Développement en JAVA

Flux, Sockets et Threads

Flux

Flux

- Un flux est caractérisé par une source (entrée) et une destination (sortie).
- Ex. d'entrées/sorties : fichier, matériel, programme, mémoire…
- Un flux encapsule des données qui peuvent prendre différents formats (octets, types primitifs, caractères, objets) et qui forment une séquence, qui elle-même peut être de diverse nature : texte, image, son…
- Ces données peuvent être transformées au cours d'un flux.

Flux

- Flux entrant :
  - Lit à partir d’une source de données (consomme la donnée).
  - Traite les éléments successivement les uns après les autres.
- Flux sortant :
  - Ecrit vers une destination (produit la donnée).
  - Traite les éléments successivement les uns après les autres.

Flux

- **Byte Streams**
  - Type de données : octet (8 bits).
  - Classes de transport : **InputStream** et **OutputStream**
  - A n’utiliser que pour des flux simples.
  - Sert de socle aux autres types de flux.

Flux

- **Byte Streams**
  - Exemple d'un programme qui lit un fichier et le duplique (utilisation des classes FileInputStream et FileOutoutStream) :

```
FileInputStream in = null;
FileOutputStream out = null;

try {
    in = new FileInputStream("resources\\lorem.txt");
    out = new FileOutputStream("resources\\sortie.txt");
    int c;
    while ((c = in.read()) != -1) {
        out.write(c);
    }
} finally {
    if (in != null) {
        in.close();
    }
    if (out != null) {
        out.close();
    }
}
```

Flux

- **Character Streams**
  - Flux spécialisé dans le traitement des caractères.
  - En Java, les caractères sont stockés au format Unicode : ce type de flux permet de gérer automatiquement les conversions de format.
  - Toutes les classes de flux de caractères héritent de Reader et Writer.

Flux

- **Character Streams**
  - Même programme réécrit avec le flux de type Character :

```
public static void main(String[] args) throws FileNotFoundException, IOException {

    FileReader inputStream = null;
    FileWriter outputStream = null;

    try {
        inputStream = new FileReader("resources\\lorem.txt");
        outputStream = new FileWriter("resources\\sortie-caracteres.txt");

        int c;
        while ((c = inputStream.read()) != -1) {
            outputStream.write(c);
        }
    } finally {
        if (inputStream != null) {
            inputStream.close();
        }
        if (outputStream != null) {
            outputStream.close();
        }
    }
}
```

Flux

- **Character Streams**
  - Il est possible de réduire le code avec un try-with-resources qui permet de s'affranchir du bloc « finally » :

```
public static void main(String[] args) throws FileNotFoundException, IOException {

    try (FileReader inputStream = new FileReader("resources\\lorem.txt");
            FileWriter outputStream = new FileWriter("resources\\sortie-caracteres.txt")) {
        int c;
        while ((c = inputStream.read()) != -1) {
            outputStream.write(c);
        }
    }

}
```

Flux

- **Buffered Streams**
  - Un Buffered Stream permet de s'affranchir des contraintes du système d'exploitation lors des opérations de lecture/écriture qui peuvent être gourmandes en ressources (accès disque, réseau, etc.).
  - Une mémoire tampon peut alors être créée par la VM pour gagner en efficience.
  - Pour cela, un flux non « bufferisé » doit être encapsulé dans un Buffer :
    - BufferedInputStream/BufferedOutputStream => pour les Byte Streams
    - BufferedReader / BufferedWriter => pour les Character Streams

