Programmation Java Avancée

Module RPCI01

Département R&T

IUT de Villetaneuse

16 novembre 2016

Plan du cours

1. La gestion des exceptions

- Levée d'une exception : la clause `throw`
- Appel d'une méthode levant une exception : clause `try / catch`
- Graphe d'héritage des exceptions
- Les `RuntimeException`
- Redirection (ou délégation) d'une exception

Levée d'une exception

Clause `throw`

Lorsqu'une méthode ne peut s'exécuter, par exemple parce qu'une situation anormale est détectée ou que les paramètres qu'on lui a fournis ne conviennent pas, au lieu de le signaler en retournant un booléen ou un code d'erreur, elle peut *lever une exception*.

Levée d'une exception

Exemple : dépiler une pile vide

```
public class Pile{
    private Object [] tabVal;
    private int indexSommet;
    . . .
    public Object depiler() throws Exception{
        Object valeur;

        if (this.estVide())
            throw new Exception("pile vide");
        valeur = this.tabVal[this.indexSommet];
        this.indexSommet--;
        return (valeur);
    }
    . . .
```

La méthode signale qu'elle est susceptible de lever une exception.

Levée d'une exception

Exemple : dépiler une pile vide

```
public class Pile{
    private Object [] tabVal;
    private int indexSommet;
    . . .
    public Object depiler() throws Exception{
        Object valeur;

        if (this.estVide())
            throw new Exception("pile vide");
        valeur = this.tabVal[this.indexSommet];
        this.indexSommet--;
        return (valeur);
    }
    . . .
```

La méthode signale qu'elle est susceptible de lever une exception.

Levée d'une exception

Si une méthode ne peut s'exécuter

- Si elle lève une exception
  1. elle crée un objet (ici de classe `Exception`), lui confie un message, puis le lance comme une bouteille à la mer.
  2. on quitte immédiatement la méthode : la fin de son code n'est donc pas exécutée.
- Si elle ne lève pas d'exception, la méthode se termine normalement.

Levée d'une exception

Si une méthode ne peut s'exécuter

- Si elle lève une exception
  1. elle crée un objet (ici de classe `Exception`), lui confie un message, puis le lance comme une bouteille à la mer.
  2. on quitte immédiatement la méthode : la fin de son code n'est donc pas exécutée.
- Si elle ne lève pas d'exception, la méthode se termine normalement.

Appel d'une méthode levant une exception

Objectif : éviter que le programme ne soit interrompu

1. appel d'une méthode susceptible de lever une exception à l'intérieur d'un bloc `try`.
2. récupération de cette exception dans un bloc `catch` dans lequel on a prévu un traitement approprié.

Exemple

```
. . .
    try{
        . . .
        Object aux = p.depiler();
        . . .
    } catch(Exception e){
        // traitement de l'exception
        System.out.println(e);
    }
. . .
```

Appel d'une méthode levant une exception

Objectif : éviter que le programme ne soit interrompu

1. appel d'une méthode susceptible de lever une exception à l'intérieur d'un bloc `try`.
2. récupération de cette exception dans un bloc `catch` dans lequel on a prévu un traitement approprié.

Exemple

```
. . .
    try{
        . . .
        Object aux = p.depiler();
        . . .
    } catch(Exception e){
        // traitement de l'exception
        System.out.println(e);
    }
. . .
```

Appel d'une méthode levant une exception

- Si la méthode appelée lève une exception, on quitte le bloc `try` et l'exception est interceptée dans le bloc `catch` dont le code est alors exécuté, précisant les actions à effectuer.
- Si la méthode appelée ne lève pas d'exception, le bloc `try` se termine et on ne passe pas dans le bloc `catch`.
- Dans tous les cas, le programme se poursuit à l'instruction suivant le bloc `catch`.

Appel d'une méthode levant une exception

- Si la méthode appelée lève une exception, on quitte le bloc `try` et l'exception est interceptée dans le bloc `catch` dont le code est alors exécuté, précisant les actions à effectuer.
- Si la méthode appelée ne lève pas d'exception, le bloc `try` se termine et on ne passe pas dans le bloc `catch`.
- Dans tous les cas, le programme se poursuit à l'instruction suivant le bloc `catch`.

Appel d'une méthode levant une exception

Bloc `try` : plusieurs méthodes peuvent lever des exceptions différentes

- Un bloc `try` peut être suivi de plusieurs blocs `catch`, chacun traitant une exception particulière.
- Les classes d'exception étant hiérarchisées, il faut traiter dans les premiers blocs `catch` les exceptions du bas de la hiérarchie et dans les derniers celles du haut de la hiérarchie.
- Une clause `finally` peut être rajoutée après les blocs `catch`. Elle est exécutée dans tous les cas (libération de ressources, fermeture de fichiers, ...).

Appel d'une méthode levant une exception

Bloc `try` : plusieurs méthodes peuvent lever des exceptions différentes

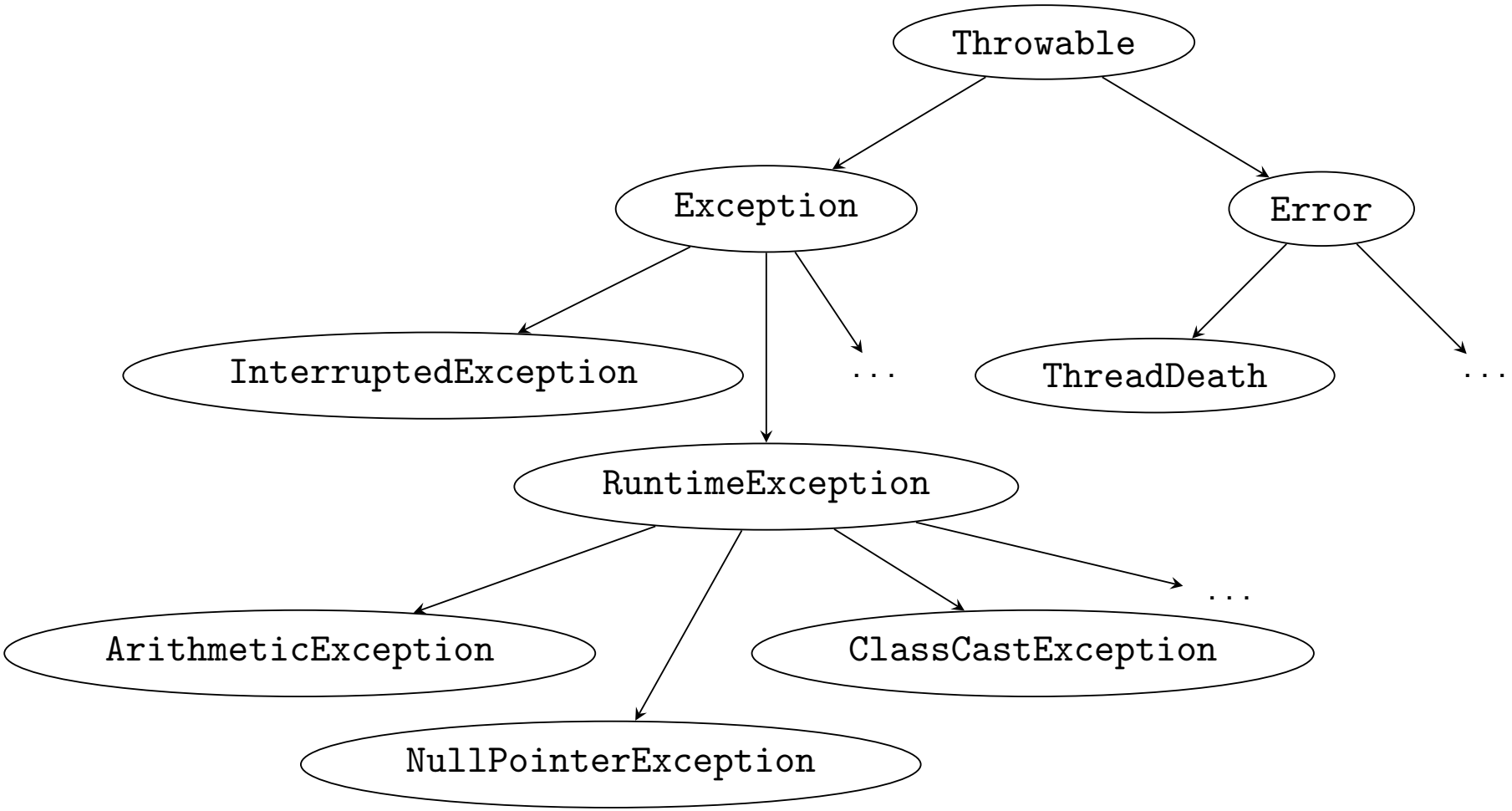
- Un bloc `try` peut être suivi de plusieurs blocs `catch`, chacun traitant une exception particulière.
- Les classes d'exception étant hiérarchisées, il faut traiter dans les premiers blocs `catch` les exceptions du bas de la hiérarchie et dans les derniers celles du haut de la hiérarchie.
- Une clause `finally` peut être rajoutée après les blocs `catch`. Elle est exécutée dans tous les cas (libération de ressources, fermeture de fichiers, ...).

Appel d'une méthode levant une exception

Bloc `try` : plusieurs méthodes peuvent lever des exceptions différentes

- Un bloc `try` peut être suivi de plusieurs blocs `catch`, chacun traitant une exception particulière.
- Les classes d'exception étant hiérarchisées, il faut traiter dans les premiers blocs `catch` les exceptions du bas de la hiérarchie et dans les derniers celles du haut de la hiérarchie.
- Une clause `finally` peut être rajoutée après les blocs `catch`. Elle est exécutée dans tous les cas (libération de ressources, fermeture de fichiers, ...).

