Java Best Practices

Un survol

Sections

Section 1 : Intro

Section 2 : Data Types

Section 3 : Methods

Section 4 : Classes and interfaces

Section 5 : Exceptions

Section 6 : Modern Java Elements

Section 7 : General Programming Rules

Section 8 : Other

Section 2

Data Types

Section 2 : Data Types

- Watch out for float and double precision
- Don’t concatenate many strings
- Pick the right data type for IDs
- Choose primitives over boxed primitives
- Understand date and time challenges

Section 2 : Data Types

Watch out for float and double precision

- Float, double (IEEE 754) :
  - pour calculs scientifiques/ingénierie
  - rapides
  - erreurs d'arrondis (0.1 est périodique en binaire)
- Pour calculs financiers :
  - calculer les cents avec des long; diviser à l'affichage
  - utiliser BigDecimal

Section 2 : Data Types

Don’t concatenate many strings

- StringBuilder.append()
- Boucle de 10 000 itérations : 2 ms versus 200 ms

Section 2 : Data Types

Pick the right data type for IDs

- Utiliser String pour business ids (visible aux utilisateurs)
  - Encode les zéros initiaux (membre 0049), gère les lettres si la politique des identifiants change.
- Ou encore le type UUID (32 caractères, ID générés presque toujours uniques).
- Utiliser long pour clés primaires (invisibles aux utilisateurs)

Section 2 : Data Types

Choose primitives over boxed primitives

- Faire int i = 0; plutôt que Integer i;
- Plus performants (5 ms versus 125 ms pour une boucle de n=10M).
- Ne peuvent pas être nuls, peuvent être comparés avec l'opérateur =.
- Utiliser les boxed primitives pour les collections.

Section 2 : Data Types

Understand date and time challenges

- Java 8 : utiliser java.time classes
- Utiliser les timestamps en UTC : Instant.now()
- LocalDate (sans fuseau horaire, pour une date de naissance ou historique p.e. 8-mai-1945)
- ZonedDateTime pour l'heure, selon un fuseau horaire

Section 3

Methods

Section 3 : Methods

- Pick the scope of method responsibility carefully
- Validate parameters
- Design method signatures carefully
- Return appropriate results
- Always use @Override

Section 3 : Methods

Pick the scope of method responsibility carefully

- One method should be responsible for **one** thing.
- CQS principle : command-query separation, une méthode modifie, lit mais pas les deux.
- Exception : push(), pop() in a stack
- Pas d'effets secondaires (side effects)
- Mauvais : Account.operation(int operationCode,..)
- Bon : Account.deposit(), .withdraw()

Section 3 : Methods

Validate parameters

- Valider les références nulles :
  - Objects.requireNonNull(param);
- Valider les index négatifs
- Lancer les exceptions si applicable
- Plus facile de détecter les bogues s'ils sont identifiés tôt

Section 3 : Methods

Design method signatures carefully

- Nom des méthodes : un verbe, court, informatif
  - √ countItems(), x itemCount()
  - Exceptions : of(), valueOf(), from()
- Préférez les interfaces aux classes :
  - √ do(List<Long> list), x do(ArrayList<Long> list)
- Maximum quatre paramètres

Section 3 : Methods

Return appropriate results

- Retourner une liste vide plutôt qu'une liste nulle
- Retourner une chaine vide plutôt qu'une chaine nulle
- Préférez les interfaces aux classes :
  - √ List<Long> getList(), x ArrayList<Long> getList()

Section 4

Classes and interfaces

Section 4 : Classes and interfaces

- Restrict access as much as possible
- Favour immutability
- Consider using static factory methods
- Consider using builders
- Design utility classes correctly
- Use dependency injection

Section 4 : Classes and interfaces

Restrict access as much as possible

- Privilégier private, puis protected, etc.
- Débuter par le plus restrictif, assouplisser au besoin
- Classes package-visible si possible, public sinon
- Méthodes privée si possible
- Constructeur protected si classe abstraite
- Champs private, sauf constantes qui peuvent être public

Section 4 : Classes and interfaces

Restrict access as much as possible

- Erreurs courantes :
  - Les test Junit 5 n'ont pas à être publics

```
@Test
void test() {..
```

  - Les controllers Spring n'ont pas à être publics

```
@RestController
class HelloController {..
```

Section 4 : Classes and interfaces

Favour immutability

- Pas de sous-classe : final class Immutable {
- Tous les champs : private final
- Tous les champs sont immuables, ou inaccessibles
- Pas de setters, ou de méthodes modifiant les champs

P.e: String, BigDecimal, LocalDate sont immuables

Section 4 : Classes and interfaces

Favour immutability

```
final class Book {
  private final String title;
  private final List<String> genres;
  public Book(String title, List<String> genres) {
    this.title = title;
    this.genres = List.copyOf(genres); //immutable copy
  }
  public String getTitle() {
  public List<String> getGenres() {
```