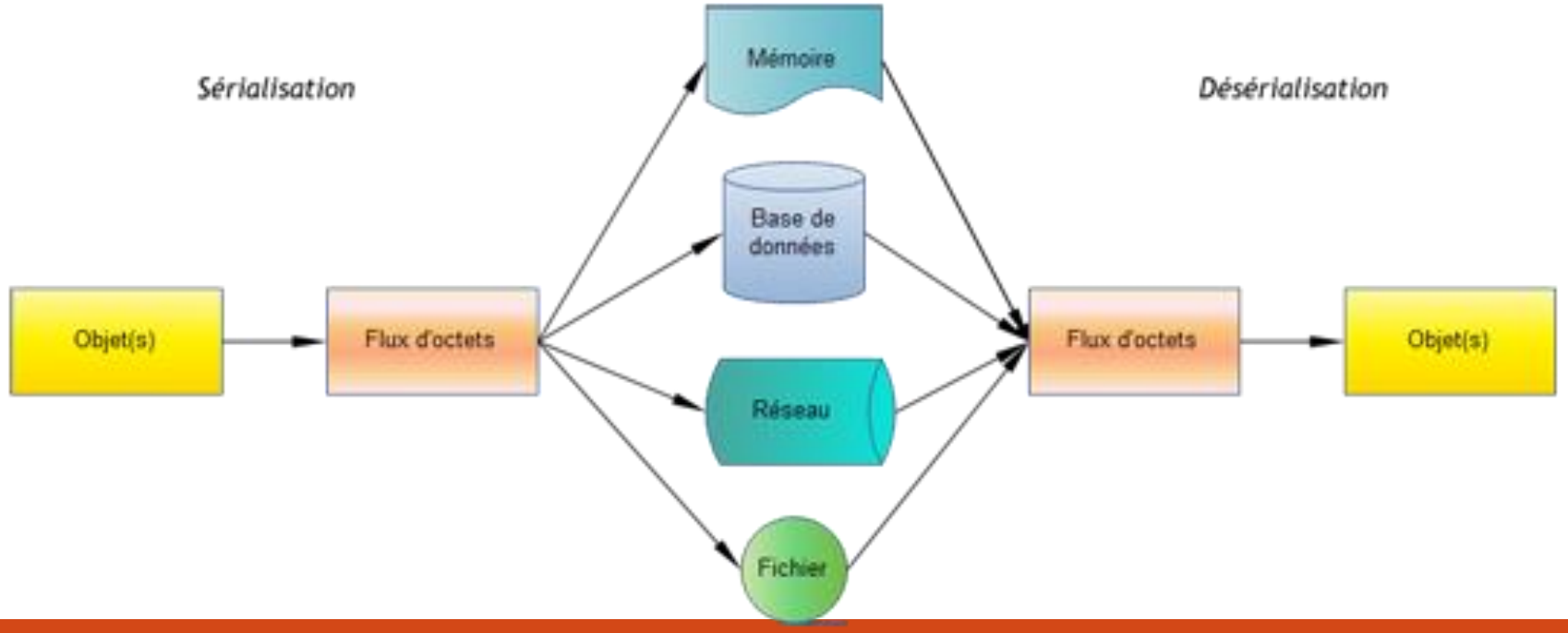
Flux

- **Buffered Streams**

```
try (
    BufferedReader in = new BufferedReader(new FileReader("resources\\lorem.txt"));
    BufferedWriter out = new BufferedWriter(new FileWriter("resources\\sortie-buffered.txt"))
) {
    String l;
    while ((l = in.readLine()) != null) {
        out.write(l);
        if (in.ready()) {
            out.newLine();
        }
    }
}
```

Flux

- **Object Streams**
  - Sérialisation/Désérialisation d'un objet : action de rendre un objet persistant pour stockage ou échange, et vice versa.



Flux

- **Object Streams**

```
// création du jeu de test d'articles
final BigDecimal[] tPrix = {
    new BigDecimal(19.99),
    new BigDecimal(9.99),
    new BigDecimal(15.99),
    new BigDecimal(3.99),
    new BigDecimal(4.99)
};
final int[]    tQuantite = {12,         8,      13,            29,       50       };
final String[]  tLibelle = {"T-shirt", "Mug", "Porte-clés", "Stylo", "Agenda"};
```

```java
// écriture dans le fichier
final String dataFile = "resources\\facture-objets";
try (
    ObjectOutputStream out = new ObjectOutputStream(
        new BufferedOutputStream(
            new FileOutputStream(dataFile)
        )
    )
) {
    out.writeObject(LocalDate.now()); // insère la date courante
    for (int i = 0; i < tPrix.length; i++) {
        out.writeObject(tPrix[i]);
        out.writeInt(tQuantite[i]);
        out.writeUTF(tLibelle[i]);
    }
}
```

```
// lecture depuis le fichier et affichage
BigDecimal prix;
int quantite;
String libelle;
LocalDate date;
BigDecimal total = BigDecimal.ZERO;

try (
    ObjectInputStream in = new ObjectInputStream(
        new BufferedInputStream(
            new FileInputStream(dataFile)
        )
    )
) {
    // récupération et affichage de la date
    date = (LocalDate) in.readObject();
    System.out.format("Commande effectuée le %s :%n",
        date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)
            .withLocale(Locale.FRANCE)
        )
    );

    // puis récupération et affichage des autres données
    while (true) { // boucle infinie(*)
        prix = (BigDecimal) in.readObject(); // lecture dans le
        quantite = in.readInt();             // même ordre que
        libelle = in.readUTF();              // l'écriture
        System.out.format("Vous avez commandé %d"
            + " %s(s) au prix de %.2f€%n", quantite, libelle, prix);
        total = total.add(prix.multiply(BigDecimal.valueOf(quantite)));
    }
} catch (EOFException e) { // (*)Exception = fin de fichier
    System.out.format("Total : %.2f€%n", total);
}
```

Flux

- **Object Streams**
  - Une exception du type ClassNotFoundException est levée si le type de l'objet récupéré ne correspond pas à celui attendu.
  - Tous les objets en relation avec l'objet sérialisé sont eux-mêmes sérialisés. Ainsi, un grand nombre d'objets peut être sérialisé avec un simple appel à la méthode writeObject() sur un seul objet.

