ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Java avancé

Concepts avancés du Java

1. Quelques particularités du langage

1.1 Évaluation paresseuse

Les expressions booléennes sont évaluées jusqu'à ce que le succès ou l'échec de l'expression totale puisse être déterminé sans ambiguïté. En conséquence, la fin d'une expression logique peut ne pas être évaluée. Dans l'exemple suivant, si `test1()` est faux, `test2()` et `test3()` ne seront pas exécutés.

```
1    if (test1() && test2() && test3()) {
2      System.out.println("expression is true");
3    } else {
4      System.out.println("expression is false");
5    }
```

1.2 Les opérateurs de décalage de bits

Aux opérateurs classiques '>>' et '<<', Java ajoute l'opérateur de décalage à droite non signé **'>>>'**, qui utilise l'extension avec des zéros sans se soucier du signe ; des zéros sont insérés aux bits de poids forts. Le décalage signé à droite **'>>'** utilise l'extension de signe : si la valeur est positive, des zéros sont insérés aux bits de poids forts, si elle est négative, des uns sont insérés pour les bits de poids forts.

Donc, >>2 permet de multiplier par 2 et <<2 de diviser par deux des entiers (dans la limite des valeurs du type).

```
1    int i=1;
2    i<<=1;    // i = 2;
3    i>>=1; // i = 1;
4
5    i=-1;
6    i<<=2; // i = -4
```

Si on décale un `char`, un `byte` ou un `short`, il sera promu en `int` avant de faire le décalage. Tous les autres opérateurs de décalage ont le même effet qu'en C.

1.2.1 Exemples

```
1     int i=-1  // i est représenté par 11111111111111111111111111111111
2     i >>>= 31;       // résultat : 1
3
4     long l = -1;
5     l >>>= 63;       // résultat : 1
6
7     short s = -1;
8     s >>>= 10;       // Erreur de troncature, donc résultat : -1
9     short s = -1;
10    s >>>= 31;       // résultat : 1
11
12    byte b = -1;
13    b >>>= 10;       // Erreur de truncature, donc résultat : -1
14    byte b = -1;     // Correct. Résultat : 1
```

1.3 Le sulfureux « goto »

Java n'a pas de `goto`, mais le mot clé est réservé.

1.4 L'opérateur « sizeof »

Java n'a pas d'opérateur `sizeof()`. Java impose la taille de chaque type primitif indépendamment du système d'exploitation. Tous les types numériques sont signés ; il n'y a pas de type non signé. C'est un choix délibéré considérant les problèmes posés par l'utilisation des types non-signés au regard du faible gain pour le développeur. Voir les discussions de ce problème sur le Web.

Chaque type primitif possède une classe associée (*wapper* en anglais).

Type primitif	Taille	Minimum	Maximum	Classe associée
boolean	-	-	-	Boolean
char	16-bit	Unicode 0	Unicode 2^{16}- 1	Character
byte	8-bit	-128	127	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	Short
int	32-bit	-2^{31}	$+2^{31}-1$	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	Long
float	32-bit	2^{-149}	$(2-2^{-23})\cdot 2^{127}$	Float
double	64-bit	2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	Double
void	-	-	-	Void

Les valeurs maximale et minimale de chaque type peuvent être obtenues en utilisant des constantes de la classe associée, par exemple :

```
Byte.MIN_VALUE
Integer.MAX_VALUE
Float.MIN_EXPONENT
```

1.5 Utilisation de labels à l'intérieur de boucles

Parce qu'il n'y a pas de `goto`, Java utilise des labels pour sortir des boucles imbriquées.

```
1    label1:
2    boucle-externe { // e.g. for (int i=0; i<10; i++)
3      boucle-interne {
4        …
5        break; // stoppe la boucle interne et retourne à la boucle externe
6        …
7        continue; // continue au début de la boucle interne
8        …
9        continue label1; // stoppe la boucle interne
10                        // et poursuit la boucle externe
11       …
12       break label1; // stoppe à la fois les boucles interne et externe
13       …
14     }
15   }
```

1.6 L'opérateur virgule

L'opérateur virgule a seulement deux usages : dans la déclaration des variables et arguments et dans une expression de boucle `for`.

```
1    int x, y;
2    for (int i = 0, j = 0; i < 10; i++, j++) { ... }
```

1.7 La boucle infinie

Le compilateur traite de la même façon la boucle infinie `while (true)` et `for (;;)`, y compris pour l'optimisation.

1.8 Signature de méthodes

La valeur de retour ne fait pas partie de la signature d'une méthode qui comprend seulement le nom de la méthode et la liste des arguments. Les deux déclarations suivantes provoquent une erreur de compilation :

```
1    class toto {
```

```
2       int f();
3       void f(); // Erreur : redéfinition de la méthode f()
4     }
```

De la même façon, la spécification d'exception ne fait pas non plus partie de la signature des méthodes. Les deux déclarations suivantes produisent une erreur de compilation :

```
1     class toto {
2       int f() throws IOException;
3       int f() throws RuntimeException; // Erreur
4     }
```

1.9 La spécification d'accessibilité

Il y a quatre types d'accessibilité des attributs et des méthodes :

- privée (`private`)
- protégé (`protected`)
- paquet (`package`)
- public (`public`)

Quand on ne spécifie aucune accessibilité, cela revient à l'accessibilité de type **paquet**. Il n'existe pas de mot clé pour signifier explicitement une accessibilité de type paquet.

Le mot clé `protected` donne aussi l'accès aux paquets. En Java, il n'est pas possible de limiter l'accessibilité à la dérivation. Cependant, une bonne pratique est de garder la distinction dans le code : une réduction de dérivation utilise le mot clé `protected` et une réduction de paquet n'utilise aucun mot clé.

La table suivante résume l'accessibilité aux membres permis par chaque type de visibilité :

Visibilité	Classe	Paquet	Sous-classe	Monde
`public`	+	+	+	+
`protected`	+	+	+	-
	+	+	-	-
`private`	+	-	-	-

2. Initialisation et nettoyage

2.1 L'objet Class

Chaque fois que l'on écrit et compile une nouvelle classe, un objet `Class` est créé et stocké, dans un fichier de même nom et suffixé `.class`. À l'exécution, quand on veut créer un objet de cette classe, la machine virtuelle Java charge le fichier `.class` correspondant si ce n'est déjà fait. Donc, un programme Java n'est pas complètement chargé au début d'une exécution, ce qui diffère de tous les autres langages.

2.2 La méthode Finalize

La méthode `finalize()` de la classe `Object` n'est pas utilisée pour libérer la mémoire au sens C++ du terme. Cette méthode est bien appelée avant que l'objet soit libéré, mais on ne peut pas savoir quand, puisque le ramasse-miette (*garbage collector*) est lancé seulement quand la machine virtuelle estime que cela est nécessaire.

Si un objet doit être supprimé (autrement que par le ramasse-miette), il faut utiliser l'idiome suivant : initialiser l'objet, puis immédiatement entrer dans un bloc '`try-catch`' pour ajouter le code qui sera utilisé par l'objet, et enfin utiliser la clause `finally` pour effectuer le nettoyage après avoir utilisé l'objet. Le bloc `finally` est toujours exécuté, qu'il y ait levé d'exception ou pas.

```
1    Foo f = new Foo();      // initialisation de l'objet
2    try {
3      bar(f);               // bloc de code utilisant l'objet
4    } catch () {
5      ...                   // code pour gérer les exceptions si nécessaire
6    } finally {
7      ...                   // bloc de code de nettoyage
8    }
```

2.3 Initialisation de données membres

Tout membre d'une classe est garanti d'avoir une valeur par défaut s'il n'est pas initialiser explicitement. La valeur par défaut correspond au zéro (*eg.*, booléen = faux, référence = null, int = 0, string = null).

```
1    class Foo {
2      int x;          // x est initialisé à 0
```

```
3       Bar bar;       // bar est initialisé à null
4     }
```

Cette garantie ne s'applique pas aux variables locales.

2.4 Ordre d'initialisation

À l'intérieur d'une classe, l'ordre d'initialisation est déterminé par l'ordre dans lequel les variables sont déclarées.

```
1     class Foo {
2       int a = 1;
3       int b = a;
4     }
```

2.5 Constructeur

Une classe possède au moins un constructeur par défaut, c'est à dire un constructeur sans argument, si le développeur n'en définit aucun. Il est donc inutile de déclarer le constructeur par défaut s'il n'y a pas de code à l'intérieur.

```
1     class Engine extends CarPart {
2       public Engine() { } // déclaration superflue
3     }
```

Cependant, si le développeur définit au moins un constructeur avec arguments, alors le constructeur par défaut n'est pas défini. Dans ce cas, il faut ajouter explicitement le constructeur par défaut, si l'on veut faire une instance avec le constructeur par défaut.

```
1     class Engine extends CarPart {
2       public Engine() { }  // déclaration non superflue pour définir
3                            // le constructeur sans argument.
4       public Engine( float price ) {
5          ...
6       }
7     }
```

Java insère automatiquement l'appel au constructeur de la classe de base par `super()` dans le constructeur des classes dérivées. Cependant, il est possible d'appeler explicitement un autre constructeur de la classe parent à l'aide de la méthode `super()` et les bons arguments. Dans ce cas, l'appel doit être la première instruction du constructeur. Si le constructeur par défaut de la classe de base n'est pas défini alors la machine virtuelle lève une exception.

```
1     class Engine extends CarPart {
2       public Engine() {
3         super(); // appel du constructeur de CarPart superflu
4         ...
```

```
5       }
6       public Engine( float price ) {
7         super(price); // appel explicite d'un constructeur de CarPart.
8         ...
9       }
10    }
```

De la même façon, il est possible d'appeler un autre constructeur de la classe avec la méthode `this()`. Toutefois, elle doit être la première instruction du constructeur.

```
1     class Engine {
2       private _horsePower;
3       public Engine( ) {
4         this(55);
5         ...
6         }
7       public Engine( int horsePower ) {
8         _horsePower = horsePower;
9         ...
10      }
11    }
```

Il n'est donc pas possible de cumuler l'appel à différents constructeurs à l'intérieur d'un même constructeur.

2.6 Initialisation statique explicite

Java permet de grouper des initialisations statiques à l'intérieur d'un bloc spécial dans une classe. Le bloc est préfixé du mot clé `static` :

```
1     class Car {
2       static int i;
3       static Engine m;
4       static {
5         i = 47;
6         m = new Engine();
7         System.out.println("m et i initialisés");
8       }
9     }
```

Ce bloc est exécuté une fois : la première fois que l'on crée un objet de cette classe ou la première fois que l'on exécute une méthode statique de cette classe. En fait, quand l'unité de compilation correspondante est chargée.

2.7 Initialisation non-statique d'instance

Java fournit un bloc similaire anonyme pour initialiser les variables non-statiques de chaque objet. Il ressemble exactement au bloc d'initialisation statique mais sans le mot clé `static`.

```
class Mugs {
  private Mug c1;

  // bloc d'initialisation
  {
    c1 = new Mug();
    System.out.println("c1 initialized");
  }
}
```

Le bloc est exécuté à chaque fois qu'une instance est créée et avant l'appel du code du constructeur choisi. C'est une façon d'ajouter un bloc d'initialisation commun à tous les constructeurs.

Dans le cas d'une hiérarchie, le bloc non-statique est appelé après le constructeur de la super-classe.

```
public class Foo {
  public Foo() {
    System.out.println("Foo");
  }
}
public final class Bar extends Foo {
  {
    System.out.println("init Bar");
  }
  public Bar( int x ) {
    super(x);
    System.out.println("Bar");
  }
  public static void main( String[] args ) {
    new Bar(5);
  }
}
```

Le résultat de l'exécution donne :

```
Foo
init Bar
Bar
```


3. Le clonage d'objet

3.1 Copie profonde via le clonage

Par défaut la méthode `clone()` de la classe de base `Object` ne permet qu'une copie superficielle. Cependant, même si la méthode de clonage est définie pour toute classe, le clonage n'est pas automatiquement disponible. Pour que le clonage soit utilisable, il faut ajouter du code spécifique.

En effet, la méthode `clone()` est protégée dans la classe `Object`. Non

seulement cela signifie qu'elle n'est pas disponible par défaut pour le client qui utilise la classe, mais cela signifie aussi que vous ne pouvez pas appeler `clone()` par une référence de la classe de base. Ainsi, les deux lignes suivantes produisent une erreur de compilation :

```
1 Integer x = new Integer(1);
2 Integer y = x.clone();
```

Cependant, une classe dérivée peut appeler la méthode `clone()` de la classe de base. Elle fournit une duplication bit à bit de l'objet de la classe dérivée et donc seulement une copie superficielle.

Pour fournir la méthode `clone()`, une classe doit :

1. Implémenter l'interface `java.lang.Cloneable`. Cette interface est vide, il s'agit d'une « *tagging interface* » :
   `public interface Cloneable { }`
   Le compilateur vérifie que la classe implémente l'interface `Cloneable`. Sinon, il lève une exception `CloneNotSupportedException` dès que l'on accède à la méthode `clone()`.
2. Surcharger la méthode `clone()`,
3. La rendre publique,
4. Faire appel à la méthode `super.clone()` à l'intérieur de la méthode `clone()` pour produire une copie bit à bit de l'objet,
5. Ajouter du code pour faire la copie particulière des éléments de la classe si nécessaire.

