

Jean-Baptiste Yunès

Java moderne

Programmation générique, fonctionnelle
et concurrente



Références sciences

Java moderne

Programmation générique,
fonctionnelle et concurrente

Jean-Baptiste Yunès



Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

*Retrouvez tous les livres de la collection
et des extraits sur www.editions-ellipses.fr*



ISBN 9782340-120617

Dépôt légal : octobre 2026

©Ellipses Édition Marketing S.A.

8/10 rue la Quintinie 75015 Paris



Le Code de la propriété intellectuelle et artistique n'autorisant, aux termes des alinéas 2 et 3 de l'article L. 122-5, d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause, est illicite » (alinéa 1^{er} de l'article L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Je pense à celles et ceux qui ont rendu possible cette île, ce livre.

Jean-Marie Rifflet qui du magma fait sortir la silice.

Philippe Narbel avec qui je crochète des atomes.

Luidnel Maignan, mon compagnon de cellule.

François Horn, un esprit sage et étonnant.

Martine, une bien sage femme.

Antoine, Luc, Juliette.

Préambule

Depuis sa première distribution publique en 1996, le langage Java a connu de nombreuses évolutions, tant sur le plan syntaxique que sémantique. Conçu initialement comme un langage orienté objet relativement simple, privilégiant la portabilité et la sûreté, Java s’est progressivement enrichi de mécanismes destinés à répondre à des besoins toujours plus complexes : abstraction de types, manipulation de comportements, et maîtrise de l’exécution concurrente.

Ces évolutions ne sont pas purement cosmétiques. Elles traduisent un changement de perspective dans la manière dont les programmes sont conçus. La généricité introduite en Java 5, les λ -expressions et les *streams* apparus en Java 8, ainsi que les API de concurrence de plus en plus sophistiquées qui ont évolué de Java 5 jusqu’à aujourd’hui, constituent autant de réponses à une même problématique fondamentale :

Comment écrire des programmes corrects, expressifs et efficaces dans un monde logiciel de plus en plus complexe ?

Cet ouvrage se propose d’explorer ces trois axes (généricité, style fonctionnel et *multithreading*) non comme des fonctionnalités isolées, mais comme des facettes complémentaires d’un même effort de modernisation du langage. Tout ceci fait désormais de Java un langage plus expressif, plus sûr et mieux adapté aux architectures logicielles et pratiques modernes de programmation.

La généricité comme abstraction

La généricité est la première modification majeure de l’histoire de Java (elle est apparue en Java 5). En permettant de paramétrer les types, elle offre au programmeur un moyen d’exprimer des invariants autrefois implicites et laissés à sa seule discipline. Les collections génériques, les méthodes paramétrées, les bornes de types et les mécanismes de variance constituent autant d’outils permettant d’écrire des codes

à la fois plus concis et abstraits, tout autant que réutilisables dans de nombreux contextes.

Cependant, cette expressivité accrue s'accompagne d'une complexité non négligeable. L'effacement de type, les jokers, la capture, les types intersection ou encore les interactions subtiles avec l'héritage rendent la généricité Java à la fois puissante et déroutante. Comprendre ces mécanismes ne relève pas uniquement d'un apprentissage syntaxique, mais nécessite une appréhension assez fine du système de types sous-jacent.

Cet ouvrage ne se contente donc pas de présenter les règles du langage : il s'attache à en expliciter les motivations, les compromis et les limites, afin de fournir au lecteur les outils conceptuels nécessaires à un usage éclairé de la généricité.

Le style fonctionnel : les comportement comme données

Avec l'introduction des λ -expressions, des interfaces fonctionnelles et de leur application sous la forme de l'API des *streams*, Java a franchi une étape supplémentaire vers un style de programmation plus déclaratif. Il devient possible de raisonner en termes de transformations de données plutôt qu'en termes de séquences d'instructions, de décrire des enchaînements de calculs plutôt que de les orchestrer explicitement.

Il ne s'agit pas pour autant d'un basculement vers un langage fonctionnel au sens strict. Java reste profondément impératif et orienté objet. Les λ -expressions de Java ne garantissent ni la pureté des fonctions, ni l'immutabilité des données, ni la transparence référentielle ; à moins de se discipliner très fortement. Pourtant, elles introduisent une nouvelle manière de structurer le code, d'en améliorer la lisibilité, et surtout de réduire la surface des erreurs, en particulier dans un contexte concurrent.

Ce livre se propose d'éclairer les questions suivantes :

- Que signifie programmer en style fonctionnel en Java ?
- Quelles garanties le langage offre-t-il réellement ?
- Lesquelles restent à la charge du programmeur ?
- Quels sont les avantages et inconvénients de cette approche hybride ?

La concurrence

La concurrence est au cœur des systèmes modernes, qu'il s'agisse de serveurs, d'applications interactives ou de traitements distribués. Java, dès ses origines, a fait le choix d'intégrer le *multithreading* au cœur du langage. Ce choix audacieux s'est révélé à la fois une force et une source de difficultés considérables.

Les mécanismes de synchronisation bas niveau, le modèle mémoire, les problèmes de visibilité, de séquençement critique ou de blocage ont longtemps rendu la programmation concurrente particulièrement délicate. Les évolutions successives de la plateforme (bibliothèques concurrentes, abstractions de plus haut niveau, immutabilité encouragée, parallélisme des *streams*) traduisent une volonté de déplacer la complexité du code vers des abstractions plus aisément maîtrisables.

Dans ce contexte, le style fonctionnel et la généricité ne sont pas de simples ajouts orthogonaux : ils participent directement à la construction de codes concurrents plus sûrs, en limitant les effets de bord, en rendant explicites les contraintes de typage et en simplifiant les échanges de données.

Public et ambition de l'ouvrage

Cet ouvrage s'adresse à des lecteurs disposant déjà d'une bonne maîtrise de Java en ce qui concerne la POO et souhaitant approfondir leur compréhension du langage au-delà de ses usages les plus courants. Il vise en particulier :

- les **apprenants** qui souhaitent enrichir leur connaissance de Java au-delà de la simple programmation orientée objet,
- les **ingénieurs** logiciel et les **développeurs** expérimentés désireux de raisonner plus finement sur leurs programmes, tant du point de vue du typage que de l'exécution. Qu'ils utilisent une version avancée de l'éco-système Java ne signifie pas toujours qu'ils en utilisent toutes les fonctionnalités ni en pleine conscience. L'ouvrage vise donc à leur fournir une compréhension plus solide et durable des concepts les plus importants.
- les **enseignants** ou **formateurs** qui cherchent un support de référence sur l'un des sujets abordés.
- les **chefs de projet** ou **décideurs** techniques qui souhaitent comprendre les concepts modernes qu'il est possible de mettre en jeu dans la phase de réalisation d'un projet.

L'objectif n'est pas de fournir un catalogue exhaustif des APIs (génériques, fonctionnelles ou concurrentes), mais de proposer une lecture structurée et argumentée des principaux concepts, appuyée par des exemples concrets, des contre-exemples et des mises en perspective avec d'autres langages et aspects théoriques.

Une approche raisonnée

En tant qu'être humain le programmeur est faillible. Les choix de conception du langage, comme les bonnes pratiques qui en découlent, prennent en compte cette

fatale réalité. Comprendre la généricité, le style fonctionnel et le *multithreading* en Java, ce n'est pas seulement apprendre à les utiliser, mais surtout comprendre quels problèmes ils cherchent à résoudre et quelles erreurs ils tentent de prévenir.

C'est à cette compréhension approfondie que cet ouvrage souhaite contribuer. En explorant Java de la version 5 à la version 21 (avec des courtes immersions en 24 et 25), cet ouvrage propose un parcours cohérent à travers un langage à la fois mature et résolument moderne. On y détaille ce qui est le plus essentiel tout en ouvrant des pistes au lecteur afin qu'il puisse par lui-même augmenter la qualité de son code, sa productivité et son expertise.

Les codes sources

L'ouvrage contient de nombreux codes sources librement utilisables qui sont tous disponibles sur GitHub :

<https://github.com/jbyunes/Java-book-code.git>

Chacun d'entre eux a été minutieusement testé avec le JDK 25 de la distribution d'Oracle. Pour des questions de mise en page, il a fallu parfois sacrifier quelques-unes des conventions d'écriture de code Java afin de permettre la représentation de programmes complets. Si une erreur s'est malencontreusement glissée, le lecteur pourra se reporter aux codes sources disponibles.

Notes

Le texte source de cet ouvrage est du code $\text{\LaTeX}$ édité grâce au logiciel libre $\text{\TeX}$ Shop dont le développement est coordonné par Richard Koch.

Le document imprimable a été obtenu grâce au système de production $\text{\TeX}$ Live 2026. Sans l'extraordinaire travail initial de Donald Knuth, comment aurions-nous pu obtenir un résultat d'une telle qualité typographique ?

Remerciements

L'auteur remercie chaleureusement :

- M. Paul de Laboulaye et les éditions Ellipses d'avoir accepté l'édition de cet ouvrage.
- MM. Philippe Narbel et François Horn (dans le désordre) d'avoir relu attentivement de très nombreuses portions du texte. Si des erreurs subsistent, l'auteur en est seul responsable.