Fichiers

Fichiers

- **Path**
  - Classe spécialisée dans la manipulation des chemins vers les fichiers et répertoires.
  - Quelques méthodes utiles :

```
// syntaxe Windows
Path path = Paths.get("C:\\home\\joe\\foo");

System.out.format("toString: %s%n", path.toString());          // toString: C:\home\joe\foo
System.out.format("getFileName: %s%n", path.getFileName());    // getFileName: foo
System.out.format("getName(0): %s%n", path.getName(0));        // getName(0): home
System.out.format("getNameCount: %d%n", path.getNameCount()); // getNameCount: 3
System.out.format("subpath(0,2): %s%n", path.subpath(0, 2));   // subpath(0,2): home\joe
System.out.format("getParent: %s%n", path.getParent());        // getParent: C:\home\joe
System.out.format("getRoot: %s%n", path.getRoot());            // getRoot: C:\
```

Fichiers

- **Lecture/Ecriture**
  - Toute méthode accédant au système de fichiers peut lancer une IOException.
  - La meilleure pratique est d'englober ces méthodes dans une instruction try-with-resources, qui permet de fermer la ressource automatiquement (fermeture d'un fichier par ex.).

```
// try-with-resources
Charset charset = Charset.forName("windows-1252");
String nomFichier = "resources\\velo.txt";
String texte = "J'ai attaché mon vélo à la barrière.";
try (BufferedWriter writer = Files.newBufferedWriter(Path.of(nomFichier), charset)) {
    writer.write(texte);
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Fichiers

- **Opérations**
  - Vérifier l'existence d'un fichier ou d'un répertoire

```
// Vérifier l'existence d'un fichier ou d'un répertoire
System.out.println(Files.exists(Path.of(nomFichier))); // true
```

  - Vérifier l'accessibilité d'un fichier

```
// Vérifier l'accessibilité d'un fichier
Path fichier = Path.of(nomFichier);
boolean estAccessible = Files.isRegularFile(fichier)
        && Files.isReadable(fichier) && Files.isExecutable(fichier);
System.out.println(estAccessible); // true
```

Fichiers

- **Opérations**
  - Supprimer un fichier ou un répertoire

```
// Supprimer un fichier ou un répertoire
try {
    Files.delete(fichier);
} catch (NoSuchFileException x) {
    System.err.format("%s: le répertoire ou le fichier n'existe pas.%n", fichier);
} catch (DirectoryNotEmptyException x) {
    System.err.format("Répertoire %s non vide.%n", fichier);
} catch (IOException x) {
    // Problèmes de droit d'accès au fichier gérés ici
    System.err.println(x);
}
```

Fichiers

- **Opérations**
  - Copier un fichier ou un répertoire

```
// Copier un fichier ou un répertoire
Path source = Path.of("resources\\orig\\velo.txt");
Path destination = Path.of("resources\\velo_copie.txt");
CopyOption[] options = { StandardCopyOption.REPLACE_EXISTING };
try {
    Files.copy(source, destination, options);
} catch (NoSuchFileException e) { //
    System.err.format("Le fichier %s n'a pas été trouvé.%n", source, e);
} catch (IOException e) {
    System.err.format("Impossible de copier : %s: %s%n", source, e);
}
```

Fichiers

- **Opérations**
  - Déplacer/renommer un fichier ou un répertoire

```
// Déplacer/renommer un fichier ou un répertoire
source = Path.of("resources\\velo_copie.txt");
destination = Path.of("resources\\velo.txt");
try {
    Files.move(source, destination, options);
} catch (IOException e) {
    System.err.println(e);
}
```

Fichiers

- **Opérations**
  - Créer un répertoire

```
// Créer un répertoire
Path nouveauRepertoire = Path.of("resources\\repertoire_test");
try {
    Files.createDirectory(nouveauRepertoire);
} catch (IOException e) {
    System.err.format("Impossible de créer le répertoire : %s: %s%n", source, e);
}
```

Fichiers

- **Opérations**
  - Afficher le contenu d'un répertoire

```
// Afficher le contenu d'un répertoire
Path repertoire = Path.of("resources");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(repertoire)) {
    for (Path file : stream) {
        System.out.println(file.getFileName());
    }
} catch (IOException | DirectoryIteratorException x) {
    System.err.println(x);
}
```

Sockets

Sockets

- Un socket est un tube de connexion synchrone entre deux entités sur un réseau (client/serveur)
- Il sert à transmettre des messages d'une IP à une autre via un port dédié.
- Il permet de faire communiquer deux entités via différents protocoles réseau (TCP/UDP)
- Modes de communication :
  - Mode connecté (Protocole TCP)
  - Mode non connecté (Protocole UDP)
- Instanciation:

```
Socket sock = new Socket (host , port );

Socket sock = new Socket (" 192.168.0.1 ", 54);
```

Sockets

- **Côté client:**
  - Ouvrir une connexion vers une adresse à laquelle envoyer des données

```
PrintWriter out =  new PrintWriter (sock . getOutputStream(), true );

BufferedReader in = new BufferedReader(new InputStreamReader (sock . getInputStream ()));
```

Sockets

- **Côté serveur:**
  - Se mettre en attente de données envoyées sur un port donné
  - Méthode accept : attente du client (multithread)

```
ServerSocket serverSocket = new ServerSocket( portNumber );

Socket clientSocket = serverSocket. accept ();
```

Sockets

- **Côté serveur:**
  - Lire et écrire des données sur un socket d'entrée

```
PrintWriter out = new PrintWriter ( clientSocket. getOutputStream (), true );

BufferedReader in = new BufferedReader( new InputStreamReader ( clientSocket. getInputStream ()));
```

Sockets

- TCP/UDP :
  - TCP et UDP sont deux protocoles d'envoi de messages en couche 4 (Transport)
- UDP
  - Gestion des ports
  - Envoi sans accusé de réception, messages légers et limités
  - Multicast (MulticastSocket)
- TCP
  - Découpage des paquets
  - Non limité
  - Accusé de réception, numérotation, cohérence

Datagrammes

- Un datagramme est un socket dédié à la transmission de messages UDP
  - OSI couche 4
  - Pas de timer, pas de garantie
  - Transmission directe et rapide, peu fiable

```
public DatagramSocket( int port) throws SocketException;

new DatagramSocket (350);  // Ouverture d'un socket UDP sur le port 350
```

- Connexion directe

```
DatagramPacket packet = new DatagramPacket(buf , buf . length );

socket . receive ( packet );
```

Sockets
Exemple

- **Objectifs :**
  - Etablir une connexion entre un processus Client et un processus Serveur.
  - Envoyer des données du Client vers le Serveur.
  - Retourner des données du Serveur vers le Client.
  - Afficher les données reçues.
  - Fermer la connexion.

Sockets
Exemple - Partie Serveur

```
final int port = 65432; // port arbitraire entre 49152 et 65535

try (ServerSocket socketServeur = new ServerSocket(port)) {
    System.out.println("Lancement du serveur");
    while (true) {
        try ( // @formatter:off
            Socket socketClient = socketServeur.accept(); // attend la sollicitation d'un client
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socketClient.getInputStream())
            );
            PrintWriter out = new PrintWriter(socketClient.getOutputStream(), true)
        ) { // @formatter:on
            System.out.println("Connexion avec : " + socketClient.getInetAddress());
            String message = in.readLine(); // récupération des données du client
            out.println(message); // envoi de la réponse au client
        }
    }
} catch (IOException e) {
    System.out.println("Erreur sur le port " + port + " ou erreur de connexion.");
}
```

Sockets
Exemple - Partie Client

```
final int port = 65432; // port du serveur
String hostName = args[0]; // nom du host fourni en paramètre (ex:localhost)