Remarque

La méthode `clone()` définie dans une classe fonctionne aussi pour les classes dérivées sans avoir à la redéfinir. Ainsi, si B est une classe dérivée de A qui définit la méthode `clone()`, alors les lignes suivantes suffisent à créer un clone de l'objet b1 :

```
B b1 = new B(); B b2 = b1.clone();
```

produit une copie de b1, selon le code de `A.clone()`, même si `clone()` n'a pas été redéfinie dans B. (C'est la magie de Java !).

3.2 Copie profonde via la sérialisation

La sérialisation est aussi une manière de cloner un objet. Il suffit de sérialiser l'objet puis de le désérialisé pour avoir la copie. Cependant, le clonage est environ quatre fois plus rapide que la sérialisation..

```
1    public static Object copy( Serializable obj )
2                            throws IOException, ClassNotFoundException {
3       ObjectOutputStream out = null;
4       ObjectInputStream in = null;
5       Object copy = null;
6
7       try {
8          // write the object
9          ByteArrayOutputStream baos = new ByteArrayOutputStream();
10         out = new ObjectOutputStream(baos);
11         out.writeObject(obj);
12         out.flush();
13         // read in the copy
14         byte data[] = baos.toByteArray();
15         ByteArrayInputStream bais = new ByteArrayInputStream(data);
16         in = new ObjectInputStream(bais);
17         copy = in.readObject();
18      } finally {
19         out.close();
20         in.close();
21      }
22
23         return copy;
24      }
25   }
```

4. Réutiliser les classes

4.1 Redéfinition

Le C++ et le C# n'implémente pas la redéfinition de méthode par défaut qui autorise qu'une méthode d'une classe de base avec une signature donnée peut être redéfinie dans une classe dérivée avec la même signature. Java au contraire autorise la redéfinition par défaut. Pour se prémunir contre ce qui pourrait être une erreur de renommage, en Java vous devez ajouter l'annotation `@Override` au-dessus de la déclaration d'une méthode redéfinie. Ainsi le compilateur génère une erreur si le prototype diverge de celui de la méthode à redéfinir.

Si une méthode de base est déclarée privée, elle n'est pas visible dans une classe dérivée. Si le programmeur définit une méthode avec la même signature dans la classe dérivée, c'est une nouvelle méthode de la classe dérivée. Avec

raison, le compilateur ne signale aucune erreur.

4.2 Le mot clé « final »

4.2.1 Attribut final et variable finale

Un attribut ou une variable locale déclarés `final` correspond à la déclaration d'une constante.

```
1 private final int CST = 5;
```

4.2.2 Attribut « final » blanc

Un attribut final blanc est un attribut qui est déclaré comme final (donc constant) mais dont la valeur n'est pas fournie au moment de la définition. Sa valeur sera donnée plus tard, au moins avant sa première utilisation, et le compilateur vérifie cela.

Cela permet d'avoir un attribut constant dans une classe qui peut avoir des valeurs différentes selon les objets, mais avec une qualité de constante. La valeur est donnée la première fois qu'elle est affectée.

```
1 class Foo {
2    final int CST;
3    Foo() { CST = 1; }
4    Foo( int val ) { CST = val; }
5    bar() { CST = 2; } // erreur de redéfinition levée à la compilation
6 }
```

4.2.3 Méthode finale

Il y a deux raisons pour définir une méthode finale :

- Sécurité : pour empêcher les redéfinitions de la méthode.
- Efficacité : permettre au compilateur de remplacer tout appel d'une méthode par son code (*inline method*).

Il faut utiliser et abuser de mot clé. Mais attention : les méthodes finales sont un frein à leur testabilité. Elles ne peuvent pas être doublées par des Mocks.

4.2.4 Argument final

Un argument final ne peut pas être modifié, mais il est possible de changer

l'objet référencé par l'argument lorsqu'il s'agit d'une référence sur un objet. En Java, il n'y a aucun moyen d'empêcher cela, contrairement à C++.

```
1 void with( final Toto g ) {
2     g = new Toto();  // Erreur
3     g.x = 2;         // pas d'erreur
4 }
```

4.2.5 Classe finale

Pour des raisons de sécurité et de sûreté, il peut être opportun d'interdit de construire des sous-classes d'une classe donnée. On pense, évidemment, à l'exemple de l'implémentation du patron Singleton. Dans l'exemple suivant, la classe `FooBar` ne peut plus être dérivée.

```
public final class FooBar{ }
```

Attention, là encore, les classes finales ne peuvent pas être doublées dans les tests.

5. Interfaces et classes internes

5.1 Interface

Une interface peut contenir des attributs, mais ceux-ci doivent être `static` et `final`, c'est-à-dire des constantes. C'est utile pour la gestion de projet quand une interface est utilisée entre deux équipes de développements et quelques constantes doivent être partagées.

```
1 interface instrument {
2     int i = 5;              // automatiquement public & static & final
3     void play( Note n );    // automatiquement public
4     String what();          // automatiquement public
5 }
```

Ces constantes peuvent être initialisées avec des expressions non constantes. Par exemple :

```
1 public interface RandVals {
2     int rint = (int)(Math.random() * 10);
3     long rlong = (long)(Math.random() * 10);
4     float rfloat = (float)(Math.random() * 10);
5     double rdouble = Math.random() * 10;
6 }
```

Ainsi, une interface est un moyen de créer des constantes liées.

```
1 public interface Month {
```

```
2     int  JANUARY = 1, FEBRUARY = 2, MARCH = 3, APRIL = 4, MAY = 5, JUNE = 6,
3       JULY = 7, AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
4       NOVEMBER = 11, DECEMBER = 12;
5 }
```

Mais on lui préférera quand même les énumérations.

```
1 public Enum Month {
2     JANUARY=1, FEBRUARY, MARCH, APRIL, MAY, JUNE,
3     JULY, AUGUST, SEPTEMBER, OCTOBER,
4     NOVEMBER, DECEMBER
5 }
```

5.2 Méthodes par défaut dans les interfaces (Defender Methods)

Les interfaces peuvent contenir l'implémentation de méthodes statiques. Les méthodes implémentées sont préfixées du mot clé `default`. C'est ce que l'on appelle des *Defender Methods*.

```
1 public interface A {
2     default void printSomething() {
3       System.out.println("something");
4     }
5 }
6
7 public class AImplementation implements A {}
```

La classe `AImplementation` peut utiliser la méthode `printSomething()`.

```
1 public class Example {
2     public static void main(String[] args) {
3       new AImplementation().printSomething();
4     }
5 }
```

5.2.1 Résolution des conflits

Que se passe-t-il si deux interfaces implémentent la même méthode par défaut et qu'une classe implémente ces deux interfaces ?

```
1 public interface A {
2     default void printSomething() {
3       System.out.println("Print something");
4     }
5 }
6 public interface B {
7     default void printSomething() {
8       System.out.println("Print something else");
9     }
10 }
11 public class C implements A, B {}
```

Le code précédent ne compile pas :

```
class C inherits unrelated defaults for printSomething() from types A and B
```

Pour résoudre le conflit, il faut faire référence explicitement à une des méthodes par défaut :

```
1 public class D implements A,B {
2     @Override
3     public void printSomething() {
4         B.super.printSomething();
5     }
6 }
```

5.2.2 Traits

Les méthodes par défaut permettent d'implémenter des *traits* en Java. Un trait offre un moyen de réutiliser le code d'une méthode dans deux classes indépendantes. Un trait est implémenté par une interface qui encapsule un ensemble cohérent de méthodes à caractère transverse et réutilisable. Plutôt que de faire une classe avec le code d'une méthode partagée par plusieurs classes qui est ensuite utilisée par agrégation, les interfaces sont un moyen plus propre de réaliser cela.

L'utilisation classique d'un trait est une interface composée de :

- une méthode abstraite qui fait le lien avec la classe sur laquelle il est appliqué,
- un certain nombre de méthodes additionnelles, dont l'implémentation est fournie par le trait lui-même car elles sont directement dérivables du comportement de la méthode abstraite.
- La classe qui implémente le trait redéfinit la méthode abstraite et profite des implémentations fournies par défaut.

Prenons l'exemple de l'interface `Comparable` en Java. Cette interface déclare une unique méthode, `compareTo()`, qui permet au développeur de spécifier la position relative de deux objets. Cette méthode est très utile, mais elle renvoie un `int`, ce qui n'est pas très explicite. Des méthodes comme `isBefore()` / `isAfter()` renvoyant des booléens, seraient plus parlantes. Comme l'interface `Comparable` appartient au JDK, nous ne pouvons pas la modifier, mais il est toujours possible de l'étendre. Elle ne contiendra que des méthodes par défaut s'appuyant sur la méthode `compareTo()` héritée de `Comparable`.

```
1 public interface Orderable<T> extends Comparable<T> {
2     // La méthode compareTo() est définie dans la super-interface Comparable
3
4     public default boolean isAfter( T other ) {
```

```
5       return compareTo(other) > 0;
6     }
7
8     public default boolean isBefore( T other ) {
9       return compareTo(other) < 0;
10    }
11
12    public default boolean isSameAs( T other ) {
13      return compareTo(other) == 0;
14    }
15  }
```

On peut l'appliquer à une classe...

```
1  public class Person implements Orderable<Person> {
2     private final String _name;
3
4     public Person( String name ) {
5       _name = name
6     }
7
8     @Override
9     public int compareTo( Person other ) {
10      return _name.compareTo(other._name);
11    }
12  }
```

... qui bénéficie aussitôt des nouvelles méthodes `isBefore()` et `isAfter()`.

```
1  public class Test {
2     public static void main(String[] args) {
3       Person laurel = new Person("Laurel");
4       Person hardy = new Person("Hardy");
5       System.out.println("Laurel vs Hardy: " + laurel.compareTo(hardy));
6       System.out.println("Laurel >  Hardy: " + laurel.isAfter(hardy));
7       System.out.println("Laurel <  Hardy: " + laurel.isBefore(hardy));
8       System.out.println("Laurel == Hardy: " + laurel.isSameAs(hardy));
9     }
10 }
```

Résultat :

```
1 Laurel compareto Hardy: 4
2 Laurel >  Hardy: true
3 Laurel <  Hardy: false
4 Laurel == Hardy: false
```

5.3 Classe de niveau supérieur

On déclare une seule classe de niveau supérieur par unité de compilation. Une unité de compilation correspond à un fichier Java portant le nom de cette classe. C'est aussi la seule classe publique de l'unité de compilation. Les autres classes du fichier ont au maximum une visibilité de type paquet.

5.4 Classe interne

Une **classe interne** est une classe définie à l'intérieur d'une autre classe. Dépendant de la façon dont elle est définie, une classe interne peut être d'un des quatre types :

1/ Une classe interne locale.

2/ Une classe interne statique.

3/ Une classe interne anonyme.

4/ Une classe membre.

5.4.1 Classe interne locale

Les classes locales sont équivalentes aux variables locales, dans le sens où elles sont créées et utilisées à l'intérieur d'un bloc. Une fois que l'on déclare une classe dans un bloc, elle peut être instanciée autant de fois que souhaité dans ce bloc. Comme les variables locales, les classes locales ne sont pas autorisées à être déclarées publiques, protégées, privées ou statiques.

Voici un exemple de code :

```
1 public class Outer1 {
2       class ListListener implements ItemListener {
3         private List<String> _list;
4         public ListListener( List<String> l ) {
5           _list = l;
6         }
7         @Override
8         public void itemStateChanged( ItemEvent e ) {
9           String s = e.getItemSelected();
10          doSomething(s);
11        }
12      }
13      private List<String> _list1 = new ArrayList<>();
14      private list<String> _list2 = new ArrayList<>();
15
16      _list1.addItemListener(new ListListener(_list1));
17      _list2.addItemListener(new ListListener(_list2));
18    }
19 }
```

Remarques

- Pour référencer l'objet correspondant à la classe externe à l'intérieur d'une méthode de la classe interne, il faut utiliser `Class.this`.

```
1 public class Outer2 {
2     private List<Inner2> _list1 = new ArrayList<>();
```

```
3
4     class Inner2 {
5       public Inner2() {
6         doSomething(Outer2.this);
7       }
8       public void doSomething( Outer2 o ) {
9       }
10    }
11
12    public void Outer2() {
13      _list1.add(new Inner2());
14    }
15 }
```

- Quelques fois, on souhaite créer des objets d'une des classes internes non statiques à partir d'une instance de la classe supérieure. Pour faire cela, on doit fournir une référence aux autres objets des classes externes dans l'expression de la création par un `new` spécial comme cela :

```
1 public class Outer3 {
2     class Contents {
3       private int _i = 11;
4       public int value() { return _i; }
5     }
6     class Destination {
7       private String _label;
8       Destination( String whereTo ) {
9         _label = whereTo;
10      }
11      String readLabel() { return _label; }
12    }
13    public static void main( String[] args ) {
14      Outer3 p = new Outer3();
15      Outer3.Contents c = p.new Contents();
16      Outer3.Destination d = p.new Destination("Tanzania");
17    }
18 }
```

La profondeur d'imbrication d'une classe importe peu :

```
1 class MMA {
2     private void f() {}
3     class A {
4       private void g() {}
5         public class B {
6           void h() {
7             g();
8             f();
9           }
10        }
11      }
12    }
13 }
14 public class MultiNestingAccess {
15    public static void main( String[] args ) {
16      MNA mna = new MNA();
17      MNA.A mnaa = mna.new A();
18      MNA.A.B mnaab = mnaa.new B();
19      mnaab.h();
```

```
20    }
21 }
```

- Il est possible de faire de l'héritage de classes internes statiques :

```
1 class WithInner {
2     class Inner {}
3 }
4 public class InheritInner extends WithInner.Inner {
5     InheritInner(WithInner wi) {
6        wi.super();
7     }
8     public static void main( String[] args ) {
9        WithInner wi = new WithInner();
10       InheritInner ii = new InheritInner(wi);
11    }
12 }
```

5.4.2 Les classes internes statiques

Les classes internes ont accès à toutes les méthodes et tous les attributs de la classe englobante. Si vous n'avez pas besoin d'une relation entre la classe interne et les objets de la classe englobante, vous pouvez faire une classe interne statique (appelée classe imbriquée). Dans ce cas seulement, la méthode et les champs sont accessibles par d'autres classes que la classe englobante et sans création d'un objet de la classe englobante. Il est également possible de créer des objets de la classe imbriquée sans créer d''objet de la classe englobante.