Chapitre 1

Quelques constructions bien pratiques

On présente dans ce chapitre un certain nombre d'éléments apparus au cours de l'évolution moderne de Java. Il ne s'agit pas d'être exhaustif mais plutôt de détailler des constructions qui sont clairement les plus immédiatement utiles. À la fois pour les chapitres qui suivent mais aussi pour permettre aux développeurs d'alléger leur code, le rendre plus expressif sans être verbeux et donc d'aller à l'essentiel.

Ces éléments permettent (en particulier) de répondre aux nombreuses critiques faites à l'écosystème Java, comme :

- la verbosité du code ;
- la multiplication des classes (à chaque concept : une classe) ;
- la multiplication des instances (une classe : des instanciations).

1.1 Gestion des ressources : `try-with-resources` (Java 7)

Une nouvelle forme de `try` est apparue en Java 7, le `try-with-resources`.

Elle permet de prendre en charge la gestion de ressources :

- en **automatisant leur libération**. Cette tâche est bien souvent désagréable à réaliser correctement ;
- en l'interfaçant **correctement avec la gestion des erreurs**. Ce problème est lui encore plus complexe, car les libérations elles-mêmes peuvent générer des erreurs en cascade.

Prenons l'exemple de la copie d'un fichier dans un autre pour illustrer le problème :

Programme 1.1: Copie de fichier, gestion des exceptions

```
public class FileCopyExample {
    public static void main(String[] args) {
        InputStream in = null;
        OutputStream out = null;
        try {
            in = new FileInputStream("source.txt");
            out = new FileOutputStream("destination.txt");
            int data;
            while ( (data=in.read()) != -1) { out.write(data); }
        } catch (Exception e) { e.printStackTrace(); }
        finally {
            if (in!=null)
                try { in.close(); }
                catch (Exception e) { e.printStackTrace(); }
            if (out!=null)
                try { out.close(); }
                catch (Exception e) { e.printStackTrace(); }
        }
    }
}
```

Le code est verbeux et déstructuré pour diverses raisons :

- on ouvre les fichiers dans le bloc `try`, mais les variables doivent être déclarées en dehors car on utilise `finally` pour fermer les fichiers ouverts (problème de portée des variables);
- dans le bloc `finally` on se retrouve à tester si les fichiers ont été ouverts, puis à les fermer dans le cas positif; mais la fermeture lève potentiellement des exceptions et on est donc conduit à imbriquer un bloc `try` si on souhaite confiner ces problèmes (gestion imbriquée des exceptions);
- les exceptions qui peuvent être levées ici et là peuvent être perdues.

1.1.1 AutoCloseable

Java 7 introduit la construction appelée `try-with-resources` qui permet préalablement à un bloc `try` de déclarer une ou plusieurs ressources qui seront contrôlées par le `try` de sorte que leur libération s'effectuera automatiquement en sortie du bloc.

C'est un équivalent Java de l'idiome RAII du C++, dont l'idée est d'inclure dans la **portée** (*scope*) d'un bloc `try` l'acquisition explicite de ressources de sorte qu'à la sortie leur libération sera implicitement réalisée.

Pour ce faire, la ressource doit implémenter l'interface `AutoCloseable` qui ne contient qu'une seule méthode abstraite :

`AutoCloseable (Java 7)`

```
public interface AutoCloseable {  
    void close() throws Exception;  
}
```

Bien entendu, la plupart des ressources des différentes APIs, et en particulier celles de `java.io` sont désormais `AutoCloseable`. On peut ainsi réécrire le code précédent à l'aide d'un `try-with-resources` :

Programme 1.2: Copie de fichier avec try-with-resources

```
public class FileCopyExample {  
    public static void main(String[] args) {  
        try (InputStream in = new FileInputStream("source.txt");  
            OutputStream out = new FileOutputStream("destination.txt")) {  
            int data;  
            while ( (data=in.read()) != -1) { out.write(data); }  
        }  
        catch (Exception e) { e.printStackTrace(); }  
    }  
}
```

On conviendra aisément que l'écriture est plus concise et la lecture plus claire.

Pour déclarer l'acquisition de ressources, il faut faire suivre le mot-clé `try` d'une liste parenthésée d'instructions déclarant des acquisitions d'`AutoCloseable`. Les variables qui capturent les ressources ont la portée du bloc `try`. Dès lors, quelque soit la façon dont on sort du bloc `try` (de façon normale ou avec exception), les ressources seront toutes automatiquement fermées dans l'ordre inverse de leur acquisition. Ceci se produira y compris en l'absence de bloc `finally`. S'il existe des blocs `catch` ou `finally`, leur code sera exécuté nécessairement après la libération des ressources.

On notera qu'un `try-with-resources` ne nécessite pas nécessairement de `catch` ni de `finally` correspondant. Un tel `try` est simplement réduit à la simple gestion automatisée de ressources.

1.1.2 Comportement vis-à-vis des exceptions

Un point important est le comportement des exceptions qui peuvent survenir à différents endroits. Quoiqu'il arrive aucune exception ne sera perdue, elles seront simplement chaînées les unes aux autres.

Le chaînage fonctionne ainsi :

- si une exception est levée dans le bloc `try`, on la conserve comme exception initiale, puis on tente les fermetures ;
- s'il n'y a pas d'exception levée dans le bloc `try`, on tente les fermetures ;
- si une exception est levée lors d'une fermeture, elle est chaînée à l'exception précédente s'il y en a une, sinon elle est conservée comme exception initiale ; puis on tente les autres fermetures. Les fermetures sont tentées dans l'ordre inverse des acquisitions ;
- on sort alors du bloc `try` avec l'exception initiale et on applique l'algorithme de recherche d'un `catch`.

Pour illustrer ceci, on peut regarder comment cela est effectivement réalisé. Tout d'abord on crée un `AutoCloseable` qui lève systématiquement une exception à sa fermeture :

```
MyAutoCloseable
public class MyAutoCloseable implements AutoCloseable {
    private String n;
    public MyAutoCloseable(String n) {
        this.n = n;
    }
    @Override public void close() throws Exception {
        throw new Exception("close()␣"+n);
    }
    @Override public String toString() {
        return n;
    }
}
```

Puis on tente plusieurs acquisitions :

Programme 1.3: Déclaration d'`AutoCloseable` dans un `try-with-resources`

```
public class AutoCloseableExample {
    public static void main(String[] args) {
        try (AutoCloseable a = new MyAutoCloseable("1");
            AutoCloseable b = new MyAutoCloseable("2")) {
            System.out.println("do␣something");
            throw new Exception("Oh␣no...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Son exécution produit :

```
do something
java.lang.Exception: Oh no...
    at autocloseable/autocloseable.Main.main(Main.java:9)
    Suppressed: java.lang.Exception: close() 2
        at autocloseable/autocloseable.MyAutoCloseable. ▶
            ▶ close(MyAutoCloseable.java:8)
        at autocloseable/autocloseable.Main.main(Main.java:10)
    Suppressed: java.lang.Exception: close() 1
        at autocloseable/autocloseable.MyAutoCloseable. ▶
            ▶ close(MyAutoCloseable.java:8)
        at autocloseable/autocloseable.Main.main(Main.java:10)
```

On observe donc :

1. qu'une exception initiale (Oh no...) est levée dans le bloc try. La machine doit donc sortir du bloc, mais auparavant elle doit libérer les ressources et ce, dans le sens inverse de leur acquisition.
2. la ressource b est libérée par appel implicite à close, mais ici l'opération lève une exception (close() 2) celle-ci est donc ajoutée à la liste des exceptions « supprimées » de l'exception initiale.
3. la ressource a est ensuite libérée, mais l'opération close lève une exception (close() 1), celle-ci est alors ajoutée à la liste.
4. la machine sort du bloc en erreur avec l'exception initiale,
5. elle recherche le catch qui correspond et exécute son code.

Depuis Java 7, le Throwable possède des méthodes supplémentaires de gestion de ces exceptions « supprimées » et dont il est souhaitable de garder la trace :

Throwable (méthodes Java 7)

```
public final void addSuppressed(Throwable exception);
public final Throwable[] getSuppressed();
```

1.1.3 Amélioration en Java 9

Avant Java 9, les ressources préexistantes au bloc try ne pouvaient être intégrées dans la gestion automatisée de leur libération. Depuis, Java 9 autorise leur intégration (sous réserve qu'elles soient de type AutoCloseable évidemment) :

Programme 1.4: Intégration d'AutoCloseable dans un try-with-resources

```
public class AutoCloseableExample {
    public static void main(String[] args) {
```

```

AutoCloseable a = new MyAutoCloseable("1");
AutoCloseable b = new MyAutoCloseable("2");
try (a; b) { // « auto closing » of pre-existing resources
    System.out.println("do_something");
    throw new Exception("Oh_no...");
} catch (Exception e) {
    System.out.println("catch:_" + a + "_" + b);
    System.out.println(e);
    for (Throwable t: e.getSuppressed()) {
        System.out.println(t);
    }
}
}

```

Celles-ci sont alors automatiquement fermées :