try ( // @formatter:off
    Socket socket = new Socket(hostName, port); // création de la socket
    BufferedReader in = new BufferedReader( // flux entrant provenant du serveur
        new InputStreamReader(socket.getInputStream())
    );
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true); // flux sortant
) { // @formatter:on
    out.println(args[1]); // envoi des données au serveur
    System.out.println(in.readLine()); // récupèration de la réponse du serveur
} catch (UnknownHostException e) {
    System.err.println("Host inconnu : " + hostName);
} catch (IOException e) {
    System.err.println("Impossible d'établir la connexion avec " + hostName);
}
```

Sockets
Exemple

- **Exercice:**
  - Implémenter l'application Client/Serveur
  - Tester l'application :
    - Ouvrir une console et aller dans le répertoire «src» du projet
  - Compiler le code java :
    - Client : javactests/socket/Client.java
    - Serveur : javactests/socket/Serveur.java
  - Exécuter l'application (ouvrir une 2èmeconsole) :
    - Console Serveur : java tests.socket.Server
    - Puis, console Client : java tests.socket.Clientlocalhost "Bonjour !"

Threads

Multithreading

- Possibilité d'exécuter plusieurs threads simultanément afin de tirer pleinement partie de la capacité du processeur.
- Un programme peut donc être scindé en plusieurs parties indépendantes qui pourront être exécutées dans le même espace mémoire, et en parallèle.
- A ne pas confondre avec le «multiprocessing», qui permet d'effectuer des traitements sur plusieurs processeurs (CPU).

Multithreading

- **Avantages :**
  - Parallélisation des opérations.
  - Amélioration de la vitesse d’exécution.
  - Indépendance des traitements.
  - Amélioration de la stabilité des programmes.

Thread

- Un thread est une sous-partie d’un processus (sous-processus) qui peut s’exécuter simultanément en parallèle d’autres threads du même processus.
- C’est une entité de traitement de tâche indépendante.
- Les threads d'un même processus se partagent sa mémoire virtuelle.
- Par contre, chaque thread possède sa propre pile d’exécution.

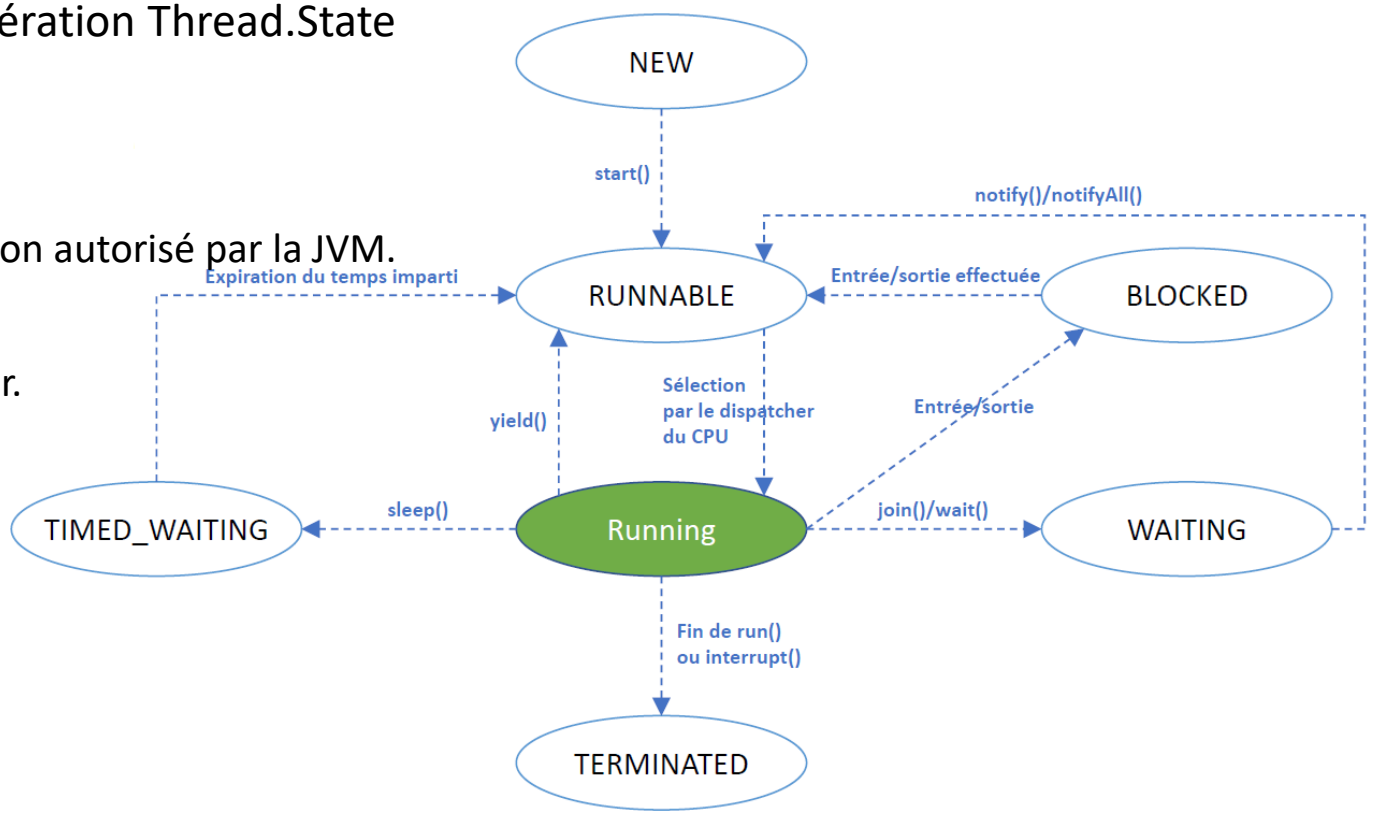
Piles

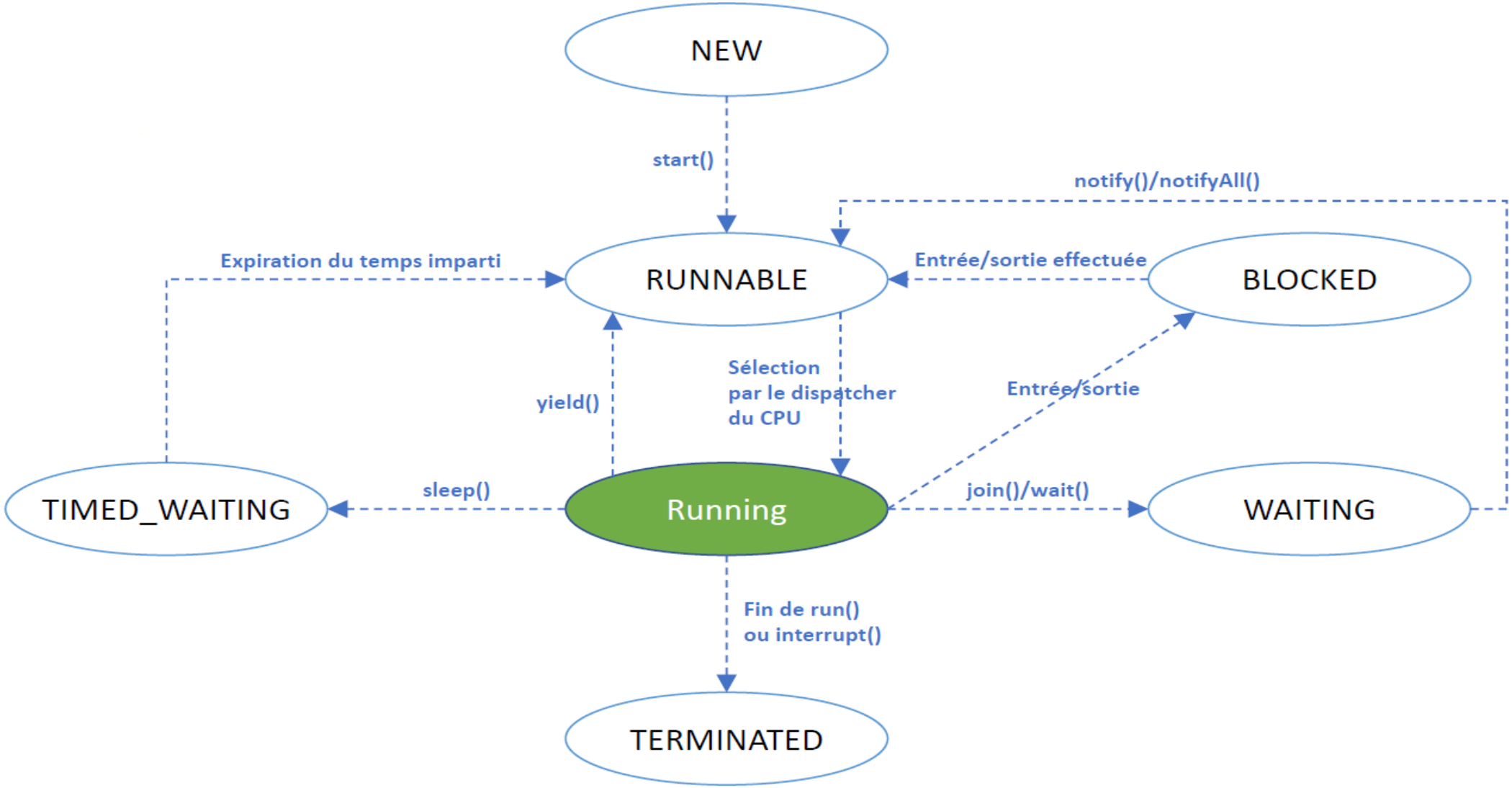
- La pile est un espace mémoire dédié à chaque thread et stockant les informations relatives à son exécution
- Stockage des objets référencés
- Désalloués par le Garbage Collector