Une classe de niveau supérieur imbriquée est une classe membre avec le modificateur `static`. C'est une classe comme n'importe quelle autre classe de niveau supérieur, sauf qu'elle est déclarée dans une autre classe ou interface. Elle est généralement utilisée comme un moyen de regrouper des classes en relation sans la création d'une nouvelle unité de compilation.

Si votre classe principale a quelques classes d'assistance plus petites qui peuvent être utilisées en dehors de la classe principale et ne font sens qu'avec votre classe principale, c'est une bonne idée d'en faire des classes de niveau supérieur imbriquées. Pour faire référence à la classe imbriquée en dehors de la classe de niveau supérieure, il faut écrire : `ClaseSupérieure.ClasseImbriquée`. Voici un exemple :

```
1 public class Filter {
2     ArrayList<Criterion> criteria = new ArrayList<Criterion>();
3     public addCriterion( Criterion c ) {
4        criteria.addElement(c);
5     }
6     public boolean isTrue( Record rec ) {
```

```
7       for (Enumeration e=criteria.elements(); e.hasMoreElements();) {
8         if (!((Criterion)e.nextElement()).isTrue(rec)) {
9           return false;
10        }
11      }
12      return true;
13    }
14    public static class Criterion {
15      String colName, colValue;
16      public Criterion( String name, String val ) {
17        colName = name; colValue = val;
18      }
19      public boolean isTrue( Record rec ) {
20        String data = rec.getData(colName);
21        return (data.equals(colValue));
22      }
23    }
24  }
```

Et quand on veut l'utiliser :

```
1 Filter f = new Filter();
2 f.addCriterion(new Filter.Criterion("SYMBOL", "SUNW"));
3 f.addCriterion(new Filter.Criterion("SIDE", "BUY"));
4 .....
5 if (f.isTrue(someRec)) //do some thing ...
```

Quelques particularités

- Il est impossible de déclarer une classe de niveau supérieur de type statique. Cela n'a pas de sens en Java puisqu'elle est déjà accessible de l'extérieur.
- Il est possible d'utiliser une classe imbriquée dans une interface.

```
1 interface IInterface {
2     static class Inner {
3       public Inner() {}
4       void f() {}
5     }
6 }
7 class A implements Iinterface.Inner { }
```

Il est ensuite possible d'utiliser `Iinterface.Inner`.

5.4.3 Classe interne anonyme

Les classes anonymes sont déclarées et instanciées dans la même instruction. Elles n'ont pas de nom, et ne peuvent être instanciées qu'une seule fois. L'exemple qui suit définit une classe dérivée anonyme de la classe `ActionListener` qui redéfinit la méthode `actionPerformed()`.

```
1 okButton.addActionListener( new ActionListener() {
2     @Override
```

```
3     public void actionPerformed( ActionEvent e ) {
4       dispose();
5     }
6  });
```

Parce qu'une classe anonyme n'a pas une déclaration normale où il est possible d'introduire le mot clé `static`, elle ne peut pas être déclarée statique.

Une méthode qui retourne une classe interne anonyme :

```
1  public class Outer4 {
2     public Contents cont() {
3       return new Contents() {
4         private int _i = 11;
5         public int value() { return _i; }
6       }; // point-virgule obligatoire
7     }
8     public static void main( String[] args ) {
9       Outer4 p = new Outer4();
10      Contents c = p.cont();
11      ... c.value()...
12    }
13 }
```

Lorsque l'on définit une classe interne anonyme et que l'on veut utiliser une variable d'une classe englobante dans la classe interne anonyme, le compilateur impose que cette variable soit de type `final`.

```
1  public class Outer5 {
2     public Destination dest( final String dest ) {
3       return new Destination() {
4         private String _label = dest;
5         public String readLabel() { return _label; }
6       };
7     }
8
9     public static void main( String[] args ) {
10      Outer5 p = new Outer5();
11      Destination d = p.dest("Tanzania");
12    }
13 }
```

Étant donné qu'une classe anonyme n'a pas de nom, elle ne peut avoir de constructeur. Si on veut une initialisation anonyme pour remplacer un constructeur, il faut utiliser le bloc d'initialisation anonyme :

```
1  public class Outer6 {
2     public Destination dest(final String dest, final float price) {
3       return new Destination() {
4               private int _cost;
5               { // A l'intérieur du bloc d'initialisation de l'instance
6                 _cost = Math.round(price);
7                 if (_cost > 100) {
8                   System.out.println("Over budget!");
9                 }
10              }
11              private String _label = dest;
```

```
12              public String readLabel() { return _label; }
13        };
14     }
15     public static void main( String[] args ) {
16        Outer6 p = new Outer6();
17        Destination d = p.dest("Tanzania", 101.395F);
18     }
19  }
```

5.4.4 Classe membre

Les classes membres sont définies à l'intérieur du corps d'une méthode. On peut déclarer une classe membre partout dans le corps de la classe externe. Elles sont utiles quand on souhaite utiliser des variables et des méthodes de la classe externe sans délégation explicite.

```
1  public class Outer7 {
2     public Destination dest(String s) {
3        class Pdestination implements Destination {
4           private String _label;
5           private Pdestination( String whereTo ) {
6              _label = whereTo;
7           }
8           public String readLabel() { return _label; }
9        }
10       return new PDestination(s);
11    }
12    public static void main( String[] args ) {
13       Outer7 p = new Outer7();
14       Destination d = p.dest("Tanzania");
15    }
16 }
```

Lorsque l'on déclare une classe membre, il n'est possible d'instancier cette classe membre que dans le contexte d'un objet de la classe externe. Elle est invisible des autres unités de compilation. Si vous souhaitez supprimer cette restriction, vous déclarez la classe membre comme une classe statique. Lorsque l'on déclare une classe membre avec le modificateur `static`, elle devient une classe de niveau supérieur imbriquée et peut être utilisée comme une classe de niveau supérieur normale.

6. Gestion des erreurs avec les exceptions

Les exceptions font partie intégrante de la programmation orientée objet et doivent être utilisées intensivement. Le langage Java est basé sur les exceptions et le compilateur oblige à les utiliser.

6.1 Définition

Une condition exceptionnelle est un problème qui empêche la poursuite d'une méthode ou d'une partie du code. Il est important de distinguer une condition exceptionnelle d'un problème normal, dans lequel vous avez suffisamment d'information dans le contexte actuel pour faire face au problème. Avec une condition d'exception, vous ne pouvez pas continuer le traitement parce que vous n'avez pas les informations nécessaires pour régler le problème dans le contexte actuel.

Une directive importante dans la gestion des exceptions est de *ne pas attraper une exception, sauf si vous savez quoi faire avec elle*.

6.2 Signature de méthode

Bien que la spécification d'exception soit imposée par le compilateur lors de l'héritage, elle ne fait pas partie de la signature de méthode, qui ne comprend que le nom de la méthode et la liste des arguments. De plus, il est possible d'utiliser une classe d'exception dérivée dans les méthodes redéfinies comme dans l'exemple ci-dessous :

```
1 public class H {
2     public void a() throws IOException {
3     }
4 }
5
6 public class G extends H {
7     @Override
8     public void a() throws EOFException {
9     }
10 }
```

Cela est possible si `EOFException` est une classe dérivée de `IOException`.

6.3 Clause Try Catch

Les exceptions sont avalées par la clause `catch` la plus proche dans l'ordre dans lequel elles sont écrites. La plus proche doit être compris selon le sens de l'héritage.

```
1 public class A {
2     @Override
3     try {
4     // Code qui lève les exceptions EOFException ou IOException
5     // où EOFException est dérivé de IOException
```

```
6       } catch (EOFException e) {
7         //
8       } catch (IOException e) {
9         //
10      }
11  }
```

6.4 La classe Throwable

La classe `Throwable` est la superclasse de toutes les erreurs et toutes les exceptions dans le langage Java. Il y a deux classes dérivées : `Error` et `Exception`.

- Une `Error` indique un problème sérieux qu'une application ne devrait raisonnablement pas capturer dans un `try-catch`. Ce type d'exception est révélateur d'une erreur dans le programme.
- Une `Exception` doit être gérée dans un '`try-catch`'. Le compilateur vous oblige à cela, exception faite des exceptions de type `RuntimeException`. qui font partie des *unchecked exceptions* et ne nécessitent pas d'être capturées.

Les méthodes de la classe `Throwable` :

- `String getMessage()`
- `String getLocalizedMessage`: Récupérer le message détaillé dans une langue considérée.
- `void printStackTrace()`
- `Throwable fillStackTrace()` : enregistre dans cet objet les informations sur l'état actuel de pile. C'est utile quand une application réémet une erreur ou une exception à partir de cette exception. On garde ainsi trace du contexte de l'erreur et pas celui de la réémission.

6.5 Passer les exceptions à la console

L'exception suivante ne sera pas attrapée et donc sera passée à la console.

```
1 public static void main( String args[] ) thows IOException {
2     FileInputStream f = new FileInputStream("main.txt");
3     f.close();
4 }
```

6.6 Convertir une exception

Si « je n'ai aucune idée de quoi faire avec cette exception, mais je ne veux

pas l'avaler ou afficher un message banal », on peut transformer cette exception en erreur de type `RuntimeException` que le compilateur n'oblige pas à attraper.

```
1     try {
2        ...
3     } catch (Exception e) {
4        throw new RuntimeException(e);
5     }
```

6.7 La clause « finally »

Dans la classe `try-catch`, qu'une exception soit levée ou pas, la clause `finally` est toujours exécutée.

```
1 try {
2     // Activités dangereuses qui peuvent lever les exceptions A, B or C
3 } catch (A a) {
4     // Code pour l'exception A
5 } catch (B b ) {
6     // Code pour l'exception B
7 } catch (C c) [
8     // Code pour l'exception C
9 } finally {
10    // Code qui est toujours exécuté.
11 }
```

Si toutes les clauses `catch` utilisent le même code, alors il faut les regrouper en une seule avec l'opérateur "|".

7. Interroger les types (Run-Time Type Identification)

Le but des RTTI est de déterminer le type d'un objet quand on ne possède qu'une référence de la classe de base vers cet objet.

7.1 L'opérateur 'instanceof' et ses équivalents

L'opérateur `instanceof` retourne vrai si l'objet sur lequel on l'applique est de la classe spécifiée ou d'une de ses super-classes. Il retourne faux si on lui passe un objet `null`.

```
1 Base x = null;
2 Base y = new Derived();          // Derived est un classe dérivée de Base.
3 x instanceof Base;               // false parce que x est null
4 y instanceof Base;               // true
5 y instanceof Derived;            // true
```

La méthode `class.isInstance(Object)` est l'équivalent dynamique de l'opérateur `instanceof`.

```
1 Base x = new Derived();
2 Base.class.isInstance(x)                → true
```

En général, on utilise `instanceof` quand on connaît à l'avance le type de classe à tester.

```
1     for (int i = 0; i < pets.size(); i++) {
2       Object o = pets.get(i);
3       if (o instanceof Cat) {
4         ((Counter)h.get("class Cat")).i++;
5       }
6       if (o instanceof Dog) {
7         ((Counter)h.get("class Dog")).i++;
8       }
9       …
10    }
```

La version avec `isInstance()` élimine le besoin de l'opérateur `instanceof` et on peut l'utiliser dynamiquement dans une boucle.

```
1  Class[] petTypes = new Class[5];
2  petTypes[0] = Cat.class;
3  petTypes[1] = Dog.class;
4
5  for (int i = 0; i < pets.size(); i++) {
6    Object o = pets.get(i);
7    for (int j = 0; j < petTypes.length; ++j) {
8      if (petTypes[j].isInstance(o)) {
9        …
10     }
11   }
12 }
```

Une autre méthode `getClass()` retourne le type exact sans prendre en compte la hiérarchie.

```
1  x.getClass() == Base.class      → false puisque x.getClass() = "Derived"
2  x.getClass().equals(Base.class)       → false
3  x.getClass() == Derived.class   → true
```

7.2 Le paquet « Reflection »

Le paquet `reflection` apporte des informations dynamiques sur les classes. Il rend le langage réflexif. La réflexivité est le moyen d'accéder et modifier dynamiquement à la représentation d'une classe. Il définit les classes `Field`, `Method` et `Constructor` qui implémentent chacune l'interface `Member` :

- `String Class.getName()`
- `Class[] Class.getInterfaces()`
- `boolean Class.isInterface()`
- `Class Class.getSuperclass()`

7.3 Chargement dynamique de classe

L'instruction `Class.forName(String)` charge dynamiquement une classe dans un programme :

```
1  Class c = Class.forName("c10.Dog");
2  Object = c.newInstance();
```

8. Sérialisation

La sérialisation est intéressante car elle permet d'implémenter la persistance légère, ce qui permet à un objet d'avoir une durée de vie qui dépasse celle du temps d'exécution du programme. Il faut noter qu'aucun constructeur, pas même le constructeur par défaut, n'est appelé dans le processus de désérialisation d'un objet de type `Serializable`.