```

do something
catch: 1 2
java.lang.Exception: Oh no...
java.lang.Exception: close() 2
java.lang.Exception: close() 1

```

1.2 Évolution des interfaces (Java 8)

Le passage sans douleur avec compatibilité ascendante à la programmation fonctionnelle (voir Chapitre 3 « Le style fonctionnel de Java 8 ») a nécessité l'évolution des interfaces de Java. Comme on va le voir, il est désormais permis d'intégrer du code dans les interfaces. Ceci pourrait sembler contraire au concept d'interface censé représenter un concept abstrait. Ces modifications apportées aux interfaces ne modifient en rien leur nature, cela reste des interfaces c'est-à-dire des spécifications abstraites de types (du moins si l'on souhaite les considérer ainsi). Il s'agit simplement d'ajouts permettant de faire évoluer les interfaces de sorte à faciliter certaines autres constructions.

C'est dans une réponse à une question postée sur le site StackOverflow que B. Goetz¹ présente cette évolution :

A key design goal was that, from the perspective of the client of an interface, default methods should be indistinguishable from "regular" interface methods. The defaultness of a method, therefore, is only interesting to the designer and implementor of the interface.

1. GOETZ, B. (2015). *Stackoverflow answer*. <https://stackoverflow.com/questions/28681737/java-8-default-methods-as-traits-safe/28684917#28684917>

Avant Java 8 les interfaces ne pouvaient contenir que :

- des méthodes publiques abstraites ;
- des champs statiques publics finaux.

Désormais depuis Java 8, elles peuvent contenir :

- des méthodes statiques publiques ;
- des méthodes avec implémentation par défaut.

Depuis Java 9, elles peuvent aussi contenir :

- des méthodes statiques privées ;
- des méthodes privées.

1.2.1 Méthodes statiques

Un argument souvent évoqué pour critiquer Java est qu'on y est conduit à créer de nombreuses classes/interfaces chacune encapsulant différentes fonctionnalités mais toutes reliées conceptuellement pour chaque type de problème.

Par exemple, une conception un peu évoluée conduit généralement à utiliser :

- des interfaces pour représenter le type ;
- des classes abstraites pour factoriser des méthodes ou fournir des implémentations par défaut ;
- des types concrets (en général cachés) ;
- des usines statiques ;
- des classes utilitaires.

Cela conduit généralement à utiliser des conventions de nommage comme :

- `Vehicle` pour désigner une interface ;
- `AbstractVehicle` pour désigner une classe abstraite factorisant des fonctionnalités communes ;
- `VehicleImpl1` pour une implémentation concrète ;
- `VehicleFactory` pour une classe usine ;
- `Vehicles` pour une classe utilitaire.

Chacune de ces classes devant être placée dans un espace de nommage adéquat avec la bonne accessibilité.

Java 8 offre la possibilité de créer des méthodes statiques dans les interfaces. Cela peut permettre de ne publier que l'interface qui contiendra à la fois les méthodes du type abstrait de données, mais aussi les fabriques statiques ou les utilitaires.

Par exemple, étant donné les types suivants :

```
Vehicle, Panhard, Oldsmobile

public interface Vehicle {}
public class Panhard implements Vehicle {
    @Override public String toString() {
        return "Panhard";
    };
}
public class Oldsmobile implements Vehicle {
    @Override public String toString() {
        return "Oldsmobile";
    };
}
```

ainsi que l'usine et la classe utilitaire suivantes :

```
VehicleFactory, Vehicles

public final class VehicleFactory {
    private VehicleFactory() {}
    public static Vehicle create(String countryCode) throws Exception {
        switch(countryCode) {
            case "fr": return new Panhard();
            case "us": return new Oldsmobile();
            default: throw new Exception("Bad_country_code");
        }
    }
}
public final class Vehicles {
    private Vehicles() {}
    public static String madeIn(Vehicle v) {
        if (v instanceof Panhard) { return "fr"; }
        if (v instanceof Oldsmobile) { return "us"; }
        return "Unknown";
    }
}
```

On peut alors écrire le programme suivant :

Programme 1.5: Test des classes Vehicle

```
public class VehiclesExample {
    public static void main(String[] args) throws Exception {
        Vehicle v = VehicleFactory.create("fr");
        System.out.println(v);
        System.out.println(Vehicles.madeIn(v));
    }
}
```

Le paquetage des véhicules contient 5 types, mais seulement 3 d'entre eux sont visibles depuis l'extérieur :

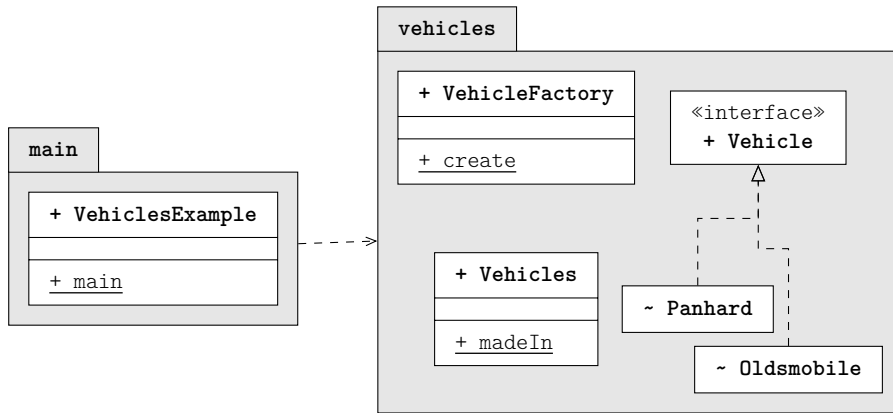


FIGURE 1.1 – Hiérarchie des véhicules.

Certains d'entre eux nécessitent l'utilisation d'une astuce permettant d'interdire l'instanciation. On peut se mettre d'accord sur le fait que de telles astuces seraient à proscrire si cela était possible¹.

Avec des méthodes statiques, le code se factorise plus avantageusement en faisant disparaître `Vehicles` et `VehicleFactory` et en insérant l'usine et l'utilitaire directement dans l'interface :

Méthode statique dans une interface : `Vehicle`

```

public interface Vehicle {
    public static Vehicle create(String countryCode) throws Exception {
        switch(countryCode) {
            case "fr":
                return new Panhard();
            case "us":
                return new Oldsmobile();
            default: throw new Exception("Bad_country_code");
        }
    }

    public static String madeIn(Vehicle v) {
        if (v instanceof Panhard) { return "fr"; }
        if (v instanceof Oldsmobile) { return "us"; }
        return "Unknown";
    }
}
    
```

1. L'utilisation d'astuces révèle les faiblesses ou la pauvreté d'un langage.

La méthode `main` n'utilise alors plus que le seul type public utile, à savoir l'interface qui fait donc aussi office d'usine et d'utilitaire :

Programme 1.6: Utilisation d'une usine statique d'interface

```
public class VehiclesExample {
    public static void main(String[] args) throws Exception {
        Vehicle v = Vehicle.create("fr");
        System.out.println(v);
        System.out.println(Vehicle.madeIn(v));
    }
}
```

On obtient donc la spécification suivante :

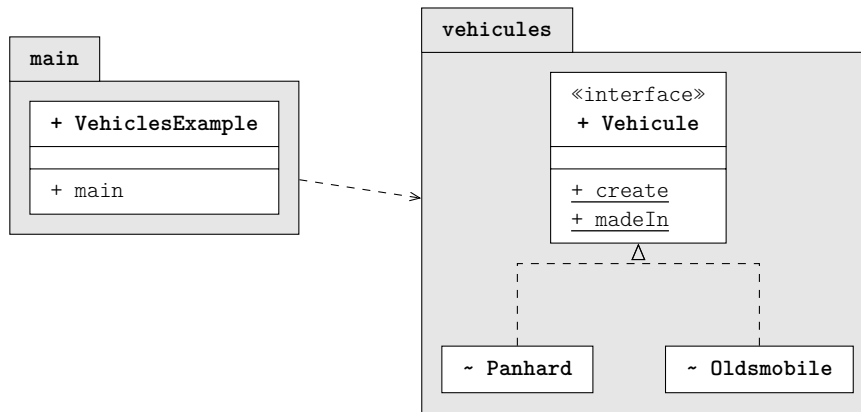


FIGURE 1.2 – Hiérarchie des véhicules.

Dans le passage à Java 8, beaucoup d'interfaces standard ont été ainsi modifiées. Par exemple, l'interface `Comparator<T>` de Java 7 ne contenait qu'une seule méthode abstraite. En Java 8, non seulement la méthode abstraite est encore présente mais sont apparues (entre autres) 9 méthodes statiques. Ces usines permettent de construire plus aisément et sûrement des comparateurs standards comme on le verra plus tard (voir Section 4.8.2 « Digression sur les comparateurs » page 239).

1.2.2 Méthodes par défaut

Un autre problème existe avec les interfaces. C'est celui des **interfaces riches** (*rich interface* vs *thin interface*).

Une interface riche est une interface contenant de nombreuses méthodes (abstraites) qui peuvent bien souvent se regrouper en des ensembles de méthodes liées

fonctionnellement les unes aux autres. On peut songer à une interface permettant d'obtenir depuis une source la valeur d'une température dans différentes unités :

TemperatureSensor : une interface riche

