

Pojo 2019

Programmation artisanale et moderne avec le bon vieil objet
Java

Joel Cavat

2019

Généralité

Présentation personnelle

- Adjoint scientifique et chargé de cours
 - Hepia (Genève) et HEIG-VD (Yverdon)
- Enseignement de cours de bachelor et formation continue
 - POO en Java
 - Base de données
 - Python3
- Partenariats
 - Design et développement
 - ID Quantique, Banque Pictet, 2CS...
- Divers
 - Scala/Java/Rust/Python + FP enthousiaste (mais non obstiné)

“Programmation artisanale et moderne avec le bon vieil objet Java”

“Programmation artisanale et moderne avec le bon vieil objet Java”

- artisanale \implies fonctionnel + bien conçu

Clean code always looks like it was written by someone who cares -

Michael Feathers - Clean Code, Robert Martin

“Programmation artisanale et moderne avec le bon vieil objet Java”

- artisanale \implies fonctionnel + bien conçu
- POJO (Martin Fowler) & moderne (prog. déclarative)

Clean code always looks like it was written by someone who cares -

Michael Feathers - Clean Code, Robert Martin

“Programmation artisanale et moderne avec le bon vieil objet Java”

- artisanale \implies fonctionnel + bien conçu
- POJO (Martin Fowler) & moderne (prog. déclarative)

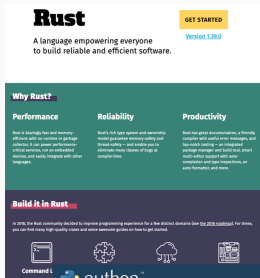
Clean code always looks like it was written by someone who cares -

Michael Feathers - Clean Code, Robert Martin

A good architecture allows major decisions to be deferred - Robert Martin,
the Clean Architecture

Présentation de Java

Présentation d'un langage lambda



Rust

A language empowering everyone to build reliable and efficient software.

GET STARTED
Version 1.39.0

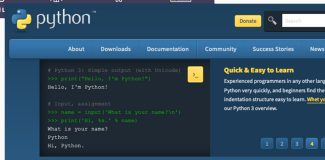
Why Rust?

Performance	Reliability	Productivity
Rust is blazing fast and memory efficient with its borrow checker. It can power performance-critical services, run on embedded devices, and easily integrate with other languages.	Rust's rich type system and ownership model guarantee memory safety and thread-safety – and enable you to eliminate many classes of bugs at compile-time.	Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling – an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

Build it in Rust

In 2019, the Rust community decided to improve programming experience for a few distinct domains (see the 2019 roadmap). For these, you can find many high-quality crates and some awesome guides on how to get started.

Command Line



python

Quick & Easy to Learn

Experienced programmers in any other language can learn Python very quickly, and beginners find the indentation structure easy to learn. [What you need to know about our Python 3 overview.](#)

```
# Python 3: Simple output (with Unicode)
msg = "Hello Python!"
print(msg)

# Input, assignment
msg = input("What is your name? ")
print(msg + ", hi!" * 3)

What is your name?
Python
Hi, Python.
```



The Scala Programming Language

Scala combines object-oriented and functional programming in one concise, high-level language. Scala's static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries.

DOWNLOAD
Getting Started
MacOS, Windows, etc.
All Previous Releases

API DOCS
Current API Docs
API Docs (other versions)
Scala Documentation
Language Specification

Scala 2.13.1

Scala runs on:

Linux, macOS, Windows, etc.



Swift

The powerful programming language that is also easy to learn.

Swift is a powerful and intuitive programming language for macOS, iOS, watchOS, tvOS and beyond. Writing Swift code is interactive and fun; the syntax is concise yet expressive, and Swift includes modern features developers love. Swift code is safe by design, and also produces software that runs lightning-fast.

```
let name = "John"
let age = 20
let isAdult = age > 18

print("Hello, \(name)! You are \(age) years old. \(isAdult ? "Adult" : "Minor").")
```

Présentation de Java

The Java™ Tutorials

Search the online

Java Tutorials Learning Paths

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later technology no longer available.

See [JDK Release Notes](#) for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Are you a student trying to learn the Java language or a professional seeking to expand your skill set? If you are feeling a bit overwhelmed by the breadth of few suggested learning paths to help you get the most from your Java learning experience.