8.1 Le mot clé transient

Les objets sérialisables sont ceux qui implémentent l'interface `Serializable` alors que les non-sérialisables sont marqués avec le mot-clé `transient` (*transitoire, éphémère*). Pour éviter que des parties sensibles d'un objet serialisable ne soient sérialisées, il faut ajouter sur chacun des attributs à protéger le mot clé `transient` qui dit "ne pas de soucier de sauver ou restaurer ces attributs, je m'en occupe."

```
private transient String _secretCode;
```

8.2 Contrôler la serialization

Par défaut, Java prend en charge la sérialisation d'objets par un code binaire propriétaire. Mais, il fournit également un mécanisme soit pour (1) étendre ce format initial (interface `Serializable`) soit pour (2) réécrire le processus de sérialisation (interface `Externisable`).

8.2.1 L'interface Serializable

Les classes qui nécessitent un traitement spécial au cours des processus de sérialisation et de désérialisation doivent mettre en œuvre des méthodes spéciales de l'interface `Serializable` dont les signatures sont les suivantes :

```
private void writeObject(ObjectOutputStream out) throws IOException
```

```
private void readObject(ObjectInputStream in) throws IOException,
  ClassNotFoundException
private void readObjectNoData() throws ObjectStreamException
```

On remarque qu'elles sont définies comme privées, ce qui signifie qu'elles ne sont appelées que par d'autres membres de cette classe. Cependant, elles ne sont pas réellement appelées par d'autres membres de cette classe, mais par les méthodes `writeObject()` `readObject()` d'objets de type `ObjectOutputStream` et `ObjectInputStream`. On peut se demander comment ces objets ont accès à des méthodes privées de votre classe. Considérons simplement que c'est la magie de la sérialisation Java.

8.2.2 L'interface Externalizable

Les méthodes `writeExternal()` et `readExternal()` de l'interface `Externalizable` sont implémentées par une classe pour donner à la classe le contrôle sur le format et le contenu du flux d'un objet et de ses supertypes.

```
public void writeExternal(ObjectOutput out) throws IOException;
public void readExternal(ObjectInput in) throws IOException,
  java.lang.ClassNotFoundException;
```


9. Concurrence

9.1 Résoudre les conflits de partage de ressources

Une ressource partagée est généralement un morceau de la mémoire sous la forme d'un objet, d'un fichier, d'un port d'entrée/sortie, ou de quelque chose comme une imprimante.

9.1.1 Méthode critique

Les collisions peuvent être évitées en faisant des méthodes **synchronisées**.

```
1  class A {
2     synchronized void f();
3     synchronized void g();
4  }
```

Lorsqu'une méthode synchronisée est appelée, l'objet appelant est verrouillé et aucune autre méthode synchronisée de cet objet ne peut être appelée jusqu'à ce que la première se termine et libère le verrou. Si `f()` est appelée pour un objet alors `g()` ne peut pas être appelé par le même objet jusqu'à ce que `f()`

soit terminée.

Il n'y a qu'un seul verrou par classe, de sorte qu'une méthode statique synchronisée peut empêcher toutes les autres méthodes d'un accès simultané.

9.1.2 Section critique

Pour se prémunir contre l'accès simultané à une partie d'une méthode de plusieurs threads, il faut utiliser le mot clé `synchronized` à l'intérieur de la méthode :

```
1  public void method() {
2    synchronized(object) {
3    }
4  }
```

Un bloc synchronisé doit avoir un objet sur lequel se synchroniser, et très souvent, on prend tout simplement l'objet lui-même : `synchronized(this)`.

9.2 Variable volatile

Le mot clé `volatile` devant une variable ou un attribut empêche le compilateur d'effectuer des optimisations telles que déplacer la variable de la pile vers le tas. Il faut utiliser une variable volatile quand elle est modifiée ou testée par des threads distincts.

9.3 Création et exécution de thread

La façon la plus simple de créer un thread est d'hériter de la classe `java.lang.Thread` qui a tout le code nécessaire pour créer et exécuter des threads. La méthode la plus importante est `run()` qui doit être redéfinie. Chaque instance de thread est exécutée en utilisant la méthode `start()`.

```
1  public class SimpleThread extends Thread {
2    private volatile int _countDown = 5;
3    private static int _threadCount = 0;
4    private int _threadNumber = ++_threadCount;
5    public SimpleThread() {
6      System.out.println("Création du thread : " + _threadNumber);
7    }
8    public void run() {
9      while (true) {
10       System.out.println("Thread " +
11                   _threadNumber + "(" + _countDown + ")");
12       if (--_countDown == 0) {
13         return;
```

```
14          }
15        }
16     }
17     public static void main( String[] args ) {
18        for (int i = 0; i < 5; i++) {
19           new SimpleThread().start();
20        }
21        System.out.println("Tous les threads lancés");
22     }
23  }
```

Si une classe hérite déjà d'une autre classe, on peut implémenter l'interface `Runnable` :

```
1  public class SimpleThread implement Runnable {
2     public SimpleThread() {
3        ..
4     }
5     @Override
6     public void run() {
7     ...
8     }
9     public static void main( String[] args ) {
10       …
11       new Thread(new SimpleThread(), "name").start();
12    }
13 }
```

Lorsque l'on crée et exécute un thread, même sans garder de référence sur cet objet, le ramasse-miette ne peut pas le nettoyer. C'est parce qu'un thread garde une référence vers lui-même lors de l'exécution. Cependant, après la fin de son exécution, le ramasse-miette peut le nettoyer.

9.4 Thread de type démon

Un thread de type démon est un thread qui est supposé fournir un service général en tâche de fond tant que le programme fonctionne, mais qui ne fait pas partie de l'essence même du programme. Pour définir un thread de type démon, il faut appeler la méthode `isDaemon(true)` avant de lancer le thread. Tous les threads créés par la suite par un thread démon sont eux-mêmes des démons.

9.5 États d'un thread

Un thread peut être dans l'un des quatre états suivants :

- nouveau
- exécutable

- mort
- bloqué / endormi.

9.5.1 Arrêter un thread

La façon normale pour tuer un thread est de stopper l'exécution de la méthode `run()`. Il faut pour cela utiliser un booléen qui indique quand terminer la méthode `run()`. Très souvent, le booléen doit être de type `volatile`.

9.5.2 Yielding

La méthode `yield()` permet de donner au thread un indice sur le fait que le thread en a fait assez et que d'autres threads peuvent avoir le CPU. Il devient moins prioritaire.

```
1  public void run() {
2    while (true) {
3      System.out.println(this);
4      if (--_coundown == 0 ) { return; }
5      yield();
6    }
7  }
```

Toutefois, `yield()` n'est utile que dans de rares situations où l'on souhaite faire des réglages fins pour optimiser la vitesse d'exécution d'une application.

9.5.3 Sleeping

La méthode `sleep()` cesse l'exécution d'un thread pendant un temps déterminé donné de millisecondes. En fait, la seule garantie est que le thread va s'arrêter au moins le laps de temps spécifié, mais il peut prendre plus de temps avant de reprendre son exécution parce que l'ordonnanceur de threads prend du temps supplémentaire pour le relancer.

La méthode doit être placée à l'intérieur d'un `try-catch` car il est possible que `sleep()` soit interrompue avant son expiration. Cela peut arriver si un autre objet a une référence au thread et appelle `interrupt()`.

En général, il est préférable d'utiliser `wait()` lorsque l'on veut réactiver un thread suspendu à l'aide `interrupt()`.

9.5.4 Priority

La méthode `setPriority()` est utilisée pour affecter une priorité à un thread. Bien que le JDK définisse 10 niveaux de priorité, cela n'est pas garanti pour tous les systèmes d'exploitation. La seule approche est de se contenter des trois niveaux MAX_PRIORITY, NORM_PRIORITY et MIN_PRIORITY pour définir les niveaux de priorité.

9.5.5 Jonction de thread

Un thread peut appeler la méthode `join()` sur un autre thread pour attendre la fin du premier thread avant de d'exécuter le second. Le lien peut être suspendu en appelant `interrupt()` sur le thread appelant.

9.5.6 Waiting

Il est important de comprendre que `sleep()` ne libère pas le verrou lorsqu'il est appelé. Par contre, la méthode `wait()` libère le verrou, ce qui signifie que d'autres méthodes synchronisées de l'objet thread peuvent être appelées pendant un `wait()`. Un `wait()` continue indéfiniment jusqu'à ce que le thread reçoive un `notify()` ou `notifyAll()`.

Toutes ces méthodes comme `sleep()` font partie de la classe de base `Object` et pas de la classe `Thread`. C'est parce qu'elles manipulent le verrou qui fait également partie de tous les objets.

Quand un objet se notifie lui-même, on doit le faire à l'intérieur d'un bloc synchronisé qui acquiert le verrou pour l'objet :

```
1  synchronized(x) {
2    x.notify();
3  }
```

La méthode `sleep()` peut être appelée à l'intérieur d'une méthode non synchronisée, puisqu'elle ne manipule pas de verrou.

9.5.7 Exemple concret : le problème du dîner des philosophes (E. Dijktra)

```
1    public class Philosopher implements Runnable {
2      private final int _id;
3      private final int _ponderFactor;
4      private Chopstick _left;
5      private Chopstick _right;
6      private Random _rand = new Random(47);
```

```
7       public Philosopher( Chopstick left, Chopstick right,
8                           int ident, int ponder ) {
9          _left = left;
10         _right = right;
11         _id = ident;
12         _ponderFactor = ponder;
13      }
14      public void run() {
15         try {
16            while (!Thread.interrupted()) {
17               System.out.println(this + " " + "thinking");
18               pause();
19               // Le philosophe devient affamé
20               System.out.println(this + " " + "prend à droite");
21               _right.take();
22               System.out.println(this + " " + "prend à gauche");
23            _left.take();
24               System.out.println(this + " " + "mange");
25               pause();
26               _right.drop();
27               _left.drop();
28            }
29         } catch (InterruptedException e) {
30            System.out.println(this + " " + "exiting via interrupt");
31         }
32      }
33      public String toString() {
34         return "Philosophe n°" + _id;
35      }
36      private void pause() throws InterruptedException {
37         if (_ponderFactor == 0) {
38            return;
39         }
40         TimeUnit.MILLISECONDS.sleep(_rand.nextInt(_ponderFactor * 250));
41      }
42   }
43   class Chopstick {
44      private boolean _taken = false;
45      public synchronized void take() throws InterruptedException {
46         while (_taken) {
47            wait();
48         }
49         _taken = true;
50      }
51      public synchronized void drop() {
52         _taken = false;
53         notifyAll();
54      }
55   }
56   class DeadlockingDiningPhilosophers {
57      public static void main( String[] args ) throws Exception {
58         final int ponder = 5;
59         final int nbOfPhilosophers = 5;
60         ExecutorService exec = Executors.newCachedThreadPool();
61         Chopstick[] sticks = new Chopstick[nbOfPhilosophers];
62         for (int i = 0; i < nbOfPhilosophers; i++) {
63            sticks[i] = new Chopstick();
64         }
65         for (int i = 0; i < nbOfPhilosophers; i++) {
66         exec.execute(new Philosopher(sticks[i]
```

```
67                                  sticks[(i + 1) % nbOfPhilosophers],
68                                  i, ponder));
69          }
70          System.out.println("Appuyer sur 'Enter' pour quitter");
71          System.in.read();
72          exec.shutdownNow();
73      }
74  }
```

9.6 Utiliser les pipes entre threads

Il faut pour cela utiliser les classes `PipeWriter` et `PipeReader`.

10. Collections d'objets

10.1 Tableau

Le tableau est le moyen le plus efficace pour stocker et accéder aléatoirement à une séquence de références d'objet. En Java, il y a une vérification des limites. Lorsqu'un index est hors limite, Java lève une exception de type `ArrayIndexOutOfBoundsException`.

Quand un tableau est créé en tant que donnée membre, ses références sont automatiquement initialisées à `null`.

10.1.1 Définition de tableau

Deux définitions sont possibles, une traditionnelle à la C et une spécifique Java qui est à privilégier :

```
1  int a[]; // Notation à la C
2  int[] a; // Notation Java
```

10.1.2 Tableau multi-dimensionnel

IL ya plusieurs façons d'initialiser un tableau umulti-dimensionnel.

Première forme d'initialisation :

```
1  int[][] a1 = {
2    { 1, 2, 3, },
3    { 4, 5, 6, },
4  };
```

Seconde forme d'initialisation :

```
1  int[][][] a2 = new int[2][2][4]; // → a2[0][0][0] = 0
```

Troisième forme d'initialisation :

```
1  int[][][] a3 = new int[pRand(7)][][];
2    for (int i = 0; i < a3.length; i++) {
3      a3[i] = new int[pRand(5)][];
4      for (int j = 0; j < a3[i].length; j++) {
5        a3[i][j] = new int[pRand(5)];
6      }
7  }
```

D'autres formes d'initialisation :

```
1  Bidule[] a; // a = null.
2  a = new Bidule[] {
3    new Bidule(), new Bidule()
4  };
5  Bidule[] b = new Bidule[5]; // chaque Bidule[i] = nulls
6  Bidule[] c = new Bidule[4];
7  for (int i = 0; i < c.length; i++) {
8    c[i] = new Bidule();
9  }
10 Bidule[] d = {
11   new Bidule(), new Bidule(), new Bidule()
12 };
```

10.1.3 Performances

Pour des raisons d'efficacité, il est toujours préférable d'utiliser des tableaux monodimensionnels plutôt que multidimensionnels. Le gain en temps d'exécution est réellement très conséquent. Il suffit de refaire l'accès aux indices par calcul. Par exemple, la version multidimensionnelle est :

```
1  int[][] a4 = new int[15][5];
2  for (int i = 0; i < 15; i++) {
3    for (int j = 0; j < 5; j++) {
4      a4[i][j]= 1;
5    }
6  }
```

et la version monodimensionnelle équivalente est :

```
1  int[] a5 = new int[15 * 5]
2  for (int i = 0; i < 15; i++) {
3    for (int j = 0; j < 5; j++) {
4      a5[i * 5 + j] = 1;
5    }
6  }
```

10.1.4 La classe Arrays

La classe `Arrays` est une classe utilitaire qui contient plusieurs méthodes statiques pour manipuler les tableaux :

- `boolean equals()` : compare 2 tableaux.