```
public interface TemperatureSensor {  
    public double celsiusValue();  
    public double fahrenheitValue();  
    public double kelvinValue();  
}
```

Il est évident que toute implémentation concrète de cette interface est contrainte de fournir du code pour toutes les méthodes abstraites (3 au total dans cet exemple), ce qui peut se révéler lourd et délicat.

Délicat, car cela laisse au programmeur l'entière responsabilité d'assurer que le contrat sémantique est correctement rempli ; en effet il y a une cohérence à assurer entre les valeurs de retour de ces méthodes.

Java 8 autorise donc à fournir des implémentations par défaut, de la manière suivante :

TemperatureSensor : méthodes par défaut

```
public interface TemperatureSensor {  
    public double celsiusValue(); // abstract  
  
    default public double fahrenheitValue() { // defaulted  
        return celsiusValue() * 9.0 / 5.0 + 32.0;  
    }  
  
    default public double kelvinValue() { // defaulted  
        return celsiusValue() + 273.15;  
    }  
}
```

Ainsi, une implémentation de cette interface ne nécessite plus qu'au minimum celle de la méthode abstraite :

ConcreteSensor : implémentation minimale

```
public class ConcreteSensor implements TemperatureSensor {  
    @Override public double celsiusValue() {  
        return Math.random() * 100; // read some value from a source  
    }  
}
```

Il est évident que le travail est plus léger.

Il est important de noter que rien n'empêche de redéfinir les méthodes par défaut si on le souhaite, il n'y a aucun mécanisme pour l'empêcher. Autrement dit une méthode marquée par `default` ne peut être `final`, la contrainte serait trop forte et briserait la compatibilité ascendante.



Il est interdit de fournir une implémentation par défaut aux méthodes de la classe `Object`.

Les seules méthodes éligibles sont propres au contrat de l'interface.

L'API de Java 8 a modifié un nombre sensible d'interfaces en y rajoutant des méthodes par défaut.

On a déjà cité `Comparator<T>` qui contient désormais 9 méthodes statiques mais aussi 7 méthodes par défaut en plus des 2 méthodes abstraites. Ces ajouts tardifs l'ont été dans le but d'interfacer ce type avec les nouveautés de la programmation fonctionnelle (voir le Chapitre 3 « Le style fonctionnel de Java 8 »).

Ainsi, il est important de noter qu'il est nécessaire de revoir les différentes API de Java lors d'un passage à Java 8 car de nombreuses fonctionnalités y ont été ajoutées et, pour bénéficier au mieux des avancées du langage il est essentiel de prendre le temps de s'y replonger¹.

1.2.3 Héritage de code *via* les interfaces

Si avant Java 8 l'implémentation multiple d'interfaces ne posait pas de problème, depuis on hérite aussi de code à travers les interfaces. Il faut donc de quoi résoudre les ambiguïtés d'héritage et permettre de faire un choix sur les méthodes entrant en conflit.

Soit par exemple les deux interfaces suivantes :

```
Drivable, Floatable
public interface Drivable {
    public default String getType() {
        return "Drivable";
    }
}
public interface Floatable {
    public default String getType() {
        return "Floatable";
    }
}
```

1. On mentionnera cette nécessité à de nombreuses reprises, ceci afin de souligner son importance cruciale en vue de maîtriser les nouveautés de Java.

Si on tente de définir trivialement une classe implémentant ces deux interfaces :

AmphibiousVehicle : conflit de méthodes par défaut

```
public class AmphibiousVehicle implements Drivable, Floatable {} // ambiguity
```

Le compilateur se plaint car il y a conflit non résolu entre les deux méthodes héritées. En effet, que signifierait un appel à `getType` sur un `AmphibiousVehicle` ?

On est donc conduit à redéfinir la méthode `getType` dans la classe afin de décider quel usage (éventuel) on fera de ces deux méthodes héritées.

Java propose donc une extension syntaxique du mot-clé `super` qui permet de rechercher dans un type immédiatement supérieur (au sens de la hiérarchie des types) une méthode :

AmphibiousVehicle : résolution de conflit

```
public class AmphibiousVehicle implements Drivable, Floatable {
    @Override public String getType() {
        return Floatable.super.getType() +
            " & " + Drivable.super.getType();
    }
}
```

Attention, des précautions sont à prendre lorsqu'on fait face à une hiérarchie de hauteur 2.

Par exemple, étant donné l'interface suivante sous-type de `Drivable` :

FastDrivable

```
public interface FastDrivable extends Drivable {}
```

On ne peut définir l'implémentation suivante :

FastCar : méthode par défaut inaccessible

```
public class FastCar implements FastDrivable {
    @Override public String getType() {
        return "Fast" + Drivable.super.getType(); // forbidden
    }
}
```

La classe `FastCar` n'a pas accès à la méthode `getType` de `Drivable` qui n'est pas un type mentionné explicitement dans la relation d'implémentation.

Il y a trois façons de s'en sortir :

1. ne pas redéfinir cette méthode qui sera donc héritée indirectement ;

2. la définir en faisant appel à celle héritée indirectement :

```
FastCar
public class FastCar implements FastDrivable {
    @Override
    public String getType() {
        return "Fast_" + FastDrivable.super.getType();
    }
}
```

3. la définir mais sans faire appel à la méthode indirectement héritée :

```
FastCar
public class FastCar implements FastDrivable {
    @Override
    public String getType() { return "Speedy..." ; }
}
```

On notera que sans préfixe, le mot-clé `super` continue de faire référence à la classe dont on hérite. Si préfixe il y a, il doit nécessairement être d'un type interface que l'on implémente explicitement et qui contient explicitement ou non une définition de la méthode à atteindre.

1.2.4 Méthodes privées dans les interfaces (Java 9)

Java 8 ayant introduit la possibilité de spécifier du code dans les interfaces, la nécessité de factoriser ce code est apparue. Comme il est fréquent de factoriser du code dans des méthodes privées, Java 9 autorise désormais la définition de méthodes privées (statiques ou non).



On rappelle que les méthodes privées comme les statiques ne sont pas éligibles à la **liaison dynamique** (*dynamic dispatch*) ; leur présence ne modifie donc en rien le comportement standard des interfaces et de leurs sous-types existants. La compatibilité ascendante est donc aussi assurée sur ce point.

Ainsi, on peut écrire :

```
FastDrivable : revisité
public interface FastDrivable extends Drivable {
    private static String concat(String s1, String s2) {
        return s1 + "_" + s2;
    }
}
```

```
@Override
public default String getType() {
    return concat("Fast", Drivable.super.getType());
}
}
```

1.3 Inférence de type avec var (Java 10)

On trouve dans la littérature les deux définitions suivantes de l'inférence de type. D'abord celle de Xavier Leroy¹ :

3.1 Introduction à l'inférence de types

Pour un langage statiquement typé, un typeur est un algorithme qui prend un programme en entrée et détermine si ce programme est typable ou non.

Inférence complète de types

Le programme source ne contient aucune déclaration de type sur les paramètres de fonctions ni sur les variables locales. Le typeur détermine le type des paramètres de fonctions d'après l'utilisation qui en est faite dans le reste du programme.

Puis, celle due à Lucas Cardelli et Peter Wegener² :

7. Type Checking and Type Inference

In conventional typed languages, the compiler assigns a type to every expression and subexpression. However, the programmer does not have to specify the type of every subexpression of every expression : type information need only be placed at critical points in a program, and the rest is deduced from the context. This deduction process is called type inference. Typically, type information is given for local variables and for function arguments and results. The type of expressions and statements can then be inferred, given that the type of variables and basic constants is known.

On définit donc, pour notre part, l'inférence de type ainsi :

Inférence de type

Définition 1.3.1 *L'inférence de type est le processus par lequel un compilateur (ou interpréteur) d'un langage de programmation déduit le type le plus approprié d'une expression, d'une variable ou d'une fonction, sans que le programmeur ait besoin de l'indiquer explicitement.*

C'est un mécanisme qui permet d'alléger le code tout en conservant la sûreté du typage.

1. LEROY, X. (1999). Notes de cours « Typage et programmation ». <https://xavierleroy.org/dea/typage/index.html>

2. CARDELLI, L., & WEGNER, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17, 471-522. <https://api.semanticscholar.org/CorpusID:2921816>

Opérateur diamant

Il existait déjà auparavant certaines formes d'inférence de type, comme avec l'opérateur « diamant » (introduit en Java 7) :

JLS 21

15.9 Class Instance Creation Expressions

If the type argument list to the class is empty — the diamond form <> — the type arguments of the class are inferred. It is legal, though strongly discouraged as a matter of style, to have white space between the "<" and ">" of a diamond. **15.9.1 Determining the Class being Instantiated**

If <> is present, then C will have its type arguments inferred in §15.9.3.

On l'utilise par exemple ainsi :

```
List<Integer> list = new ArrayList<>();
```

La présence de l'opérateur diamant (<>) ne signifie pas qu'on ne passe pas de valeur pour le paramètre d'instanciation du générique mais qu'on laisse le compilateur deviner lequel.

Ce dernier utilise alors le contexte pour retrouver (assez trivialement) qu'on souhaite une instanciation avec `Integer`, car aucun autre choix ne convient (pour les détails, se reporter au Chapitre 2 « La généricité de Java 5 »).

Inférence avec var (Java 10)

L'inférence à la déclaration des variables¹ utilise le mot-réserve `var` qui permet d'omettre un type lors d'une déclaration :