New To Java



The following trails are most useful for beginners:

Building On The Foundation



Ready to dive deeper into the technology? See the fc

<https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>

Historique

- Java7 (2011)
 - try-with-resources / AutoCloseable

```
1 String test() {  
2     try (BufferedReader br =  
3         new BufferedReader(new FileReader("Filename.txt"))) ) {  
4         return br.readLine();  
5     } catch (IOException e) {  
6         return "";  
7     }  
8 }
```

Historique

- Java8 (LTS) - 2014
 - expressions lambda / `@FunctionalInterface`
 - `Stream`
 - `Optional`
 - `CompletableFuture<T>`, `CompletionStage<T>`
 - méthodes par défaut et méthodes statiques dans les interfaces
- Java9 - 2017
 - méthodes privées dans les interfaces

Présentation de Java

Future<Java>

- text block

```
1  """
2  SELECT *
3  FROM Table1;
4  """;
```

- expression switch et pattern matching ?

```
1  String greeting = switch( person ) {
2      case Person(name, age) if age >= 18 -> {
3          return "Good morning Mr. " + name;
4      }
5      case _ -> { return "Hi kid!"; }
6  }
```

- Value Object
- Paramètres primitifs pour les génériques

P00

- Héritage
- Encapsulation
- P???

POO

- Héritage 🤨 -> composition
- Encapsulation -> POO ?? vraiment ?
- Polymorphisme
 - héritage basé sur un abstraction (interface)
 - contractuel, se concentre sur le comportement
 - `Database db = new InMemoryDatabase()`

Héritage 🤨

Cas du Properties: represents a persistent set of properties

- Each key and its corresponding value in the property list is a string

```
1 public class Properties extends Hashtable<Object, Object> {  
2     public Object put(Object key,  
3         Object value); // hérité de Hashtable  
4 }
```

javadoc Properties

Présentation de Java

Polymorphisme (de sous-typage)

- Extensibilité
- Découplage

```
1 interface Database { ... }  
2 // Les implémentations dépendent d'une abstraction  
3 class FakeDatabase implements Database { ... }  
4 class InMemoryDatabase implements Database { ... }  
5 class MySQLDatabase implements Database { ... }
```

```
1 public void runServerWith(Database db) {  
2     /* La logique métier dépend également  
3     d'une abstraction */  
4 }
```

```
1 // inject. de dép.  
2 Database db = new FakeDatabase();  
3 runServerWith( db );
```

- Polymorphisme ad-hoc (surcharge)

```
1 db.save( user ); // appel à save(User user)
2 db.save( contract ); // appel à save(Contract contract)
```

Au lieu de :

```
1 db.saveUser( user );
2 db.saveContract( contract );
```

- Polymorphisme paramétrique

```
1 int i = List.of(1,2,3).get(0);
2 String s = List.of("Vivement", "l'apéro").get(0);
```

Présentation de Java

- Renforcement du contrôle des types à la compilation (Typesafety)

```
1 Vector v = Vector.of(1.0, 2.0)
2 TransposedVector t = Vector.of(2.0, -4.0).t();
3
4 // Addition si compatible
5 Vector res1 = v.add(v); // Ok !
6 Vector res2 = t.add(t); // Ok !
7 Vector res3 = v.add(t); // ne compile heureusement pas
8
9 // Produit scalaire si compatible
10 double d1 = t.dot(v); // Ok !
11 double d2 = v.dot(t); // ne compile heureusement pas non plus
```

$$\vec{x}^t \cdot \vec{y} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} = \sum_{i=1}^n x_i \cdot y_i$$

- Expressivité / style déclaratif

```
1 cat.eat( bird );  
2 Vector v = Vector.fillWith(4.0).repeat(10);
```

```
1 people.stream()  
2     .filter( p -> p.age() >= 18 )  
3     .map( p -> p.email() )  
4     .forEach( email -> spam(email) );
```

Les limites du polymorphisme

Live Coding

Design d'API

- Conseil de design
- Enum Pattern

Conseil de design. . .

- UML 🙄
 - trop centré sur la structure
 - “The design is the code, the code is the design” (Implementing DDD, 2013, Vaughn Vernon)
 - The Code is the Design, 1992, Jack Reeves

Conseil de design. . .

- TDD 🤔
 - test-first programming concepts
 - KISS/YAGNI ok
 - vision trop limitée

Conseil de design. . .

- Embrassez les grands principes: KISS, Yagni, DRY, TDA, SoC, SOLID

Conseil de design. . .

- Suggestif
 - un bon design est celui qui suggère son utilisation (The Design of Everyday Things, Donald Norman)



Conseil de design. . .

- **Ecrire un extrait de code qui décrit comment on souhaite utiliser notre API/lib**
 - en supposant que les fonctionnalités existes
 - code qui devrait compiler (ou pas) et s'exécuter
 - (comment utiliser mon API plutôt que comment je vais le mettre en oeuvre)
 - centré sur l'expérience utilisateur et sur un comportement général

Conseil de design. . .