Section 4 : Classes and interfaces

Favour immutability

```
String title = ..
List<String> genres = ..
Book book = new Book(title, genres);

genres.clear(); //does not change book's state
book.getGenres().clear(); //runtime error
                          //cannot modify immutable list
```

Section 4 : Classes and interfaces

Consider using static factory methods

```
final class Book {
  private final String title;
  private final List<String> genres;
  private Book(String title, List<String> genres) {
    this.title = title;
    this.genres = List.copyOf(genres); //immutable copy
  }

  public static Book of(String title, List<String>..) {
    return new Book(title, genres);
  }
```

Section 4 : Classes and interfaces

Consider using static factory methods

- Les *factory methods* ont des noms

```
new Rectangle(5, 5) → Rectangle.squareOf(5);
```

- Les *factory methods* peuvent gérer les caches

```
private static Text of(String title) {
  return (title == null) ? Text.EMPTY : new Text(title);
}
```

- Les *factory methods* peuvent instancier un sous-type

Section 4 : Classes and interfaces

Consider using builders

Quand :

- Une classe a beaucoup de champs
- Certains champs sont obligatoires, ont une valeur défaut

Buts :

- éviter les constructeurs avec trop de paramètres
- éviter trop de constructeurs

Section 4 : Classes and interfaces

Consider using builders

Par exemple :

```
CsvReader reader = CsvReader.of(filename)
  .withSeparator(“;“)
  .withHeaderLines(1)
  .withCharset(StandardCharsets.ISO_8859_1)
  .withIgnoreLinesStartingWith(“#“);
  .build(); //call private constructor

String[] lines = reader.readLines();
```

Section 4 : Classes and interfaces

Design utility classes correctly

Une classe est une collection de fonctions

eg : Math.sin(), Math.cos()

Classe utilitaire :

- méthodes et champs statiques
- constructeur privé sans paramètres

Section 4 : Classes and interfaces

Use dependency injection

```
class OrderServiceImpl implements OrderService {
   private final PaymentService paymentService;
   private final InventoryService inventoryService;

   public OrderServiceImpl() {
     this.paymentService = new PaymentServiceImpl();
     this.inventoryService = new InventoryServiceImpl();
   }

   @Override
   public processOrder(..) {
```

X

Section 4 : Classes and interfaces

Use dependency injection

Injection par constructeur (la plus commune) :

```
class OrderServiceImpl implements OrderService {
   private final PaymentService paymentService;
   private final InventoryService inventoryService;

   public OrderServiceImpl(PaymentService ps, ..) {  √
     this.paymentService = ps;
     this.inventoryService = is.
   }

   @Override
   public processOrder(..) {
```

Section 4 : Classes and interfaces

Use dependency injection

- Plus facile à corriger
- Plus facile à tester unitairement
- Soutenu par de nombreux cadriciels :
  - Spring, Guice, Dagger

Section 5

Exceptions

Section 5 : Exceptions

- Throw appropriate exceptions
- Throw exceptions in if blocks
- Design catch blocks carefully
- Use meaningful error messages

Section 5 : Exceptions

Throw appropriate exceptions

- Vérifier la validation des paramètres
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `IllegalArgumentException`
  - `IllegalStateException`
  - `UnsupportedOperationException`
- Ne pas lancer RuntimeException (trop générale)

Section 5 : Exceptions

- **Throw exceptions in if blocks**

```
void withdraw(BigDecimal amount) {
  if (amount != null) {
    if (amount.compareTo(BigDecimal.ZERO >= 0) {
      if (balance.compareTo(amount)) >= 0) {
        balance = balance.substract(amount); //happy path
      } else {
        throw new InsufficientFundsException(..
      }
    } else {
      throw new IllegalArgumentException(..
    } else {
    throw new NulllPointerException(..
  }
}
```

X

Section 5 : Exceptions

- **Throw exceptions in if blocks**

```
void withdraw(BigDecimal amount) {
  if (amount == null) {
    throw new NulllPointerException(..
  } else if (amount.compareTo(BigDecimal.ZERO) < 0) {
    throw new IllegalArgumentException(..
  } else if (balance.compareTo(account) < 0) {
    throw new InsufficientFundsException(..
  }

  balance = balance.substract(amount); //happy path √
  ..
```

Section 5 : Exceptions

- **Design catch blocks carefully**

```
private String retrieveUsername() throws UsernameNotFoundException {
  try {
    String filename = ..
    BufferedReader r = new BufferedReader(new FileReader(filename));
    String line = r.readLine();
  } cath (IOException cause) {
    //exception translation √
    String message = ..
    throw new UsernameNotFoundException(message, cause);
  }
}
```

Si l'exception est lancée, la cause (fichier inexistant, mauvais format, sans droit de lecture, etc.) est imbriquée dans l'exception.