Graphe d'héritage des exceptions



Graphe d'héritage des exceptions

Les branches du graphe

- `Error` : erreurs graves non réparables. Elles ne peuvent être traitées dans un bloc `try/catch`.
- `Exception` : erreurs récupérables :
  - `InterruptedException` : exceptions qui doivent nécessairement être traitées dans un bloc `try/catch`.
  - `RuntimeException` : exceptions qui n'exigent pas d'être traitées dans un bloc `try/catch` mais on a quand même intérêt à le faire.

Graphe d'héritage des exceptions

Les branches du graphe

- `Error` : erreurs graves non réparables. Elles ne peuvent être traitées dans un bloc `try/catch`.
- `Exception` : erreurs récupérables :
  - `InterruptedException` : exceptions qui doivent nécessairement être traitées dans un bloc `try/catch`.
  - `RuntimeException` : exceptions qui n'exigent pas d'être traitées dans un bloc `try/catch` mais on a quand même intérêt à le faire.

RuntimeException sans bloc try/catch

Exemple

```
int tab[] = new int[10];

for (int i = 0; i < 11; i++)
    tab[i] = i;
System.out.println("fin normale de l'exécution");
```

A l'exécution

- java.lang.ArrayIndexOutOfBoundsException est levée au moment où i vaut 10.
- Le message de fin n'est pas affiché : arrêt brutal du programme.

RuntimeException sans bloc try/catch

Exemple

```
int tab[] = new int[10];

for (int i = 0; i < 11; i++)
    tab[i] = i;
System.out.println("fin normale de l'exécution");
```

A l'exécution

- `java.lang.ArrayIndexOutOfBoundsException` est levée au moment où i vaut 10.
- Le message de fin n'est pas affiché : arrêt brutal du programme.

RuntimeException avec bloc try/catch (conseillé)

Exemple

```
int tab[] = new int[10];

try{
    for (i = 0; i < 11; i++)
        tab[i] = i;
} catch(ArrayIndexOutOfBoundsException e){
    System.out.println("erreur de programmation");
}
System.out.println("fin normale de l'exécution");
```

A l'exécution

- affichage : "erreur de programmation" et "fin normale de l'exécution"
- l'erreur est détectée, contrôlée et le programme se termine normalement.

RuntimeException avec bloc try/catch (conseillé)

Exemple

```
int tab[] = new int[10];

try{
    for (i = 0; i < 11; i++)
        tab[i] = i;
} catch(ArrayIndexOutOfBoundsException e){
    System.out.println("erreur de programmation");
}
System.out.println("fin normale de l'exécution");
```

A l'exécution

- affichage : "erreur de programmation" et "fin normale de l'exécution"
- l'erreur est détectée, contrôlée et le programme se termine normalement.

Redirection (ou délégation) d'une exception

Exemple

```
public class Pile{
    private Object [] tabVal;
    private int indexSommet = -1; // pile vide
    . . .
    public Object depiler() throws Exception{
        Object aux = this.tabVal[this.indexSommet];
        this.indexSommet--;
        return(aux);
    }
```

Exécution

- une `ArrayIndexOutOfBoundsException` est levée lors de l'appel à `this.tabVal[-1]`, qui est redirigée vers l'appelant par `depiler()`
- la méthode `depiler()` lève donc une exception sans qu'il y ait de `throw new Exception(. . .)` dans le corps de cette méthode.

Redirection (ou délégation) d'une exception

Exemple

```
public class Pile{
    private Object [] tabVal;
    private int indexSommet = -1; // pile vide
    . . .
    public Object depiler() throws Exception{
        Object aux = this.tabVal[this.indexSommet];
        this.indexSommet--;
        return(aux);
    }
```

Exécution

- une `ArrayIndexOutOfBoundsException` est levée lors de l'appel à `this.tabVal[-1]`, qui est redirigée vers l'appelant par `depiler()`
- la méthode `depiler()` lève donc une exception sans qu'il y ait de `throw new Exception(. . .)` dans le corps de cette méthode.

Redirection (ou délégation) d'une exception

Autre exemple

```
public class Pile{
    private ArrayList pile;
    ...
    public Object depiler() throws Exception{
        return(this.pile.remove(this.pile.size()-1));
    }
```

Exécution

- Lorsque la pile est vide, `this.pile.size()` vaut 0.
- l'appel à `this.pile.remove(-1)` lève une `IndexOutOfBoundsException` qui est redirigée vers l'appelant par la méthode `depiler()`

Redirection (ou délégation) d'une exception

Autre exemple

```
public class Pile{
    private ArrayList pile;
    ...
    public Object depiler() throws Exception{
        return(this.pile.remove(this.pile.size()-1));
    }
```

Exécution

- Lorsque la pile est vide, `this.pile.size()` vaut 0.
- l'appel à `this.pile.remove(-1)` lève une `IndexOutOfBoundsException` qui est redirigée vers l'appelant par la méthode `depiler()`

Redirection (ou délégation) d'une exception

Autre exemple

```
public class Pile{
    private ArrayList pile;
    ...
    public Object depiler() throws Exception{
        return(this.pile.remove(this.pile.size()-1));
    }
```

Exécution

- Lorsque la pile est vide, `this.pile.size()` vaut 0.
- l'appel à `this.pile.remove(-1)` lève une `IndexOutOfBoundsException` qui est redirigée vers l'appelant par la méthode `depiler()`

Redirection et levée d'une exception dans la même méthode

Exemple

```
public void monDepilage(int i) throws Exception{
    if (i == 0)
        throw new Exception(" i vaut 0 " ); // levée
    Object aux = this.pile.depiler();  // redirection
    . . .
}
```

Filtre de réponse utilisant une exception

Exemple

```
String mac = null;
boolean ok = false;

do{
    try{
        System.out.print("adresse MAC ? : ");
        mac = sc.next();
        Protocoles.verifierMac(mac); // lève une exception
        ok = true;
    } catch (Exception e) {System.out.println(e);}
} while(!ok);
```

Exécution

- au début, `ok` est mis à `false`
- si la méthode `Protocoles.verifierMac(mac)` ne lève pas d'exception, `ok` est mis à `true` et la boucle se termine.
- si elle lève une exception, `ok` reste à `false` et la boucle est répétée.

Filtre de réponse utilisant une exception

Exemple

```
String mac = null;
boolean ok = false;

do{
    try{
        System.out.print("adresse MAC ? : ");
        mac = sc.next();
        Protocoles.verifierMac(mac); // lève une exception
        ok = true;
    } catch (Exception e) {System.out.println(e);}
} while(!ok);
```

Exécution

- au début, `ok` est mis à `false`
- si la méthode `Protocoles.verifierMac(mac)` ne lève pas d'exception, `ok` est mis à `true` et la boucle se termine.
- si elle lève une exception, `ok` reste à `false` et la boucle est répétée.

2. Les fichiers en java

- Écriture et lecture simples
- Écriture et lecture d'un objet simple

Les classes gestionnaires de fichiers

- organisées en couches.
- se définissent et se construisent les unes par rapport aux autres.
- la plupart des méthodes de ces classes lancent des exceptions du type `IOException` qu'il faut déléguer.

Écriture simple

```
import java.io.*;
import java.util.*;

public class Outil{
    public static void enregistrerTexte(String nomFich)
            throws IOException{
        PrintWriter pw = new PrintWriter(nomFich);
        Scanner sc = new Scanner(System.in);
        String ligne;
        do {
            System.out.print("donnez une ligne de texte
                              (! pour terminer) : ");
            ligne = sc.next();
            if (!ligne.equals("!"))
                pw.println(ligne);
        } while(!ligne.equals("!"));
        pw.close();
     } // fin enregistrerTexte
. . .
```

Lecture simple

```
import java.io.*;
import java.util.*;

public class Outil{
    public static void chargerTexte(String nomFich)
            throws IOException{
        FileReader f = new FileReader(nomFich);
        BufferedReader bIn = new BufferedReader(f);
        String ligne = bIn.readLine();
        while (ligne != null){
            System.out.println(ligne); // traitement minimal
            ligne = bIn.readLine();
        }
        bIn.close();
        f.close();
    } // fin chargerTexte
. . .
```

Classe de test

```
import java.io.*;

public class TestEcritureLecture{
    public static void main (String[] args){
        try {
            Outil.enregistrerTexte("essai.txt");
            Outil.chargerTexte("essai.txt");
        } catch(IOException e){System.out.println(e);}
    }  // fin main
} // fin class
```

Écriture d'un objet simple

```
public class Machine{
    private String nom;
    private String adresseIP;
    private String adresseMAC;
    . . .
    public void enregistrerTexte(String nomFichier)
            throws IOException{
        PrintWriter pOut = new PrintWriter(nomFichier);
        pOut.print(this); // appelle le toString() de l'objet
        pOut.close();
    } // fin enregistrerTexte

    public String toString(){
        return (new String(this.nom + ";" + this.adresseIP
                           + ";" + this.adresseMAC));
    } // fin toString
    . . .
```

Lecture d'un objet simple

```
public class Machine{
    . . .
    public void chargerTexte(String nomFich)
            throws IOException{
        FileReader f = new FileReader(nomFich);
        BufferedReader bIn = new BufferedReader(f);
        String ligne = bIn.readLine();
        StringTokenizer st = new StringTokenizer(ligne,";");
        this.nom = st.nextToken();
        this.adresseIP = st.nextToken();
        this.adresseMAC = st.nextToken();
        bIn.close();
        f.close();
   } // fin chargerTexte
. . .
```

Classe de test

```
import java.io.*;

public class TestMachine{
    public static void main (String[] args){
        try{
            Machine m1 = new Machine("pollux","10.0.0.1",
                "04:04:04:04:04:04");
            m1.enregistrerTexte("machine.txt");
            Machine m2 = new Machine();
            m2.chargerTexte("machine.txt");
            System.out.println(m2);
        } catch(IOException e){System.out.println(e);}
    } // fin main
} // fin class
```