- `void fill()` : remplit le tableau avec une valeur donnée.
- `void sort()` : trie le tableau.
- `boolean binarySearch()` : recherche dans un tableau trié.
- `List<?> asList()` : retourne un tableau sous forme d'une liste.

Pour copier un tableau, il faut utiliser la méthode `System.arrayCopy()`. Toutefois, cela ne copie que les références des objets et pas leur contenu.

La recherche dans un tableau nécessite d'implémenter l'interface `Comparator` qui oblige à définir la méthode `compare()`.

```
1    public class AlphabeticComparator implements Comparator {
2       @Override
3       public int compare( Object o1, Object o2 ) {
4          String s1 = (String)o1;
5          String s2 = (String)o2;
6          return s1.toLowerCase().compareTo(s2.toLowerCase());
7       }
8    }
9    AlphabeticComparator comp = new AlphabeticComparator();
10   Arrays.sort(sa, comp);
11   int index = Arrays.binarySearch(sa, sa[10)], comp);
```

10.2 Conteneur

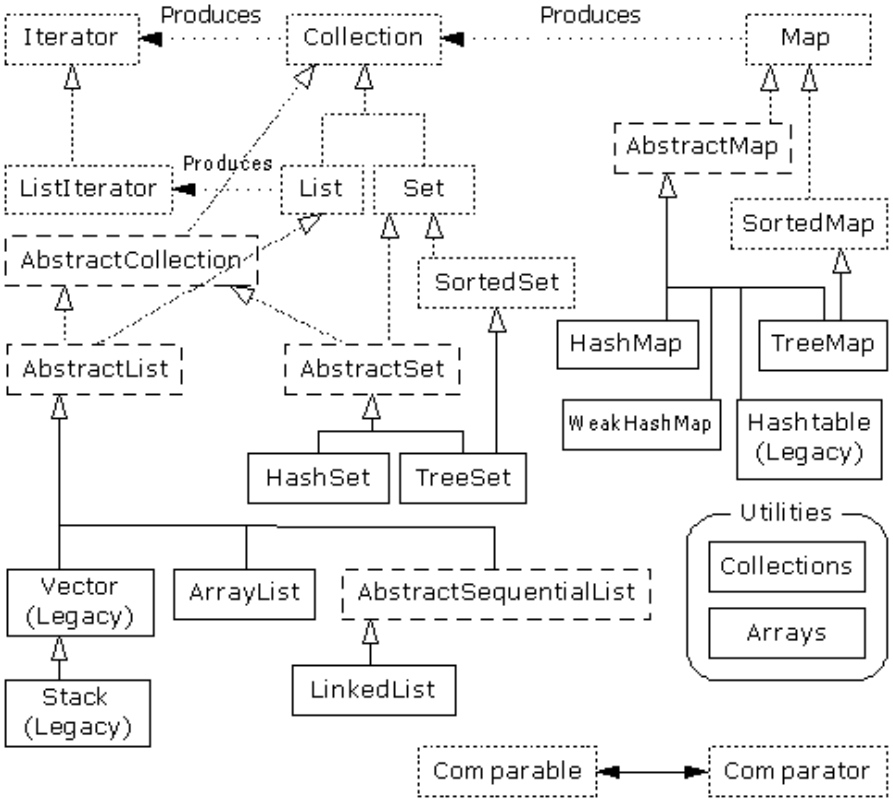

Figure 1: La hiérarchie des conteneurs.

Les conteneurs sont représentés par 2 classes de base : `Collection` et `Map`.

10.2.1 Collection

Une Collection est un groupe d'éléments individuels. La classe associée est **Collections** avec ses méthodes statiques correspondant à des utilitaires de manipulation de collection. Il y a deux classes abstraites dérivées de la classe `Collection : List` et `Map`.

1/ List

- **ArrayList** : une liste implémentée comme un tableau. Elle permet un accès rapide aux éléments, mais elle est lente quant à l'insertion ou la suppression d'éléments dans le milieu de la liste.
- **LinkedList** : une liste chaînée pou des accès séquentiels, avec des insertions et des suppressions rapides dans le milieu de la liste. Par contre, elle est relativement lente pour un accès aléatoire.
- Remarque : `Vector`, `Stack` sont synchronisés ce qui rend ces deux conteneurs lents à l'utilisation. Quand la synchronisation n'est pas nécessaire, il est préférable d'utiliser `ArrayList` pour les vecteurs et `LinkedList` pour les piles.

2/ **Set :** chaque élément est garanti d'être unique. Les éléments doivent définir la méthode `hashCode()` (voir la section 10.2.3).

- **HashSet** : pour les ensembles où la recherche d'éléments est importante. Ce doit être la classe par défaut pour les ensembles de type **Set**.
- **LinkedHashSet** : équivalent à `HashSet` mais maintient l'ordre dans lequel on insère les éléments.
- **TreeSet** : un ensemble ordonné avec l'algorithme du tri tas. L'ordre entre les éléments est donné par la méthode de comparaison des éléments `equals()`.

10.2.2 Map

Une Map est un tableau associatif contenant des paires d'objets de type clé-value. Il y a deux classes dérivées abstraites. L'utilisation de ce type nécessite

d'implémenter la méthode `hashCode()` et de définir la méthode `equals()`.

3/ **AbstractMap** : elle définit les opérations de base : `put`, `get`, `containsKey`, `constainsValue`, `size`, `isEmpty`.

- **HashMap** : Ce type doit être utilisé à la place de `Hashtable`, parce qu'il n'est pas synchronisée et est donc plus rapide.
- **LinkedHashMap** : le tableau peut être parcouru comme une `LinkedList`. On récupère les paires dans l'ordre d'insertion, ou dans l'ordre donné par la date d'accès des éléments.

4/ **SortedMap**

- **TreeMap** : implémentation basée sur les arbres rouges et noirs. Il permet de garder le contenu trié.
- **IdentityHashMap** : permet d'utiliser == au lieu de `equals` et est donc plus rapide pour les cas concernés.

10.2.3 Code de hachage

Dans son livre « Effective Java », Joshua Bloch donne une recette pour générer un code de hachage consistant pour la méthode `hashCode()` :

1/ Définir une variable entière initialisée avec une constante, en général un nombre premier, par exemple 17.

2/ Pour chaque attribut caractéristique de la classe (en fait les attributs qui sont utilisés par la méthode `equals()`), il faut calculer un code de hachage comme suit :

Type de la variable	Calcul
boolean v	v = (f ? 0 : 1)
byte, char, short, int v	v = (int)f;
long v	v = (int)(f >>> 32))
float v	v = Float.floatToIntBits(f);
double long l	Double.doubleToLongBits(f); v = (int)(l ^ (l >>> 32))
Object v	v = f.hashCode()
Array v	Appliquer les règles précédentes sur chaque élément.

3/ Combiner les codes hachages calculés au-dessus : `result = 37*result + v;`

4/ Retourner `result`.

Exemple

```
1    public class XXX {
2       private String _s;
3       private int _id;
4       @Override
5       public int hashCode() {
6          int result = 17;
7          result = 37 * result + _s.hashCode();
8          result = 37 * result + _id;
9       }
10      @Override
11      public boolean equals( Object o ) {
12         return o instanceof XXX && _s.equals((((XXX)o)._s)
13                                 && _id = ((XXX)o)._id;
14      }
15   }
```

10.3 Rendre une collection ou une map non-modifiable

Le modificateur final placé devant une variable de type Collection ou Map n'empêche pas de modifier le contenu. Pour empêcher la modification du contenu, il faut utiliser la classe utilitaire `Collections` et ses méthodes :

```
1  List<Int> a = Collections.unmodifiableList(new ArrayList<Int>());
2  a.add(5);        // error UnsupportedOperationException
3  Set<Int> s = Collections.unmodifiableSet(new HastSet<Int>());
4  s.add(1);        // error UnsupportedOperationException
5  Map<Int,Int> m = Collections.unmodifiableMap(new hashMap<Int, Int>());
6  m.put(5, 5);     // error UnsupportedOperationException
```

10.4 Synchroniser une Collection ou une Map

De façon basique, on utilisera le mot clé `synchronized` :

```
synchronized(list) { }
```

Une version plus protégée utilise les méthodes de la classe Collections :

```
List<T> l = Collections.synchronizedList(new ArrayList<TW>());
Set<Y> s = Collections.synchronizedSet(new HashSet<T>());
Map<E,V> m = Collections.synchronizedMap(new HashMap<E,V>());
```

10.5 Copie profonde

Les deux classes `CopyOnWriteArrayList` et `CopyOnWriteArraySet` créent des copies distinctes et profonde d'une collection.

10.6 Gestion des références

La bibliothèque `java.lang.ref` contient un ensemble de classes qui permettent une plus grande souplesse dans la gestion du ramasse-miettes. Ces classes sont particulièrement utiles pour le cas d'objets qui consomment beaucoup de mémoire. Chacune fournit différents niveaux d'indirection pour le ramasse-miette si l'objet en question est accessible par un des objets de référence ci-dessous :

- SoftReference
- WeakReference
- PhantomReference

On utilise des objets de ces classes lorsque l'on souhaite conserver une référence sur un objet, mais aussi permettre au ramasse-miette de libérer cet objet si la mémoire vient à manquer.

On accomplit cela en utilisant une référence intermédiaire entre l'objet et sa référence, et il ne doit pas y avoir de références classiques à l'objet. On garde une référence sur cet objet mais l'objet référencé peut ne plus exister. Vous pouvez utiliser la référence comme un cache. C'est une implémentation du patron de conception Procuration (Proxy).

11. Quelques classes utiles

11.1 Les classes String et StringBuffer

La classe `StringBuffer` est une classe associée à la classe `String`. La classe `String` est une classe immuable alors que `stringBuffer` est une classe modifiable. Avec `StringBuffer` toutes les modifications portent sur l'objet lui-même, et non sur une copie de l'objet comme pour `String`, ce qui consomme moins de ressources.

11.2 StringJoiner

La classe `StringJoiner` permet de concaténer des `String` séparées par un délimiteur. Elle fonctionne selon le patron Monteur (*Builder*) :

```
String join = new StringJoiner(",")
        .add("a")
        .add("b")
        .add("c")
        .toString();
assert join.equals("a,b,c");

String join = String.join(",", asList("a", "b", "c"));
assert join.equals("a,b,c");
```

11.3 Nombre à grande précision

Java inclut deux classes pour faire de l'arithmétique de grande précision :

- **BigInteger** permet la représentation intégrale de valeur entière de n'importe quelle taille sans perte d'information durant les opérations.
- **BigDecimal** est l'équivalent pour des nombres arbitraires à virgule fixe. La représentation à virgule fixe utilise un nombre fixe de chiffres après la virgule et un facteur d'échelle. Chaque bit du facteur d'échelle représente l'inverse d'une puissance de 10 (ou de 2). Ainsi, la première décimale est 1/10, la seconde 1/100, etc. Par exemple 1.23 sera représenté par 1230 avec un facteur d'échelle de 1/1000. Cette représentation permet d'augmenter la vitesse d'exécution des opérations de calcul.

11.4 Préférences

La classe `java.pref.Preferences` est utilisée pour stocker et récupérer des paires de valeurs persistantes ; des données qui restent sur le disque entre deux exécutions du programme.

Le stockage de ces préférences est fait par le système d'exploitation et dépend donc de celui-ci (sur Windows dans les registres, sur Linux dans les dossiers de préférences, etc), mais cela est fait de façon transparente pour l'utilisateur.

```
prefs = Preferences.userNodeForPackage(this.getClass());
```

Pour stocker et récupérer les valeurs :

```
        Stocker                          Récupérer
prefs.put(key, s)                  b = prefs.get(key);

prefs.putBoolean(key, b);          b = prefs.getBoolean(key);
prefs.putInt(key, i);              i= prefs.getInteger(key);
prefs.putLong(key, l);             l= prefs.getLong(key);
```

```
prefs.putDouble(key, d);           d = prefs.getDouble(key);
prefs.putFloat(key, f);            f= prefs.getFloat(key);
prefs.putByteArray(key, ba);       ba = prefs.getByteArray(key);

prefs.clear();
allkeys = prefs.keys();
prefs.remove(key);
```

11.5 Ensemble de bits

La classe `BitSet` implémente un vecteur de bits qui augmente dynamiquement. Un `BitSet` est utilisé si vous voulez stocker efficacement un grand nombre d'informations binaires. Il est efficace seulement du point de vue de la taille, au détriment de la vitesse d'accès. Il est légèrement plus lent que l'utilisation d'un tableau d'un type natif.

```
BitSet x = new BitSet(1024);
```

La taille minimale de `BitSet` est celle d'un long : 64 bits. Cela signifie que si vous stockez quelque chose de plus petit, comme 8 bits, un `BitSet` perd beaucoup de place. Il est plus efficace de créer sa propre classe, ou tout simplement d'utiliser un tableau si la mémoire est un problème.