- Détails:
  - Adresses des fonctions
  - Références d’objets

Threads
Cycle de vie

- Le statut du thread est encapsulé dans l'énumération Thread.State
- Un thread peut prendre les états suivants :
  - NEW : L'objet Thread est instantié, non démarré.
  - RUNNABLE : Invocation de la fonction start mais non autorisé par la JVM.
  - RUNNING : Thread en cours d'execution.
  - BLOCKED : en attente de l'obtention d'un moniteur.
  - WAITING : en attente d'action d'un autre thread.
  - TIMED_WAITING : en attente d'action d'un autre thread pendant un temps déterminé.
  - TERMINATED : Fin de la méthode run




NEW
start()
notify()/notifyAll()
Expiration du temps imparti
RUNNABLE
Entrée/sortie effectuée
BLOCKED
Sélection
par le dispatcher
du CPU
yield()
Entrée/sortie
TIMED_WAITING
sleep()
Running
join()/wait()
WAITING
Fin de run()
ou interrupt()
TERMINATED

Thread

- **Priorités:**
  - Un thread possède une propriété qui précise sa priorité d'exécution.
  - Pour déterminer ou modifier la priorité d'un thread, la classe Thread contient les méthodes suivantes :
    - **int getPriority()** : retourne la priorité du thread
    - **void setPriority(int)** : modifie la priorité du thread
    - La valeur de la priorité se situe entre Thread.MIN_PRIORITY et Thread.MAX_PRIORITY (généralement entre 1 et 10).
    - La valeur de priorité normale est Thread.NORM_PRIORITY

Thread

Méthode	Description
static Thread currentThread()	Renvoyer l'instance du thread courant
void interrupt()	Demander l'interruption du thread
boolean isAlive()	Renvoyer un booléen qui indique si le thread est actif ou non
boolean isInterrupted()	Renvoyer un booléen qui indique si le thread a été interrompu
void join()	Attendre la fin de l'exécution du thread
void run()	Contenir les traitements à exécuter
static void sleep(long millis)	Endormir le thread pour le délai exprimé en millisecondes précisé en paramètre
void start()	Lancer l'exécution des traitements : associer des ressources systèmes pour l'exécution et invoquer la méthode run()
static void yield()	Demander au scheduler de laisser la main aux autres threads

Thread

- **Implémentation:**
  - Via l’interface Runnable
  - Il est possible d'implémenter l'interface Runnable.
  - Celle-ci ne définit qu'une seule méthode run() dont l'implémentation doit contenir les traitements à exécuter.