```
var list = new ArrayList<Integer>();
```

L'inférence ne peut fonctionner que s'il existe un contexte qui puisse permettre de déterminer un type adéquat qui n'est pas explicitement spécifié.

Ici le compilateur utilise l'expression d'initialisation pour déterminer le type de la variable. Attention, dans notre cas le type trouvé sera **exactement** celui de l'expression à droite, soit `ArrayList<Integer>`.

Cette écriture allégée a l'avantage de ne contenir qu'un seul point de définition du type, ce qui facilite le réusinage. Bien qu'il soit possible d'utiliser `var` à de nombreux endroits, il est fortement déconseillé de l'utiliser abusivement.

1. GOETZ, B. (2022). *JEP 286 : Local-Variable Type Inference*. <https://openjdk.org/jeps/286>

Par exemple il vaut mieux éviter de l'utiliser sur les types primitifs :

```
var i = 3;  
var z = i/3.5;
```

Ceci n'améliore en rien ni l'écriture, ni la lecture et certainement pas le réusinage ; bien au contraire car cela impose par exemple au relecteur de ce code d'analyser (par lui-même) les expressions afin d'en retrouver le type ; les humains sont tellement faillibles qu'il vaut mieux éviter.

var dans les boucles étendues

Un usage fréquent et très raisonnable est de l'utiliser dans les boucles étendues :

```
var l = List.of(1,2,3,4);  
for (var e: l) {  
    System.out.println(e);  
}
```

L'avantage (important) est qu'un réusinage qui conduirait à changer le type de la variable `l` (qui devra rester un `Iterable<T>`) ne conduirait pour autant à aucune modification de la boucle. Le code est donc rendu plus abstrait en faisant disparaître les types, laissant le compilateur trouver le bon ; ceci sans pour autant sacrifier la lisibilité.

1.3.1 var et les types anonymes

L'inférence de type avec `var` est plus puissante qu'il n'y paraît. Elle permet de typer des variables par un type qui n'est pas vraiment exprimable dans le langage, par exemple un type anonyme.

Soit le code suivant :

```
Object o = new Object() {  
    public int v = 666;  
};  
System.out.println(o.v); //forbidden
```

La dernière ligne provoque une erreur de compilation. En effet, bien que `o` désigne un objet (d'un sous-type de `Object`) ayant un attribut de nom `v`, il n'est pas possible d'y accéder puisque `o` est typé par `Object` et que ce type ne contient pas de champ `v` (ceci ne fonctionnerait pas non plus avec une méthode). Une telle définition se révèle alors assez limitée.

Notons que le type concret de l'objet est un type possédant un nom mais que ce nom n'est pas exprimable dans un code source.

L'inférence *via* `var` fonctionne parfaitement dans ce cas car le compilateur connaît le type anonyme puisqu'il l'a fabriqué ; le type est anonyme pour le programmeur mais pas pour le compilateur.

Le code suivant est donc correct :

```
var o = new Object() { public int v = 666; };  
System.out.println(o.v); // correct!
```

Il produit à son exécution :

```
666
```

Ceci fonctionne car comme on l'a déjà expliqué, `var` capture le type de l'expression d'initialisation, et ce quel que soit ce type, y compris s'il n'est pas exprimable explicitement dans le code source.

En examinant le bytecode généré, on peut y apercevoir le nom du type généré :

```
Code:  
0: new          #16      // class mypackage/Test$1  
3: dup  
4: invokespecial #18      // Method mypackage/Test$1."<init>":()V  
7: astore_1  
8: getstatic     #19      // Field java/lang/System.out:Ljava/io/PrintStream;  
11: aload_1  
12: getfield      #25      // Field mypackage/Test$1.v:I  
15: invokevirtual #29      // Method java/io/PrintStream.println:(I)V  
18: return
```

Le type est `Test$1` où `Test` est le nom de la classe qui contient la classe anonyme et `1` le nom de la classe anonyme déterminé par le compilateur.



S'il est possible de créer une classe dont le nom contient \$, par conséquent un nom comme `Test$1`, cela pose de véritables problèmes car le nom entrerait possiblement en conflit avec ceux des classes anonymes.

JLS 21

3.8 Identifiers

The **dollar sign should be used only in mechanically generated source code** or, rarely, to access pre-existing names on legacy systems.

Java 11 permet aussi d'utiliser `var` avec les λ -expressions (voir la Section 3.3.2 « `var`, annotations et λ -expressions (Java 11) » page 130).

1.3.2 Restrictions

Le mot réservé `var` ne peut pas être utilisé pour la spécification d'un type de retour de méthode :

```
var f() { return 12; }
```

Le mot réservé `var` ne peut pas être utilisé pour définir plusieurs variables :

```
var i = 12, j = 13; //forbidden
```

Le mot réservé `var` ne peut être utilisé pour définir des champs :

```
class K { var i = 3; }
```

Le mot réservé `var` ne peut être employé pour typer des paramètres (où il est évident qu'il n'y a pas de contexte disponible pour inférer un type).

1.4 Nouveau switch et expressions-switch (Java 14)

L'instruction `switch` ayant de nombreux inconvénients, Java 14¹ a définitivement adopté une nouvelle syntaxe pour les `switch` afin de les rendre à la fois moins verbeux et surtout plus sûrs.

Avec le `switch` traditionnel (qui continue d'exister bien entendu), on écrivait des choses comme :

Programme 1.7: `switch` traditionnel

```
public class TestSwitch {
    public static void main(String[] args) {
        try (var scanner = new Scanner(System.in)) {
```

1. BIERMAN, G. (2022a). *JEP 361 : Switch Expressions*. <https://openjdk.org/jeps/361>

```

System.out.print("Give me a car brand: ");
String brand = scanner.nextLine().toLowerCase();
String country = "unknown";
switch (brand) {
    case "peugeot":
    case "renault":
    case "hotchkiss":
        country = "french";
        break;
    case "ford":
    case "buick":
        country = "american";
}
System.out.println(brand+" is a "+country+" car brand.");
}
}
}

```

Ce type de code possède plusieurs inconvénients :

- il est nécessaire de mettre des `break` sous peine d'avoir de sérieux ennuis. D'ailleurs l'absence de `break` dans les derniers cas posera certainement un problème lorsqu'il s'agira d'ajouter de nouveaux cas ;
- il n'y a pas d'obligation à avoir un cas par défaut (`default`). Il est donc difficile d'assurer que tous les cas sont effectivement considérés ;
- il possède pas mal de redondance (répétition des `case`). Si cette répétition peut être évitée en Java 14 où l'on peut utiliser `case v1, v2, ...`, il vaut bien mieux utiliser le nouveau `switch` qui comble aussi les défauts précédents.

Une nouvelle syntaxe a donc été introduite permettant de pallier aux défauts de la forme initiale :

Programme 1.8: `switch` de Java 14

```

public class TestSwitch {
    public static void main(String[] args) {
        try (var scanner = new Scanner(System.in)) {
            String brand = scanner.nextLine();
            String country = "unknown";
            switch(brand.toLowerCase()) {
                case "peugeot", "renault", "hotchkiss" -> country = "french";
                case "ford", "buick" -> country = "american";
            }
            System.out.println(brand+" is a "+country+" car brand.");
        }
    }
}

```

La syntaxe du nouveau switch consiste en :

- le mot-clé `case` suivi d'une liste de valeurs séparées par une virgule ;
- le symbole fonctionnel `->` ;
- une expression ou un bloc de code.



Attention, on ne peut mélanger dans un même switch ancienne et nouvelle syntaxe, il faut choisir.

Le code est plus concis et lisible, d'autant qu'il n'est plus nécessaire d'y mettre des `break` dont l'absence ou l'oubli conduit à trop de problèmes. Les cases du nouveau switch sont nécessairement exclusifs.

1.4.1 Expression switch (Java 14)

Cette nouvelle syntaxe permet aussi d'obtenir l'expression-switch. On rappelle qu'une expression produit une valeur tandis qu'une instruction non. Dans l'exemple précédent, le code consiste à positionner une même variable à différentes valeurs (selon les cas). L'idée est donc de faire en sorte que le switch **produise** une valeur, laquelle pourra ensuite être captée dans une autre expression.

Pour se faire, on va donc externaliser l'affectation :

Programme 1.9: Expression-switch

```
public class TestSwitch {  
    public static void main(String[] args) {  
        try (var scanner = new Scanner(System.in)) {  
            System.out.print("Give me a car brand: ");  
            String brand = scanner.nextLine();  
            String country = switch (brand.toLowerCase()) {  
                case "peugeot", "renault", "hotchkiss" -> "french";  
                case "ford", "buick" -> "american";  
                default -> "unknown";  
            };  
            System.out.println(brand + " is a " + country + " car brand.");  
        }  
    }  
}
```

Dans le cas d'un tel switch, à droite de l'opérateur fonctionnel (`->`), c'est une expression qui est attendue (dans les cas simples). La valeur de cette expression déterminera la valeur du switch lorsque le cas correspond. En retour de l'expression-switch la valeur est donc captée dans la variable `country`.

Il faut noter que dans le cas d'une telle expression-switch, le compilateur vérifie que tous les cas possibles sont couverts¹ ce qui améliore la sûreté du code.

Comme il n'est pas possible d'être exhaustif sur les valeurs du type String, c'est le cas default qui permet d'assurer que tout est correctement traité. En cas d'absence de default, le compilateur refusera de compiler le code.

Retour de valeur : yield

Dans le cas où l'on souhaite réaliser un traitement plus compliqué que de renvoyer la valeur d'une expression, on peut utiliser un bloc d'instructions.

En cas, il faut un moyen de spécifier la valeur de l'expression-switch; c'est le mot-réserve yield qui permet de signifier que la valeur de l'expression qui suit sera la valeur de l'expression-switch :

Programme 1.10: yield pour retourner une valeur

