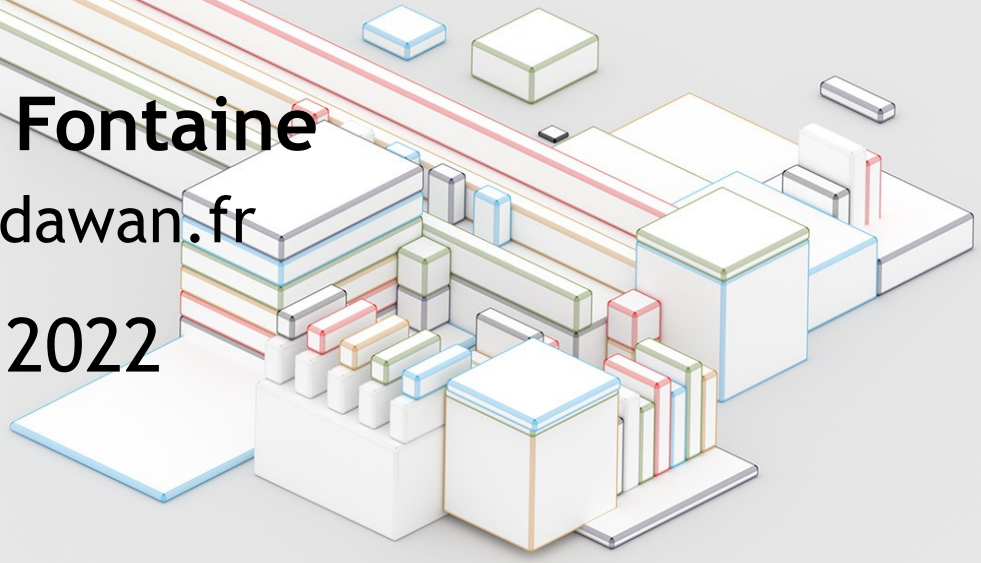
Dawan
Training is our way

Java SE Approfondissement



Christophe Fontaine
cfontaine@dawan.fr

21/11/2022

Objectifs

- Connaître et maîtriser les concepts avancés de la programmation Java

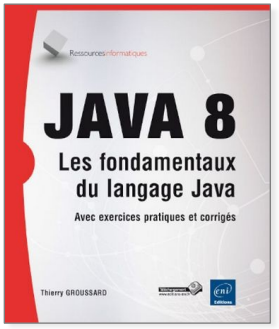
- Réaliser et déployer des applications complètes, performantes et maintenables

Durée : 2 jours

Pré-requis : avoir suivi la formation Java initiation ou posséder les connaissances équivalentes

Bibliographie

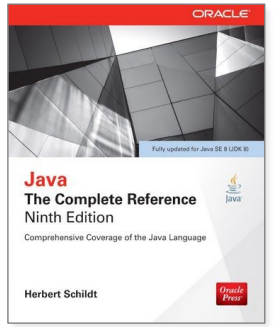
- **Java 8 Les fondamentaux du langage Java**
  Thierry Groussard
  Éditions ENI - Juillet 2014



- **Java Tête la première** (couvre Java 5.0)
  Kathy Sierra, Bert Bates
  Editions O'REILLY - Novembre 2006

- **Java The Complete Reference**
  Herbert Schildt
  Edition Oracle Press - 9th edition - Juin 2015

- **Java Platform Standard Edition 8 Documentation**
  https://docs.oracle.com/javase/8/docs/



- **Développons en Java**
  https://www.jmdoudoux.fr/java/dej/index.htm

Plan

- Accéder à des bases de données
- Généricité
- Patrons de conception (Design patterns)
- Interfaces graphiques Swing
- Threads
- Introspection (Reflection API)
- Communications réseau (Socket)
- Spécificités de la plate-forme Java

Installation du JDK

- **Téléchargement**
  - Oracle JDK : https://www.oracle.com/java/technologies/downloads/
  - OpenJDK : https://adoptium.net/
- **Paramétrages des variables d'environnement**