```
public class MonThread implements Runnable {

    @Override
    public void run() {
        // traitements...
    }

    public static void main(String[] args) {

        // en décomposant
        Runnable r = new MonThread();
        Thread t = new Thread(r);
        t.start();

        // ou équivalent en 1 ligne...
        new Thread(new MonThread()).start();

    }

}
```

Thread

- **Implémentation**
  - Via un héritage de la classe Thread
  - Il est possible d'hériter de la classe Thread et de redéfinir la méthode run().
  - Remarque : la classe Thread implémente l'interface Runnable.

```
public class MonAutreThread extends Thread {

    @Override
    public void run() {
        // traitement...
    }

    public static void main(String[] args) {
        Thread t = new MonAutreThread();
        t.start();
    }

}
```

Synchronisation de threads

- Quand plusieurs threads sont en cours d'exécution simultanément, il peut arriver qu'ils doivent accéder à la même ressource (mémoire, fichier…).

- Les accès concurrents à une même ressource peuvent engendrer des résultats incohérents.

Synchronisation de threads

- **Exemple:** Sans synchronisation

```
public class Decompte {
    public void afficherDecompte(String nomThread) {
        for (int i = 5; i > 0; i--) {
            System.out.println("Compteur " + nomThread + " ---> " + i);
        }
    }
}
```

Synchronisation de threads

- **Exemple:** Sans synchronisation

```
public class ThreadDecompte extends Thread {
    Decompte decompte;

    ThreadDecompte(String nomThread, Decompte decompte) {
        this.setName(nomThread);
        this.decompte = decompte;
    }

    @Override
    public void run() {
        System.out.println("Thread " + getName() + " - Démarrage.");
        decompte.afficherDecompte(getName());
        System.out.println("Thread " + getName() + " - Terminé.");
    }
}
```

Synchronisation de threads

- **Exemple:** Sans synchronisation

```
public class TestThread {
    public static void main(String[] args) {
        Decompte decompte = new Decompte();
        ThreadDecompte t1 = new ThreadDecompte("t1", decompte); // même objet
        ThreadDecompte t2 = new ThreadDecompte("t2", decompte); // même objet, autre thread
        t1.start();
        t2.start();
        try {
            t1.join(); // attend la fin du thread 1
            t2.join(); // attend la fin du thread 2
        } catch (Exception e) {
            System.out.println("Interrompu !");
        }
    }
}
```

Synchronisation de threads

- **Exemple:** Sans synchronisation

