Par l'annotation @ComponentScan et en indiquant le paquetage racine à partir duquel rechercher les composants à charger automatiquement
  - Par l'annotation @Component sur les beans à charger automatiquement
- Supprimer le bean Hello de la configuration
- Marquer la classe Hello avec l'annotation @Component

TP: Configuration

- Modifier la classe Display en conséquence
  - Suppression du constructeur
  - Injection de la dépendance au bean Hello par l'utilisation de l'annotation @Autowired
  - Tester et observer
- Supprimer toutes les méthodes de la configuration et marquer la classe avec l'annotation @Component
  - Tester et observer

AOP

Présentation

- Les principes du paradigme objet supportent une meilleure capitalisation / réutilisation
  - Par l'encapsulation et l'emploi de différents patrons de conception
  - La POO améliore le développement logiciel au niveau architectural
  - Elle reste cependant limitée au niveau comportemental
- Le paradigme AOP permet de mieux séparer les préoccupations au niveau comportemental
  - La définition d'un objet résulte généralement de la composition de diverses préoccupations: métier, performance, sécurité, authentification, etc.
  - Le but de l'AOP est un découplage et une meilleure réutilisation de préoccupations transverses

Présentation

- L'AOP se combine bien à l'IoC
  - Elle permet un tissage en cours d'exécution de différentes préoccupations
  - Le pattern Proxy utilisé par l'IoC se prête bien au tissage de comportement en cours d'exécution
- Exemples d'application:
  - Logging
  - Gestion des transactions
  - Mise en cache
  - Sécurité

Principes

- Les aspects représentent des blocs de code réutilisables pouvant être injecté à l'application en cours d'exécution
  - Évite la duplication de code et permet une meilleure réutilisation de bouts de code
  - Facilite la maintenance
  - Améliore la qualité, notamment en rendant le code plus lisible

- Concepts associés aux aspects
  - **Aspect:** module définissant des greffons et leurs points d'activation
  - **Greffon (Advice):** un programme activable à un certain point d'exécution du système, précisé par un point de jonction
  - **Tissage (Weaving) :** insertion statique ou dynamique dans le système logiciel de l'appel aux greffons
  - **Point de coupe (P*ointcut*) :** endroit du logiciel où est inséré un greffon par le tisseur d'aspect
  - **Point de jonction (Join Point)** endroit spécifique dans le flot d'exécution du système, où il est valide d'insérer un greffon (avant, autour de, à la place ou après l'appel de la fonction)
  - **Préoccupation transverse:** sous-programmes distinct couvrant un aspect de la programmation

AspectJ

- AspectJ est un langage supportant le paradigme AOP
- Il a été proposé et initialement développé par Xerox au début des années 2000
  - Sur le base du langage Java
  - Dans l'environnement Eclipse
- Il supporte le tissage statique et dynamique
- Il est bien intégré dans l'environnement SPRING
- Pour aller plus loin avec AspectJ
  - https://www.eclipse.org/aspectj/doc/released/progguide/index.html

TP: AOP

- Ajouter une dépendance à Log4j dans le fichier pom.xml et importer les librairies
- Configurer log4j pour afficher les logs sur la sortie standard (fichier log4j.properties)
- Modifier le code des classes Hello et Display afin de tracer les appels aux opérations
  - Respectivement getMessage() et displayMessage()
  - En instanciant un Logger au niveau de chaque classe
- Exécuter et observer le résultat

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-reload4j</artifactId>
</dependency>
```

```
log4j.rootLogger=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{dd-MM-yyyy HH:mm:ss} %-5p %c{1} - %m%n
```

TP: AOP

- Nous allons maintenant "aspectiser" les traces
- Il s'agit de créer un aspect qui va tisser le comportement du log aux fonctions
- Il faut pour ce faire ajouter une dépendance à AspectJ dans le fichier pom.xml
- Nous allons créer une annotation @Loggable que nous utiliserons pour injecter du comportement aux fonctions
- Ensuite nous créons notre aspect qui définit
  - Un point de coupe
  - Des greffons: avant, pendant et après l'appel
- L'aspect est un simple bean chargé automatiquement et annoté @Aspect
- Le point de coupe est une méthode
  - Ne retournant rien
  - Annoté @Pointcut avec en paramètre la condition du tissage (annotation, appel, exception,…)

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Loggable {}
```