```
public class TestSwitch {
    public static void main(String[] args) {
        try (var scanner = new Scanner(System.in)) {
            System.out.print("Give me a car brand: ");
            String brand = scanner.nextLine();
            String country = switch (brand.toLowerCase()) {
                case "peugeot", "renault", "hotchkiss" -> "french";
                case "ford", "buick" -> "american";
                default -> {
                    System.out.println("oups");
                    yield "unknown";
                }
            };
            System.out.println(brand + " is a " + country + " car brand.");
        }
    }
}
```

yield est, pour les expressions-switch, un équivalent de return pour les méthodes.

Inférence de type pour l'expression-switch

Il est permis de renvoyer des valeurs de types différents pour les cas et de laisser l'inférence fonctionner afin de déterminer un type adéquat pour l'expression-switch.

1. Ce qui est bien nécessaire sinon quelle serait la valeur de l'expression pour un cas non couvert ?

On peut donc écrire :

Programme 1.11: Expression-switch : inférence de type

```
public class SwitchInferenceExample {  
    public static void main(String[] args) {  
        for (int v=1; v<5; v++) {  
            var i = switch(v) {  
                case 1 -> Integer.MAX_VALUE;  
                case 2 -> Double.valueOf(2.5);  
                case 3 -> 4.5f;  
                default -> (short)0;  
            };  
            System.out.println(i);  
        }  
    }  
}
```

Dans ce cas, le compilateur détermine que `double` est un type adéquat. On ne détaillera pas le mécanisme d'inférence qui est très complexe, mais il suffit de savoir que le compilateur détermine le **plus petit sur-type commun** (*lub* pour *least upper bound*)

JLS 21

4.10.4 Least Upper Bound

The least upper bound, or "lub", of a set of reference types is a shared supertype that is more specific than any other shared supertype (that is, no other shared supertype is a subtype of the least upper bound).

Ici il ne s'agit que de nombres (éventuellement emboîtés pour certains) alors le compilateur prend le plus grand type capable de stocker les valeurs, et donc le type `double`. Sa préférence allant vers un type primitif.

Dans le cas suivant :

```
public class SwitchInferenceExample {  
    public static void main(String[] args) {  
        for (int v=1; v<3; v++) {  
            var i = switch(v) {  
                case 1 -> 1;  
                default -> true;  
            };  
            System.out.println(i);  
        }  
    }  
}
```

Le type inféré est (voir Section 2.14 « Types intersections » page 115 pour les détails concernant un tel type) :

```
Object & Comparable<?> & Constable & Serializable
```

Dans ce type celui qui « domine » est le premier, c'est-à-dire `Object` ; on ignorera donc les autres dans notre justification. Le plus important est de comprendre pourquoi c'est `Object`.

Les types primitifs `int` et `boolean` n'ont pas de plus petit super type commun, par conséquent le compilateur emboîte les valeurs (de la sorte il passe dans le monde des objets où l'on sait qu'il existe nécessairement un plus grand type). On obtient donc `Integer` et `Boolean`. Ces deux types n'ayant qu'un seul plus petit sur-type commun qui est `Object`, celui-ci est donc choisi.

1.4.2 switch et null (Java 21)

Java 17 a introduit en incubation la possibilité d'inclure `null` dans les cas d'un `switch` et l'a définitivement intégré en version 21.

Auparavant, il fallait être défensif en testant la nullité de la référence avant le `switch` :

```
Integer i = ...;
if (i==null) {
    System.out.println("null");
} else {
    switch (i) {
        case 1 -> System.out.println("1");
        case 2 -> System.out.println("2");
        default -> System.out.println("big");
    }
}
```

Désormais il est possible de tester la nullité dans les cas eux-mêmes :

```
switch (i) {
    case null -> System.out.println("null");
    case 1 -> System.out.println("1");
    case 2 -> System.out.println("2");
    default -> System.out.println("big");
}
```

Il est aussi possible de factoriser les cas `null` et `default` si besoin est.

On peut ainsi écrire :

```
switch (i) {  
    case null , default -> System.out.println("oops");  
    case 1      -> System.out.println("1");  
    case 2      -> System.out.println("2");  
}
```

1.5 record (Java 17)

Les *records* ont été introduits en Java 14 puis définitivement intégrés en Java 17 (voir BIERMAN, 2024a). L'idée est de permettre de créer des classes élémentaires sans payer le prix de la verbosité.

En effet, en Java « classique » si l'on souhaite créer un type élémentaire permettant de représenter, par exemple, des points dans l'espace, on écrit généralement quelque chose comme :

Point : un type défini classiquement

```
public class Point {  
    public final int x;  
    public final int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
    @Override  
    public String toString() {  
        return "Point[" + x + ", " + y + "]";  
    }  
}
```

Pour une fonctionnalité somme toute minimale, on est conduit à produire un nombre non négligeable de lignes de code plutôt inutiles et répétitives (*boilerplate code*). Bien entendu, la plupart des outils nous aident à synthétiser ce genre de construction mais le résultat est là : de nombreuses lignes de code, quoiqu'on fasse.

Les *records* sont donc une manière plus élégante et sûre de synthétiser un tel type en en décrivant que l'essentiel.

Ainsi, le type précédent peut être décrit de façon extrêmement concise à l'aide d'un `record` :

record : la synthèse du type `Point`

```
public record Point(int x, int y) {}
```

Une et une seule ligne de code pour obtenir le même résultat (à peu de choses près et en faveur du `record`).

1.5.1 Composants d'un record

On appelle **composants** (*components*) les éléments typés de la liste qui suit le nom du `record`. Les composants définissent à la fois des champs, des méthodes et un constructeur (tous synthétisés). Dans notre exemple, `x` et `y` sont les composants.

La liste ordonnée des composants définit :

- des **champs** `private`, `final` de noms et types indiqués ;
- un **constructeur canonique** (*canonical constructor*) avec des arguments ordonnés de la même façon ;
- des **getters** pour ces composants ;
- une **redéfinition** adéquate de `toString`, `hashCode` et `equals`.



Attention les getters ne sont pas synthétisés en utilisant la convention habituelle (`getAttribut`, une convention héritée des JavaBeans) mais comme méthodes portant le nom des attributs.

Ainsi, on peut utiliser ce type de la manière suivante :

```
Point p = new Point(5,6);
System.out.println(p);
System.out.print("Ordinateur is " + p.y());
```

qui produit à l'exécution :

```
Point[x=5, y=6]
Ordinateur is 6
```

Une autre différence importante est que les champs du `record` sont finaux, on ne peut donc les modifier. Les instances créées sont alors **non modifiables**.



Attention, la synthèse des *getters* empêche la définition de composants qui auraient pour nom celui d'une méthode héritée d'`Object`.

JLS 21

8.10.1 Record Components

It is a compile-time error for a record declaration to have a record component with the name `clone`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, or `wait`.

record variadique

Puisque la définition d'une liste de composant correspond à la définition implicite d'un constructeur de même signature, il est donc autorisé de construire un record variadique.

Comme pour les méthodes variadiques, l'argument variadique doit être le dernier de la liste :

Tuple : record variadique

```
public record Tuple(int ...values) {}
```

Ce type s'utilise très naturellement :

Programme 1.12: record variadique, exemple d'un tuple d'entiers

```
public class RecordExample {
    public static void main(String[] args) {
        Tuple t = new Tuple(1, 2, 3, 4, 5);
        System.out.println(t.values()); // On récupère un tableau...
    }
}
```

On peut donc utiliser de tels types record comme de petites collections légères :

IntTuple

```
public record IntTuple(int... values) implements Iterable<Integer> {
    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int idx = 0;
            @Override public boolean hasNext() { return idx < values.length; }
            @Override public Integer next() { return values[idx++]; }
        };
    }
}
```

On peut l'utiliser comme dans le programme suivant :

Programme 1.13: record : variadique comme exemple d'un itérable d'entiers

```
public class RecordExample {
    public static void main(String[] args) {
        var t = new IntTuple(1, 2, 3, 4, 5);
        System.out.println(t.values());
        for (var e: t) { System.out.println(e); }
    }
}
```

Ce qui produit à l'exécution :