Section 5 : Exceptions

Use meaningful error messages

```
class InsufficientFundsException extends Exception {

 //force caller to provide these parameters
 InsufficientFundsException(BigDecimal balance, BigDecimal amount) {
    super(format(balance, amount));
  }

  private static String format(BigDecimal balance, BigDecimal amount) {
   String pat = "balance={0}; amount={1}";
   return MessageFormat.format(pat, balance, amount);
  }
}
```

Section 5 : Exceptions

Use meaningful error messages

```
void withdraw(BigDecimal amount) throws InsufficientFundsException {
  if (amount == null) {
    throw new InsufficientFundsException(balance, amount);
  } else if (amount.compareTo(BigDecimal.ZERO) < 0) {
    throw new InsufficientFundsException(balance, amount);
  } else if (balance.compareTo(amount) < 0 {
    throw new InsufficientFundsException(balance, amount);
  }

  this.balance = balance.substract(amount);
  ..
}
```

En production, il se peut que le message de l'exception est la seule information indiquant la cause d'une malfonction.

Section 6

Modern Java Elements

Section 6 : Modern Java Elements

- [Java 7] Use try-with-resources
- [Java 10] Use var correctly
- [Java 14] Use records
- [Java 14] Prefer switch expressions
- [Java 16] Prefer pattern matching

Section 6 : Modern Java Elements

[Java 7] Use try-with-resources

```
void writeFile(String path, String message) {
  try (Writer writer = new BufferedWriter(new FileWriter(path))) {
    writer.write(message);
  }

  ..
}
```

Pas besoin de fermer le FileWriter, ni le BufferedWriter

Section 6 : Modern Java Elements

[Java 10] Use var correctly

```
var message = "Hello World"; //inferred as String
var base = 10_000L;          //inferred as long
```

√ √

Le compilateur infère le type voulu

```
var flag = 0x7f;    //inferred as int, byte expected
var mask = 0x7fff;  //inferred as int, short expected
var base = 10_000;  //inferred as int, long expected
var value = 0.5;    //inferred as double, float expected
```

X X X X

Le compilateur n'infère pas le type voulu

Section 6 : Modern Java Elements

[Java 10] Use var correctly

```
var list = List.of();
```
X

Le compilateur infère que list est List<Object>

```
var set = new HashSet<String>; //inferred as HashSet<String>
```
X

```
Set<String> set = new HashSet<String>;
```
√

La variable set devrait être typée avec un type générique

Section 6 : Modern Java Elements

[Java 10] Use var correctly

```
var items = getItems();
items.add("Alpha");
..
for (var item : items) {
  processItem(item);
}

void processItem(String item) {
void processItem(CharSequence item) {
void processItem(Object item) {
```

Pour comprendre le code, il faut connaitre le type de retour de getItems().
Pire : si l'auteur de getItems() change son type de retour, il modifie le comportement de notre code !

Section 6 : Modern Java Elements

[Java 14] Use records

```
record Book(String title, String author) {}
```

équivaut à :

```
final class Book {
  private final String title;
  private final String author;

  public Book(String title, String author) {
    this.title = title;
    this.author = author;
  }

  public String title() { return title; }
  public String author() { return author; }
  //implement equals()
  //implement hashCode()
  //implement toString()
```

Section 6 : Modern Java Elements

[Java 14] Use records

Records versus Lombok @Value

- Lombok permet l'héritage
- Lombok permet un @Builder
- ..mais Lombok requiert une librairie externe

Section 6 : Modern Java Elements

[Java 14] Prefer switch expressions

```
String result = switch(value) {
  case 1 -> "One";
  case 2 -> "Two";
  default -> "Many";
};
```

plutôt que :

```
switch(value) {
  case 1 :
    result = "One";
    break;
  case 2 :
    result = "Two";
    ..
};
```

Section 6 : Modern Java Elements

[Java 16] Pattern matching

```
switch(obj) {
  case Integer i -> processInteger(i);
  case Double d -> processDouble(d);
  default : processObject(obj);
};
```

plutôt que :

```
if (obj instanceof Integer i) {
  processInteger(i);
} else if (..
```

ou pire, plutôt que :

```
if (obj instanceof Integer) {
  Integer i = (Integer)obj;
  processInteger(i);
} else if (..
```

Section 7

General Programming Rules

Section 7 : General Programming Rules

- Follow popular naming conventions
- Use libraries
- Don’t optimise code too early
- Program to interfaces where appropriate