TP: AOP

- Nous allons mettre en place 3 types de greffon:
  - Celui exécuté avant l'appel à l'opération identifié par l'annotation @Before
  - Celui exécuté pendant l'appel à l'opération identifié par l'annotation @Around
  - Celui exécuté après l'appel à l'opération identifié par l'annotation @AfterReturning
- Il est également nécessaire d'activer les aspects au niveau de la configuration
  - Par l'utilisation de l'annotation @EnableAspectJAutoProxy
- Tester et observer
- Mettre en œuvre un autre aspect qui modifie la valeur de retour de la méthode getMessage de la classe Hello
- Implanter un aspect qui log le nombre d'appels à chaque méthode
- Implanter un aspect qui décryptent les chaînes de caractères avant d'appeler une méthode et qui cryptent les chaînes de caractères en sortie

Exemples d'application: Lombok

- Project Lombok est une bibliothèque Java qui s'intègre à votre éditeur et à vos outils de construction, simplifiant le code Java et réduisant les répétitions.

- Elle y parvient en permettant aux annotations de remplacer ce qui serait autrement un code verbeux et répétitif, comme les getters, setters, les constructeurs ou les méthodes equals(), hashCode() et toString(), entre autres.

- L'objectif premier de Lombok est de réduire ce code sans valeur ajoutée et de rendre le code Java plus concis, plus lisible et plus facile à maintenir. Il améliore la productivité des développeurs en automatisant ces tâches banales au moyen d'annotations.

Exemples d'application: Lombok Principales annotations

- **@Getter/@Setter** : Génère automatiquement des getters et setters pour vos champs
- **@NoArgsConstructor, @RequiredArgsConstructor, @AllArgsConstructor** : Génère des constructeurs sans arguments, avec arguments requis ou avec tous les arguments, respectivement.
- **@Data** : Un raccourci pour @ToString, @EqualsAndHashCode, @Getter sur tous les champs, @Setter sur tous les champs non finaux, et @RequiredArgsConstructor
- **@Builder** : Implémente le modèle Builder, facilitant la construction d'objets.

Exemples d'application: Lombok Principales annotations

- **@SneakyThrows** : Permet de lancer des exceptions vérifiées sans les déclarer dans la clause throws de la méthode.

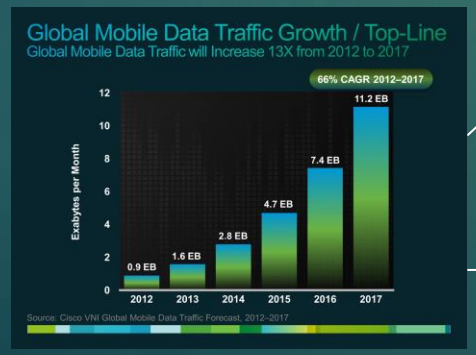
- **@Value** : Crée une classe immuable en marquant tous les champs comme privés et finaux et en générant des getters, un constructeur, des méthodes equals(), hashCode() et toString().

- **@Slf4j** : fournit un champ logger, simplifiant la journalisation au sein de la classe

Persistence

Persistance des données

- Les applications manipulent des données
  - Ces données peuvent provenir de différentes sources: fichiers ou base de données
  - Ces données peuvent être stockées pour pouvoir être réutilisées ou partagées
  - La persistance permet aux données d'exister même quand les applications qui les manipulent ne s'exécutent plus



ORM – Object Relational Mapping

- La persistance d'objets dans des bases de données relationnelles passe par une traduction entre deux paradigmes
  - Le paradigme objet d'une part, basé sur des objets et des pointeurs
  - Le paradigme entité/relation d'autre part, basé sur des colonnes et des clefs
  - **Question**: Quels sont les différences fondamentales entre les deux paradigmes ?
- Lorsque l'on veut persister des objets java dans des bases relationnelles, il est nécessaire de définir un mapping entre les deux mondes
  - C'est précisément l'objectif d'un ORM
  - Et c'est ce que JPA permet de spécifier
- Les avantages de l'utilisation d'un ORM sont nombreux:
  - Gain de productivité
  - Meilleure séparation des préoccupations
  - Meilleure maintenabilité