  Dans le menu Windows, taper : **env**

  Choisir → Modifier les variables d'environnement du système

  ↳ Variable d'environnement …

  Dans **variables système**
  - Créer une variable **`JAVA_HOME`** qui contient le chemin vers le dossier contenant le JDK
  - Modifier la variable **`PATH`** en ajoutant **`%JAVA_HOME%\bin`**
- **Vérification**

  `> javac -version` → doit afficher la version de java

Configuration d'Eclipse

- À partir de eclipse **4.18**, le JRE qui va exécuter eclipse est intégrée sous forme de plugin (openjdk 17)

  Il n'est plus nécessaire de le configurer dans le fichier **eclipse.ini** avec l'option **-vm**

```
-vm
C:\Program Files\Java\jdk1.8.0_351
```

- Dans windows → préférence
  - filtre sur **jre**

    Installed JREs → Add → choisir :
    - Standard VM
    - JRE home : `C:\Program Files\Java\jdk1.8.0_351\jre`
    - JRE Name : `jdk1.8.0_351`

Configuration d'Eclipse

- filtre sur **compiler**
  - JDK Compilance → Compiler compliance level → 1.8
- filtre sur **text editors**
  - cocher : Insert spaces for tabs
  - cocher : remove multiple spaces and backpace/delete
- filtre sur **spelling**
  - décocher : enable spelling
- filtre sur **formatter**
  - java → code style → Formatter
  - new → profil name : Eclipse
  - indentation : tab policy choisir **space only**