```
1
2
3
4
5
```

1.5.2 Constructeur canonique

Il peut être nécessaire de définir soi-même le constructeur canonique, par exemple dans le cas où l'on souhaite réaliser des contrôles sur les valeurs passées. Il y a deux façons de procéder :

- par **définition** (*normal canonical constructor*);
- ou par **enrichissement** (*compact canonical constructor*).

Ce que précise la spécification du langage :

JLS 21

8.10.4 Record Constructor Declarations

a record class has a canonical constructor, declared explicitly or implicitly, that initializes all the component fields of the record class.

8.10.4.1 Normal Canonical Constructors

A (non-compact) constructor in the declaration of record class R is the **canonical constructor** of R if its signature is override-equivalent to the derived constructor signature of R.

8.10.4.2 Compact Canonical Constructors

A compact constructor declaration is a succinct form of constructor declaration, only available in a record declaration. It declares the **canonical constructor** of a record class without requiring the record components of the class to be manually repeated as formal parameters of the constructor.

Par définition (*normal canonical constructor*)

La première façon de définir le constructeur canonique est de le définir intégralement, en utilisant donc la signature respectant la déclaration des composants.

Ainsi on peut écrire :

record : définition du constructeur canonique

```
public record Point(int x, int y) {  
    public Point(int x, int y) { // canonical constructor, explicit definition  
        if (x<0 || y<0) throw new IllegalArgumentException("bad");  
        this.x = x;  
        this.y = y;  
    }  
}
```



Si on définit soi-même le constructeur canonique, il est impératif d'initialiser correctement les champs.

Par enrichissement (*compact canonical constructor*)

L'enrichissement du constructeur canonique consiste à déléguer la synthèse de l'initialisation des champs au compilateur tout en autorisant le rajout de fonctionnalités préliminaires à ces initialisations (en général des contrôles sur les valeurs des paramètres). De la sorte, seul le code « utile » apparaît dans la définition du type.

JLS 21

8.10.4.2 Compact Canonical Constructors

The intent of a compact constructor declaration is that only code to validate or normalize parameters needs to be given in the constructor body; the remaining initialization code is supplied by the compiler.

Par exemple, le code de définition du constructeur canonique précédent peut être simplement agrémenté du contrôle de la validité des paramètres de la manière suivante :

record : enrichissement du constructeur canonique

```
public record Point(int x, int y) {  
    public Point { // canonical constructor enrichment  
        if (x<0 || y<0) {  
            throw new IllegalArgumentException("bad");  
        }  
    }  
}
```

La syntaxe consiste à déclarer le code d'enrichissement du constructeur sous une forme particulière : le nom du type, **pas de liste de paramètres**, puis un code qui **sera exécuté avant** l'exécution du constructeur canonique synthétisé.



Il est interdit de définir et d'enrichir en même temps le constructeur canonique.

1.5.3 Constructeurs additionnels

Il est tout à fait possible d'ajouter des variantes de constructeur, mais il est alors impératif de faire appel à un constructeur (canonique ou non) en première instruction. Il est aussi impératif que la chaîne de ces appels termine par un appel au constructeur canonique.

On peut donc écrire :

record : constructeurs additionnels

```
public record Point(int x, int y) {  
    public Point {  
        if (x<0 || y<0) {  
            throw new IllegalArgumentException("bad");  
        }  
    }  
    public Point() { this(0); }  
    public Point(int x) { this(x,0); }  
}
```

Définition explicite des getters

Une définition explicite des getters est possible, il suffit de respecter le typage qui doit être conforme à la déclaration des composants :

record : définition explicite d'un getter

```
public record Point(int x, int y) {  
    public int x() { System.out.println("in_x()"); return this.x; }  
}
```

1.5.4 Ajout de méthode

Il est possible d'enrichir un type `record` avec des méthodes qu'elles soient d'instances ou statiques. Ceci peut permettre par exemple d'ajouter des getters JavaBeans.

record : ajout de méthodes

```
public record Point(int x, int y) {  
    public double distanceTo(Point p) {  
        int dx = x-p.x;  
        int dy = y-p.y;  
        return Math.sqrt(dx*dx+dy*dy);  
    }  
    public static double distanceBetween(Point p1, Point p2) {  
        return p1.distanceTo(p2);  
    }  
    public int getX() { return x(); }  
    public int getY() { return y(); }  
}
```

1.5.5 Restrictions

Certaines choses sont interdites concernant les record :

- un record ne peut étendre une classe. De façon implicite un record étend la classe `Record` et il est interdit d'en faire mention explicitement ;
- un record est un type `final`, on ne peut donc étendre un record. Il ne peut donc pas être non plus `abstract` ;
- il est interdit d'ajouter des champs d'instance à un record et il n'est pas non plus possible de définir des blocs d'initialisation d'instance. Par contre il est possible de définir des champs statiques ainsi que des blocs d'initialisation statique ;
- il n'est pas non plus possible d'y définir des méthodes natives.

1.6 Types scellés (Java 17)

Le scellement d'un type, qui est apparu en Java 17¹, permet de spécifier quels sont ses sous-types acceptables et de permettre au compilateur de vérifier que ces implémentations sont effectivement fournies et pas plus (on verra un peu plus loin que cela permet aussi d'assurer des filtrages exhaustifs).

Effet, lors d'une conception, il est fréquent de faire apparaître un type abstrait et des sous-types correspondants. Mais, il est parfois souhaitable de préciser que ce sous-typage est fermé à une liste prédéfinie de sous-types acceptables ; ceci parce que le domaine d'application n'en autorise que certains.

Le mot-clé `sealed` (associé à `permits`) permet de spécifier une telle contrainte. Ainsi fait, on sera certain qu'on ne trouvera pas d'implémentation « exotique ».

1. BIERMAN, G. (2024b). *JEP 409 : Sealed Classes*. <https://openjdk.org/jeps/409>

Par exemple, dans un jeu on souhaiterait ne faire apparaître que deux types de **personnages** (*character*) :

- les **gentils** (*friendly*);
- et les **méchants** (*wicked*).

UML représente ce cas ainsi :

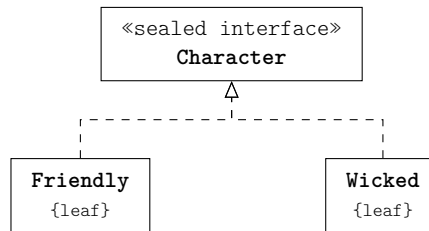


FIGURE 1.3 – Hiérarchie close en UML.

Avant Java 17, il n'était pas possible de l'assurer puisqu'au minimum on expose une interface (*Character*) et par conséquent cela laisse la liberté (parfois non désirée) d'en réaliser n'importe quelle implémentation. Pour exprimer cette contrainte on peut désormais écrire :

```

Character, Friendly, Wicked
sealed public class Character permits Friendly, Wicked {}
final public class Friendly extends Character {}
final public class Wicked extends Character {}
  
```

La clause *permits* spécifie la liste des seuls sous-types directs acceptables pour le **type scellé** (*sealed type*) *Character*. Non seulement ce sont les seuls sous-types acceptables mais leur existence est impérative.

Toute violation de cette règle conduit le compilateur à émettre une erreur. Par exemple, si l'on omet la définition de la classe *Wicked*, le compilateur indique quelque chose comme :

```

Test.java:3: error: cannot find symbol
sealed class Character permits Friendly, Wicked {}
                                         ^
    symbol: class Wicked
Test.java:3: error: invalid permits clause
sealed class Character permits Friendly, Wicked {}
                                         ^
    (illegal reference to supertype Wicked)
2 errors
  
```

Si on tente de créer une sous-classe qui n'apparaît pas dans la clause `permits` le compilateur indique :

```
Character.java:5: error: class is not allowed to extend sealed class: ►
    ► Character (as it is not listed in its 'permits' clause)
public class Nobody extends Character {} // compile-time error
    ~
1 error
```

D'autre part, on remarque que dans notre exemple (simple) les sous-classes sont marquées avec `final`. En effet tout sous-type d'un type scellé doit être :

- lui-même scellé avec `sealed`, dans ce cas le sous-type n'autorise que les sous-types éventuellement spécifiés par sa propre clause `permits` ;
- réouvert avec `non-sealed`, dans ce cas le sous-type n'interdit rien quant à son propre sous-typage ;
- `final` et dans ce cas rien n'est plus permis.

1.6.1 Sous-types sealed

On peut par exemple autoriser explicitement à ce qu'il y ait deux sous-types de gentils : les **sympas** (*kind*) et les **super-sympas** (*nice*).

La hiérarchie scellée est donc la suivante :

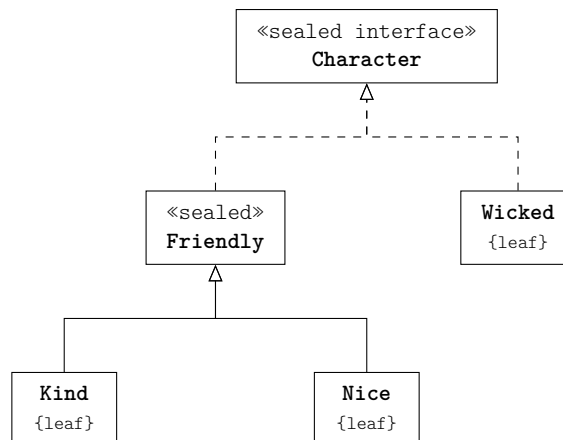


FIGURE 1.4 – Hiérarchie close à plusieurs niveaux.

Il est donc nécessaire de fournir une nouvelle définition du type `Friendly` pour l'ouvrir au sous-typage et déclarer ses sous-types acceptables.

Ce qui s'écrit :

Friendly

