Formation Programmation Java Avancée

Présentation de la formation

Cette formation avancée Java s'étale sur 5 jours et couvre les principaux aspects de la programmation avancée en Java : concurrence, communication réseau, administration, réflexion, et évolutions du langage.

Objectifs pédagogiques

À l'issue de la formation, le participant sera en mesure de :

- Mettre en œuvre la programmation concurrente à base de threads
- Utiliser différentes techniques de communication (Socket, RMI, JMS)
- Administrer une application Java via JMX
- Surveiller la JVM
- Utiliser la programmation réflexive et les annotations

Plan (1/6)

- Programmation concurrente
- Communication par socket
- RMI et JMS
- Administration via JMX
- Réflexion, extensions et nouveautés versions Java

Plan (2/6)

Programmation concurrente

- Concepts de base
- Threads, Runnable
- Synchronisation, verrous
- Problèmes classiques
- Extensions Java 5/7/8
- Travaux pratiques

Sommaire (3/6)

Communication par socket

- Concepts réseaux
- Client/serveur
- Sérialisation
- Serveur séquentiel/concurrent
- Peer to Peer
- Travaux pratiques

Sommaire (4/6)

RMI et JMS

- ORB et RMI
- Développement client/serveur RMI
- Sécurité et chargement de classes
- JMS : concepts, messages, modèles
- Travaux pratiques

Sommaire (5/6)

Administration, Réflexion, Extensions

- JMX : concepts, MBeans
- Supervision, adaptateurs
- Réflexion et annotations
- Extensions Java 5 à 9
- Exercices de synthèse

Sommaire (6/6)

Nouveautés Java 10 à 21

- Type local var (`var`)
- API Collections (List.copyOf, Set.of, etc.)
- Switch expressions
- Text blocks
- Records
- Sealed classes
- Pattern matching
- Virtual threads (Project Loom)
- API Flow (Reactive Streams)
- Améliorations de la JVM et du GC
- Autres nouveautés (instanceof amélioré, etc.)

Introduction à la programmation concurrente

Pourquoi la programmation concurrente ?

- Les applications modernes doivent souvent gérer plusieurs tâches en même temps : serveurs web, applications bancaires, systèmes embarqués…
- La concurrence permet d’exploiter au mieux les ressources matérielles (multi-cœurs).
- Elle est essentielle pour la réactivité (UI), la performance (traitement parallèle), et la robustesse (isolation des tâches).

Cas d’usage en entreprise

- Traitement de commandes en parallèle dans un site e-commerce.
- Gestion simultanée de multiples connexions clients sur un serveur.
- Calculs financiers distribués (ex : calcul de risques en banque).
- Traitement d’images ou de vidéos en temps réel.

Définition : Thread

Un thread est un flux d’exécution indépendant au sein d’un programme.

- Chaque thread possède sa propre pile d’exécution.
- Plusieurs threads partagent la mémoire du processus.
- Permet d’exécuter plusieurs tâches en parallèle.

Pourquoi utiliser des threads ?

- Pour paralléliser des traitements lourds (ex : calculs, traitements de fichiers).
- Pour améliorer la réactivité d’une application (ex : interface utilisateur qui reste fluide).
- Pour gérer plusieurs clients en même temps sur un serveur.

Définition : Runnable

- `Runnable` est une interface fonctionnelle qui définit une tâche à exécuter dans un thread.
- Permet de séparer la logique métier de la gestion du thread.

Exemple d'utilisation :

```
Runnable tache = () -> System.out.println("Tâche exécutée");
Thread t = new Thread(tache);
t.start();
```

Quand utiliser Runnable plutôt que Thread ?

- Quand on veut séparer la logique métier de la gestion du thread.
- Quand on souhaite réutiliser la même tâche dans différents contextes (ex : dans un pool de threads).

Programmation concurrente

Concepts de la programmation multithread

- Modèle d’activités de Java (Runnable et Thread)
- Création/destruction des threads
- Ordonnancement
- Synchronisation
- Problèmes classiques
- Extensions Java 5/7/8

Thread et Runnable (1/3)

Définition

Un thread est un flux d'exécution indépendant au sein d'un programme.

- `Thread` : classe de base pour créer un thread
- `Runnable` : interface fonctionnelle pour définir le code à exécuter

Thread et Runnable (2/3)

Exemple de création d’un thread avec Thread

```
class MonThread extends Thread {
    public void run() {
        System.out.println("Thread lancé !");
    }
}
MonThread t = new MonThread();
t.start();
```

Thread et Runnable (3/3)

Exemple avec Runnable

```
Runnable r = () -> System.out.println("Thread via Runnable");
Thread t = new Thread(r);
t.start();
```

Création et destruction des threads

- `start()` pour lancer un thread
- `run()` contient le code à exécuter
- Un thread terminé ne peut pas être relancé
- Gestion de la fin de vie d’un thread

Ordonnancement des threads

- L’ordonnancement dépend de la JVM et de l’OS
- Méthodes : `yield()`, `sleep()`, `join()`
- Priorités de thread

Synchronisation des threads (1/2)

- Problème : accès concurrent à des ressources partagées
- Solution : synchronisation
- Mot-clé `synchronized` sur méthodes ou blocs