Dawan
Training is our way

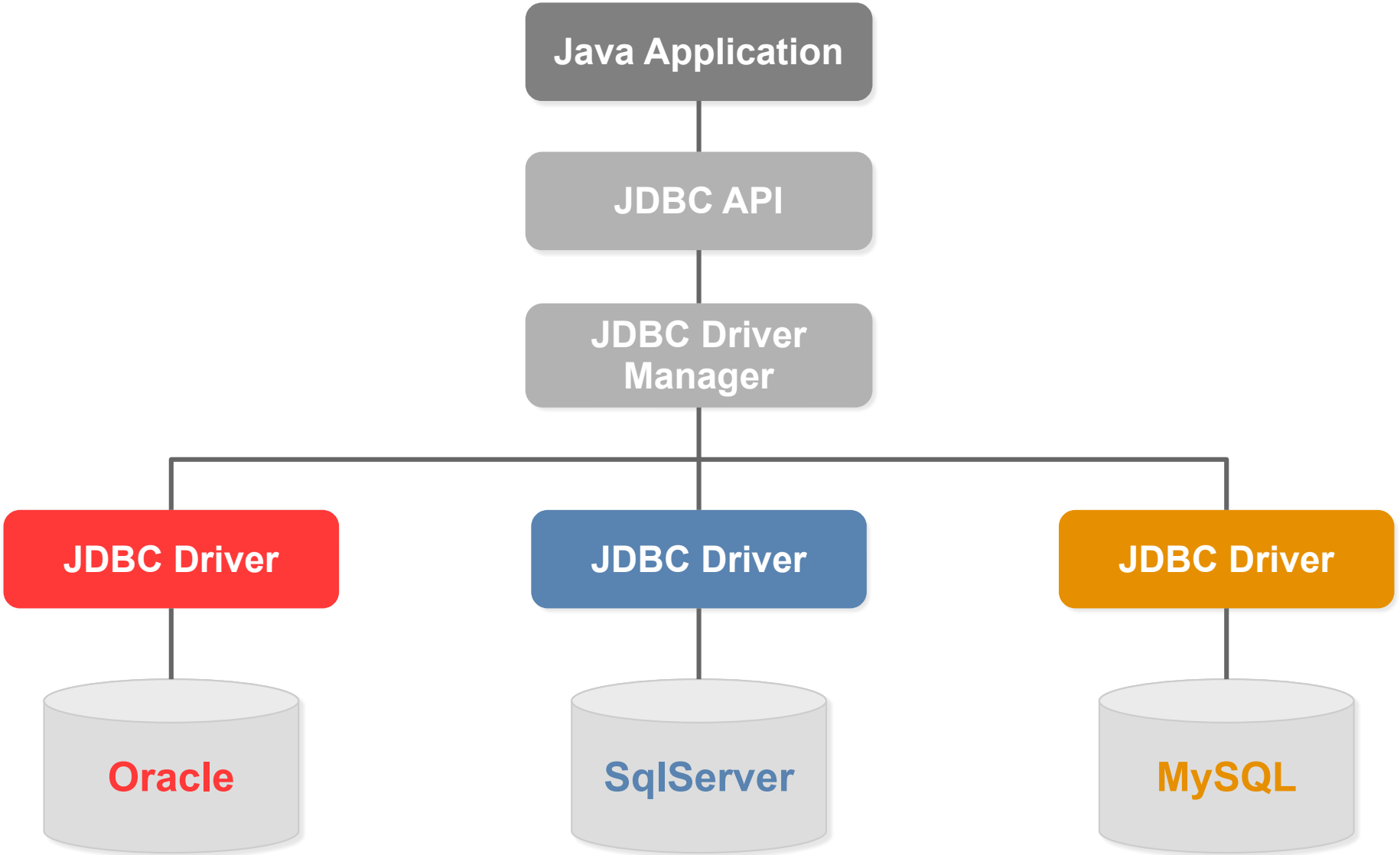
Accéder à des bases de données

JDBC

- JDBC (**J**ava **D**ata **B**ase **C**onnectivity) est l'API Java pour accéder à des bases de données relationnelles avec le langage SQL
- Les interfaces et les classes se trouvent dans le package **java.sql**
- JDBC ne fournit pas les classes qui implémentent les interfaces
- C'est le driver JDBC qui implémente ces interfaces
- Les drivers dépendent du Système de gestion de base de donnée auquel ils permettent d'accéder

  Ils sont fournis par les éditeurs des SGDB (mysql, mariadb, postgreSQL, oracle database, SQL server …)

JDBC



Principe d'accès à une base de données avec JDBC

- Chargement du pilote de la base de de données
- Ouverture d'une connexion (**`Connection`**) à la base de données
- Création d'un objet (**`Statement`** ou **`PreparedStatement`**) qui va permettre de d'exécuter la requête SQL sur la connexion
- S'il y en a, récupération du résultat de la requête dans un objet (**`ResultSet`**), et exploitation de ce résultat
- Fermeture de la connexion

Chargement du pilote de la base de donnée

- Le driver d'une base de données est une classe Java qui implémente de l'interface **`java.sql.Driver`**
- Il faut ajouter le chemin de la classe du driver dans le classpath (fichier.jar)
  - **en ligne de commande**

    avec l'option **`-classpath`** de la commande java

    ```
    > java -classpath CheminDuDriverJDBC.jar
    ```
  - **avec eclipse**

    projet → properties

    filtre sur **java build path:**
    - onglet : libraries
    - Add External JARs… → `CheminDuDriverJDBC.jar`

Chargement du pilote de la base de donnée

- **Avant JDBC 4.0**

  Il faut charger dynamiquement la classe du driver par appel de la méthode `Class.forName("nom classe");`

  Le nom de la classe varie suivant la base de données

MySQL	com.mysql.cj.jdbc.Driver
MariaDb	org.mariadb.jdbc.Driver
H2	org.h2.Driver
PostgreSQL	org.postgresql.Driver
Oracle	oracle.jdbc.driver.OracleDriver

Chargement du pilote de la base de donnée

- **À partir de JDBC 4.0 (Java 6)**

  Lors du chargement d'un JAR, Java examine le contenu du répertoire **META-INF** du fichier

  S'il y trouve un fichier **`services/java.sql.Driver`**, alors il charge les classes définies dans ce fichier et les enregistre en tant que pilotes JDBC

  Il n'est plus nécessaire de charger le driver avec `Class.forName()`

Connection

- La méthode **`getConnection`** de **DriverManager** permet de créer la Connexion
  - **`getConnection`**`(String url)`
  - **`getConnection`**`(String url, Properties prop)`
  - **`getConnection`**`(String url, String user, String password)`
- L'url est composée de 3 parties séparées par :
  - protocole
  - nom du SGDB
  - détails de connexion (spécifiques à la base de données)

    pour MySql → `//url du serveur:port/nom de la base`

```
jdbc:mysql://localhost:3306/zoo
```