3. Les sockets

- Utilisation des sockets
- Communications
- Programmes clients/serveur en mode non connecté
- Programmes clients/serveur en mode connecté

Sockets

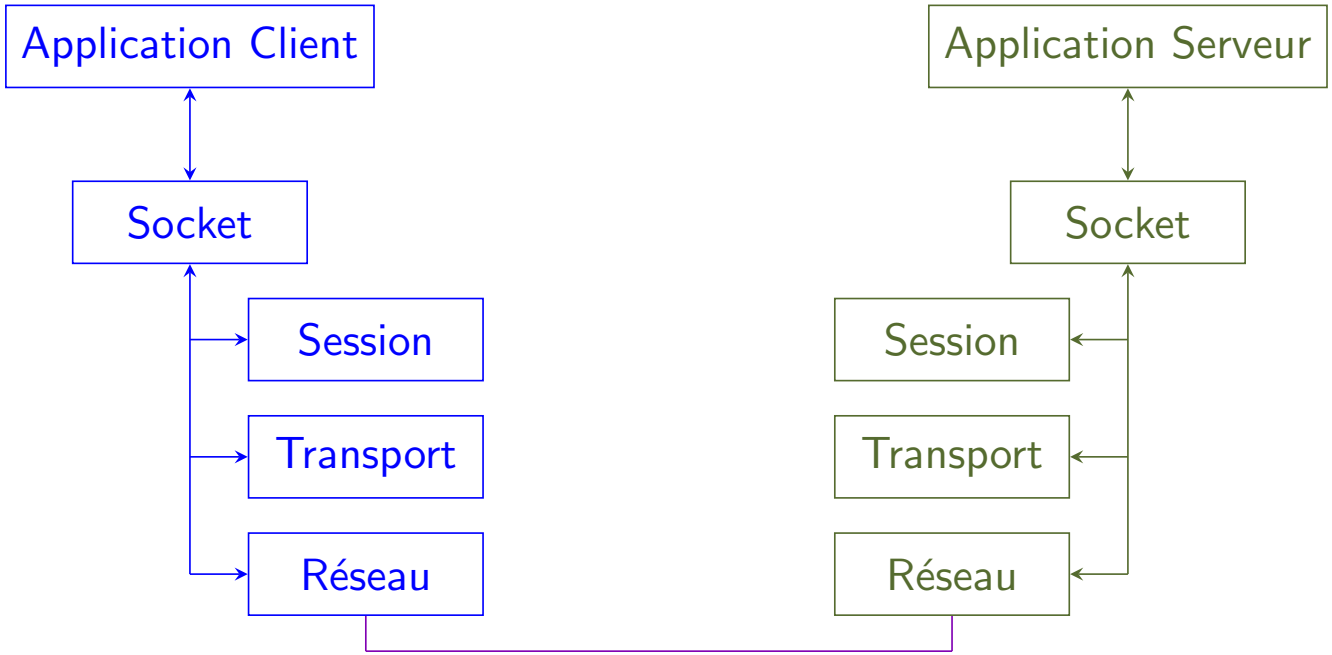
Utilisation des sockets

- mécanisme essentiel à la communication entre processus distants : permet de construire des applications réparties selon le modèle client/serveur.
- concerne les couches 3, 4 et 5

Sockets

Utilisation des sockets

- mécanisme essentiel à la communication entre processus distants : permet de construire des applications réparties selon le modèle client/serveur.
- concerne les couches 3, 4 et 5



API SOCKET (Application Program Interface)

Sockets accessibles en utilisant des API

- bibliothèque de primitives permettant d'assurer l'échange de données entre deux processus distants.
- interface entre les applications et les couches réseaux (3, 4 et 5)
- existe dans de nombreux langages (`C`, `java`, ...)

Paramètres pour l'établissement de la liaison

- protocole de transport utilisé
- adresse IP de la machine distante
- port de la machine distante

API SOCKET (Application Program Interface)

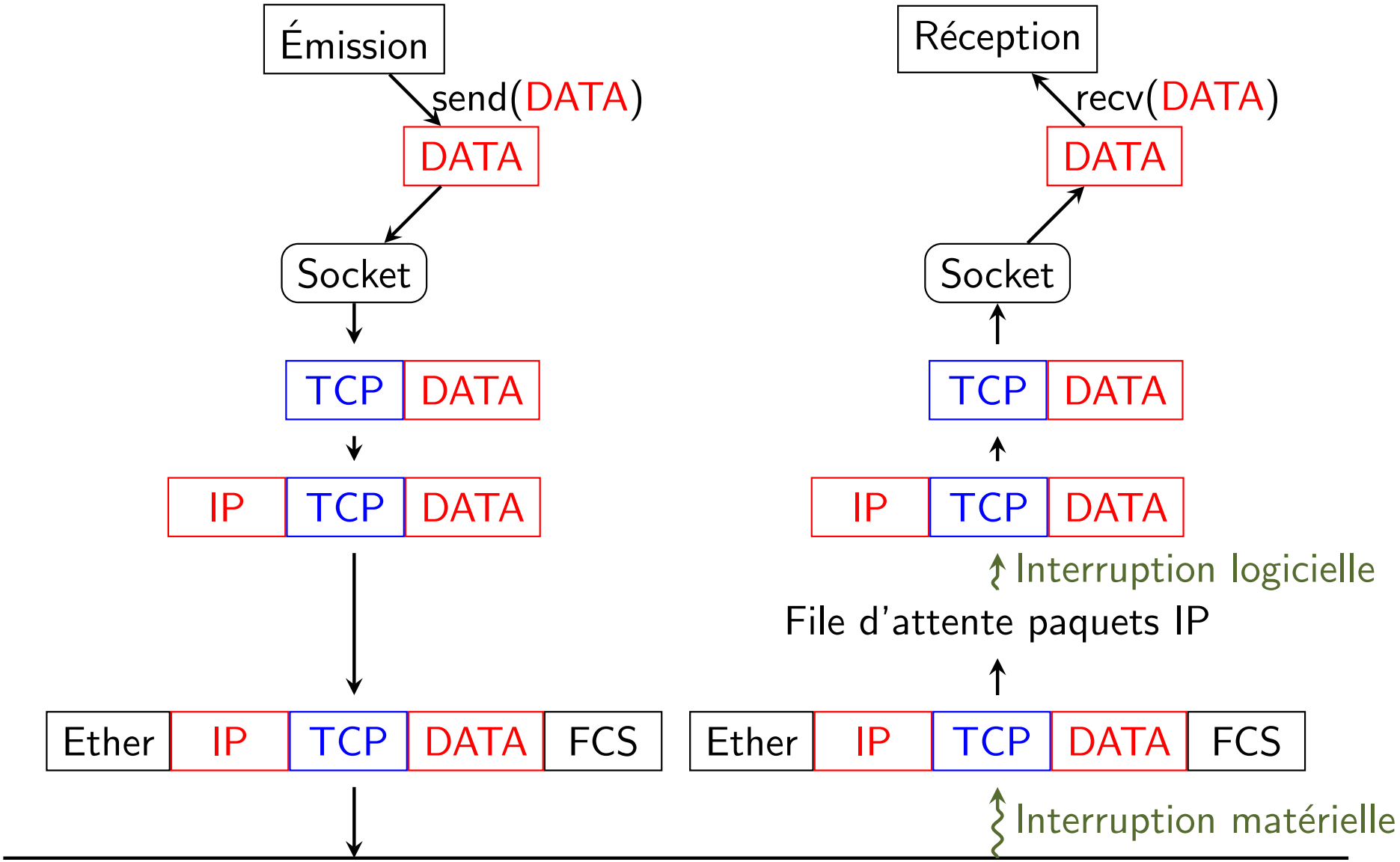
Sockets accessibles en utilisant des API

- bibliothèque de primitives permettant d'assurer l'échange de données entre deux processus distants.
- interface entre les applications et les couches réseaux (3, 4 et 5)
- existe dans de nombreux langages (`C`, `java`, ...)

Paramètres pour l'établissement de la liaison

- protocole de transport utilisé
- adresse IP de la machine distante
- port de la machine distante

Suites d'encapsulations (protocole de transport = TCP)



Communication entre sockets

Communication suivant le mode utilisé

non connecté : par échange de datagrammes (par ex. paquets UDP)

connecté : par l'intermédiaire de fichiers en lecture/écriture.

Rangement dans des tampons (mode connecté)

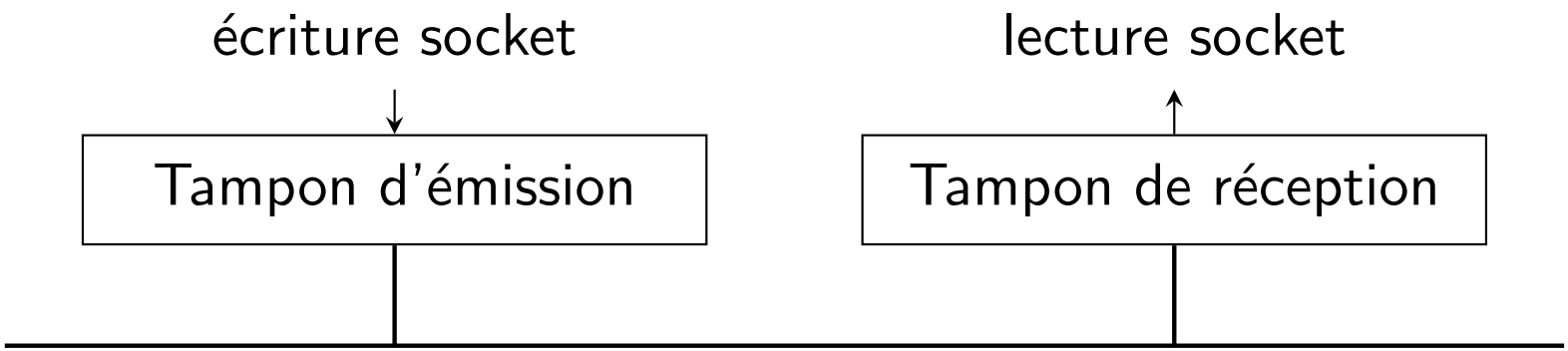
Utilisation de mémoire tampon

- mécanisme de rangement dans des mémoires tampons mis en œuvre par les sockets en mode connecté
- pour chaque connexion, il existe un tampon d'émission et un tampon de réception

Rangement dans des tampons (mode connecté)

Utilisation de mémoire tampon

- mécanisme de rangement dans des mémoires tampons mis en œuvre par les sockets en mode connecté
- pour chaque connexion, il existe un tampon d'émission et un tampon de réception



Appels bloquants ou non bloquants

Socket bloquante

lecture bloquante tant qu'il n'y a rien à lire dans le tampon de réception. Dès leur réception, les données sont envoyées à l'application (même si le nombre d'octets reçus est inférieur au nombre d'octets demandé par l'application).