Synchronisation des threads (2/2)

Exemple

```
public synchronized void increment() {
    compteur++;
}
```

Ou :

```
synchronized(obj) {
    // section critique
}
```

Les moniteurs

- Chaque objet Java possède un moniteur
- Utilisé pour la synchronisation
- Méthodes : `wait()`, `notify()`, `notifyAll()`

Problèmes classiques du multithread (1/2)

- Interblocage (deadlock)
- Famine (starvation)
- Conditions de course (race conditions)

Problèmes classiques du multithread (2/2)

Exemple d’interblocage

```
// Deux threads se bloquent mutuellement
synchronized(a) {
    synchronized(b) {
        // ...
    }
}
```

Prévention et détection de l’interblocage

- Ordonnancer les acquisitions de verrous
- Utiliser des timeouts
- Outils de détection (jconsole, VisualVM)

Extensions du modèle de concurrence (Java 5+)

- `Callable<T>`, `Future<T>`
- `ExecutorService`
- Collections concurrentes

Exemple : Callable et Future

```
Callable<Integer> tache = () -> 42;
Future<Integer> resultat = executor.submit(tache);
System.out.println(resultat.get());
```

Le modèle Fork/Join (Java 7)

- Permet le parallélisme massif
- Découpage récursif des tâches
- Classes : `ForkJoinPool`, `RecursiveTask`, `RecursiveAction`

Exemple Fork/Join

```
class SommeTask extends RecursiveTask<Integer> {
    // ...
}
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new SommeTask(...));
```

Extensions Java 8 : CompletableFuture

- Programmation asynchrone
- Chaînage de tâches
- Gestion des exceptions

Exemple CompletableFuture

```
CompletableFuture.supplyAsync(() -> "Hello")
    .thenApply(s -> s + " World")
    .thenAccept(System.out::println);
```

Outils de gestion de la concurrence

- Verrous partagés/exclusifs (`ReentrantLock`)
- Sémaphores (`Semaphore`)
- Barrières cycliques (`CyclicBarrier`)

Exemple : ReentrantLock

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    // section critique
} finally {
    lock.unlock();
}
```

Travaux pratiques (1/2)

TP : Application multithread avec contraintes de concurrence

- Créer une classe qui incrémente un compteur partagé par plusieurs threads
- Synchroniser l’accès au compteur

Travaux pratiques (2/2)

TP : Détection d’interblocage

- Créer deux threads qui se bloquent mutuellement
- Utiliser VisualVM pour détecter l’interblocage

Exercices

Exercice 1 enrichi : Thread qui affiche les nombres de 1 à 10

- Objectif : créer un thread qui affiche les nombres de 1 à 10 avec une pause entre chaque nombre.
- Utilisez la classe `Thread` et la méthode `sleep()` pour temporiser.

```
public class Compteur extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep(500); // Pause de 500 ms
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
// Pour lancer :
```

Exercice 2 enrichi : Producteur/consommateur

- Objectif : synchroniser deux threads, un producteur et un consommateur, avec un buffer partagé.
- Utilisez `wait()` et `notify()` pour la synchronisation.

```
class Buffer {
    private int data;
    private boolean disponible = false;
    public synchronized void produire(int valeur) throws InterruptedException {
        while (disponible) wait();
        data = valeur;
        disponible = true;
        notifyAll();
    }
    public synchronized int consommer() throws InterruptedException {
        while (!disponible) wait();
        disponible = false;
        notifyAll();
        return data;
    }
}
```

Exercice 3 enrichi : ExecutorService pour 10 tâches

- Objectif : lancer 10 tâches en parallèle avec un pool de threads.
- Utilisez `ExecutorService` et la méthode `submit()`.

```
ExecutorService pool = Executors.newFixedThreadPool(4);
for (int i = 0; i < 10; i++) {
    int id = i;
    pool.submit(() -> {
        System.out.println("Tâche " + id + " exécutée par " + Thread.currentThread().getName());
    });
}
pool.shutdown();
```

- Observez la répartition des tâches entre les threads du pool.

Quiz

1. Quelle est la différence entre `Thread` et `Runnable` ?
2. À quoi sert le mot-clé `synchronized` ?
3. Citez deux problèmes classiques du multithread.

Synthèse

- Les bases de la concurrence en Java
- Les outils modernes (Executor, Fork/Join, CompletableFuture)
- Les bonnes pratiques de synchronisation

Étude de cas détaillée — L'hôtel et la réservation de chambres

Présentation de l'exemple

Nous allons modéliser un hôtel où plusieurs clients (threads) tentent de réserver des chambres en parallèle. Cet exemple servira de fil rouge pour illustrer la concurrence, la synchronisation et les problèmes classiques du multithread.

Étape 1 : Modélisation de l'hôtel

- L'hôtel possède un nombre fixe de chambres.
- Chaque chambre peut être réservée par un seul client à la fois.
- Les clients sont modélisés par des threads.

Étape 2 : Classe Hotel et synchronisation