Statement

- **Création**
  Un objet de type **`Statement`** s'obtient en appelant la méthode **`createStatement()`** de l'interface **`Connection`**

```
Statement smt = connection.createStatement();
```

- **Exécution**
  - **`executeQuery`**`(String req)`
    retourne un objet de type **ResultSet** (SELECT)

```
ResultSet rs = smt.executeQuery("SELECT nom, prix FROM articles");
```

  - **`executeUpdate`**`(String req)`
    retourne un nombre d'objets modifiés (INSERT, DELETE, UPDATE)

```
int count = smt.executeUdate("DELETE FROM articles");
```

Prepared Statement

- L'interface **`PreparedStatement`** étend **`Statement`**
- Il ajoute la possibilité de paramétrer des requêtes SQL
- Les instances de **`PreparedStatement`** s'utilisent quand une même requête doit être exécutée plusieurs fois, avec des paramètres différents

```
PreparedStatement ps = connection.prepareStatement(
 "INSERT INTO Articles (nom, prix) values (?, ?)");
```

- **?** → paramètres, dont la valeur est donnée avec les méthodes **`setType`**`(int numParam,Type valeur)`
- L'indexation des paramètres commence à **1**

```
ps.setString(1,"batterie");
ps.setDouble(2, 76.0);
```

Clés générées

- Pour récupérer la valeur de la clé primaire générée par la base de donnée:
  - On ajoute en paramètre à executeUpdate après la requête : **`Statement.RETURN_GENERATED_KEYS`**
  - Après l'exécution de la requête, on récupère la clé primaire générée avec la méthode **`getGeneratedKeys()`**

```
Statement stmt = connection.createStatement();
int rowCount = smt.executeUpdate("INSERT INTO articles (nom,
prix) VALUES ('Stylo', 1.7)",Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next()) {
    // récupération de la clé primaire
    int idStylo = rs.getInt(1);
}
```

Resultset

- Le résultat d'une requête est stocké dans un objet de type **`ResultSet`**
- Un **`ResultSet`** peut être vu comme un tableau de résultats, dont chaque colonne est un champ, et chaque ligne un enregistrement
- La méthode **`next()`** : déplace le curseur sur la ligne suivante
  Elle retourne **false** si le curseur a dépassé la fin du tableau, sinon elle retourne **true**
- Les méthodes
  ```
  Type getType(String nomColonne)
  Type getType(int numColonne)
  ```
  permettent de lire la valeur des colonnes
- L'indexation des colonnes commence à **1**

JDBC Exemple

```
try {
    Class.forName( "com.mysql.jdbc.Driver" );
    cnx = DriverManager.getConnection(
                    "jdbc:mysql://localhost/test","root", "");
    Statement st = cnx.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM articles");
    while(rs.next()) {
        // …
        rs.getString(1)
        // …
    }
    cnx.close();
}catch(  ) {
    // …
}
```

Transaction

- Une transaction est un ensemble d'une ou plusieurs requêtes exécutées en tant qu'unité, de sorte que soit toutes les requêtes sont exécutées, soit aucune des requêtes n'est exécutée

- **Désactivation du mode auto-commit**

  Par défaut un connexion est en mode auto-commit, Chaque instruction SQL est automatiquement validée juste après son exécution

  Pour grouper plusieurs requêtes dans une transaction il faut désactiver le mode auto-commit

```
cnx.setAutoCommit (false);
```

Transaction (commit et rollback)

- La méthode **`commit()`** de Connexion permet de valider les requêtes
- Sinon la méthode **`rollback()`** va permettre d'annuler les requêtes

```
try{
    cnx.setAutoCommit(false);
    Statement stmt = cnx.createStatement();
    String SQL = "INSERT INTO Employees
                        VALUES (2000, 'John', 'Doe')";
    stmt.executeUpdate(SQL);
    // … autre requête
    cnx.commit();
 } catch(SQLException se){
     cnx.rollback();
 }
```