écriture bloquante uniquement si le tampon en émission est plein. Les octets rangés dans le tampon sont émis sur le réseau tant que le tampon n'est pas vide.

Socket non bloquante

lecture d'un tampon de réception vide, retourne une erreur.

écriture dans un tampon d'émission plein, retourne une erreur.

Appels bloquants ou non bloquants

Socket bloquante

lecture bloquante tant qu'il n'y a rien à lire dans le tampon de réception. Dès leur réception, les données sont envoyées à l'application (même si le nombre d'octets reçus est inférieur au nombre d'octets demandé par l'application).

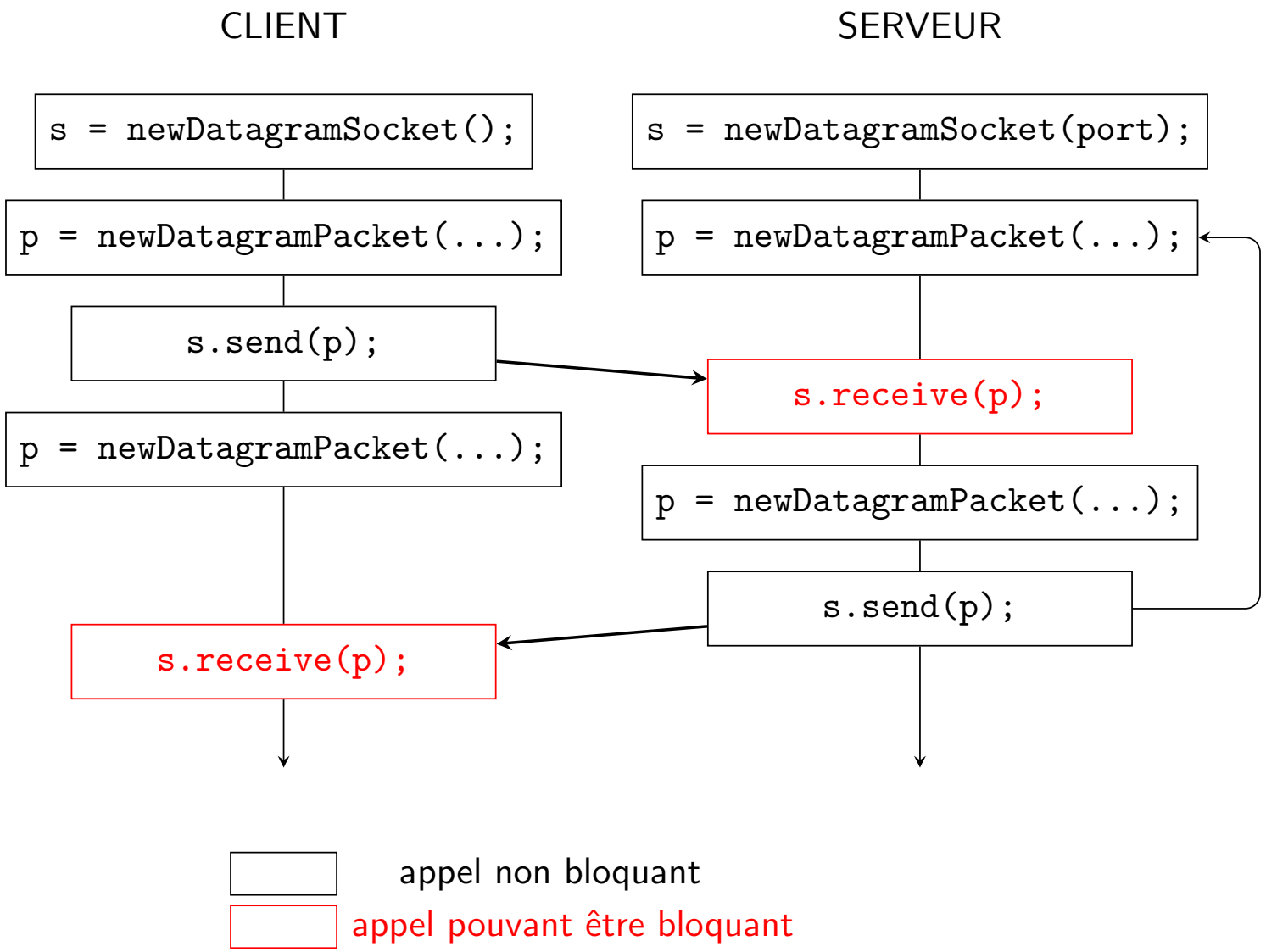
écriture bloquante uniquement si le tampon en émission est plein. Les octets rangés dans le tampon sont émis sur le réseau tant que le tampon n'est pas vide.

Socket non bloquante

lecture d'un tampon de réception vide, retourne une erreur.

écriture dans un tampon d'émission plein, retourne une erreur.

Mode non connecté en Java



Algorithme du client en mode non connecté

1. Créer une `DatagramSocket` sans préciser de numéro de port (le système d'exploitation attribue automatiquement un numéro de port local au client).
2. Préparer un `DatagramPacket` (paquet UDP) qui contiendra la requête au serveur.
3. Utiliser la `DatagramSocket` pour envoyer le paquet au serveur.
4. Utiliser la `DatagramSocket` pour lire la réponse du serveur (reçue dans un `DatagramPacket`).
5. Fermer la `DatagramSocket`.

Algorithme du client en mode non connecté

1. Créer une `DatagramSocket` sans préciser de numéro de port (le système d'exploitation attribue automatiquement un numéro de port local au client).
2. Préparer un `DatagramPacket` (paquet UDP) qui contiendra la requête au serveur.
3. Utiliser la `DatagramSocket` pour envoyer le paquet au serveur.
4. Utiliser la `DatagramSocket` pour lire la réponse du serveur (reçue dans un `DatagramPacket`).
5. Fermer la `DatagramSocket`.

Algorithme du client en mode non connecté

1. Créer une `DatagramSocket` sans préciser de numéro de port (le système d'exploitation attribue automatiquement un numéro de port local au client).
2. Préparer un `DatagramPacket` (paquet UDP) qui contiendra la requête au serveur.
3. Utiliser la `DatagramSocket` pour envoyer le paquet au serveur.
4. Utiliser la `DatagramSocket` pour lire la réponse du serveur (reçue dans un `DatagramPacket`).
5. Fermer la `DatagramSocket`.

Algorithme du client en mode non connecté

1. Créer une `DatagramSocket` sans préciser de numéro de port (le système d'exploitation attribue automatiquement un numéro de port local au client).
2. Préparer un `DatagramPacket` (paquet UDP) qui contiendra la requête au serveur.
3. Utiliser la `DatagramSocket` pour envoyer le paquet au serveur.
4. Utiliser la `DatagramSocket` pour lire la réponse du serveur (reçue dans un `DatagramPacket`).
5. Fermer la `DatagramSocket`.

Algorithme du client en mode non connecté

1. Créer une `DatagramSocket` sans préciser de numéro de port (le système d'exploitation attribue automatiquement un numéro de port local au client).
2. Préparer un `DatagramPacket` (paquet UDP) qui contiendra la requête au serveur.
3. Utiliser la `DatagramSocket` pour envoyer le paquet au serveur.
4. Utiliser la `DatagramSocket` pour lire la réponse du serveur (reçue dans un `DatagramPacket`).
5. Fermer la `DatagramSocket`.

Algorithme du serveur en mode non connecté

- Créer une `DatagramSocket` en précisant le numéro de port sur lequel le serveur écoutera.
- Répéter indéfiniment :
  - Lire la requête du client :
    - Préparer un `DatagramPacket` (paquet UDP) qui recevra la requête du client.
    - Utiliser la `DatagramSocket` pour lire la requête d'un client (reçue dans le `DatagramPacket`).
  - Envoyer la réponse au client :
    - Préparer un `DatagramPacket` qui contiendra la réponse au client.
    - Utiliser la `DatagramSocket` pour répondre au client (réponse contenue dans le `DatagramPacket`).

  Fin Répéter
- Fermer la `DatagramSocket`.

Algorithme du serveur en mode non connecté

- Créer une `DatagramSocket` en précisant le numéro de port sur lequel le serveur écoutera.
- Répéter indéfiniment :
  - Lire la requête du client :
    - Préparer un `DatagramPacket` (paquet UDP) qui recevra la requête du client.
    - Utiliser la `DatagramSocket` pour lire la requête d'un client (reçue dans le `DatagramPacket`).
  - Envoyer la réponse au client :
    - Préparer un `DatagramPacket` qui contiendra la réponse au client.
    - Utiliser la `DatagramSocket` pour répondre au client (réponse contenue dans le `DatagramPacket`).

  Fin Répéter
- Fermer la `DatagramSocket`.