Section 7 : General Programming Rules

Follow popular naming conventions

- Les classes : noms en format CamelCase

  `CustomerAccount, PurchaseOrder, TaskScheduler`

- Les interfaces, noms ou adjectifs en -able

  `Cloneable, Comparable, Runnable, Serializable`

  `Throwable est une exception (sans jeu de mot)`

- Éviter acronymes, sauf les plus connus : HTTP, URL
- HttpUrl plutôt que HTTPURL

Section 7 : General Programming Rules

Follow popular naming conventions

- Les méthodes : verbes en format camelCase

```
readLine(), startsWith() : cas général

getField(), setField() : getter/setter

isEmpty(), isEnabled(), hasChildren() : booléen

toString(), toUpperCase() : convertion

of(), valueOf() : factory method
```

Section 7 : General Programming Rules

Follow popular naming conventions

- Les champs et variables locales : noms en camelCase

```
firstName, orderDate : cas général

String name : object au singulier

List<String> names : collection au pluriel

Long.MAX_VALUE, Math.PI : constantes en majuscules

Month.APRIL : litéral dans une énumération
```

Section 7 : General Programming Rules

Follow popular naming conventions

- Les paquetages : strictement en minuscule

  `java.awt.datatransfer, java.net.http`

  `javax.imageio, javax.swing.filechooser`

- corporatif : nom de domaine inversé

  `ca.qc.gouv.tresor.division.`

Section 7 : General Programming Rules

Use libraries (Spring, Apache, Guava)

Avantages :

- Bénéficier de l'expertise pour des problèmes communs
- Mieux testé, mieux documenté, plus efficace
- Bénéficier des améliorations futures

Inconvénients (mineurs) :

- Problèmes communs, peut être inapplicable pour vos besoins spécifique
- Augmente la taille des applications
- Votre responsabilité de mettre à jour les nouvelles versions des librairies

Section 7 : General Programming Rules

Don’t optimise code too early

- Débuter avec un code claire et structuré
- Une fois complété, mesurer la performance
- Identifier les goulots d’étranglement
- Seulement si la performance souffre, optimiser.
- Utiliser des outils de profilage : Jprofiler, VisualVM

“Premature optimization is the root of all evil“ -Donald Knuth

“Make it works, make it right, make it fast“ -Robert C Martin

Section 7 : General Programming Rules

Program to interfaces where appropriate

- Débuter en écrivant l'interface
- Référer à la classe seulement à l'instantiation :

```
CsvReader reader = new CsvReader(..);
```
X

```
BufferedReader reader = new CsvReader(..);
```
√

```
reader.readLine();
```

- De même, les variables devraient être typées :

```
ArrayList<String> names = new ArrayList<>();
```
X

```
List<String> names = new ArrayList<>();
```
√

Section 8

Other

Section 8 : Other

- Avoid Java serialization
- Package by feature, not by layer
- Follow unit test best practices

Section 8 : Other

Avoid Java serialization

- Serialization : objets → byte streams
- Crée en 1997, binaire (pas human-readable)
- Faille de sécurité à la déserialisation (code malveillant)
- Alternative : JSON
- Libraries : Jackson, GSON.

Section 8 : Other

Package by feature, not by layer

- Par couche:

  packages : controllers, services, entities, etc.

- Par fonction :

  un package *account* qui contient :

  AccountController, AccountService, UserEntity, RoleEntity, ..

  un package *products* qui contient :

  ProductController, ProductService, ProductEntity, CategoryEntity..

  un package *ordering* qui contient :

  OrderingController, OrderringService, PurchaseOrderEntity..

Section 8 : Other

Package by feature, not by layer

- Par couche:
  - Difficile de naviguer d'un package à l'autre
  - Le nombre de classes par package augmente dans le temps
  - Toutes les classes doivent être publiques

Section 8 : Other

Package by feature, not by layer

- Par fonction :
  - On navigue souvent à l'intérieur d'un même package
  - Le nombre de packages augmente dans le temps, mais chaque package aura un nombre restreint et similaire de classes
  - Les controllers, services, etc. peuvent être package-visible (diminue le couplage inter-modules).
  - On peut réutiliser les packages-fonction dans d'autres projets

Section 8 : Other

Follow unit test best practices

- Les tests doivent être indépendants des autres
- Les tests ne doivent pas dépendrent de l'ordre dans lequel ils sont exécutés
- Le test fait GWT :

  Given condition, When action, Then result

  (aussi appelé Arrange-Act-Assert)
- Le nom des tests doit être informatif

  givenWrongPassword_WhenLogin_ThenFail()

Section 8 : Other

Follow unit test best practices

- Viser au moins 80% de couverture de code
- Tester le *happy path* et les *wrong paths*
- Les tests devraient rester rapides à exécuter
  - Il faut les exécuter à chaque *commit*.