Exemple d'utilisation :

```
1    BitSet bb = new BitSet();
2    Byte bt = (byte)rand.nextInt();
3    for (int i = 7; i >= 0; i-- ) {
4      if (((1 <<i) & bt) != 0) {
5        bb.set(i);
6      } else {
7        bb.clear(i);
8      }
9    }
10
11   String bbits = new String();
12   for (int j=0; j<8; j++ ) {
13     bbits += (bb.get(j) ? "1" : "0");
14   }
15   System.out.println("bit pattern =" + bbits);
```

11.6 La classe Optional

La classe `Optional` est un conteneur pour une valeur qui peut être `null`. En utilisant un `Optional` comme type d'une variable ou d'une valeur de retour d'une méthode, on oblige le client à se poser la question du test de l'existence de la valeur. Ainsi, c'est maintenant le développeur de la méthode qui indique au client que la valeur de retour peut être nulle et donc qu'il doit tester l'existence

de la valeur avant de l'utiliser.

```
Optional<String> hello = Optional.of("hello");

assertTrue(hello.isPresent());
assertEquals("hello".equals(hello.get()));
```

Si la valeur contenue est `null` :

```
Optional<Object> absent = Optional.ofNullable(null);

assertFalse(absent.isPresent());
absent.get() // throws java.util.NoSuchElementException: No value present
```

Les méthodes de la classe `Optional` permettent de transformer/filtrer la valeur contenue, pour retourner une valeur par défaut ou des exceptions.

12. Expression lambda

Une expression lambda est une méthode anonyme. L'intérêt des lambdas est de rendre le code beaucoup moins verbeux. Elles permettent par exemple de remplacer les classes anonymes qui implémentent une interface avec une seule méthode abstraite. Prenons l'exemple du tri d'une collection. Un code classique utilisant une classe anonyme est :

```
1  List<Integer> numbers = Arrays.asList(10, 1, 1000, 100);
2
3  Collections.sort(numbers, new Comparator<Integer>()) {
4    @Override
5    public int compare( Integer a, Integer b ) {
6      return a.compareTo(b);
7    }
8  });
```

L'équivalent en utilisant une lambda est :

```
Collections.sort(numbers, (a, b) -> a.compareTo(b));
```

12.1 Interface fonctionnelle

Les expressions lambda correspondent à des types spécifiés par des interfaces avec exactement une méthode abstraite (nommées SAM : *Single Abstract Method*).

Java propose l'annotation `@FunctionalInterface` pour s'assurer qu'une interface ne déclare qu'une seule méthode abstraite. Mais, le compilateur Java considère toute interface répondant à la définition comme une interface

fonctionnelle, que l'annotation `@FunctionInterface` soit présente ou non. Il est quand même préférable de mettre l'annotation de manière à prévenir les développeurs que cette interface est une SAM potentiellement déjà utilisée par des lambdas.

```
1    @FunctionalInterface
2    public interface Function<T, R> {
3      R apply(T t);
4    }
```

Si une deuxième méthode abstraite est ajoutée, la compilation échoue.

12.2 Méthodes par défaut dans les interfaces

Si une interface fonctionnelle contient une méthode abstraite et plusieurs méthodes par défaut, il s'agit toujours d'une interface fonctionnelle.

Toutefois, il n'est pas possible d'accéder aux méthodes statiques depuis une lambda.

```
1    @FunctionalInterface
2    public interface A {
3      default void printSomething() {
4        System.out.println("something");
5      }
6      void print();
7    }
```

La ligne suivante provoque une erreur de compilation.

```
A doesNotCompile = () -> printSomething(); // Erreur
```

12.3 Écriture des lambdas

Plusieurs raccourcis sont utilisables rendant l'écriture de moins en moins verbeuses.

Soit une interface fonctionnelle définissant une méthode abstraite avec deux paramètres de type `int` et une valeur de retour de type `int` . Les lignes suivantes sont équivalentes.

```
(int x, int y) -> { return x + y; }
```

Les types des paramètres peuvent être inférés par le compilateur.

```
(x, y) -> { return x + y }
```

Le mot clé `return` est facultatif.

```
(x, y) -> { x + y }
```

Étant donné qu'il n'y a qu'une instruction, les accolades sont facultatives :

```
(x, y) -> x + y
```

Au cas où une lambda ne possède aucun paramètre, l'écriture est appelée "*burger arrow*" :

```
() -> x
```

12.4 Quelques interfaces fonctionnelles bien utiles

Java définit un certain nombre d'interfaces fonctionnelles génériques dans le paquet `java.util.function`.

12.4.1 Function<T, R> : (T) → R

Une `Function` prend un argument de type T et retourne un résultat de type R. Dans l'exemple suivant, la méthode par défaut `compose()` de l'interface `Function` applique la fonction lambda `toInteger` puis la fonction lambda `toString` alors que la méthode `andThen()` applique `toString` puis `toInteger`.

```
1  Function<Integer, String> toString = n -> String.valueOf(n);
2  Function<String, Integer> toInteger = s -> Integer.valueOf(s);
3
4  assertEquals("4", toString.apply(4));
5  assertEquals(4, toInteger.apply("4"));
6
7  assertEquals("4", toString.compose(toInteger).apply("4")));
8  assertEquals(4, toString.andThen(toInteger).apply(4));
```

12.4.2 BiFunction<T, U, R> : (T, U) → R

C'est une spécialisation d'une `Function` qui prend deux arguments et retourne un résultat.

```
1  BiFunction<Integer, String, String>
2                      concat = (Integer i, String s) -> s + ": " + i;
3  assertEquals("un: 1", concat.apply(1, "un"));
```

12.4.3 UnaryOperator<T> : (T) → T

C'est une `Function` qui prend un argument et retourne un résultat du même type. Par exemple la fonction `identity` retourne toujours la valeur passée en argument.

```
1  assertEquals("un", UnaryOperator.identity().apply("un"));
```

12.4.4 BinaryOperator<T> : (T, T) → T

Un `BinaryOperator` est une spécialisation d'une `BiFunction` dont les paramètres et le résultat partagent le même type.

```
1  BinaryOperator<String>
2              concatString = (s1, s2) -> s1.concat(": ").concat(s2);
3
4  assertEquals("un: 1", concatString.apply("un", "1"));
```

12.4.5 Predicate<T> : (T) → boolean

Un `Predicate` prend un argument de type T et retourne un booléen.

```
1  Predicate<String> isEmpty = s -> s == null || s.isEmpty();
2  Predicate<String> isTrimmed = s -> s.equals(s.trim());
3
4  assertTrue(isEmpty.test(null));
5  assertTrue(isEmpty.test(""));
6  assertFalse(isEmpty.test("not empty"));
7
8  assertTrue(isEmpty.negate().and(isTrimmed).test("not empty"));
9  assertFalse(isEmpty.negate().and(isTrimmed).test(" not empty "));
10
11 assertTrue(isEmpty.or(isTrimmed).test(""));
12 assertTrue(isEmpty.or(isTrimmed).test("not empty"));
13
14 assertTrue(Predicate.isEqual("coucou").test("coucou"));
```

- `negate` inverse le prédicat,
- `and` et `or` permettent de chaîner des prédicats selon l'opérateur logique,
- La méthode statique `isEqual` teste l'égalité de deux objets selon `Object::equals`.

12.4.6 BiPredicate<T> : (T, T) → boolean

Un `BiPredicate` prend deux arguments et retourne un booléen.

12.4.7 Supplier<T> : () → T

Un `Supplier` ne prend pas d'argument et produit un résultat de type T.

```
1  Supplier<String> emptyString = () -> "";
2
3  assertEquals("", emptyString.get());
```

12.4.8 Consumer<T> : (T) → ()

Un `Consumer` prend un argument de type T mais ne retourne pas de résultat.

```
1  Consumer<String> print = s -> System.out.println(s);
2  Consumer<String> hello = s -> System.out.printf("Hello %s !", s);
3
4  print.accept("something");
5  print.andThen(hello).accept("JC"); // JC Hello JC !
```

- La méthode `andThen()` permet de chaîner les consommateurs.

12.5 Le mot clé this

Une différence essentielle entre l'utilisation d'une classe anonyme et les expressions lambda est l'utilisation du mot-clé `this`. Pour les classes anonymes, le mot clé `this` référence l'instance de la classe anonyme, alors que pour les expressions lambda, `this` référence la classe qui contient l'expression lambda.

12.6 Capture de variables

Les lambdas peuvent capturer une variable non statique définie en dehors du corps de la lambda. Par exemple, cette lambda capture la variable x :

```
1  int x = 5;
2  return y -> x + y;
```

Mais pour que cette déclaration soit valide, la variable doit être constante. Soit elle est marquée par le modificateur `final`, soit elle ne doit pas être modifiée après sa définition.

Par exemple, ces lignes de code :

```
1  Integer n = 4;
2  Function <Integer, Integer> modulo = (Integer a) -> a% n;
3  n = 8;
```

soulèvent l'erreur :

```
Local variable referenced from a lambda expression must be final or
  effectively final.
```

Cet autre exemple soulève la même erreur :

```
1  int count = 0;
2  List<String> strings = Arrays.asList("a", "b", "c");
3  strings.forEach(s -> {
4    count++; // erreur: impossible de modifier la valeur de count
5  });
```

12.7 Accès aux membres et variables statiques

Contrairement aux variables locales, les membres de classe sont accessibles

en écriture.

```
1  public class Lambda {
2    private static int _staticNumber;
3    private int _number;
4
5    public void write() {
6      Function<Integer, Integer> f1 = (Integer a) -> _staticNumber = a;
7      Function<Integer, Integer> f2 = (Integer a) -> _number = a;
8    }
9  }
```

12.8 Transparence d'exception

Si une exception doit être lancée à l'intérieur d'une lambda, l'interface fonctionnelle doit déclarer cette exception. L'exception n'est pas propagée à la méthode englobante. Le code suivant ne compile pas puisque l'interface fonctionnelle utilisée ne lève pas d'exception :

```
1  void appendAll(Iterable<String> values, Appendable out) throws IOException{
2    UnaryOperator<String> function = s -> out.append(s);
3
4    values.forEach(s -> {
5      function(s)          // erreur: impossible de lever IOException
6                           // Consumer.accept(T) ne le permet pas
7    });
8  }
```

Remarque : le `consumer` en question dans le message d'erreur est caché dans la définition de la méthode `forEach`.

Une manière de contourner cela, est de définir sa propre interface fonctionnelle qui étend l'interface `UnaryOperator` et lève l'exception `IOException`.

12.9 Instanciation de classes abstraites en utilisant une lambda

Une classe abstraite, même si elle ne déclare qu'une seule méthode abstraite, ne peut pas être instanciée avec une lambda.

Par exemple, les deux lignes suivantes ne compilent pas. Les deux classes abstraites `Ordering` et `CacheLoader` appartiennent à la bibliothèque Guava.

```
Ordering<String> order = (a, b) -> ...;
CacheLoader<String, String> loader = (key) -> ...;
```

L'argument le plus courant contre cela est que l'implantation d'une classe abstraite de cette façon pourrait conduire à l'exécution de code caché, par exemple le constructeur de la classe abstraite ou un bloc d'initialisation statique.

Une solution de contournement simple consiste à ajouter des méthodes fabriques pour convertir une lambda en une instance :

```
Ordering<String> order = Ordering.from((a, b) -> ...);
CacheLoader<String, String> loader = CacheLoader.from((key) -> ...);
```

12.10 Références de méthodes

Toujours dans le but de rendre le langage moins verbeux, Java a introduit le mot clef « :: » pour extraire des références de méthodes.

Par exemple, on peut remplacer :

```
Function<Integer, String> toString = n -> String.valueOf(n);
```

par :

```
Function<Integer, String> toString = String::valueOf;
```

On peut faire référence à des méthodes statiques à partir de la classe mais aussi on peut aussi faire référence à des méthodes depuis une instance :

```
1  String hello = new String("hello");
2  Predicate<String> startsWith = hello::startsWith;
3
4  assertTrue(startsWith.test("he"));
```

et aux constructeurs :

```
Supplier<String> newString = String::new;
```

et aussi aux méthodes d'instance sans référence d'objet en particulier :

```
1  List<String> names = Arrays.asList("Avranches", "Bayeux", "Caen",
  "Honfleur");
2  Collections.sort(names, String::compareToIgnoreCase);
```

En résumé :

Lambda expression	Équivalent par référence
`x -> String.valueOf(x)`	`String::valueOf`
`x -> x.toString()`	`Object::toString`
`() -> x.toString()`	`x::toString`
`() -> new ArrayList<>()`	`ArrayList::new`

Bien sûr, les méthodes en Java peuvent être surchargées. Les classes peuvent avoir plusieurs méthodes avec le même nom mais des paramètres différents. Il en va de même pour les constructeurs. Par exemple, l'appel `ArrayList::new` pourrait se référer à l'un des différents constructeurs. La méthode effectivement utilisée dépend de l'interface fonctionnelle utilisée.

13. Les flux

Java définit la classe `Stream`. Un objet `Stream` est une séquence d'éléments sur laquelle on applique une chaîne de traitements. Le flux est une implémentation en programmation fonctionnelle de la notion d'itérateur. Couplés aux expressions lambdas, les flux proposent une représentation moins verbeuse et des implémentations plus efficaces des itérations sur des séquences d'éléments. Lorsque cela est possible, il faut privilégier un flux à une itération.