Algorithme du serveur en mode non connecté

- Créer une `DatagramSocket` en précisant le numéro de port sur lequel le serveur écoutera.
- Répéter indéfiniment :
  - Lire la requête du client :
    - Préparer un `DatagramPacket` (paquet UDP) qui recevra la requête du client.
    - Utiliser la `DatagramSocket` pour lire la requête d'un client (reçue dans le `DatagramPacket`).
  - Envoyer la réponse au client :
    - Préparer un `DatagramPacket` qui contiendra la réponse au client.
    - Utiliser la `DatagramSocket` pour répondre au client (réponse contenue dans le `DatagramPacket`).

  Fin Répéter
- Fermer la `DatagramSocket`.

Algorithme du serveur en mode non connecté

- Créer une `DatagramSocket` en précisant le numéro de port sur lequel le serveur écoutera.
- Répéter indéfiniment :
  - Lire la requête du client :
    - Préparer un `DatagramPacket` (paquet UDP) qui recevra la requête du client.
    - Utiliser la `DatagramSocket` pour lire la requête d'un client (reçue dans le `DatagramPacket`).
  - Envoyer la réponse au client :
    - Préparer un `DatagramPacket` qui contiendra la réponse au client.
    - Utiliser la `DatagramSocket` pour répondre au client (réponse contenue dans le `DatagramPacket`).

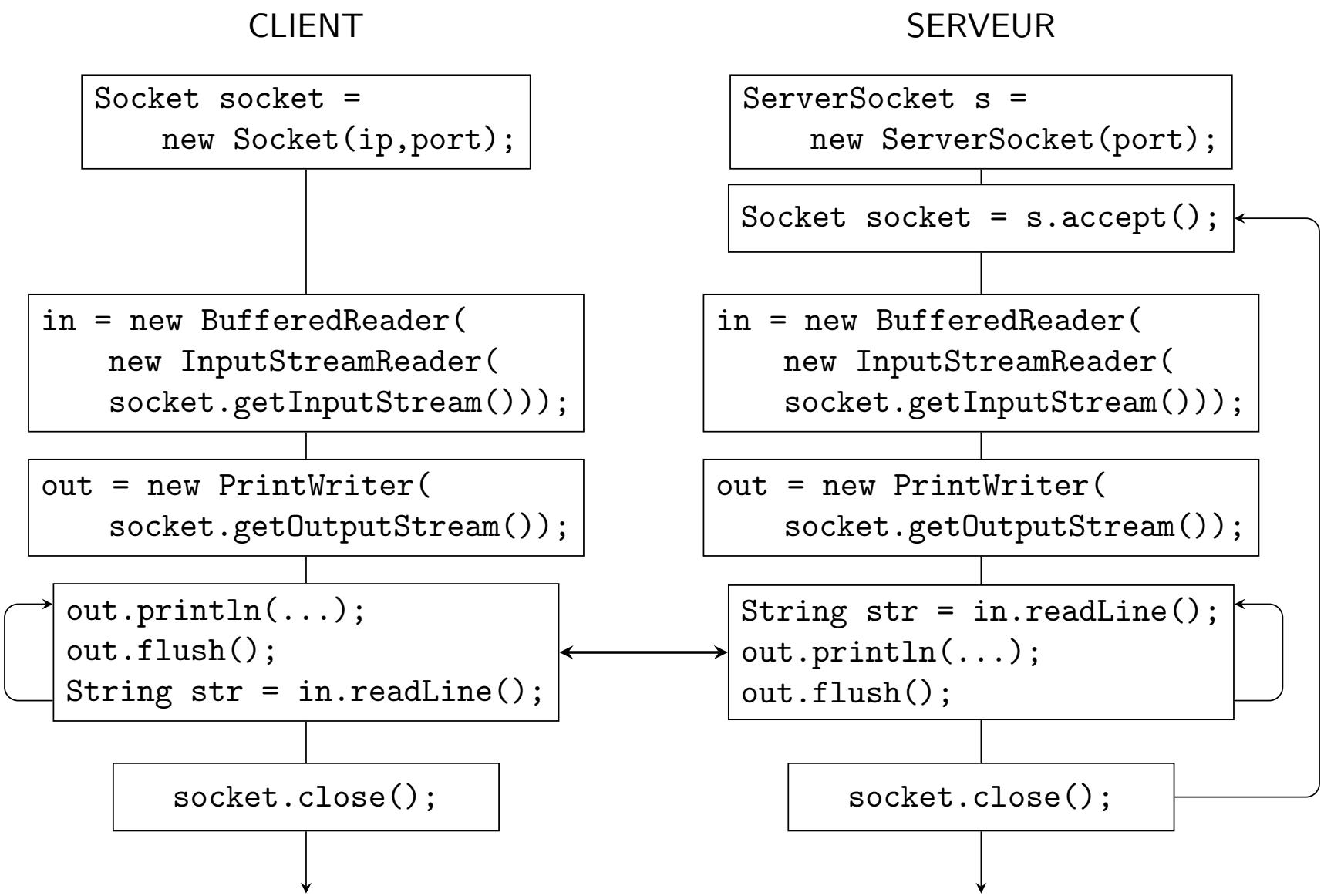
  Fin Répéter
- Fermer la `DatagramSocket`.

Algorithme du serveur en mode non connecté

- Créer une `DatagramSocket` en précisant le numéro de port sur lequel le serveur écoutera.
- Répéter indéfiniment :
  - Lire la requête du client :
    - Préparer un `DatagramPacket` (paquet UDP) qui recevra la requête du client.
    - Utiliser la `DatagramSocket` pour lire la requête d'un client (reçue dans le `DatagramPacket`).
  - Envoyer la réponse au client :
    - Préparer un `DatagramPacket` qui contiendra la réponse au client.
    - Utiliser la `DatagramSocket` pour répondre au client (réponse contenue dans le `DatagramPacket`).

  Fin Répéter
- Fermer la `DatagramSocket`.

Mode connecté en Java



Algorithme du client en mode connecté

1. Créer une socket vers un serveur écoutant sur un port donné : adresse IP du serveur et numéro de port du service (le système d'exploitation attribue automatiquement pour cette connexion un numéro de port local au client).
2. Créer un flux entrant et un flux sortant en utilisant la socket (ces flux se connectent sur les flux sortant et entrant du serveur).
3. Répéter tant que nécessaire
   - Écrire/lire dans ces flux.

   Fin Répéter
4. Fermer les flux.
5. Fermer la socket.

Algorithme du client en mode connecté

1. Créer une socket vers un serveur écoutant sur un port donné : adresse IP du serveur et numéro de port du service (le système d'exploitation attribue automatiquement pour cette connexion un numéro de port local au client).
2. Créer un flux entrant et un flux sortant en utilisant la socket (ces flux se connectent sur les flux sortant et entrant du serveur).
3. Répéter tant que nécessaire
   - Écrire/lire dans ces flux.

   Fin Répéter
4. Fermer les flux.
5. Fermer la socket.

Algorithme du client en mode connecté

1. Créer une socket vers un serveur écoutant sur un port donné : adresse IP du serveur et numéro de port du service (le système d'exploitation attribue automatiquement pour cette connexion un numéro de port local au client).
2. Créer un flux entrant et un flux sortant en utilisant la socket (ces flux se connectent sur les flux sortant et entrant du serveur).
3. Répéter tant que nécessaire
   - Écrire/lire dans ces flux.

   Fin Répéter
4. Fermer les flux.
5. Fermer la socket.

Algorithme du client en mode connecté

1. Créer une socket vers un serveur écoutant sur un port donné : adresse IP du serveur et numéro de port du service (le système d'exploitation attribue automatiquement pour cette connexion un numéro de port local au client).
2. Créer un flux entrant et un flux sortant en utilisant la socket (ces flux se connectent sur les flux sortant et entrant du serveur).
3. Répéter tant que nécessaire
   - Écrire/lire dans ces flux.

   Fin Répéter
4. Fermer les flux.
5. Fermer la socket.

Algorithme du client en mode connecté

1. Créer une socket vers un serveur écoutant sur un port donné : adresse IP du serveur et numéro de port du service (le système d'exploitation attribue automatiquement pour cette connexion un numéro de port local au client).
2. Créer un flux entrant et un flux sortant en utilisant la socket (ces flux se connectent sur les flux sortant et entrant du serveur).
3. Répéter tant que nécessaire
   - Écrire/lire dans ces flux.

   Fin Répéter
4. Fermer les flux.
5. Fermer la socket.

Algorithme du serveur en mode connecté

Algorithme du serveur

- Créer une `ServerSocket` écoutant sur un port donné.
- Se mettre à l'écoute des connexions entrantes.
- Répéter pour chaque connexion entrante :
  - Accepter la connexion : une nouvelle Socket est créée puis passée en paramètre à un thread qui gère la connexion.

  Fin Répéter

Algorithme du thread

- Répéter tant que nécessaire :
  - Lire et/ou écrire sur la nouvelle socket.

  Fin Répéter
- Fermer la socket.

Algorithme du serveur en mode connecté

Algorithme du serveur

- Créer une `ServerSocket` écoutant sur un port donné.
- Se mettre à l'écoute des connexions entrantes.
- Répéter pour chaque connexion entrante :
  - Accepter la connexion : une nouvelle Socket est créée puis passée en paramètre à un thread qui gère la connexion.

  Fin Répéter

Algorithme du thread

- Répéter tant que nécessaire :
  - Lire et/ou écrire sur la nouvelle socket.

  Fin Répéter
- Fermer la socket.

Algorithme du serveur en mode connecté

Algorithme du serveur

- Créer une `ServerSocket` écoutant sur un port donné.
- Se mettre à l'écoute des connexions entrantes.
- Répéter pour chaque connexion entrante :
  - Accepter la connexion : une nouvelle Socket est créée puis passée en paramètre à un thread qui gère la connexion.

  Fin Répéter

Algorithme du thread

- Répéter tant que nécessaire :
  - Lire et/ou écrire sur la nouvelle socket.