```
public class Hotel {
    private boolean[] chambres;
    public Hotel(int nbChambres) {
        chambres = new boolean[nbChambres];
    }
    public synchronized boolean reserver(int clientId) {
        for (int i = 0; i < chambres.length; i++) {
            if (!chambres[i]) {
                chambres[i] = true;
                System.out.println("Client " + clientId + " a réservé la chambre " + i);
                return true;
            }
        }
        return false; // Plus de chambres disponibles
    }
}
```

Étape 3 : Classe Client (Thread)

```
public class Client extends Thread {
    private Hotel hotel;
    private int id;
    public Client(Hotel hotel, int id) {
        this.hotel = hotel;
        this.id = id;
    }
    public void run() {
        boolean success = hotel.reserver(id);
        if (!success) {
            System.out.println("Client " + id + " n'a pas pu réserver de chambre.");
        }
    }
}
```

Étape 4 : Lancement de la simulation

```
public class Main {
    public static void main(String[] args) {
        Hotel hotel = new Hotel(3); // 3 chambres
        for (int i = 0; i < 10; i++) {
            new Client(hotel, i).start();
        }
    }
}
```

Exercice guidé : Ajout de fonctionnalités

1. Ajouter une méthode pour libérer une chambre.
2. Simuler des clients qui réservent puis libèrent une chambre après un temps aléatoire.
3. Gérer l’attente si aucune chambre n’est disponible (utiliser wait/notify).

Variante : Gestion de l’attente (wait/notify)

```
public synchronized boolean reserverAvecAttente(int clientId) throws InterruptedException {
    while (aucuneChambreDisponible()) {
        wait();
    }
    // ... réservation ...
    notifyAll();
    return true;
}
```

Exercice avancé : Prévention de l’interblocage

- Simuler plusieurs hôtels et des clients qui veulent réserver dans deux hôtels à la fois.
- Montrer comment un mauvais ordre de réservation peut provoquer un deadlock.
- Proposer une solution (ordonnancement, timeout).

Parallélisme vs Concurrence

Définition du parallélisme

- Le parallélisme consiste à exécuter plusieurs tâches exactement en même temps, en exploitant plusieurs cœurs de processeur.
- Différent de la concurrence, qui consiste à gérer plusieurs tâches qui progressent de façon entrelacée (pas forcément en même temps).

Parallélisme en entreprise

- Traitement de gros volumes de données (Big Data, ETL)
- Calculs scientifiques ou financiers répartis sur plusieurs cœurs
- Traitement d’images, vidéos, ou rendu 3D

Exemple de parallélisme en Java

```
IntStream.range(0, 1000).parallel().forEach(i -> traiter(i));
```

- Utilisation des streams parallèles pour exploiter tous les cœurs disponibles.

Les pools de threads avec ExecutorService

Pourquoi un pool de threads ?

- Créer un thread pour chaque tâche est coûteux (ressources, temps de création/destruction).
- Un pool de threads permet de réutiliser un nombre limité de threads pour exécuter un grand nombre de tâches.
- Meilleure gestion des ressources et des performances.

Créer un pool de threads avec ExecutorService

- Utiliser la fabrique Executors pour créer différents types de pools :

```
ExecutorService pool1 = Executors.newFixedThreadPool(4); // Pool fixe de 4 threads
ExecutorService pool2 = Executors.newCachedThreadPool(); // Pool extensible
ExecutorService pool3 = Executors.newSingleThreadExecutor(); // Un seul thread
```

Méthodes principales d’ExecutorService

- `submit(Runnable/Callable)` : soumet une tâche à exécuter
- `shutdown()` : arrête le pool après exécution des tâches en cours
- `invokeAll(Collection)` : exécute une collection de tâches et attend leur fin
- `invokeAny(Collection)` : exécute une collection de tâches et retourne le résultat de la première terminée

Exercice 1 enrichi : Thread qui affiche les nombres de 1 à 10

- Objectif : créer un thread qui affiche les nombres de 1 à 10 avec une pause entre chaque nombre.

```
public class Compteur extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep(500); // Pause de 500 ms
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
// Pour lancer :
new Compteur().start();
```

Exercice 2 enrichi : Producteur/consommateur

- Objectif : synchroniser deux threads, un producteur et un consommateur, avec un buffer partagé.
- Utilisez `wait()` et `notify()` pour la synchronisation.

```
class Buffer {
    private int data;
    private boolean disponible = false;
    public synchronized void produire(int valeur) throws InterruptedException {
        while (disponible) wait();
        data = valeur;
        disponible = true;
        notifyAll();
    }
    public synchronized int consommer() throws InterruptedException {
        while (!disponible) wait();
        disponible = false;
        notifyAll();
        return data;
    }
}
```

Exercice 3 enrichi : ExecutorService pour 10 tâches

- Objectif : lancer 10 tâches en parallèle avec un pool de threads.
- Utilisez `ExecutorService` et la méthode `submit()`.