DAO

- Le pattern **DAO** (Data Access Object) permet d'isoler la couche métier de la couche de persistance
  - Permet de centraliser les requête SQL dans un seul objet
  - Permet de changer facilement de système de stockage de données (Bdd, XML …)
- Avec l'objet DAO, on va réaliser les opérations CRUD
  - Créer l'objet en base (INSERT)
  - Rechercher l'objet en base pour le recréer (find, RETRIEVE)
  - Mettre à jour l'objet en base (UPDATE)
  - Supprimer l'objet en base (DELETE)
- On aura **un DAO par objet Métier**

Object Relational Mapping

- Concept permettant de connecter un modèle objet à un modèle relationnel
- Couche qui va interagir entre l'application et la base de données
- **Avantages**
  - Gain de temps au niveau du développement d'une application
  - Abstraction de toute la partie SQL
  - La portabilité de l'application d'un point de vue SGBD
- **Inconvénients**
  - L'optimisation des frameworks/outils proposés
  - La difficulté à maîtriser les frameworks/outils

ORM: JPA

- Une API (Java Persistence API)
- Des implémentations

- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

ORM: Exemple Entité

```java
@Entity
@Table(name="products")
public class Product {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    private String name;

    private double price;

    @ManyToMany(cascade=CascadeType.ALL)
    @JoinTable(name="products_suppliers")
    private Set<Supplier> suppliers;

    ...
}
```

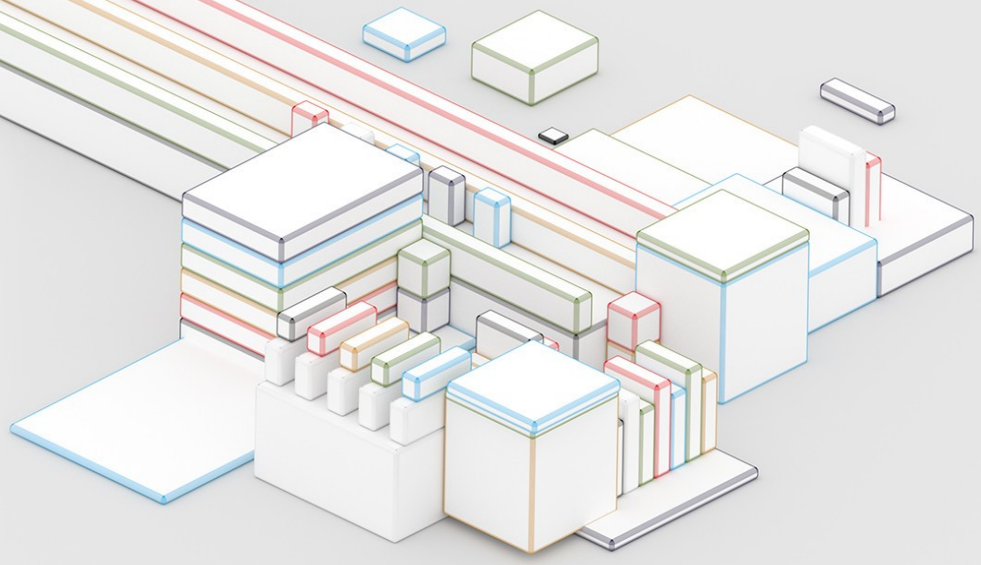
```java
@Entity
@Table(name="suppliers")
public class Supplier {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    @Column(nullable=false, unique=true, length=200)
    private String name;

    @ManyToMany(mappedBy="suppliers", cascade=CascadeType.ALL)
    private Set<Product> products;

    ...
}
```

Généricité



Généricité

- Depuis la version 5.0, Java autorise la définition de classes et d'interfaces contenant un (des) paramètre(s) représentant un (des) type(s)
- Cela permet d'écrire une structure qui pourra être personnalisée au moment de l'instanciation à tout type d'objet
- **Motivation :** Homogénéité garantie
- **Inférence de type :**
  - principe d'erasure à la compilation
  - pas de duplication de code
- **Aucune incidence sur la JVM :** les casts restent en interne mais deviennent sûrs (sans levée d'exceptions)