  Fin Répéter
- Fermer la socket.

Algorithme du serveur en mode connecté

Algorithme du serveur

- Créer une `ServerSocket` écoutant sur un port donné.
- Se mettre à l'écoute des connexions entrantes.
- Répéter pour chaque connexion entrante :
  - Accepter la connexion : une nouvelle Socket est créée puis passée en paramètre à un thread qui gère la connexion.

  Fin Répéter

Algorithme du thread

- Répéter tant que nécessaire :
  - Lire et/ou écrire sur la nouvelle socket.

  Fin Répéter
- Fermer la socket.

Algorithme du serveur en mode connecté

Algorithme du serveur

- Créer une `ServerSocket` écoutant sur un port donné.
- Se mettre à l'écoute des connexions entrantes.
- Répéter pour chaque connexion entrante :
  - Accepter la connexion : une nouvelle Socket est créée puis passée en paramètre à un thread qui gère la connexion.

  Fin Répéter

Algorithme du thread

- Répéter tant que nécessaire :
  - Lire et/ou écrire sur la nouvelle socket.

  Fin Répéter
- Fermer la socket.

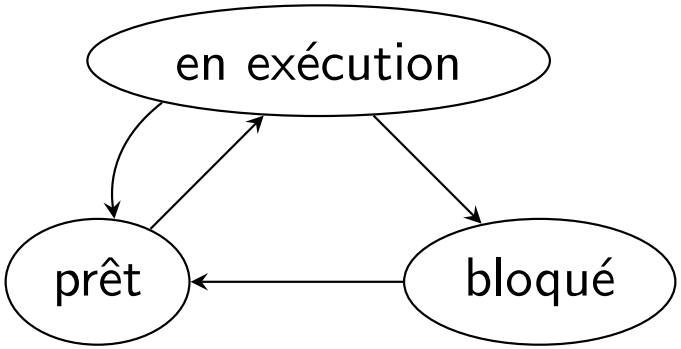
4. Les threads

- Rappels
- Les threads : des processus particuliers
- Les threads en `java`
- Héritage de la classe `Thread`
- Implémentation de l'interface `Runnable`
- La terminaison d'un thread

Rappels sur les processus

- Un processus est un programme en cours d'exécution.
- Dans le cas général, plusieurs processus semblent s'exécuter simultanément (multi-tâches)
- En réalité, l'ordonnanceur répartit le temps CPU entre les différents processus.
- Le système alloue de la mémoire à chaque processus.
- Un processus a besoin de ressources pour s'exécuter : il dispose d'un ensemble de registres du processeur qui sont sauvegardés par le système lorsque le processus n'est pas en cours d'exécution.

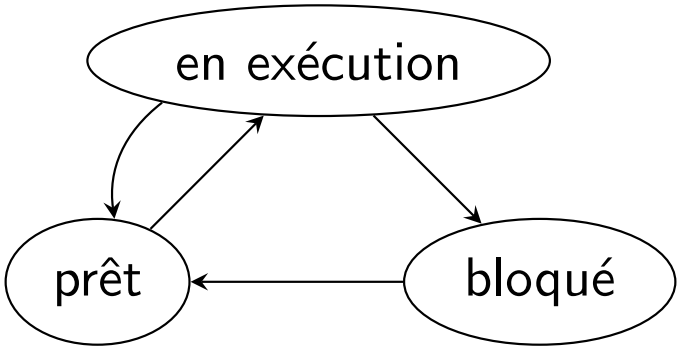
Les différents états d'un processus



Différents états d'un processus

- En cours d'exécution (`running`) : il s'exécute sur le processeur.
- Prêt : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en cours d'exécution).
- Bloqué : en attente de ressource ou en sommeil.

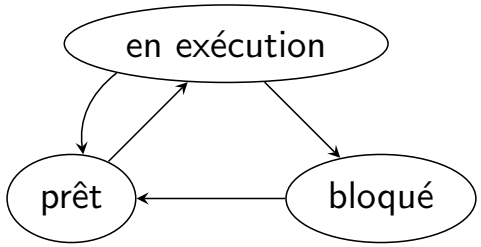
Les différents états d'un processus



Différents états d'un processus

- En cours d'exécution (`running`) : il s'exécute sur le processeur.
- Prêt : le processus est prêt à s'exécuter, mais n'a pas le processeur (occupé par un autre processus en cours d'exécution).
- Bloqué : en attente de ressource ou en sommeil.

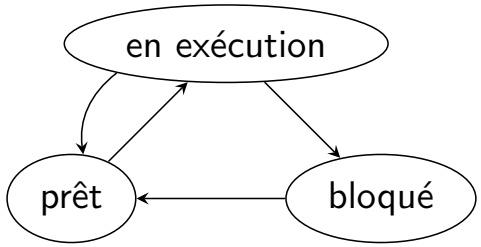
Les transitions entre états d'un processus



Transitions entre états

- en exécution → bloqué :
  - S'il reçoit l'ordre de dormir.
  - S'il a besoin d'une ressource non disponible.
- en exécution → prêt :
  - S'il a épuisé son quantum de temps CPU.
- prêt → en exécution :
  - Si l'ordonnanceur lui donne la main.
- bloqué → prêt :
  - Si son temps de sommeil est épuisé.
  - Si la ressource qu'il attendait s'est libérée.

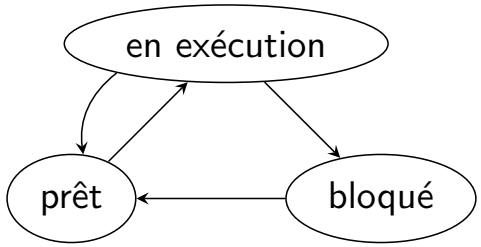
Les transitions entre états d'un processus



Transitions entre états

- en exécution → bloqué :
  - S'il reçoit l'ordre de dormir.
  - S'il a besoin d'une ressource non disponible.
- en exécution → prêt :
  - S'il a épuisé son quantum de temps CPU.
- prêt → en exécution :
  - Si l'ordonnanceur lui donne la main.
- bloqué → prêt :
  - Si son temps de sommeil est épuisé.
  - Si la ressource qu'il attendait s'est libérée.

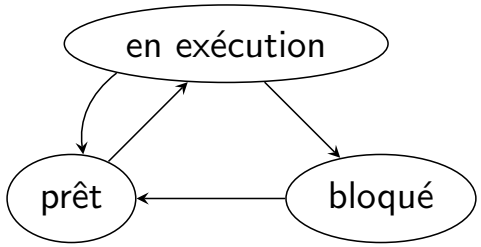
Les transitions entre états d'un processus



Transitions entre états

- en exécution → bloqué :
  - S'il reçoit l'ordre de dormir.
  - S'il a besoin d'une ressource non disponible.
- en exécution → prêt :
  - S'il a épuisé son quantum de temps CPU.
- prêt → en exécution :
  - Si l'ordonnanceur lui donne la main.
- bloqué → prêt :
  - Si son temps de sommeil est épuisé.
  - Si la ressource qu'il attendait s'est libérée.

Les transitions entre états d'un processus



Transitions entre états

- en exécution → bloqué :
  - S'il reçoit l'ordre de dormir.
  - S'il a besoin d'une ressource non disponible.
- en exécution → prêt :
  - S'il a épuisé son quantum de temps CPU.
- prêt → en exécution :
  - Si l'ordonnanceur lui donne la main.
- bloqué → prêt :
  - Si son temps de sommeil est épuisé.
  - Si la ressource qu'il attendait s'est libérée.

Caractéristiques des threads

- Un thread est un processus léger à l'intérieur d'un processus.
- Un processus peut posséder plusieurs threads (multi-threading) : exécution concurrente à l'intérieur d'un processus de plusieurs tâches.
- Un processus possède au moins un thread qui exécute le programme principal, habituellement la fonction `main()`.

Threads *vs.* sous-processus

Différences entre threads et sous-processus

- Les sous-processus issus d'un même processus ont leur propre espace d'adressage et doivent, pour communiquer, utiliser des moyens de communication spécifiques (tubes).
- Les threads issus d'un même processus partagent les ressources allouées à celui-ci : temps processeur, mémoire (segments de code et de données).

Intérêt des threads

- Communication très simple (mais parfois dangereuse) grâce aux données partagées.
- Partage du temps alloué au processus père entre plusieurs threads, chacun d'eux exécutant une fonction précise.

Threads *vs.* sous-processus

Différences entre threads et sous-processus

- Les sous-processus issus d'un même processus ont leur propre espace d'adressage et doivent, pour communiquer, utiliser des moyens de communication spécifiques (tubes).
- Les threads issus d'un même processus partagent les ressources allouées à celui-ci : temps processeur, mémoire (segments de code et de données).

Intérêt des threads

- Communication très simple (mais parfois dangereuse) grâce aux données partagées.
- Partage du temps alloué au processus père entre plusieurs threads, chacun d'eux exécutant une fonction précise.

Threads *vs.* sous-processus

Quand utiliser des threads ?

Lorsque l'application contient des tâches répétitives et/ou grosses consommatrices de temps CPU.

- calculs
- animation
- attente de connexions

Autres exemples

- Un serveur peut répondre à des demandes de connexion de clients en créant un thread par client.
- Le lancement de la machine virtuelle java lance un processus de base qui contient plusieurs threads :
  - le thread principal : celui qui exécute le `main()`.
  - le thread dédié au garbage collector.
  - le thread dédié au traitement des événements clavier et souris.

Threads *vs.* sous-processus

Quand utiliser des threads ?

Lorsque l'application contient des tâches répétitives et/ou grosses consommatrices de temps CPU.