```
ExecutorService pool = Executors.newFixedThreadPool(4);
for (int i = 0; i < 10; i++) {
    int id = i;
    pool.submit(() -> {
        System.out.println("Tâche " + id + " exécutée par " + Thread.currentThread().getName());
    });
}
pool.shutdown();
```

- Observez la répartition des tâches entre les threads du pool.

Introduction à la communication par socket

Pourquoi communiquer par socket ?

- Permet à des applications de dialoguer à travers un réseau (local ou Internet).
- Indispensable pour les architectures client/serveur, les microservices, les applications mobiles connectées.

Cas d’usage en entreprise

- Systèmes de réservation en ligne (hôtel, avion, etc.).
- Applications bancaires (transactions entre agences).
- Jeux en ligne multijoueurs.
- Systèmes de supervision et de collecte de logs à distance.

Définition : Socket

- Une socket est un point de terminaison pour la communication entre deux machines.
- Elle permet d’envoyer et de recevoir des données via le réseau.
- En Java, on utilise `Socket` (client) et `ServerSocket` (serveur).

Pourquoi utiliser TCP plutôt qu’UDP ?

- TCP garantit la livraison, l’ordre et l’intégrité des messages (ex : transactions bancaires, commandes e-commerce).
- UDP est plus rapide mais sans garantie (ex : streaming vidéo, jeux en temps réel).

Communication par socket

Concepts réseaux

- Modèle OSI (rappels)
- TCP vs UDP
- Ports, adresses IP

Communication en mode connecté (TCP)

- Socket côté client
- Socket côté serveur
- Streams d’entrée/sortie

Exemple : Client TCP

```
Socket socket = new Socket("localhost", 1234);
OutputStream out = socket.getOutputStream();
out.write("Hello".getBytes());
out.close();
socket.close();
```

Exemple : Serveur TCP séquentiel

```
ServerSocket server = new ServerSocket(1234);
Socket client = server.accept();
InputStream in = client.getInputStream();
// ...
client.close();
server.close();
```

Serveur concurrent

- Un thread par client
- Utilisation d’un pool de threads

Exemple : Serveur TCP concurrent

```
ServerSocket server = new ServerSocket(1234);
while (true) {
    Socket client = server.accept();
    new Thread(() -> handleClient(client)).start();
}
```

Sérialisation en Java

- Permet d’envoyer des objets sur le réseau
- Interface `Serializable`
- Streams d’objets (`ObjectInputStream` , `ObjectOutputStream`)

Exemple : Sérialisation

```
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
out.writeObject(monObjet);
```

Communication en mode non connecté (UDP)

- Utilisation de `DatagramSocket`, `DatagramPacket`
- Pas de connexion persistante

Exemple : Client UDP

```
DatagramSocket socket = new DatagramSocket();
byte[] data = "Hello".getBytes();
DatagramPacket packet = new DatagramPacket(data, data.length, InetAddress.getByName("localhost"), 1234);
socket.send(packet);
socket.close();
```

Exemple : Serveur UDP

```
DatagramSocket server = new DatagramSocket(1234);
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
server.receive(packet);
String msg = new String(packet.getData(), 0, packet.getLength());
server.close();
```

Modèle Peer to Peer

- Chaque nœud peut être client et serveur
- Utilisation dans les applications distribuées

Travaux pratiques (1/2)

TP : Client/serveur séquentiel

- Écrire un serveur qui reçoit un message et répond "OK"
- Écrire un client qui envoie un message

Travaux pratiques (2/2)

TP : Serveur concurrent

- Modifier le serveur pour gérer plusieurs clients en parallèle

Exercices

Exercice 1 enrichi : Client TCP

- Objectif : écrire un client TCP qui envoie une chaîne à un serveur et affiche la réponse.
- Utilisez la classe `Socket`, les flux d'entrée/sortie (`InputStream`, `OutputStream`).

```
Socket socket = new Socket("localhost", 1234);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out.println("Bonjour serveur");
String reponse = in.readLine();
System.out.println("Réponse : " + reponse);
socket.close();
```

Exercice 2 enrichi : Serveur UDP

- Objectif : écrire un serveur UDP qui affiche les messages reçus.
- Utilisez `DatagramSocket` et `DatagramPacket`.

```
DatagramSocket server = new DatagramSocket(1234);
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
server.receive(packet);
String msg = new String(packet.getData(), 0, packet.getLength());
System.out.println("Message reçu : " + msg);
server.close();
```

Exercice 3 enrichi : Sérialisation d'objet via socket

- Objectif : sérialiser un objet et l'envoyer via un socket.
- Utilisez `ObjectOutputStream` et `ObjectInputStream`.

```
// Côté client
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
out.writeObject(monObjet);

// Côté serveur
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
MonObjet obj = (MonObjet) in.readObject();
```

Quiz

1. Quelle est la différence entre TCP et UDP ?
2. À quoi sert la sérialisation ?
3. Comment gérer plusieurs clients sur un serveur ?

Synthèse

- Les bases de la communication réseau en Java
- Les modèles client/serveur et peer-to-peer
- La sérialisation d’objets

Étude de cas — Hôtel en client/serveur TCP

Présentation

On reprend l’exemple de l’hôtel : cette fois, l’hôtel est un serveur TCP, et chaque client est une application qui se connecte pour réserver une chambre.

Étape 1 : Serveur Hôtel TCP

- Le serveur écoute sur un port.
- Pour chaque connexion, il traite la demande de réservation.
- Utilisation d’un thread par client.

```
ServerSocket server = new ServerSocket(12345);
Hotel hotel = new Hotel(3);
while (true) {
    Socket clientSocket = server.accept();
    new Thread(() -> traiterClient(clientSocket, hotel)).start();
}
```

Étape 2 : Client TCP

- Le client se connecte au serveur et envoie une demande de réservation.
- Il reçoit la réponse (succès ou échec).