13.1 Fonctionnement des flux

Un flux se compose d'une source (un tableau, une collection, etc), d'opérations intermédiaires et d'une opération terminale qui produit le résultat. La plupart des opérations de flux acceptent en paramètre une interface fonctionnelle, ce qui permet d'utiliser des expressions lambdas.

Le code suivant est un exemple de traitement en flux d'une liste :

```
1  List<String> myList = Arrays.asList("a1", "a2", "c2", "c1", "b1");
2  myList.stream()
3      .filter(s -> s.startsWith("c"))
4      .map(String::toUpperCase)
5      .sorted()
6      .forEach(System.out::println);
// Sortie :
// C1
// C2
```

Une opération *intermédiaire* maintient le flux ouvert et permet d'enchaîner plusieurs opérations sur le même flux. Les méthodes `filter()` et `map()` dans l'exemple ci-dessus sont des opérations intermédiaires.

Une opération *terminale* est indispensable pour consommer un flux. En effet, une caractéristique importante des opérations intermédiaires est l'évaluation paresseuse. Considérons l'exemple suivant où une opération terminale est manquante :

```
1  Stream.of("d2", "a2", "b1", "b3", "c")
2        .filter(s -> {
3          System.out.println("filter: " + s);
4          return true;
5        });
```

Lors de l'exécution de cette instruction, rien n'est imprimé sur la console. C'est parce que les opérations intermédiaires ne sont exécutées que lorsqu'une

opération terminale est présente.

13.2 Différents types de flux

On peut créer un flux de différentes façons :

- à partir d'une collection en utilisant la méthode `stream()` :

```
List<City> cities = Arrays.asList(
        new City("Marseille", 855393),
        new City("Paris", 2220445),
        new City("Caen", 108365),
        new City("Rouen", 110618));
assertEquals(cities.stream().count(), 4);
```

- en utilisant les classes `IntStream`, `DoubleStream`, `LongStream` pour créer des flux de type primitifs numériques :

```
assertEquals(IntStream.range(0, 10).sum(), 45);
```

- ou encore en profitant des méthodes de la classe utilitaire `Stream` :

```
Stream.of("a", "b", "c").forEach(System.out::println);
Stream.builder().add("a").add("b").add("c")
        .build()
        .forEach(System.out::println);
```

Un `Stream` peut-être infini. Dans ce cas, il doit y avoir une opération stoppante :

```
Random random = new Random();
Stream.generate(() -> random.nextInt())
      .limit(10)
      .forEach(System.out::println);
new Random().ints()
            .limit(10)
            .forEach(System.out::println);
```

Parfois, il est utile de transformer un flux d'objets en un flux de types primitifs et vice versa. Pour cela, le flux d'objets définit les méthodes intermédiaires `mapToInt()`, `mapToLong()` et `mapToDouble` :

```
Stream.of("a1", "a2", "a3")
        .map(s -> s.substring(1))
        .mapToInt(Integer::parseInt)
        .max()
        .ifPresent(System.out::println);
// Sortie:
// 3
```

À l'inverse un flux de types primitifs peut être transformé en flux d'objets par l'intermédiaire de la méthode intermédiaire `mapToObj()` :

```
IntStream.range(1, 4)
```

```
            .mapToObj(i -> "a" + i)
            .forEach(System.out::println);
// Sortie :
// a1
// a2
// a3
```

13.3 Ordre de traitement

L'ordre de traitement d'un flux dépend totalement de sa composition. Considérons l'exemple ci-dessous :

```
Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> {
            System.out.println("filter: " + s);
            return true;
        })
       .forEach(s -> System.out.println("forEach: " + s));
// Sortie :
filter:  d2
forEach: d2
filter:  a2
forEach: a2
filter:  b1
forEach: b1
filter:  b3
forEach: b3
filter:  c
forEach: c
```

L'ordre du résultat peut être surprenant. Une approche naïve consiste à considérer que les opérations sont appliquées horizontalement les unes après les autres sur tous les éléments du flux. Mais en fait chaque élément se déplace verticalement le long de la chaîne. La première chaîne "d2" passe le `filter` puis le `forEach`, avant que la seconde chaîne "a2" ne commence a être traitée.

Ce comportement peut permettre de réduire le nombre réel d'opérations effectuées sur chaque élément, comme dans l'exemple suivant :

```
Stream.of("d2", "a2", "b1", "b3", "c")
        .map(s -> {
            System.out.println("map: " + s);
            return s.toUpperCase();
        })
        .anyMatch(s -> {
            System.out.println("anyMatch: " + s);
            return s.startsWith("A");
        });
// Sortie :
// map:      d2
// anyMatch: D2
// map:      a2
// anyMatch: A2
```

L'opération `AnyMatch` stoppe la chaîne dès que le prédicat donné est vérifié. Ceci est vrai pour le deuxième élément passé "A2". En raison de l'exécution verticale de la chaîne de flux, l'opération `map` ne sera exécutée que deux fois dans ce cas.

13.3.1 Pourquoi l'ordre importe

L'exemple suivant est constitué de deux opérations intermédiaires `map` et `filtrer` et l'opération terminale `forEach`. Examinons de nouveau comment ces opérations sont exécutées :

```
1  Stream.of("d2", "a2", "b1", "b3", "c")
2        .map(s -> {
3            System.out.println("map: " + s);
4            return s.toUpperCase();
5        })
6        .filter(s -> {
7            System.out.println("filter: " + s);
8            return s.startsWith("A");
9        })
10      .forEach(s -> System.out.println("forEach: " + s));
```

```
// Sortie :
// map:     d2
// filter:  D2
// map:     a2
// filter:  A2
// forEach: A2
// map:     b1
// filter:  B1
// map:     b3
// filter:  B3
// map:     c
// filter:  C
```

Les opérations `map` et `filter` sont appelées cinq fois puisqu'il y a cinq éléments dans la collection alors que l'opération `forEach` est appelée une fois.

Nous pouvons réduire considérablement le nombre réel d'exécutions si l'on change l'ordre des opérations :

```
1  Stream.of("d2", "a2", "b1", "b3", "c")
2        .filter(s -> {
3            System.out.println("filter: " + s);
4             return s.startsWith("a");
5        })
6        .map(s -> {
7            System.out.println("map: " + s);
8            return s.toUpperCase();
9        })
10       .forEach(s -> System.out.println("forEach: " + s));
```

```
// Sortie :
// filter:  d2
```

```
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

Cette fois, l'opération `map` n'est appelée qu'une seule fois de sorte que la chaîne d'opérations s'effectue beaucoup plus vite pour un plus grand nombre d'éléments d'entrée. Il faut donc accorder une grande attention à la composition des flux.

Étendons l'exemple ci-dessus avec l'opération de tri, `sorted` :

```
1  Stream.of("d2", "a2", "b1", "b3", "c")
2          .sorted((s1, s2) -> {
3              System.out.printf("sorted: %s; %s\n", s1, s2);
4              return s1.compareTo(s2);
5          })
6          .filter(s -> {
7              System.out.println("filter: " + s);
8              return s.startsWith("a");
9          })
10         .map(s -> {
11             System.out.println("map: " + s);
12             return s.toUpperCase();
13         })
14         .forEach(s -> System.out.println("forEach: " + s));
```

L'exécution de cet exemple produit la sortie suivante :

```
sorted: a2; d2
sorted: b1; a2
sorted: b1; d2
sorted: b1; a2
sorted: b3; b1
sorted: b3; d2
sorted: c; b3
sorted: c; d2
filter: a2
forEach: a2
filter: b1
filter: b3
filter: c
filter: d2
```

Tout d'abord, l'opération de tri est exécutée sur la totalité de la collection d'entrée. En d'autres termes `sorted` est exécuté horizontalement. Dans ce cas, `sorted` est appelé huit fois pour plusieurs combinaisons des éléments de la collection d'entrée de façon à les trier.

Encore une fois, nous pouvons optimiser la performance en réorganisant la chaîne :

```
1  Stream.of("d2", "a2", "b1", "b3", "c")
```

```
        .filter(s -> {
            System.out.println("filter: " + s);
            return s.startsWith("a");
        })
        .sorted((s1, s2) -> {
            System.out.printf("sort: %s; %s\n", s1, s2);
            return s1.compareTo(s2);
        })
        .map(s -> {
            System.out.println("map: " + s);
            return s.toUpperCase();
        })
        .forEach(s -> System.out.println("forEach: " + s));
```

```
// Sortie :
// filter:  d2
// filter:  a2
// filter:  b1
// filter:  b3
// filter:  c
// map:     a2
// forEach: A2
```

Dans cet exemple, l'opération `sorted` n'est jamais appelée car `filter` réduit la collection d'entrée à un seul élément.

13.4 Réutiliser les flux

Les flux ne peuvent pas être réutilisés. Dès que l'opération terminale est achevée, le flux est fermé :

```
Stream<String> stream = Stream.of("d2", "a2", "b1", "b3", "c")
                              .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

L'appel de `noneMatch` après `AnyMatch` sur les mêmes résultats de flux lève une exception de type `IllegalStateException`.

Pour surmonter cette limitation, nous devons créer une nouvelle chaîne de flux pour chaque opération terminale que nous voulons exécuter. Par exemple, nous pouvons créer un fournisseur de flux pour construire un nouveau flux avec toutes les opérations intermédiaires déjà configurées :

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Chaque appel à `get()` construit un nouveau flux sur lequel nous pouvons appeler l'opération terminale souhaitée.

13.5 Les opérations de base sur les flux

La plupart des exemples de code de cette section utilisent la définition d'une classe Ville caractérisée par un nom et une population. La liste de Ville utilisée est la suivante :

```
List<City> cities = Arrays.asList(
        new City("Marseille", 855393),
        new City("Paris", 2220445),
        new City("Caen", 108365),
        new City("Rouen", 110618));
```

13.5.1 Opérations intermédiaires

Une opération intermédiaire retourne un nouveau flux qui peut être utilisé par les opérations de flux qui suivent.

- **filter** : exclure du flux tous les éléments qui ne correspondent pas à un prédicat.

```
cities.stream()
      .filter(c -> c.getName().startsWith("R"))
      .forEach(System.out::println);
// Sortie :
// Rouen
```

- **map** : applique une transformation des éléments un-à-un pour produire un nouveau flux.

```
cities.stream()
        .map(City::getPopulation)
        .sorted()
        .forEach(System.out::println);
// Sortie :
108365
110618
855393
2220445
```

- **flatMap** : met à plat un flux hiérarchique. On peut par exemple transformer un flux `Stream<List<City>>` en un flux `Stream<City>` :

```
Stream.<List<City>>builder()
      .add(asList(new City("Caen", 108365), new City("Marseille", 855393))
      .add(asList(new City("Rouen", 110618), new City("Paris", 2220445))
      .build()
      .flatMap(cities -> cities.stream())
      .filter(city -> city.getName().endsWith("n"))
      .forEach(System.out::println);
// Sortie :
// Caen
// Rouen
```

L'opération `flatMap` est d'une utilisation intéressante avec classe `Optional`. Elle permet d'éviter les nombreux contrôles de nullité. Prenons l'exemple d'une structure très hiérarchisée comme celui-ci :

```
1  class A {
2     AA aa;
3  }
4
5  class AA {
6      AAA aaa;
7  }
8
9  class AAA {
10     String foo;
11 }
```

Le parcours de cette structure impose de nombreux tests de nullité pour empêcher les `NullPointerExceptions` :

```
1  A a = new A();
2  if (a != null && a.aa != null && a.aa.aaa!= null){
3      System.out.println(a.aa.aaa.foo);
4  }
```

Le même parcours en utilisant `flatMap` et `Optional` donne :

```
1  Optional.of(new A())
2      .flatMap(o -> Optional.ofNullable(o.a))
3      .flatMap(n -> Optional.ofNullable(n.aa))
4      .flatMap(i -> Optional.ofNullable(i.aaa))
5      .ifPresent(System.out::println);
```

Chaque appel à `flatMap` renvoie un flux qui peut être vide. Dans ce cas, le traitement en flux s'arrête.

- **peek** : effectue une opération de type `Consumer` sur chacun des éléments sans changer quoique que ce soit dans le flux. C'est une méthode essentiellement utilisée pour déboguer entre les opérations d'un flux.

```
1  cities.stream()
2        .filter(c -> c.getName().endWith("n"))
3        .peek(System.out::println)
4        .filter(c -> c.population() > 100000)
5        .peek(System.out::println)
6        .collect(Collectors.toSet());
```

```
// Sortie :
// Caen
// Rouen
// Rouen
```

- **distinct** : exclut les multiples occurrences d'un même élément en s'appuyant sur la méthode `equals()` attachée aux éléments pour tester l'égalité.

- **sorted** : trie les éléments d'un flux en fonction de l'ordre imposé par un `Comparator`.

```
1  cities.stream()
2        .sorted((c1, c2) -> c1.getName().compareTo(c2.getName()))
3        .forEach(System.out::println);
```

```
// Sortie :
Marseille
Paris
Caen
Rouen
```

- **skip, limit** : veille à ce que les opérations ultérieures ne voient pas les premiers éléments ou ne voient qu'un nombre limité d'éléments.

```
1  cities.stream()
2        .limit(2)
3        .skip(1)
4        .findFirst()
5        .ifPresent(System.out::print);
```

```
// Sortie :
// Paris
```

13.5.2 Opérations terminales

- **forEach** : applique une fonction sur chaque élément du flux.

```
1  cities.stream()
2     forEach(c -> System.out.println(c.getName() + ": " +
  c.getPopulation()));
```

```
// Sortie :
// Marseille: 855393
// Paris: 2220445
// Caen: 108365
// Rouen: 110618
```

- **reduce** : combine tous les éléments du flux en un seul résultat. La méthode se décline en trois versions.