- calculs
- animation
- attente de connexions

Autres exemples

- Un serveur peut répondre à des demandes de connexion de clients en créant un thread par client.
- Le lancement de la machine virtuelle java lance un processus de base qui contient plusieurs threads :
  - le thread principal : celui qui exécute le `main()`.
  - le thread dédié au garbage collector.
  - le thread dédié au traitement des événements clavier et souris.

Threads *vs.* sous-processus

Quand utiliser des threads ?

Lorsque l'application contient des tâches répétitives et/ou grosses consommatrices de temps CPU.

- calculs
- animation
- attente de connexions

Autres exemples

- Un serveur peut répondre à des demandes de connexion de clients en créant un thread par client.
- Le lancement de la machine virtuelle java lance un processus de base qui contient plusieurs threads :
  - le thread principal : celui qui exécute le `main()`.
  - le thread dédié au garbage collector.
  - le thread dédié au traitement des événements clavier et souris.

Threads *vs.* sous-processus

Quand utiliser des threads ?

Lorsque l'application contient des tâches répétitives et/ou grosses consommatrices de temps CPU.

- calculs
- animation
- attente de connexions

Autres exemples

- Un serveur peut répondre à des demandes de connexion de clients en créant un thread par client.
- Le lancement de la machine virtuelle java lance un processus de base qui contient plusieurs threads :
  - le thread principal : celui qui exécute le `main()`.
  - le thread dédié au garbage collector.
  - le thread dédié au traitement des événements clavier et souris.

Les threads en java

Deux possibilités

- Hériter de la classe `Thread`.
- Implémenter l'interface `Runnable`.

Hériter de la classe Thread

Exemple

```
public class MonThread extends Thread{
    public MonThread(){
        super();
        . . .
    }
    public void run{
        // on met ici le code à exécuter, c'est-à-dire ce que
        // le thread devra faire lorsqu'il aura la main
    }
}
```

Utilisation

```
MonThread th = new MonThread();
th.start();
```

Les méthodes `run()` et `start()`

`public void run()`

- La redéfinition de cette méthode indique ce que le thread doit faire quand il s'exécutera.
- Elle contient toujours (directement ou indirectement) une répétitive.
- Elle n'est pas appellée directement mais par l'intermédiaire de la méthode `start()`.

`public void start()`

- met le thread à l'état prêt, ce qui lui permet de démarrer dès que possible.
- c'est la machine virtuelle Java qui décide du moment où le thread va s'exécuter : appel de la méthode `run()` du thread.
- le programmeur rédige la méthode `run()` et utilise la méthode `start()`.

Les méthodes `run()` et `start()`

`public void run()`

- La redéfinition de cette méthode indique ce que le thread doit faire quand il s'exécutera.
- Elle contient toujours (directement ou indirectement) une répétitive.
- Elle n'est pas appellée directement mais par l'intermédiaire de la méthode `start()`.

`public void start()`

- met le thread à l'état prêt, ce qui lui permet de démarrer dès que possible.
- c'est la machine virtuelle Java qui décide du moment où le thread va s'exécuter : appel de la méthode `run()` du thread.
- le programmeur rédige la méthode `run()` et utilise la méthode `start()`.

Exemple

Classe ThreadAffiche

```
public class ThreadAffiche extends Thread{
    private int numero;
    private int sommeil;

    public ThreadAffiche(int numero, int sommeil, int priorite){
        super();
        this.numero = numero;
        this.sommeil = sommeil;
        this.setPriority(priorite);
    }
    . . .
```

Exemple

Classe ThreadAffiche (suite)

```
    . . .
    public void run(){
        for (int i = 0; i < 5; i++){
            System.out.print(" ["+ this.numero +"] ");
            try{
                this.sleep(sommeil);
            } catch(InterruptedException e){
                System.out.println(e);
            }
        }
    } // fin run
} // fin de la classe ThreadAffiche
```

Exemple

Classe de test

```
public class TestAffiche{
    public static void main(String[] args){
        ThreadAffiche tha = new ThreadAffiche(1,10,1);
        ThreadAffiche thb = new ThreadAffiche(2,10,1);
        tha.start(); thb.start();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            try{Thread.sleep(10);
            } catch(InterruptedException e){
                System.out.println(e);}
        }
    } // fin main
} // fin de la classe TestAffiche
```

Un affichage possible à l'éxécution

[main] [2] [1] [main] [2] [1] [main] [1] [2] [main] [2] [1] [main] [2] [1]

Exemple

Classe de test

```
public class TestAffiche{
    public static void main(String[] args){
        ThreadAffiche tha = new ThreadAffiche(1,10,1);
        ThreadAffiche thb = new ThreadAffiche(2,10,1);
        tha.start(); thb.start();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            try{Thread.sleep(10);
            } catch(InterruptedException e){
                System.out.println(e);}
        }
    } // fin main
} // fin de la classe TestAffiche
```

Un affichage possible à l'éxécution

[main] [2] [1] [main] [2] [1] [main] [1] [2] [main] [2] [1] [main] [2] [1]

Exemple

Modification des priorités et du temps de sommeil

```
public class TestAffiche{
    public static void main(String[] args){
        ThreadAffiche tha = new ThreadAffiche(1,2,8);
        ThreadAffiche thb = new ThreadAffiche(2,10,1);
        tha.start(); thb.start();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            try{Thread.sleep(10);
            } catch(InterruptedException e){
                System.out.println(e);}
        }
    } // fin main
} // fin de la classe TestAffiche
```

Affichage à l'éxécution

[1] [2] [main] [1] [1] [1] [1] [main] [2] [main] [2] [main] [2] [main] [2]

Exemple

Modification des priorités et du temps de sommeil

```
public class TestAffiche{
    public static void main(String[] args){
        ThreadAffiche tha = new ThreadAffiche(1,2,8);
        ThreadAffiche thb = new ThreadAffiche(2,10,1);
        tha.start(); thb.start();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            try{Thread.sleep(10);
            } catch(InterruptedException e){
                System.out.println(e);}
        }
    } // fin main
} // fin de la classe TestAffiche
```

Affichage à l'éxécution

[1] [2] [main] [1] [1] [1] [1] [main] [2] [main] [2] [main] [2] [main] [2]

Implémentation de l'interface Runnable

Classe ThreadAfficheRunnable

```
public class ThreadAfficheRunnable implements Runnable{
    private int numero;
    private int sommeil;
    private Thread th;

    public ThreadAfficheRunnable(int numero, int sommeil,
            int priorite){
        this.numero = numero;
        this.sommeil = sommeil;
        this.th = new Thread(this);
        this.th.setPriority(priorite);
    }

    public void demarrer(){
        this.th.start();
    }
    . . .
```

Implémentation de l'interface Runnable

Classe ThreadAfficheRunnable (suite)

```
    ...
    public void run(){
        for (int i = 0; i < 5; i++){
            System.out.print(" ["+ this.numero +"] ");
            try{
                this.th.sleep(sommeil);
            } catch(InterruptedException e){
                System.out.println(e);
            }
        }
    } // fin run
} // fin de la classe ThreadAfficheRunnable
```

Implémentation de l'interface Runnable

Classe de test

```
public class TestAfficheRunnable{
    public static void main(String[] args){
        ThreadAfficheRunnable tha =
            new ThreadAfficheRunnable(1,10,1);
        ThreadAfficheRunnable thb =
            new ThreadAfficheRunnable(2,10,1);
        tha.demarrer(); thb.demarrer();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            try{Thread.sleep(10);}
            catch(InterruptedException e){System.out.println(e);}}
    } // fin main
} // fin de la classe TestAfficheRunnable
```

Affichage possible à l'éxécution

[main] [2] [1] [main] [2] [1] [main] [1] [2] [main] [2] [1] [main] [2] [1]

Implémentation de l'interface Runnable

Classe de test

```
public class TestAfficheRunnable{
    public static void main(String[] args){
        ThreadAfficheRunnable tha =
            new ThreadAfficheRunnable(1,10,1);
        ThreadAfficheRunnable thb =
            new ThreadAfficheRunnable(2,10,1);
        tha.demarrer(); thb.demarrer();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            try{Thread.sleep(10);}
            catch(InterruptedException e){System.out.println(e);}}
    } // fin main
} // fin de la classe TestAfficheRunnable
```

Affichage possible à l'éxécution

[main] [2] [1] [main] [2] [1] [main] [1] [2] [main] [2] [1] [main] [2] [1]

Threads avec paramètre Runnable

Si un thread est construit avec en paramètre un objet Runnable, le code à exécuter par ce thread doit figurer dans la méthode run() du paramètre.

```
MaClasse mc = new MaClasse(...);
Thread th = new Thread(mc);
th.start();
```

Le thread th est prêt à exécuter la méthode run() de l'objet Runnable mc.

Threads avec paramètre Runnable

Si un thread est construit avec en paramètre un objet Runnable, le code à exécuter par ce thread doit figurer dans la méthode run() du paramètre.

```
MaClasse mc = new MaClasse(...);
Thread th = new Thread(mc);
th.start();
```

Le thread th est prêt à exécuter la méthode run() de l'objet Runnable mc.

Comment forcer un thread à terminer avant les autres ?

Modification de la classe `ThreadAfficheRunnable`

```
public void join() throws InterruptedException{
    this.th.join();
}
```

Comment forcer un thread à terminer avant les autres ?

Classe de test

```
public static void main(String[] args){
    try{ThreadAfficheRunnable tha =
            new ThreadAfficheRunnable(1,10,1);
        ThreadAfficheRunnable thb =
            new ThreadAfficheRunnable(2,10,1);
        tha.demarrer();
        tha.join(); // tha termine avant les autres
        thb.demarrer();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            Thread.sleep(10);}
    } catch(Exception e){System.out.println(e);}
} // fin main
```

Affichage à l'éxécution

[1] [1] [1] [1] [1] [main] [2] [main] [2] [main] [2] [main] [2] [main] [2]

Comment forcer un thread à terminer avant les autres ?