```
Socket socket = new Socket("localhost", 12345);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out.println("RESERVER");
String reponse = in.readLine();
System.out.println("Réponse serveur : " + reponse);
```

Exercice guidé : Extensions

1. Permettre au client de libérer une chambre.
2. Gérer plusieurs types de requêtes (réserver, libérer, état des chambres).
3. Ajouter la sérialisation pour transmettre des objets (ex : infos client).

Exercice avancé : Serveur concurrent robuste

- Utiliser un pool de threads pour limiter le nombre de clients simultanés.
- Gérer la déconnexion inattendue d’un client.

Introduction à RMI et JMS

Pourquoi utiliser RMI ?

- Permet d’appeler des méthodes à distance comme si elles étaient locales.
- Simplifie la création d’applications distribuées (ex : gestion centralisée de stocks, calculs répartis).

Cas d’usage RMI en entreprise

- Systèmes de gestion centralisée (ERP, CRM).
- Calculs scientifiques répartis sur plusieurs serveurs.
- Plateformes de trading ou de réservation avec logique métier partagée.

Définition : RMI (Remote Method Invocation)

- Technologie Java permettant d’invoquer des méthodes sur des objets distants.
- Utilise la sérialisation pour transmettre les paramètres et résultats.

Pourquoi utiliser JMS ?

- Pour la communication asynchrone entre applications (découplage).
- Pour la fiabilité et la scalabilité (ex : file d’attente de commandes, notifications, workflow).

Cas d’usage JMS en entreprise

- File d’attente de commandes e-commerce.
- Systèmes de notification (alertes, emails).
- Orchestration de processus métier (workflow, BPM).

RMI et JMS

Principes généraux des ORB

- ORB : Object Request Broker
- Permet l’appel de méthodes à distance
- RMI : Remote Method Invocation

Modèle RMI (1/2)

- Interfaces distantes
- Classes d’implémentation
- Stubs et skeletons

Modèle RMI (2/2)

- Service de nommage (RMI Registry)
- Déploiement du serveur et du client

Exemple : Interface distante

```
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Exemple : Implémentation

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
    public String sayHello() { return "Hello, world!"; }
}
```

Exemple : Serveur RMI

```
Hello obj = new HelloImpl();
Naming.rebind("Hello", obj);
```

Exemple : Client RMI

```
Hello stub = (Hello) Naming.lookup("rmi://localhost/Hello");
System.out.println(stub.sayHello());
```

Contraintes de sécurité et chargement de classes

- Politique de sécurité (policy file)
- Chargement dynamique de classes

Travaux pratiques RMI

- Créer une application RMI simple (Hello World)
- Étendre avec des méthodes supplémentaires

JMS : Java Message Service

- Communication asynchrone par messages
- Concepts : producteurs, consommateurs, messages

Modèle de base JMS

- Interfaces et classes principales
- Connexion, session, destination, message

Formes de messages JMS

- TextMessage, ObjectMessage, BytesMessage, MapMessage

Communication point à point

- File de messages (Queue)
- Un producteur, un consommateur

Communication publish/subscribe

- Topic
- Plusieurs abonnés

Exemple : Envoi d’un message JMS

```
TextMessage msg = session.createTextMessage("Hello JMS");
producer.send(msg);
```

Travaux pratiques JMS

- Envoyer et recevoir des messages avec ActiveMQ ou Artemis

Exercices

Exercice 1 enrichi : Interface distante et serveur RMI

- Objectif : créer une interface distante et un serveur RMI.
- Utilisez `Remote`, `UnicastRemoteObject`, et le registre RMI.

```
public interface Calculatrice extends Remote {
    int addition(int a, int b) throws RemoteException;
}

public class CalculatriceImpl extends UnicastRemoteObject implements Calculatrice {
    public int addition(int a, int b) { return a + b; }
}

// Enregistrement
Calculatrice calc = new CalculatriceImpl();
Naming.rebind("Calculatrice", calc);
```

Exercice 2 enrichi : JMS queue

- Objectif : envoyer un message JMS à une queue et le consommer.
- Utilisez `ConnectionFactory`, `Session`, `Queue`, `MessageProducer`, `MessageConsumer`.

```
// Envoi
TextMessage msg = session.createTextMessage("Hello JMS");
producer.send(msg);

// Réception
Message message = consumer.receive();
if (message instanceof TextMessage) {
    System.out.println(((TextMessage) message).getText());
}
```

Quiz

1. Qu’est-ce qu’un ORB ?
2. À quoi sert le RMI Registry ?
3. Citez deux types de messages JMS.

Synthèse

- Appels distants avec RMI
- Communication asynchrone avec JMS

Étude de cas — Hôtel avec RMI

Présentation

L’hôtel expose ses services de réservation via RMI. Les clients invoquent à distance les méthodes de réservation/libération.

Étape 1 : Interface distante

```
public interface HotelRMI extends Remote {
    boolean reserver(int clientId) throws RemoteException;
    void liberer(int chambreId) throws RemoteException;
}
```

Étape 2 : Implémentation serveur