```
Thread t2 - Démarrage.
Thread t1 - Démarrage.
Compteur t2 ---> 5
Compteur t2 ---> 4
Compteur t1 ---> 5
Compteur t1 ---> 4
Compteur t2 ---> 3
Compteur t2 ---> 2
Compteur t1 ---> 3
Compteur t1 ---> 2
Compteur t2 ---> 1
Compteur t1 ---> 1
Thread t1 - Terminé.
Thread t2 - Terminé.
```

Synchronisation de threads

- Avec synchronisation:
  - Utilisation des méthodes wait() et notify()
- Les threads peuvent se synchroniser entre eux grâce à l'envoi de messages.
- Les 2 méthodes principales de ce type de synchronisation sont wait() et notify() (et sa variante notifyAll()).
- A noter : un moniteur d'objet ne peut être utilisé que par un et un seul thread actif à la fois. Voici une représentation simplifiée de la commutation de contexte entre 2 threads :

Synchronisation de threads

- Utilisation des méthodes wait() et notify()
- **wait() :**
  - Force le thread courant à attendre qu'un autre thread appelle la méthode notify() ou notifyAll() sur le même objet.
  - Pour cela, le thread courant doit être en possession du verrou (activé grâce au mot-clé «synchronized»).
  - Il existe 3 signatures de la méthode wait() :
  - **wait()** : attente infinie.
  - **wait(long timeout)** : attente pendant la durée spécifiée, puis le thread est automatiquement réveillé.
  - **wait(long timeout, int nanos)** : idem, mais durée spécifiée plus précise.

Synchronisation de threads

- Utilisation des méthodes wait() et notify()
- **notify() / notifyAll() :**
  - Permettent de réveiller un (notify()) ou plusieurs (notifyAll()) threads qui ont été mis en attente par la méthode wait() d'un même objet.
  - **notify()** : permet de réveiller arbitrairement l'un des threads en attente. Le choix du thread est effectué par la JVM. Cette méthode est utilisée notamment dans le cadre d'exclusions mutuelles sur des tâches similaires interthread. Mais dans la plupart des cas, il est préférable d'appeler la méthode notifyAll().
  - **notifyAll()** : permet de réveiller tous les threads en attente sur l'objet. Les threads réveillés se termineront de la manière habituelle, comme n'importe quel autre thread.

Synchronisation de threads

- **Exemple : le Poète (pattern Producteur / Consommateur)**
- Considérons un poète qui écrit et envoie des vers à destination d'un lecteur.
- Le poète, parfois en manque d'inspiration, n'envoie pas les vers toujours à la même fréquence. De la même manière, le lecteur ne lit pas toujours régulièrement les vers.
- Il faut donc une synchronisation entre le poète et le lecteur afin que ce dernier reçoive les vers dans l'ordre, et que le premier ne soit pas sollicité s'il n'a rien à envoyer.
- Un objet partagé par les deux threads permet de stocker une valeur et de gérer son accès par les threads en les synchronisant.

Synchronisation de threads

- **Exemple : le Poète (pattern Producteur / Consommateur)**

```
public class TestPoete {
    public static void main(String[] args) {
        Data data = new Data();
        (new Thread(new Poete(data))).start();
        (new Thread(new Lecteur(data))).start();
    }
}
```

```java
public class Poete implements Runnable {
    private Data data;

    public Poete(Data data) { this.data = data; }

    @Override
    public void run() {
        String poesie[] = { // @formatter:off
            "Demain, dès l'aube, à l'heure où blanchit la campagne,",
            "Je partirai. Vois-tu, je sais que tu m'attends.",
            "J'irai par la forêt, j'irai par la montagne.",
            "Je ne puis demeurer loin de toi plus longtemps."
        }; // @formatter:on
        Random random = new Random();

        for (String vers : poesie) {
            data.envoyer(vers);
            try {
                Thread.sleep(random.nextInt(5000)); // simule une activité quelconque
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread interrompu.");
            }
        }
        data.envoyer("TERMINE");
    }
}
```

```java
public class Lecteur implements Runnable {
    private Data data;

    public Lecteur(Data data) {
        this.data = data;
    }

    public void run() {
        Random random = new Random();
        for (String message = data.recevoir(); !message.equals("TERMINE"); message = data.recevoir()) {
            System.out.format("Message reçu: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000)); // simule une activité quelconque
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread interrompu.");
            }
        }
    }
}
```

```java
public class Data {

    private String message;
    private boolean transfert = true;

    public synchronized void envoyer(String message) {
        while (!transfert) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread interrompu.");
            }
        }
        transfert = false;
        this.message = message;
        notifyAll();
    }
```

```java
    public synchronized String recevoir() {
        while (transfert) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Thread interrompu.");
            }
        }
        transfert = true;
        notifyAll();
        return message;
    }

} // fin class Data
```

Threads

- **Exercice**
- Soit un tableau de 10 entiers.
- Le but du programme est de multiplier par 2 chaque valeur du tableau et d'afficher ce tableau.
- Pour gagner en efficience, le tableau sera séparé en 2 parties de longueur équivalente (à 1 élément près).
- Chaque partie sera traitée séparément par un thread.
- Lorsque les traitements seront terminés, le tableau final sera reconstitué et affiché.