- **Ecrire un extrait de code qui décrit comment on souhaite utiliser notre API/lib**
 - en supposant que les fonctionnalités existes
 - code qui devrait compiler (ou pas) et s'exécuter
 - (comment utiliser mon API plutôt que comment je vais le mettre en oeuvre)
 - centré sur l'expérience utilisateur et sur un comportement général
- **"Fake it till you make it"**

Conseil de design. . .

- Ecrire un extrait de code qui décrit comment on souhaite utiliser notre API/lib
 - en supposant que les fonctionnalités existes
 - code qui devrait compiler (ou pas) et s'exécuter
 - (comment utiliser mon API plutôt que comment je vais le mettre en oeuvre)
 - centré sur l'expérience utilisateur et sur un comportement général
- "Fake it till you make it"
- "acting as if" technique thérapeutique, 1920, Alfred Adler

Account v0.1

```
1  if( account1.getAmount() >= 10000 ) {  
2      account1.setAmount( account1.getAmount() - 10000 );  
3      account2.setAmount( account2.getAmount() + 10000 );  
4  }
```


Account v0.2

```
1 boolean transferOk = account1.transfer(account2, 10000);
```

Account v0.3

```
1 // Transaction t = account1.transfer(10000); // shouldn't compile !  
2 Transaction t = account1.transfer(10000).to(account2);  
3 if( t.isSuccess() ) { ... }
```

Account v0.4

```
1 Transaction t = account1.transfer(10000).to(account2);
2 if( t.isSuccess() ) {
3     Account sender = t.sender();
4     Account receiver = t.receiver();
5     log( sender.owner() + " now have " + sender.amount() );
6     log( receiver.owner() + " now have " + receiver.amount() );
7 }
8 broker.publish(t);
```

Account v1.0

```
1 // TransactionService.transfer(account1, account2, 100000) (alternative)
2 account1.transfer(10000).to(account2).ifSuccess (
3     (Account sender, Account receiver, int amount) -> {
4         log( sender.owner() + " now have " + sender.amount() );
5         log( receiver.owner() + " now have " + receiver.amount() );
6         log( amount + " has been transferred correctly...");
7     })
8 .ifTransactionFailure( (String reason) -> log("Failed with " + reason))
9 .ifExternalException( (Exception e) -> log(e.getMessage()) )
10 .andThen( t -> broker.publish(t) );
11 )
```

Enum Pattern

- Représentation d'une énumération "avancée" (type algébrique de données, *sum type*)
- Possède une sémantique propre (par opposition aux `Optional`, `Try...`)

```
1  enum Transaction {  
2      Success(Account,Account,int),  
3      TransactionFailure(String),  
4      ExternalFailure(Exception)  
5  }  
6  
7  match (transaction) {  
8      case Success(a1, a2, amount) => ...  
9      case TransactionFailure(reason) => ...  
10     case ExternalFailure(e) => ...  
11 }
```

Enum Pattern Java (version déclarative)

```
1 interface Transaction {
2     class Success implements Transaction {...}
3     class TransactionFailure implements Transaction {...}
4     class ExternalFailure implements Transaction {...}
5
6     Transaction ifSuccess(Consumer<Account,Account,Integer> consumer);
7     Transaction ifTransactionFailure(Consumer<String> consumer);
8     Transaction ifExternalFailure(Consumer<Exception> consumer);
9 }
10
11 transaction.ifSuccess( (a1, a2, amount) -> ... )
12     .ifTransactionFailure( reason -> ... );
13     .ifExternalFailure( e -> ... );
```

Enum Pattern Java (version old-school)

```
1  interface Transaction {
2      class Success implements Transaction {...}
3      class TransactionFailure implements Transaction {...}
4      class ExternalFailure implements Transaction {...}
5
6      boolean isSuccess();
7      boolean isTransactionFailure();
8      boolean isExternalFailure();
9      Success getSuccess() throws UnsupportedOperationException;
10     String getTransactionFailure() throws UnsupportedOperationException;
11     Exception getExternalFailure() throws UnsupportedOperationException;
12 }
13
14 if( transaction.isSuccess() ) {
15     Account a1 = transaction.getSuccess().sender();
16     ...
17 }
```

Le type Try

- Résultat qui peut soit retourner une valeur calculée avec succès, soit donner lieu à une exception

```
1 enum Try<T> {  
2     Success(T),  
3     Failure(Exception)  
4 }  
  
1 Try<Integer> t = trySomething();
```