```
public class HotelRMIImpl extends UnicastRemoteObject implements HotelRMI {
    // ... même logique que précédemment ...
}
```

Étape 3 : Client RMI

```
HotelRMI hotel = (HotelRMI) Naming.lookup("rmi://localhost/Hotel");
boolean ok = hotel.reserver(42);
if (ok) System.out.println("Chambre réservée !");
```

Exercice guidé : Extensions

1. Ajouter la gestion de la libération de chambre.
2. Permettre à un client de demander la liste des chambres libres.
3. Simuler plusieurs clients concurrents.

Exercice avancé : Sécurité et tolérance aux pannes

- Ajouter une politique de sécurité RMI.
- Gérer la reconnexion d’un client après une coupure.

Introduction à JMX

Pourquoi administrer une application avec JMX ?

- Pour surveiller l’état d’une application en production (métriques, logs, alertes).
- Pour modifier dynamiquement des paramètres sans redémarrer.
- Pour automatiser la supervision et la maintenance.

Cas d’usage JMX en entreprise

- Supervision de serveurs d’application (Tomcat, JBoss).
- Monitoring de la consommation mémoire, du nombre de connexions, etc.
- Déclenchement d’actions automatiques en cas d’alerte.

Définition : MBean

- Un MBean (Managed Bean) est un composant Java exposant des attributs et opérations pour l’administration.
- Permet d’interagir avec l’application via des outils comme JConsole ou VisualVM.

Administration des applications : JMX

Modèle JMX (Java Management eXtension)

- Concepts : MBeans, MBeanServer
- Supervision et administration

MBeans et MBeanServers

- MBean : JavaBean exposant des attributs et opérations
- MBeanServer : registre centralisé

Exemple : MBean simple

```
public interface MonMBean {
    int getValeur();
    void setValeur(int v);
}
public class Mon implements MonMBean {
    // ...
}
```

Mise en place d’une couche d’administration

- Enregistrement des MBeans
- Accès via JConsole ou VisualVM

Console d’administration (JConsole)

- Surveillance de la JVM
- Gestion des MBeans

Communication via adaptateurs et connecteurs

- Adaptateurs HTTP, RMI
- Accès distant à l’administration

Travaux pratiques JMX

- Créer un MBean pour surveiller un compteur
- Accéder au MBean via JConsole

Exercices

Exercice 1 enrichi : MBean

- Objectif : créer un MBean exposant un attribut et une opération.
- Implémentez une interface `MonMBean` et une classe `Mon`.

```
public interface MonMBean {
    int getValeur();
    void setValeur(int v);
}
public class Mon implements MonMBean {
    private int valeur;
    public int getValeur() { return valeur; }
    public void setValeur(int v) { valeur = v; }
}
```

Exercice 2 enrichi : JConsole

- Objectif : utiliser JConsole pour modifier la valeur d’un attribut d’un MBean.
- Lancez votre application avec le MBean enregistré dans le MBeanServer, puis connectez-vous avec JConsole.

Quiz

1. Qu’est-ce qu’un MBean ?
2. Comment accéder à un MBean à distance ?

Synthèse

- Supervision et administration avec JMX
- Outils de monitoring Java

Introduction à la réflexion et aux extensions Java

Pourquoi la réflexion en Java ?

- Pour découvrir dynamiquement les classes, méthodes, attributs (ex : frameworks, outils de test, ORM).
- Pour instancier des objets ou appeler des méthodes sans connaître leur type à la compilation.

Cas d’usage de la réflexion

- Frameworks de mapping objet-relationnel (Hibernate).
- Outils de sérialisation/désérialisation (JSON, XML).
- Génération dynamique de code ou de proxies (Spring, CDI).

Définition : Réflexion

- Ensemble d’API permettant d’inspecter et de manipuler dynamiquement les classes, méthodes, champs, annotations.
- Utilisée pour l’automatisation, la configuration dynamique, les tests avancés.

Programmation réflexive et extensions Java

Objectifs et principes de la réflexion

- Découverte dynamique des classes et objets
- Instanciation et invocation dynamique

Exemple : Réflexion

```
Class<?> clazz = Class.forName("java.util.Date");
Object obj = clazz.getDeclaredConstructor().newInstance();
```

Découverte dynamique des informations

- Méthodes : `getMethods()`, `getFields()`, `getConstructors()`

Instanciation et invocation dynamique

- Créer un objet dynamiquement
- Appeler une méthode par réflexion

Réflexivité et annotations (Java 5+)

- Définir et utiliser des annotations personnalisées
- Lire les annotations par réflexion

Exemple : Annotation personnalisée

```
@interface MonAnnotation {
    String valeur();
}
```

Exemple : Utilisation d’une annotation

```
@MonAnnotation(valeur = "exemple")
public class Exemple {}
```

Travaux pratiques réflexion

- Créer une annotation et l’utiliser sur une classe
- Lire l’annotation par réflexion

Exercices

Exercice 1 enrichi : Réflexion

- Objectif : lister toutes les méthodes d'une classe via réflexion.
- Utilisez la classe `Class` et la méthode `getMethods()`.