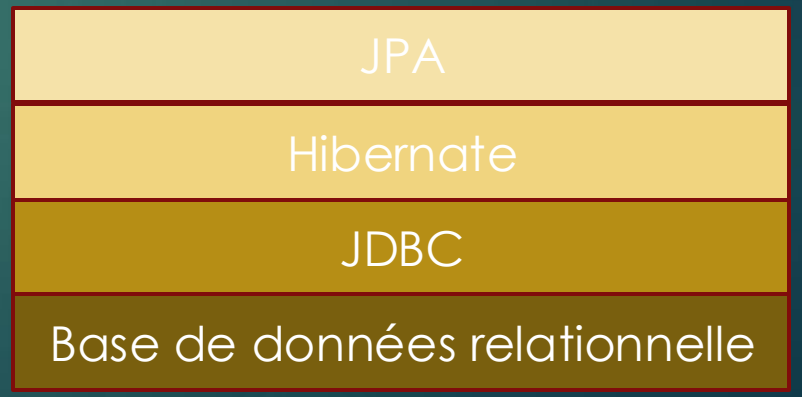
JPA – Java Persistence API

- Java offre des API permet de gérer la persistance des API dans les bases de données, relationnelles ou non
  - Accès
  - Persistance
  - Gestion

- JPA est supporté par un grand nombre d'outils
  - Hibernate, Eclipse Link, Oracle, etc.

JPA: Présentation

- JPA permet de cacher au développeur les détails techniques de la persistance
  - Qui est du ressort de la base de données
  - Cela permet de se focaliser sur les problèmes métiers
- Les API de JPA sont définies dans les paquetages javax.persistence.*
  - Cf. https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html
- Les API de JPA ne définissent que des interfaces
  - Qui sont implantées par différents outils:
    - Eclipse Link, Hibernate, OpenJPA ou Data Nucleus



JPA: Configuration

- Comme pour les beans, il y a deux manières de configurer JPA
  - Par les annotations ou par les fichiers XML
  - Dans le cas où les deux sont présents, les fichiers XML ont la priorité
- La définition des mappings est basée sur l'utilisation de métadonnées sous forme d'annotations
- Les objets devant être persistés dans des bases de données sont marqués par l'annotation @Entity
- Une entité est un composant scannable
- Elle définit le mapping entre:
  - Objet et table
  - Propriétés et colonnes
  - Contraintes d'intégrités
- Elle est manipulable comme un objet
  - On peut lui ajouter des méthodes par exemple

```
@Entity
@Table(
  name = "persons",
  uniqueConstraints = {
    @UniqueConstraint(columnNames = {"firstName", "lastName"})
  }
)
public class Person {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @Column(name = "id", nullable = false)
  private Long id;
  private String firstName;
  @Column(length = 50)
  private String lastName;
  private int age;
}
```

JPA: Règles de mapping

- L'annotation @Entity sert à indiquer que les instances d'une classe seront persistées dans une table
- Les champs de la classe doivent être publiques et non statiques ni finaux
- Sans aucune autre information, JPA applique des règles de mapping par défaut, à savoir:
  - Entité ⇒ Table de même nom
  - Attribut ⇒ Colonne de même nom
    - En particulier, l'attribut id génère une clef primaire

JPA: Règles de mapping

- Sur les types également, JPA applique des règles de mapping par défaut, selon les SGBD
  - String ⇒ varchar(255)
  - Long ⇒ BIGINT
  - LocalDate, LocalDateTime => DATE, DATETIME
  - …
- Ces règles sont cependant modifiables par des annotations
  - Nom des tables et des colonnes
  - Champs requis
  - Taille des champs
  - …

JPA: Relations entre entités

- Les objets en Java sont reliés par des références. Dans les bases de données, les tables sont reliées par des jointures, basées sur des clefs

- Dans le cas de références à des tableaux d'objets ou des références croisées, JPA offre également un certain nombre d'annotations pour préciser le mapping