Classe de test

```
public static void main(String[] args){
    try{ThreadAfficheRunnable tha =
            new ThreadAfficheRunnable(1,10,1);
        ThreadAfficheRunnable thb =
            new ThreadAfficheRunnable(2,10,1);
        tha.demarrer();
        tha.join(); // tha termine avant les autres
        thb.demarrer();
        for (int i = 0; i < 5; i++){
            System.out.print(" [main] ");
            Thread.sleep(10);}
    } catch(Exception e){System.out.println(e);}
} // fin main
```

Affichage à l'éxécution

[1] [1] [1] [1] [1] [main] [2] [main] [2] [main] [2] [main] [2] [main] [2]

Terminer l'exécution d'un thread

Comment stopper un thread ?

- Un thread doit terminer "naturellement" : il n'y a pas de méthode particulière pour l'arrêter.
- Le thread termine quand sa méthode `run()` termine.

Comment stopper un thread depuis l'extérieur ?

- programmer dans sa méthode `run` une boucle qui s'exécute tant qu'un booléen est `vrai`.
- positionner ce booléen à `faux` pour stopper le thread.

Terminer l'exécution d'un thread

Comment stopper un thread ?

- Un thread doit terminer "naturellement" : il n'y a pas de méthode particulière pour l'arrêter.
- Le thread termine quand sa méthode `run()` termine.

Comment stopper un thread depuis l'extérieur ?

- programmer dans sa méthode `run` une boucle qui s'exécute tant qu'un booléen est `vrai`.
- positionner ce booléen à `faux` pour stopper le thread.

Stopper un thread de l'extérieur

Structure

```
public class MonThread implements Runnable{
    . . .
    private boolean vivant;
    public MonThread(){
        . . .
        vivant=true;}
    public void setVivant(boolean vivant){
        this.vivant= vivant;}
    public void run(){
        while (vivant){
            . . .}
    }
}
```

Arrêt depuis l'extérieur

```
mt.setVivant(false);
```

Stopper un thread de l'extérieur

Structure

```
public class MonThread implements Runnable{
    . . .
    private boolean vivant;
    public MonThread(){
        . . .
        vivant=true;}
    public void setVivant(boolean vivant){
        this.vivant= vivant;}
    public void run(){
        while (vivant){
            . . .}
    }
}
```

Arrêt depuis l'extérieur

```
mt.setVivant(false);
```

5. Synchronisation de threads

- Objectif et mécanisme de base
- Gestion des threads en attente

Objectif de la synchronisation de threads

Objectif

Éviter les conflits d'accès concurents de plusieurs threads à une même ressource.

Solution

Déclarer `synchronized` les méthodes critiques de la ressource.

Objectif de la synchronisation de threads

Objectif

Éviter les conflits d'accès concurents de plusieurs threads à une même ressource.

Solution

Déclarer `synchronized` les méthodes critiques de la ressource.

Synchronisation : exemple

Une ressource partagée

```
public class Ressource{
    public Ressource(){}

    public void afficher(int numero){
        for (int i = 0; i < 3; i++){
            System.out.print(numero + " ");
            try{
                Thread.sleep(500); // attente d'une demi seconde
            } catch(InterruptedException e){
                System.out.println(e);
            }
        }
    }
}
```

Synchronisation : exemple

Un thread

```
public class ThreadConcurent implements Runnable{
    private int numero;
    private Ressource ressource; // ressource partagée
    private Thread th;

    public ThreadConcurent(int numero,Ressource ressource){
        this.numero=numero;
        this.ressource=ressource;
        this.th = new Thread(this);
        this.th.start(); // démarrage à la fin du constructeur
    }

    public void run(){
        this.ressource.afficher(this.numero);
    }
}
```

Synchronisation : exemple

Classe de test : ressource partagée par 3 threads concurrents

```
public class Test{
    public static void main(String[] args){
        Ressource ressourcePartagee = new Ressource();
        ThreadConcurent tc1 =
            new ThreadConcurent(1,ressourcePartagee);
        ThreadConcurent tc2 =
            new ThreadConcurent(2,ressourcePartagee);
        ThreadConcurent tc3 =
            new ThreadConcurent(3,ressourcePartagee);
    }
}
```

Exécution

```
1 2 3 1 2 3 1 2 3
```

Synchronisation : exemple

Classe de test : ressource partagée par 3 threads concurrents

```
public class Test{
    public static void main(String[] args){
        Ressource ressourcePartagee = new Ressource();
        ThreadConcurent tc1 =
            new ThreadConcurent(1,ressourcePartagee);
        ThreadConcurent tc2 =
            new ThreadConcurent(2,ressourcePartagee);
        ThreadConcurent tc3 =
            new ThreadConcurent(3,ressourcePartagee);
    }
}
```

Exécution

```
1 2 3 1 2 3 1 2 3
```

Synchronisation : attente des threads

Les threads appelant une ressource sont mis en attente dans une pile

On déclare `synchronized` la méthode appelée.

```
public synchronized void afficher(int numero){
    . . .
}
```

Exécution

```
1 1 1 3 3 3 2 2 2
```

- le premier qui invoque la méthode termine son affichage.
- le troisième attend que le premier termine avant de pouvoir utiliser la ressource.
- le second attend que le troisième termine.

Mise en attente avec `wait()`

Un thread peut être mis en attente par une méthode synchronisée qu'il appelle en utilisant `wait()`.

```
public synchronized void methode1(){
    while (!condition)
        this.wait();
        // c'est le thread appelant la méthode
        // qui est mis en attente
    ...
}
```

Débloquer un thread avec `notify()`

Pour débloquer thread en attente, il faut qu'un autre thread appelle une autre méthode synchonisée de la classe qui invoque la méthode `notify()` :

```
public synchronized void methode2(){
    ...
    this.notify();
    // le thread en attente est débloqué
    ...
}
```

Exemple

Classe Ressource

```
public class Ressource{
    private int valeur;
    public Ressource(){
        this.valeur=0;
    }
    public synchronized void afficher() throws Exception{
        while (this.valeur==0)
            this.wait();
        System.out.println("affichage : " + this.valeur);
    }
    public synchronized void incrementer(){
        System.out.println("incrémentation");
        this.valeur++;
        this.notify();
    }
}
```

Exemple

Utilisation

Une instance de la classe `Ressource` est utilisée par deux threads, un thread afficheur et un thread incrémenteur :

```
Ressource ressource = new Ressource();
ThreadAfficheur tha = new ThreadAfficheur(ressource);
ThreadIncrementeur thb = new ThreadIncrementeur(ressource);
. . .
```

Exemple

Exemple d'afficheur

Dans sa méthode `run()` le thread afficheur appelle la méthode `afficher()` de la ressource :

```
public void run(){
    try{
        this.ressource.afficher();
    } catch(Exception e){System.out.println(e);}
} // end run
```

Exemple d'incrémenteur

Dans sa méthode `run()` le thread incrémenteur appelle la méthode `incrementer()` de la ressource :

```
public void run(){
    this.ressource.incrementer();
} // end run
```

Exemple

Exemple d'afficheur

Dans sa méthode `run()` le thread afficheur appelle la méthode `afficher()` de la ressource :

```
public void run(){
    try{
        this.ressource.afficher();
    } catch(Exception e){System.out.println(e);}
} // end run
```

Exemple d'incrémenteur

Dans sa méthode `run()` le thread incrémenteur appelle la méthode `incrementer()` de la ressource :

```
public void run(){
    this.ressource.incrementer();
} // end run
```

Exemple

Classe de test

```
import java.util.Scanner;

public class Test{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        Ressource ressource = new Ressource();
        ThreadAfficheur tA =
            new ThreadAfficheur(ressource);
        ThreadIncrementeur tI =
            new ThreadIncrementeur(ressource);
        tA.demarrer();
        System.out.print("donnez votre nom : "); // pour attendre
        String s = sc.next();
        tI.demarrer();
    } // fin main
} // fin de la classe Test
```

Exemple

Exécution

Lorsque le thread afficheur tA démarre, il est mis en attente par la méthode `afficher()` de la ressource (car la variable `valeur` vaut 0) jusqu'à ce que le thread incrementeur tI invoque la méthode `incrementer()` de la ressource et que l'appel à `notify()` débloque tA.

Rappel : méthodes `incrementer()` et `afficher()` de `Ressource`

```
public synchronized void incrementer(){
    System.out.println("incrémentation");
    this.valeur++;
    this.notify();
}
public synchronized void afficher() throws Exception{
    while (this.valeur==0)
        this.wait();
    System.out.println("affichage : " + this.valeur);
}
```

Exemple

Affichage

```
donnez votre nom : X
incrémentation
affichage : 1
```

Remarque

Si l'on démarre `tI` avant `tA`, l'appel à `notify()` est fait "dans le vide".

```
tI.demarrer();
tA.demarrer();
System.out.print("donnez votre nom : "); // pour attendre
String s = sc.next();
```

Affichage

```
donnez votre nom : incrémentation
affichage : 1
// attente de la saisie du nom
```

Exemple

Affichage

```
donnez votre nom : X
incrémentation
affichage : 1
```

Remarque

Si l'on démarre `tI` avant `tA`, l'appel à `notify()` est fait "dans le vide".

```
tI.demarrer();
tA.demarrer();
System.out.print("donnez votre nom : "); // pour attendre
String s = sc.next();
```

Affichage

```
donnez votre nom : incrémentation
affichage : 1
// attente de la saisie du nom
```

Exemple

Affichage

```
donnez votre nom : X
incrémentation
affichage : 1
```

Remarque

Si l'on démarre `tI` avant `tA`, l'appel à `notify()` est fait "dans le vide".

```
tI.demarrer();
tA.demarrer();
System.out.print("donnez votre nom : "); // pour attendre
String s = sc.next();
```

Affichage

```
donnez votre nom : incrémentation
affichage : 1
// attente de la saisie du nom
```