```
Class<?> clazz = Class.forName("java.util.Date");
for (Method m : clazz.getMethods()) {
    System.out.println(m.getName());
}
```

Exercice 2 enrichi : Instanciation dynamique

- Objectif : instancier une classe dont le nom est fourni en paramètre.
- Utilisez `Class.forName()` et `getDeclaredConstructor().newInstance()`.

```
String nomClasse = "java.util.Date";
Object obj = Class.forName(nomClasse).getDeclaredConstructor().newInstance();
System.out.println(obj);
```

Quiz

1. À quoi sert la réflexion en Java ?
2. Comment lire une annotation à l’exécution ?

Synthèse

- Programmation réflexive
- Utilisation des annotations

Extensions Java 5 à 9

Génériques

- Définition et syntaxe
- Avantages des génériques

Exemple : Génériques

```
List<String> liste = new ArrayList<>();
liste.add("Hello");
```

Énumérations

- Définition et utilisation

Exemple : Enum

```
enum Jour { LUNDI, MARDI, ... }
```

Autoboxing / Unboxing

- Conversion automatique entre types primitifs et objets

Exemple : Autoboxing

```
Integer i = 5; // autoboxing
int j = i;     // unboxing
```

Lambda-expressions et interfaces fonctionnelles

- Syntaxe des lambdas
- Utilisation avec les API Java

Exemple : Lambda

```
Runnable r = () -> System.out.println("Hello");
```

Streams de Java 8

- Programmation fonctionnelle sur collections
- Opérations : map, filter, reduce

Exemple : Stream

```
List<String> noms = Arrays.asList("Alice", "Bob");
noms.stream().filter(n -> n.startsWith("A")).forEach(System.out::println);
```

Modules de Java 9

- Modularisation du code
- Fichier `module-info.java`

Exemple : module-info.java

```
module mon.module {
    requires java.base;
}
```

Nouveautés Java 10 à 21

Introduction : Pourquoi suivre l’évolution de Java ?

- Java évolue rapidement avec un cycle de publication semestriel.
- Les nouvelles versions apportent des fonctionnalités qui simplifient le code, améliorent la performance et la sécurité.
- Maîtriser ces nouveautés permet d’écrire du code plus moderne, lisible et efficace.

Java 10 : Le type local `var` (1/2)

Définition

- Le mot-clé `var` permet de déclarer une variable locale sans préciser explicitement son type.
- Le type est déduit par le compilateur à la compilation.

Java 10 : Le type local `var` (2/2)

Exemples

```java
var message = "Bonjour"; // String
var nombres = List.of(1, 2, 3); // List<Integer>
var map = new HashMap<String, Integer>(); // HashMap<String, Integer>
```

- `var` ne peut être utilisé que pour les variables locales (pas pour les champs de classe ni les paramètres).

Java 9/10+ : Collections immuables (1/2)

Définition

- Les collections immuables ne peuvent pas être modifiées après leur création.
- Java 9 introduit des méthodes de fabrique pour créer facilement des listes, ensembles et maps immuables.

Java 9/10+ : Collections immuables (2/2)

Exemples

```
List<String> noms = List.of("Alice", "Bob");
Set<Integer> ids = Set.of(1, 2, 3);
Map<String, Integer> map = Map.of("a", 1, "b", 2);
```

- Toute tentative de modification (add, remove) lève une exception.

Java 14 : Switch expressions (1/2)

Définition

- Les switch expressions permettent d'utiliser `switch` comme une expression qui retourne une valeur.
- Syntaxe plus concise, moins d'erreurs (pas d'oubli de break).

Java 14 : Switch expressions (2/2)

Exemples

```
String type = switch (jour) {
    case MONDAY, FRIDAY -> "Début/fin de semaine";
    case SATURDAY, SUNDAY -> "Week-end";
    default -> "Milieu de semaine";
};
```

- Peut être utilisé dans une affectation ou un return.

Java 15 : Text blocks (1/2)

Définition

- Les text blocks permettent d'écrire des chaînes de caractères multilignes plus lisibles.
- Délimitées par trois guillemets `"""`.

Java 15 : Text blocks (2/2)

Exemples

```
String json = """
{
  "nom": "Alice",
  "age": 30
}
""";
```

- Pratique pour le JSON, SQL, HTML, etc.

Java 16 : Records (1/3)

Définition

- Les records sont des classes immuables et concises pour représenter des « data objects ».
- Génèrent automatiquement les méthodes equals, hashCode, toString, et les accesseurs.

Java 16 : Records (2/3)

Exemple simple

```
public record Point(int x, int y) {}
Point p = new Point(1, 2);
System.out.println(p.x()); // 1
```

Java 16 : Records (3/3)

Cas d’usage

- DTO (Data Transfer Object)
- Résultats de requêtes
- Valeurs de configuration

Java 17 : Sealed classes (1/2)

Définition

- Les sealed classes permettent de restreindre quelles classes peuvent hériter d'une classe ou implémenter une interface.
- Améliore la sécurité et la maintenabilité du code.

Java 17 : Sealed classes (2/2)

Exemple

```
public sealed class Animal permits Chien, Chat {}
final class Chien extends Animal {}
final class Chat extends Animal {}
```

- Seules les classes listées après `permits` peuvent hériter.

Java 16+ : Pattern matching pour instanceof (1/2)

Définition

- Simplifie le test de type et le cast dans une seule instruction.

Java 16+ : Pattern matching pour instanceof (2/2)

Exemple