JPA: Relations entre entités

- Il y a différents types de relations entre entités et JPA permet d'annoter les références afin de spécifier les arités des relations et les cycles de vie associés
  - 1..1 : @OneToOne ⇒ Stratégie de mapping: JoinColumn
  - 1..* : @OneToMany ⇒ Stratégie de mapping: JoinTable
  - *..1 : @ManyToOne ⇒ Stratégie de mapping: JoinColumn
  - *..* : @ManyToMany ⇒ Stratégie de mapping: JoinTable

- Pour chaque type de relation, il est possible d'en préciser les propriétés
  - Optional: La relation n'est pas obligatoire
  - MappedBy: nom de la référence inverse
  - Cascade: Action à réaliser en cas de modification (CRUD)

JPA: Relations entre entités

- Pour chaque classe référencée, un identifiant est généré
  - Il est possible de changer cet identifiant par l'annotation @JoinColumn
- Pour les arités n-aires, il est possible de générer une table d'association par l'annotation @JoinTable
  - Le choix d'utiliser l'annotation @JoinTable plutôt que @JoinColumn est une question de compromis entre performance et empreinte mémoire
- Chargement des entités référencées
  - Eager: chargée systématiquement
  - Lazy: chargée à la demande
  - Le choix de l'une ou l'autre stratégie est une question de compromis entre performance et empreinte mémoire

JPA: Héritage entre entités

- Peu de SGBD supporte l'héritage entre tables
- Pour autant, l'héritage est une caractéristique importante du paradigme Objet
- JPA supporte l'héritage entre entités avec différentes stratégies de mapping possibles
  - MAPPED_SUPER_CLASS: La classe parent n'est pas persistée
  - SINGLE_TABLE: Une seule table utilisée pour une hiérarchie de classes
  - JOINED: L'héritage est représenté par une jointure entre table parente et tables filles
  - TABLE_PER_CLASS: l'héritage se manifeste par une table par entité

```
@MappedSuperclass
public abstract class Person {...}

@Entity
@DiscriminatorValue("Patient")
public class Patient extends Person {..}
```

```
@Entity
@Inheritance(strategy =
InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "person_type")
public class Person {...}

@Entity
@DiscriminatorValue("Patient")
public class Patient extends Person {..}
```

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "person_type")
public class Person {...}

@Entity
@DiscriminatorValue("Patient")
public class Patient extends Person {..}
```

```
@Entity
@Inheritance(strategy =
InheritanceType.TABLE_PER_CLASS)
public class Person {...}