Classe Générique

```
class name<T1, T2, … , Tn> { /* ... */ }
```

```
public class Box<T> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
public static void main(String[] args){
    Box<Integer> intBox = new Box<>();
    intBox.set(42);
    Box<String> strBox = new Box<>() ;
    strBox.set("Test");
}
```

- **Conventions de nommage des paramètres de type**

E : Element **K** : Key **N** : Number

T : Type **V** : Value **S,U,V** … : 2ème, 3ème, 4ème type

Méthode Générique

- Les méthodes génériques sont des méthodes qui introduisent leurs propres paramètres de type
- Les méthodes d'instance, de classe et les constructeurs peuvent être des méthodes génériques

```
public class GenMethod {

    public static <T> boolean isEqual(T v1, T v2){
        return v1.equals(v2);
    }

    public static void main(String args[]){
       boolean t1=GenMethod.<String>isEqual("test","hello");
       boolean t2=GenMethod.<Integer>isEqual(42,30+12);
    }
}
```

Contraintes sur les types génériques

- Les génériques permettent d'imposer qu'un type T en étend un autre (classe ou interface)

```
public class MyClass<T extends Comparable>{…}
```

- Contraintes multiples sur un type générique → **&**

```
public class MyClass<T extends Comparable &  Cloneable>{…}
```

La liste des types

  - ne peut comporter qu'une unique classe, qui doit être déclarée en premier
  - peut comporter plusieurs interfaces

Implémentation des génériques

- **Construction d'une instance**

  On ne peut pas construire une nouvelle instance d'un type T avec : `new T()` ou `T.class.newInstance()`

  Il faut utiliser :

```
public static <T> T newInstance(Class<T> clazz){
   return clazz.newInstance();
}
```

- **Membres statiques**

  On ne peut pas référencer un type générique déclaré au niveau de la classe dans des membres statiques

```
public class MyClass<T> {
   private static T t ;  // erreur
   public static T newInstance(T t) { //…  } // erreur
```

Type <?>

- Une classe générique ne peut étendre aucune version d'elle-même

  List<T> n'étend jamais **List<U>**, quelle que soit la relation entre **T** et **U**

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; // Ne compile pas
```

- Pour résoudre ce problème, on peut utiliser **?** (wildcard)

  List<?> correspond à une liste de type inconnue

```
List<String> ls = new ArrayList<String>();
List<?> lo = ls;
```

- Pour éviter d'ajouter d'autre éléments que des String dans la **List<?>**, On interdit l'utilisation des méthodes qui prennent **?** en paramètre

Contraintes sur les type<?>

- **Type ? extension d'un type**

  On peut imposer que le type **?** en étende un autre

```
List<Integer> lstInt = Arrays.asList(5, 2, 10) ;
List<?  extends Number> lstNum = lstInt ;
//lstNum.add((Integer)2) ; // Erreur ne compile pas
int val =  (int) lstNum.get(1) ; // 2
```

- **Type ? super-type d'un type**

  On peut imposer d'être le super-type d'un type donné, avec ? **super** T (tous les types dont le type T est un type dérivé)