```
if (obj instanceof String s) {
    System.out.println(s.toUpperCase());
}
```

- Plus besoin de caster explicitement après le test.

Java 21 : Pattern matching pour switch (1/2)

Définition

- Permet de faire un switch sur le type d’un objet, pas seulement sur des valeurs.

Java 21 : Pattern matching pour switch (2/2)

Exemple

```
switch (obj) {
    case String s -> System.out.println("Chaîne: " + s);
    case Integer i -> System.out.println("Entier: " + i);
    default -> System.out.println("Autre");
}
```

- Facilite l’écriture de code générique et sûr.

Java 21 : Virtual threads (Project Loom) (1/2)

Définition

- Les virtual threads sont des threads légers gérés par la JVM.
- Permettent de créer des milliers de threads sans surcoût mémoire.

Java 21 : Virtual threads (2/2)

Exemple

```
Thread.startVirtualThread(() -> System.out.println("Thread virtuel"));
```

- Idéal pour les applications serveur à forte concurrence (ex : serveurs web, microservices).

Java 9+ : API Flow (Reactive Streams) (1/2)

Définition

- L’API Flow permet de gérer des flux de données asynchrones et réactifs.
- Basée sur le modèle Publisher/Subscriber.

Java 9+ : API Flow (2/2)

Exemple

```
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
publisher.subscribe(new Flow.Subscriber<>() {
    public void onSubscribe(Flow.Subscription s) { s.request(1); }
    public void onNext(String item) { System.out.println(item); }
    public void onError(Throwable t) { t.printStackTrace(); }
    public void onComplete() { System.out.println("Terminé"); }
});
publisher.submit("Hello");
publisher.close();
```

Autres nouveautés (1/2)

Améliorations JVM et API

- Nouveaux ramasse-miettes (G1, ZGC, Shenandoah)
- API Files améliorée (`Files.writeString` , etc.)
- Compact Number Formatting
- Foreign Function & Memory API (FFM)

Autres nouveautés (2/2)

Exemples

```
// Files.writeString
Path path = Path.of("exemple.txt");
Files.writeString(path, "Bonjour Java 17+");

// Compact Number Formatting
NumberFormat fmt = NumberFormat.getCompactNumberInstance(Locale.US, NumberFormat.Style.SHORT);
System.out.println(fmt.format(1200)); // "1.2K"
```

Programmation réactive en Java

Qu’est-ce que la programmation réactive ?

- La programmation réactive est un paradigme qui consiste à concevoir des systèmes capables de réagir à des flux de données et à des événements de manière asynchrone et non bloquante.
- Elle repose sur la propagation des changements : lorsqu’une donnée change, tous les composants qui en dépendent sont automatiquement notifiés et réagissent.

Comment fonctionne la programmation réactive ?

- Utilise des flux (streams) de données asynchrones : les données sont produites et consommées au fil du temps.
- Les composants s'abonnent à des flux et réagissent aux nouvelles valeurs, erreurs ou à la fin du flux.
- Basée sur le modèle Publisher/Subscriber (éditeur/abonné).
- En Java, l'API Flow (depuis Java 9) et des librairies comme RxJava, Project Reactor, permettent d'implémenter ce modèle.

Intérêt de la programmation réactive

- Permet de gérer efficacement un grand nombre d'événements ou de requêtes simultanées (ex : serveurs web, applications temps réel).
- Améliore la réactivité et la scalabilité des applications.
- Facilite la gestion de l'asynchronisme et des traitements parallèles sans bloquer les threads.
- Idéal pour les architectures modernes (microservices, streaming de données, interfaces utilisateur dynamiques).

Exemple simple avec l’API Flow (Java 9+)

```
SubmissionPublisher<String> publisher = new SubmissionPublisher<>();
publisher.subscribe(new Flow.Subscriber<>() {
    public void onSubscribe(Flow.Subscription s) { s.request(1); }
    public void onNext(String item) { System.out.println("Reçu : " + item); }
    public void onError(Throwable t) { t.printStackTrace(); }
    public void onComplete() { System.out.println("Terminé"); }
});
publisher.submit("Bonjour");
publisher.close();
```

Exercices de synthèse

Exercice 1

- Créer une application multithread qui communique via sockets

Exercice 2

- Implémenter un service RMI pour calculer la factorielle

Exercice 3

- Créer un MBean pour surveiller l’utilisation mémoire

Exercice 4

- Utiliser la réflexion pour charger dynamiquement une classe et exécuter une méthode

Cas pratiques

Cas pratique 1 : Application multithread complète

- Développer une application qui gère plusieurs clients via threads et sockets
- Synchroniser l’accès à une ressource partagée

Cas pratique 2 : Système distribué avec RMI

- Créer un mini-système de chat distribué avec RMI

Cas pratique 3 : Supervision d’application avec JMX

- Ajouter des MBeans pour exposer des métriques
- Utiliser JConsole pour surveiller l’application

Cas pratique 4 : Utilisation avancée des streams et lambdas

- Manipuler des collections avec les API fonctionnelles de Java 8

Merci pour votre attention !

N’hésitez pas à poser vos questions.