@Entity
public class Patient extends Person {..}
```

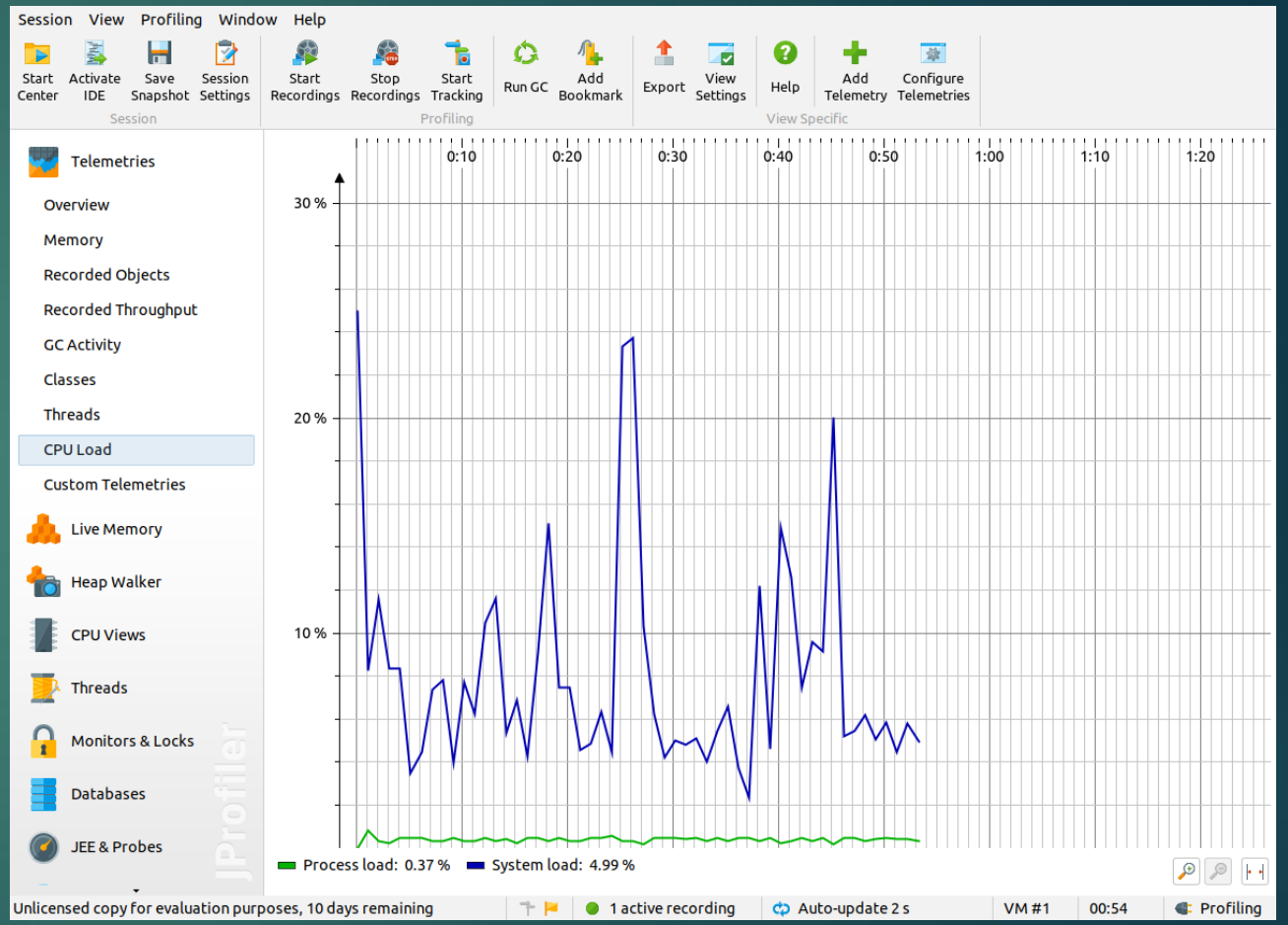
JPA: Exercices

- Compléter les classes se trouvant dans le paquetage jpa
  - Ajouter les informations de mapping
  - Ajouter les relations
  - Ajouter les contraintes d'intégrité
- Vérifier le résultat par l'exécution du fichier ActorDAO

Pour aller plus loin

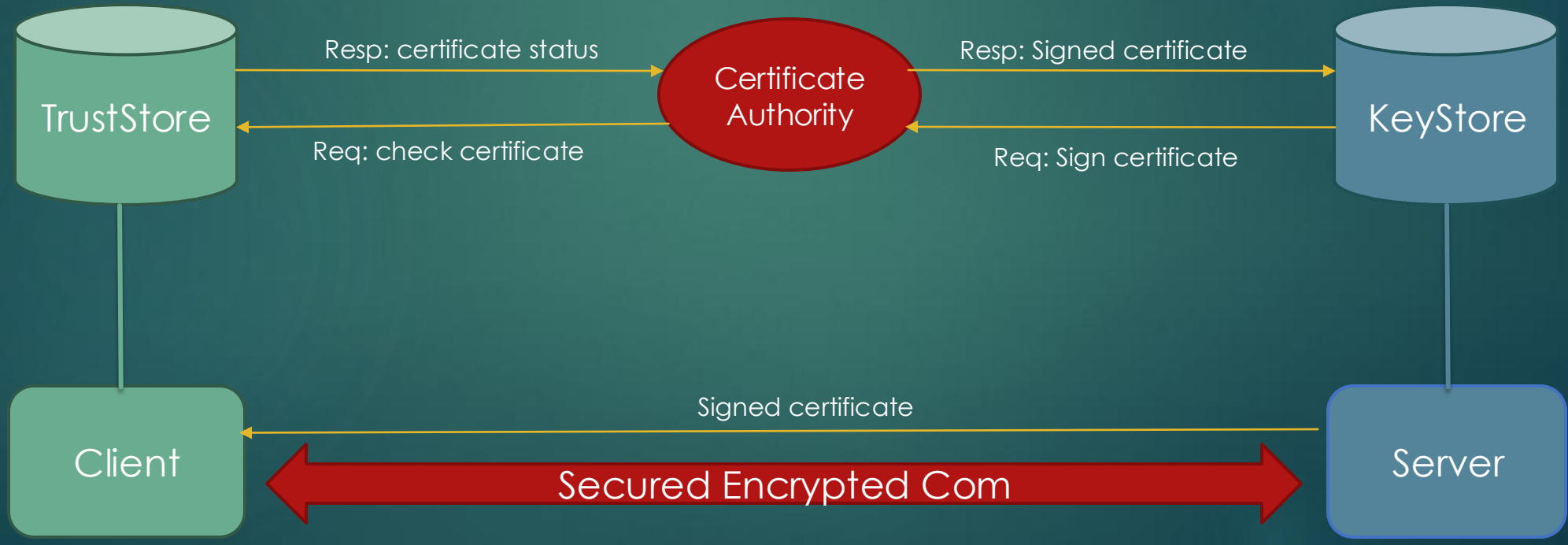
Profiling

- En complément aux tests, il peut être intéressant d'avoir une idée plus précise de la performance générale de vos applications
  - Utilisation de la mémoire
  - Utilisation du processeur
  - Statistiques sur les appels de méthodes
    - Détection de contentions et mise en place de patterns pour les corriger
- Télécharger l'outil JProfiler (License d'évaluation):
  - https://www.ej-technologies.com/download/jprofiler/files
- Installer le plugin JProfiler dans Intellij
- Vous pouvez lancer une exécution avec le profiling



Transport Layer Security - TLS 1.3

- Permet le transport sécurisé de données à travers le réseau
  - Cryptage au niveau de l'émetteur
  - Décryptage au niveau du destinataire



Transport Layer Security - TLS 1.3

- Génération des certificats avec openssl
  - mkdir ssl ssl/private ssl/certs #dossier pour enregistrer les clefs
  - openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout ssl/private/myselfsigned.key -out ssl/certs/myselfsigned.crt
  - openssl dhparam -out ssl/certs/dhparam.pem 2048

- Installation du certicat dans l'environnement java
  - sudo keytool -import -noprompt -trustcacerts –alias mycert -file ssl/certs/myselfsigned.crt -keystore /opt/jdk-21/lib/security/cacerts

JVM: options essentielles

- Mémoire
  - Taille du tas