1/ La première version réduit un flux d'éléments à un élément du flux exactement. Elle prend un paramètre qui est une fonction d'accumulation de type `BinaryOperator` qui retourne un élément du type du flux. L'exemple suivant retourne la ville la plus peuplée :

```
1  cities.stream()
2        .reduce((c1, c2) -> c1.getPopulation() > c2.getPopulation()? c1 : c2)
3        .ifPresent(System.out::println);
```

```
// Sortie :
// Paris
```

2/ La deuxième méthode accepte en plus une valeur d'identité. La valeur d'identité est à la fois la valeur initiale de la réduction et le résultat par défaut

s'il n'y a pas d'élément dans le flux. Dans cet exemple, l'élément d'identité est une instance anonyme : `new City("", 0)`. La fonction d'accumulation est la même que dans la première version. Dans l'exemple ci-dessous, la méthode est utilisée pour construire une nouvelle ville avec les noms agrégés et la somme des populations de toutes les villes :

```
City result = cities.stream()
        .reduce(new City("", 0), (c1, c2) -> {
            c1.population += c2.getPopulation();
            c1.name += c2.getName();
            return c1;
        });
System.out.format("name=%s; population=%s", result.name, result.age);
```

```
// Sortie :
// name=MarseilleParisCaenRouen; population=3294821
```

3/ La troisième version accepte un troisième paramètre : une fonction de combinaison de type `BinaryOperator` qui sert à regrouper deux résultats intermédiaires. Cette fonction n'a d'intérêt que pour l'exécution parallèle du flux :

```
Integer populationSum = cities.parallelStream()
 .reduce(0,
        (sum, c) -> {
            System.out.format("accumulator: sum=%s; city=%s\n", sum, c);
             return sum += c.getPopulation();
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, s um2);
            return sum1 + sum2;
        });
```

```
// Sortie :
// accumulator: sum=0; city=Caen
// accumulator: sum=0; city=Rouen
// accumulator: sum=0; city=Marseille
// accumulator: sum=0; city=Paris
// combiner: sum1=855393; sum2= 3075838
// combiner: sum1=3075838; sum2=110618
// combiner: sum1=3184201; sum2=2331063
```

Comme l'accumulateur est appelé en parallèle, la fonction de combinaison est nécessaire pour réduire les valeurs cumulées séparées.

- **collect** : stocke les éléments d'un flux dans un conteneur, tel qu'un `Set`, une `Collection` ou une `Map`. La méthode introduit une nouvelle interface appelée `Collector`. Cette interface est un peu difficile à comprendre, mais heureusement, il y a une classe utilitaire `Collectors` pour générer toutes sortes de conteneurs utiles. Par exemple :

```
cities.stream()
      .map(City::getName)
```

```
3         .collect(Collectors.toList());
// Sortie :
// [Marseille, Paris, Caen, Rouen]
```

Pour retourner un ensemble au lieu d'une liste, il suffit d'utiliser `Collectors.toSet()`.

Afin de transformer les éléments du flux en une `map`, nous devons spécifier comment les clés et les valeurs doivent être incluses. Les clés mappées doivent être uniques, sinon une exception `IllegalStateException` est levée. Il est possible de passer une fonction de fusion comme paramètre supplémentaire pour contourner l'exception et construire une valeur à partir de plusieurs pour une même clé :

```
1  Map<Integer, List<String>> strings = Stream.of("a", "bb", "cc")
2      .collect(Collectors.toMap(
3          c -> c.length(),        // clé
4          c -> c.getName(),             // valeur
5          (name1, name2) -> name1 + ";" + name2)); // fonction de fusion
6  System.out.println(map);
// Sortie :
// {1="a", 2="bb;cc"}
```

Les collecteurs sont extrêmement polyvalents. Ils permettent également créer des agrégations sur les éléments du flux, des jointures ou des statistiques avec `averagingInt()` ou `Collectors.summarizingInt`.

L'exemple suivant regroupe les chaînes de caractères selon leur longueur :

```
1  Map<Integer, List<String>> strings = Stream.of("a", "bb", "cc")
2       .collect(Collectors.groupingBy(s -> s.length()));
3  strings.forEach((l, s) -> System.out.format("longueur %s: %s\n", l, s));
// Sortie :
// longueur 1: ["a", "a"]
// longueur 2: ["bb", "cc"]
```

L'exemple suivant regroupe toutes les villes en une seule chaîne :

```
1  String phrase = citiesc
2      .stream()
3      .filter(c -> c.getPopulation() >= 500000)
4      .map(c -> c.getName())
5      .collect(Collectors.joining(" et ", "Les villes ",
6                                  " sont supérieures  500000 habitants."));
7  System.out.println(phrase);
// Sortie :
// Les villes Marseile et paris sont supérieures 500000 habitants.
```

Il est également possible de construire son propre collecteur. Par exemple, nous voulons transformer toutes les villes du flux en une seule chaîne constituée de tous les noms en lettres majuscules séparées par le caractère "|". Pour y parvenir, nous créons un nouveau collecteur via `Collector.of()`. Nous

devons passer les quatre ingrédients d'un collecteur : un fournisseur, un accumulateur, un combinateur et un finisseur.

```
Collector<City, StringJoiner, String> cityNameCollector =
    Collector.of(
        () -> new StringJoiner(" | "),                  // fournisseur
        (j, c) -> j.add(c.getName().toUpperCase()),     // accumulateur
        (j1, j2) -> j1.merge(j2),                       // combinateur
        StringJoiner::toString);                        // finisseur
String names = cities.stream()
                     .collect(cityNameCollector);
System.out.println(names);
```

```
// Sortie :
MARSEILLE | PARIS | CAEN | ROUEN
```

Le fournisseur construit initialement une `StringJoiner` avec le séparateur approprié. L'accumulateur est utilisé pour ajouter chaque nom de ville en majuscule à la `StringJoiner`. Le combinateur sait comment fusionner deux `StringJoiners` en un seul. La dernière étape du module de finition construit la chaîne souhaitée à partir du `StringJoiner`.

- **toArray** : stocke les éléments d'un flux dans un tableau. Elle permet de palier le fait que l'opération `collect` ne permet pas de construire des tableaux.
- **min, max** : recherche l'élément minimum, maximum selon une relation d'ordre.

```
assertEquals(IntStream.rangeClosed(1, 10)
        .min()
        .getAsInt()
    , 1);
```

- **count** : compte le nombre d'éléments dans le flux.

```
assertEquals(cities.stream()
        .filter(c -> c.getPopulation() >= 500000)
        .count()
    , 2);
```

- **sum** : retourne la somme de tous les éléments.

```
assertEquals(IntStream.rangeClosed(1, 10).sum(), 55);
```

- **noneMatch, anyMatch, allMatch** : vérifie que zéro/un/des éléments répondent à un `Predicate`.

```
assertTrue(cities.stream().allMatch(c -> c.getName().startsWith("M")));
assertTrue(cities .stream().noneMatch(c -> c.getPopulation() == 1000000));
assertTrue(cities .stream().anyMatch(c -> "Paris".equals(c.getName())));
```

- **concat :** concatène des `Stream`.

```
IntStream.concat( IntStream.range(0, 4), IntStream.range(4, 6))
```

```
2           .forEach(System.out::print);
// Sortie :
// 012345
```

- **findFirst, findAny** : retourne le premier ou un élément du flux s'il existe en tant qu'`Optional`. Il s'agit d'opérations de court-circuit.

```
1  cities.stream()
2        .findFirst()
3        .ifPresent(System.out::print);
// Sortie :
// Marseille
```

ou encore n'importe quel élément :

```
1  cities.stream()
2        .parallel()
3        .findAny()
4        .ifPresent(System.out::print);
```

Le résultat de `findAny` n'est pas déterministe, le résultat peut varier d'une exécution à l'autre.

13.6 Flux parallèle

Par défaut, les opérations effectuées sur un flux sont séquentielles. Pour paralléliser les processus, il suffit d'utiliser la méthode de `Collection parallelStream()` à la place de la méthode `stream()`. On peut aussi changer l'état d'un `Stream` au cours de son utilisation en utilisant les méthodes `sequential()` et `parallel()`. Toutefois, il ne s'agit pas d'utiliser le parallélisme sur chaque `Stream` en espérant un gain de temps de traitement. Ça peut même être le contraire. Il convient donc de s'assurer de la pertinence de son utilisation.

Les flux parallèles utilisent les valeurs de configuration du `ForkJoinPool` disponible via la méthode statique `ForkJoinPool.commonPool()`. La taille du pool de thread dépend de la quantité de cœurs des processeurs physiques disponibles :

```
1 ForkJoinPool commonPool = ForkJoinPool.commonPool();
2 System.out.println(commonPool.getParallelism());
// Sortie :
// 3
```

Le nombre de thread utilisable par défaut peut être diminué ou augmenté en définissant le paramètre de la JVM suivant :

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Afin d'analyser le comportement d'exécution d'un flux en parallèle l'exemple

suivant affiche des informations sur le thread courant :

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
             s, Thread.currentThread().getName());
         return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

```
// Sortie :
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:     c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:     b1 [main]
forEach: B1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:     a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

Comme on peut le voir, le flux parallèle utilise les 3 threads disponibles de `ForkJoinPool` pour l'exécution des opérations de flux. La sortie peut varier d'exécution en exécution parce que le comportement des threads utilisés est non déterministe.

Étendons l'exemple avec une opération de flux supplémentaire, `sorted` :

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
             s, Thread.currentThread().getName());
         return s.toUpperCase();
     })
     .sorted((s1, s2) -> {
         System.out.format("sorted: %s <> %s [%s]\n",
             s1, s2, Thread.currentThread().getName());
         return s1.compareTo(s2);
     })
```

```
18      .forEach(s -> System.out.format("forEach: %s [%s]\n",
19          s, Thread.currentThread().getName()));
```

```
// Sortie :
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:     b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
map:     a1 [ForkJoinPool.commonPool-worker-2]
map:     c2 [ForkJoinPool.commonPool-worker-3]
sorted:  A2 <> A1 [main]
sorted:  B1 <> A2 [main]
sorted:  C2 <> B1 [main]
sorted:  C1 <> C2 [main]
sorted:  C1 <> B1 [main]
sorted:  C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

Le résultat peut paraître étrange au premier abord. Il semble que `sorted` soit exécuté de manière séquentielle sur le thread principal seul. En fait, `sorted` sur un flux parallèle utilise la nouvelle méthode `Arrays.parallelSort()` qui décide seule si le tri est effectué de manière séquentielle ou en parallèle en fonction de la longueur.

Lors de l'utilisation de la méthode `reduce()`, nous avons déjà constaté qu'il y avait l'appel d'une méthode de recombinaison des résultats intermédiaires sur les flux parallèles mais pas sur les flux séquentiels. Voyons quels threads sont réellement appelés :

```
1  List<City> cities = Arrays.asList(
2      new City("Marseille", 855393),
3      new City("Paris", 2220445),
4      new City("Caen", 108363),
5      new City("Rouen", 110618));
6
7  cities.parallelStream()
8          .reduce(0,
9           (sum, c) -> {
10              System.out.format("accumulator: sum=%s; city=%s [%s]\n",
11                  sum, c, Thread.currentThread().getName());
12              return sum += c.getPopulation();
13          },
14          (sum1, sum2) -> {
15              System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",
16                  sum1, sum2, Thread.currentThread().getName());
17              return sum1 + sum2;
18          });
```

```
// Sortie :
accumulator: sum=0; city=Caen; [main]
accumulator: sum=0; city=Marseille;    [ForkJoinPool.commonPool-worker-3]
accumulator: sum=0; city=Rouen;  [ForkJoinPool.commonPool-worker-2]
accumulator: sum=0; city=Paris;  [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=855393; sum2=2220445;  [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=2220445; sum2=110618;  [ForkJoinPool.commonPool-worker-2]
combiner:    sum1=3075838; sum2=218983;  [ForkJoinPool.commonPool-worker-2]
```

La sortie de la console révèle que l'accumulateur et les fonctions de combinaison sont exécutés en parallèle.

En résumé, on peut dire que les flux parallèles peuvent apporter un bon coup de pouce à la performance de flux de grande quantité d'éléments d'entrée. Mais gardez à l'esprit que certaines opérations de flux parallèles comme `reduce` ou `collect` ont besoin d'opérations supplémentaires (opérations de recombinaisons) qui ne sont pas nécessaires lors de l'exécution séquentielle.

Il est important de noter que le parallélisme n'est pas gratuit. Vous ne pouvez pas simplement échanger un flux séquentiel pour un parallèle et espérer que les résultats seront identiques sans autre réflexion. Il existe des propriétés à considérer concernant votre flux, ses opérations et la destination de ses données avant de pouvoir paralléliser un flux.

13.7 Map

Les `Map` ne sont pas compatibles avec les `Stream`. Toutefois, elles disposent de la méthode parcours `forEach()` qui utilise un `BiConsumer` pour consommer tous les couples d'une `Map`.

```
1  Map<String, Integer> map = new HashMap<>();
2  map.put("un", 1);
3  map.put("deux", 2);
4  map.put("trois", 3);
5
6  map.forEach((key, value) -> System.out.printf("%s(%d) ", key, value));
```

```
// Sortie :
// trois(3) un(1) deux(2)
```