```
sealed public class Friendly extends Character permits Kind, Nice {}
```

Et, ce qui impose la définition des sous-types correspondants :

Kind, Nice

```
final public class Kind extends Friendly {}  
final public class Nice extends Friendly {}
```

1.6.2 Sous-types non-sealed

On peut aussi laisser la liberté d'ouvrir intégralement le sous-type `Wicked` à n'importe sous-typage en utilisant la clause `non-sealed` :

Wicked

```
non-sealed public class Wicked extends Character {}
```

Puis définir n'importe quel(s) sous-type(s) (lesquels pouvant être scellés ou non, etc) :

Zombie, Brute

```
public class Zombie extends Wicked {}  
public class Brute extends Wicked {}
```

1.6.3 Sous-types implicites

Il est possible de ne pas spécifier de clause `permits`, mais en ce cas les seuls sous-types autorisés doivent être définis dans la même unité de compilation (c'est-à-dire le même fichier code source `.java`).

JLS 21

8.1.6 Permitted Direct Subclasses

If the declaration of a sealed class `C` lacks a `permits` clause, then the permitted direct subclasses of `C` are as follows :

- If `C` is not an enum class, then its permitted direct subclasses are those classes **declared in the same compilation unit** as `C`.

Ainsi le compilateur peut reconstruire aisément la liste des sous-types acceptés.

1.7 Filtrage par motif structurel (Java 16)

Le filtrage a été introduit en Java 13¹, a été définitivement adopté en Java 16² puis a subi différentes améliorations dans les versions ultérieures.

On peut définir ce concept de la manière suivante :

Filtrage par motif

Définition 1.7.1 *Le **filtrage par motif structurel** (structural pattern matching) est la possibilité de tester qu'une donnée structurée est d'une forme donnée et si c'est le cas d'y associer une action éventuellement paramétrée par les valeurs qui s'y trouvent.*

1.7.1 Filtrage basique (Java 16)

Le filtrage basique consiste à déterminer si une variable est d'un type donné. Une telle opération est traditionnellement réalisée à l'aide d'une **transtypage vers le bas** (*downcasting*).

On écrit alors quelque chose comme :

```
Object o = ...;
if (o instanceof String) {           // test if downcast is possible
    String s = (String)o;           // effective downcast
} else if (o instanceof Number) {
    Number n = (Number)o;
}
```

C'est évidemment à la fois extrêmement redondant (trois occurrences de `String` et `Number`) et très sensible aux erreurs.

Le mécanisme de **filtrage** permet le transtypage sans la lourdeur évoquée et repose sur :

- un **prédicat de typage**;
- une **capture du transtypage** réalisé dans une variable cible.

Il utilise l'opérateur `instanceof` pour construire le **prédicat et la capture en une seule expression** de la façon suivante :

```
Object o = ...;
if (o instanceof String s) {         // test and capture if possible
```

1. BIERMAN, G. (2022b). *JEP 394 : Pattern Matching for instanceof*. <https://openjdk.org/jeps/394>

2. BIERMAN, G. (2023b). *JEP 441 : Pattern Matching for switch*. <https://openjdk.org/jeps/441>

```
//play with s
} else if (o instanceof Number n) {
    //play with n
}
```

`o instanceof String` est le **prédicat** et `s` la variable qui **capture** l'objet avec le type correct si le prédicat est vrai.

Une fois la capture dans la variable réalisée, cette dernière est immédiatement utilisable, y compris dans la même expression. La visibilité de cette variable est le flot d'exécution concerné lorsque le prédicat est vérifié (cette notion de visibilité est similaire à l'analyse effectuée par le compilateur pour vérifier qu'une variable a été initialisée, c'est-à-dire une analyse de flot).

```
Object o = ...;
if (o instanceof String s && s.length()>10) { ... }
```

Ce code capture dans la variable `s` l'instance désignée par `o` si son type est `String`. Une fois cette capture réalisée la suite de l'expression peut être évaluée.

Attention, la visibilité peut être surprenante mais un code comme :

```
Object o = "Hello, World!";
if (! (o instanceof String s)) {
    System.out.println("Not a string");
    return;
} else {
    System.out.println(s.toUpperCase());
}
System.out.println(s.toUpperCase());
```

est correct ! En effet, si le prédicat est vérifié, le flot d'exécution « sort » du `if` avec la capture réalisée.

1.7.2 Filtrage avec `switch` (Java 21)

Ce filtrage a beaucoup évolué car il a été proposé en incubation à partir de Java 17 et maintes fois remanié.

Le filtrage de Java 16 conduit possiblement à une longue liste de cas, c'est pourquoi il est désormais possible de l'employer avec un `switch` (on préférera le nouveau `switch` qui assure que tous les cas sont couverts et donc impose si nécessaire la présence d'un cas par défaut).

Étant donné le type :

```
public record Pair(Object first, Object second) {}
```

Pair

On peut écrire :

Programme 1.14: Filtrage avec switch

```
public class PatternExample {  
    public static void main(String[] args) {  
        Object o = new Pair(13, "foo");  
        switch(o) {  
            case Pair p -> System.out.println(p.first()+" "+p.second());  
            default      -> System.out.println("what_ do?");  
        }  
    }  
}
```



Il est bon de rappeler que les cas des switch sont examinés dans l'ordre, y compris avec les nouveaux switch; à l'exception de la clause default et du cas null qui peuvent apparaître n'importe où.

Il n'est pas forcément nécessaire de faire apparaître un cas par défaut si le compilateur peut inférer que tous les cas sont couverts. Ainsi :

```
Integer v = 3;  
switch(v) {  
    case Integer i -> System.out.println("Integer");  
}
```

est correct. En effet, la classe Integer est finale et donc aucun autre type n'est possible. Mais ceci fonctionne tout aussi bien :

```
Integer v = 3;  
switch(v) {  
    case Number n -> System.out.println("Number");  
}
```

et aucun autre cas ne peut-être spécifié non plus !

En effet, si on place un cas Integer avant le cas Number ce dernier ne pourra être atteint et le compilateur se plaindra (les codes morts sont interdits), et bien évidemment

le cas `Integer` ne peut-être placé à la suite car il serait couvert par le cas `Number` (voir dominance Section 1.7.6 « Dominance des cas » page 40).

Ce filtrage permet de réécrire le code de l'interface `Vehicle` (voir programme 1.4, page 9) de la façon suivante :

Vehicle : filtrage par motif structurel

```
public interface Vehicle {
    public static Vehicle create(String countryCode) throws Exception {
        return switch(countryCode) {
            case "fr" -> new Panhard();
            case "us" -> new Oldsmobile();
            default -> throw new Exception("Bad_country_code");
        };
    }
    public static String madeIn(Vehicle v) {
        return switch(v) {
            case Panhard p -> "fr";
            case Oldsmobile o -> "us";
            default -> "unknown";
        };
    }
}
```

1.7.3 Filtrage avec gardes (Java 21)

Il est possible de compléter un cas par une condition supplémentaire afin de raffiner la capture.

Ceci sert notamment à éviter d'alourdir le traitement d'une capture avec une litanie de tests de sous-cas particuliers.

Pour cela on peut compléter le cas avec une expression `when` permettant de tester un prédicat :

Programme 1.15: Filtrage avec garde

```
public class GuardExample {
    public static void main(String[] args) {
        try (var scan = new Scanner(System.in)) {
            System.out.print("What? ");
            switch (scan.nextLine()) {
                case "y", "Y"
                    -> System.out.println("You win");
                case String s when s.length() < 10
                    -> System.out.println("Short string");
                case String s when s.length() >= 10
```

```

        -> System.out.println("Long_string");
    default
        -> System.out.println("Weird");
    }
}
}
}
}

```



La clause `when` ne peut être utilisée qu’avec une clause de filtrage. On ne peut écrire `case 13 when f()==12`.

D’autre part, il faut faire particulièrement attention à la dominance des cas (voir Section 1.7.6 « Dominance des cas » page 40).

1.7.4 Filtrage et classes scellées

L’exhaustivité des clause est vérifiée lorsque les types sont scellés. Ainsi étant donnés les types scellés suivants :

Shape, Oval, Triangle, Rectangle

```

public sealed abstract class Shape permits Oval, Triangle, Rectangle {}
public final class Oval extends Shape {}
public final class Triangle extends Shape {}
public final class Rectangle extends Shape {}

```

Le switch suivant est exhaustif :

Programme 1.16: switch : exhaustivité sur types scellés

```

public class PatternMatchingExample {
    public static void main(String[] args) {
        Shape f = null;
        String s = switch(f) {
            case Oval o    -> "Oval";
            case Rectangle o -> "Rectangle";
            case Triangle o  -> "Triangle";
            case null       -> "null";
        };
        System.out.println(s);
    }
}

```

Ceci fonctionne car la classe `Shape` est abstraite (cela fonctionnerait pareillement s’il s’agissait d’une interface). Si on enlève son caractère abstrait, alors il faudra ou bien un cas par défaut ou bien un cas couvrant le type `Shape`.