    - –Xms : taille initiale du tas, -Xmx: taille max du tas
    - -XX:Permize: taille du PermGen, où sont enregistrées les méta-données des classes
    - Ex: -Xms2048m –Xmx:4096m
  - Taille de la pile de threads
    - -Xss : taille de la pile pour chaque thread
- Garbage collector
  - -verbose:gc : Afficher les opérations du GC – date et durée

JVM: options essentielles

- Chargement des classes
  - -XX:+TraceClassLoading
  - -XX:+TraceClassUnloading
- Profilage:
  - -Xprof
  - -Xrunhprof

Conclusion

Synthèse de la formation

- Les fondements du langage Java : syntaxe, typage, paradigmes objet et fonctionnel.
- Les évolutions majeures : génériques, lambdas, streams, modularisation, records, sealed classes, threads virtuels, string templates.
- Les API essentielles : collections, concurrence, I/O, tests (JUnit, Mockito), sécurité.
- Les bonnes pratiques : gestion de la mémoire, robustesse, lisibilité, maintenabilité.
- Les outils modernes : JShell, IntelliJ, Git, modules, compilation native.

Conclusion

- L'évolution du langage et des spécifications de la JVM repose sur le travaille d'une large communauté très active
  - Mises à jour régulières
  - Nouveautés et corrections
  - S'inspire des contributions des autres langages
- Au-delà du JDK, il existe un riche éco-système et des projets soutenus par des organismes reconnus
  - Apache, Eclipse, …

Conclusion

- Il existe également un riche éco-système de langages qui sont compilés pour la JVM et qui apportent des contributions significatives
  - Groovy, xTend, kotlin, scala, …
- Java est dans la pratique très utilisé avec des outils de gestion de cycle de vie:
  - JUnit, Ant, maven, gradle, jenkins