```
List<Integer> lstInt = Arrays.asList(1,  2,  3) ;
List<?  super Integer> lstNum = lstInt ;
lstNum.add(2) ; //ok
// Compile, mais n'est pas sûr(le type de retour peut
int i = (Integer)lstNum.get(0); // être Object)
```

Dawan
Training is our way

Patrons de conception (Design patterns)

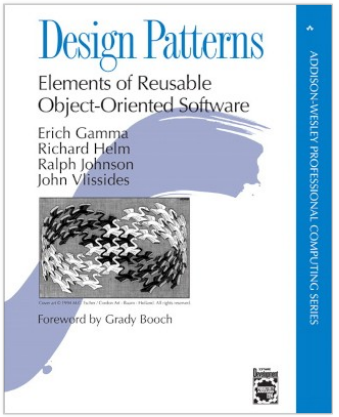
Présentation

1995 : GoF « Gang of Four »

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Auteurs du livre de référence :

Design Patterns :Elements of Reusable Object-Oriented Software



- Solution standard pour un problème d'architecture
  - Solution à disposition
  - Éprouvée, de qualité
  - Connue, permettant la communication
- à appliquer au code, particulièrement de POO
- 23 patterns répertoriés dans 3 classes

  https://refactoring.guru/fr/design-patterns

Patterns de création

Création d'objets sans instanciation directe d'une classe

Singleton	Classe qui ne pourra avoir qu'une seule instance
Factory	Introduction d'une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective
Abstract Factory	Création d'objets regroupés en famille sans devoir connaître les classes concrètes destinées à la création de ces objets
Prototype	Création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage
Builder	Séparation de la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets avec des implantations différentes

Patterns de composition (de structure)

Composition de groupes d'objets

Adapter	Conversion de l'interface d'une classe existante pour garantir la collaboration entre clients de cette interface
Bridge	Séparation des aspects conceptuels d'une hiérarchie de classes de leur implantation
Composite	Offrir une profondeur variable à la composition d'objets conception basée sur un arbre
Decorator	Ajout dynamique de fonctionnalités supplémentaires à un objet
Facade	Regroupement des interfaces d'un ensemble d'objets pour le simplifié (interface unifiée)
Flyweight	Faciliter le partage d'un ensemble important d'objet
Proxy	Construction d'un objet se substituant à un autre objet et qui contrôle son accès

Patterns de comportement

Modélisation des communications inter-objets et du flot de données

Interpreter	fournir un cadre objets pour évaluer/interpréter les expressions d'un langage
Iterator	parcours (accès séquentiel) d'une collection d'objets
Memento	sauvegarder et restaurer l'état d'un objet
Observer	construire une dépendance entre un sujet et des observateurs (avec notifications)
State	adaptation du comportement d'un objet en fonction de son état interne
Strategy	adaptation du comportement et des algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec le client
Template Method	report dans les sous-classes d'une étape de création d'un objet
Visitor	construction d'une opération à réaliser sur les éléments d'une collection d'objets ; ajout d'opérations sans modification de la classe de ses objets

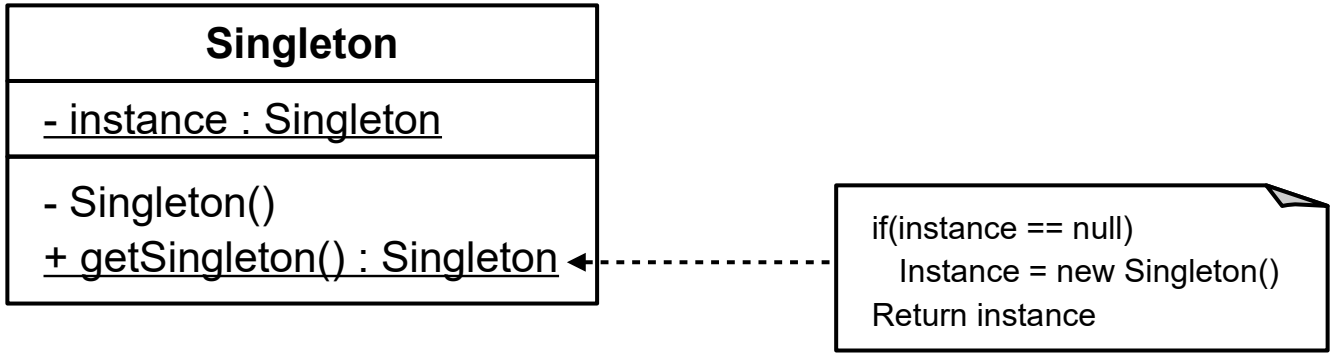
Singleton

- **Objectif**

  A tout moment une (et une seule) instance d'une classe existe pour une application

- **Solution**

  La classe possède une méthode statique **getInstance(),** qui renvoie une instance statique, et a son constructeur privé



Factory

On utilise le **FactoryMethod** lorsque :

- une classe ne peut anticiper la classe de l'objet qu'elle doit construire
- une classe délègue la responsabilité de la création à ses sous-classes, tout en concentrant l'interface dans une classe unique