

Architecture logicielle

IUT de Paris – S4
Jean-François Brette

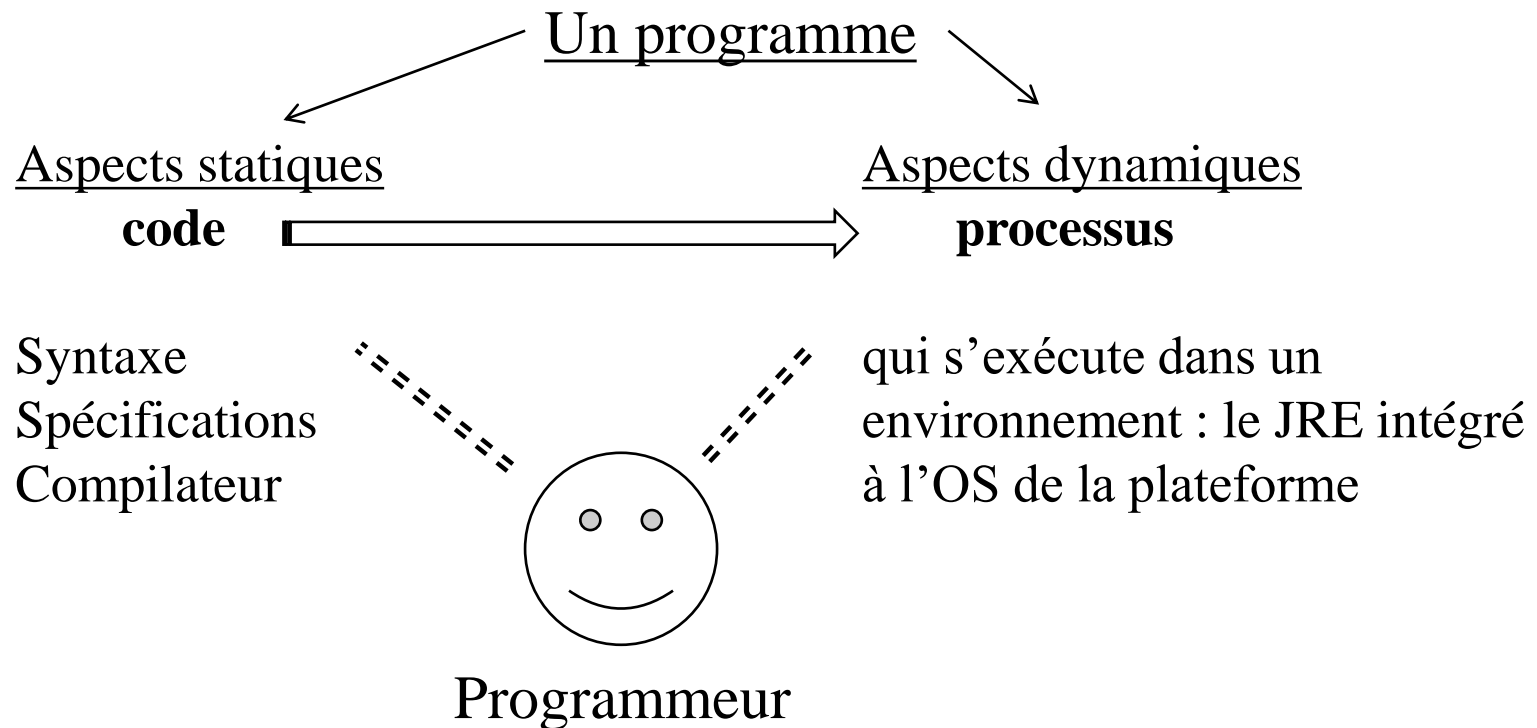
1. Parallélisme - Multithreading

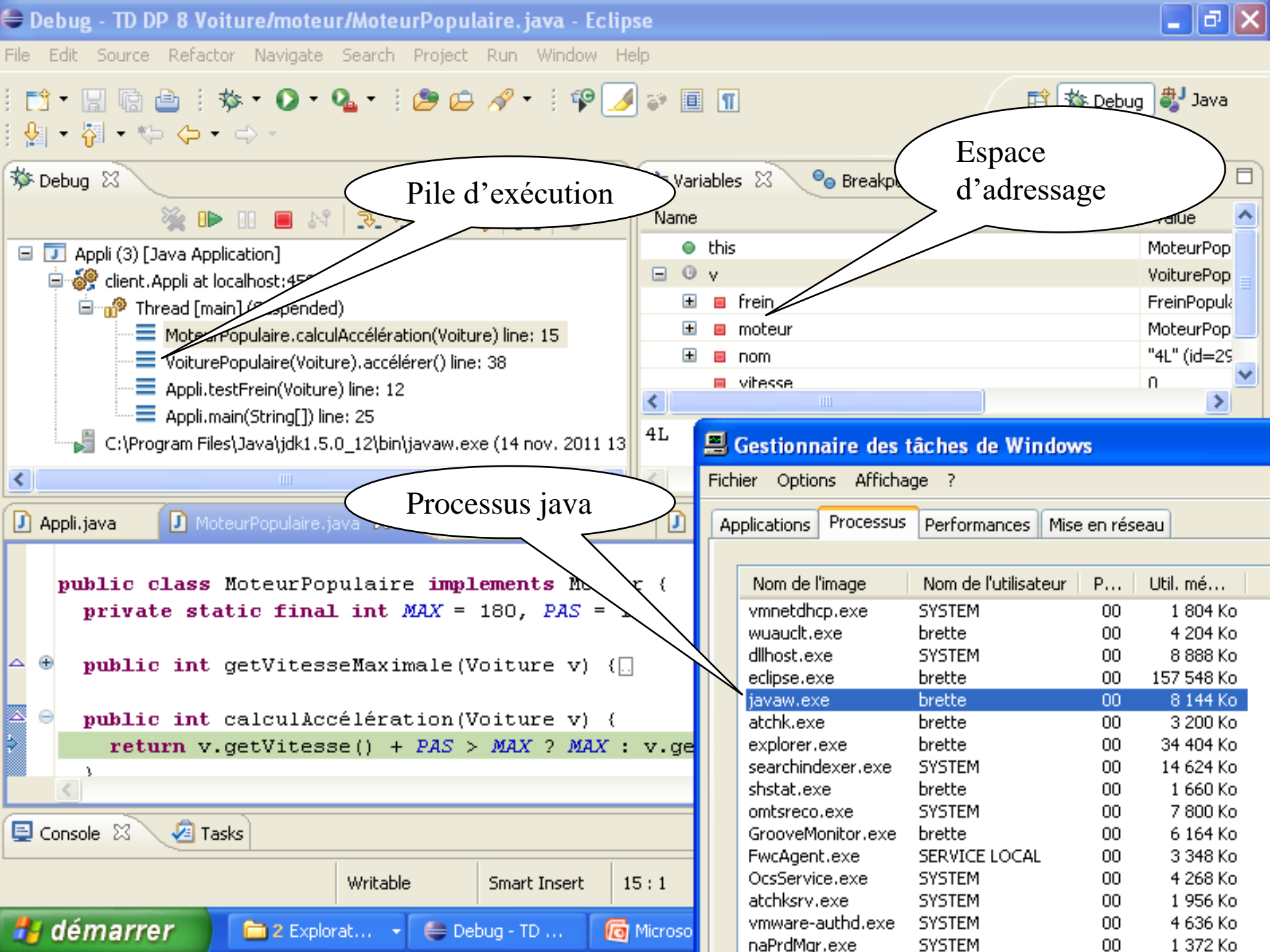
Processus légers (threads)

Thread - Runnable

Programme

(aspects statiques et dynamiques)





Processus vs thread

Processus (“lourd”) pile d’exécution propre

espace d’adressage propre

Processus léger (thread) créé au sein d’un processus lourd

pile d’exécution + esp. d’adressage propre

+ espace d’adressage partagé avec d’autres processus légers

Exécution en **parallèle** de plusieurs tâches (non bloquantes)

machine monoprocesseur => le scheduler

distribue le temps processeur (règle du tourniquet)

(ex.: anim.graph. + saisie texte + téléchargement d’une image)

Partage **concurrentiel** de données (ressources) entre plusieurs tâches

conflits, sections critiques, etc

Exécuter une activité (Runnable) dans un Thread

Interface `java.lang.Runnable`, classe `java.lang.Thread`

```
public interface Runnable {  
    // runnable signifie exécutable dans un thread propre  
    public abstract void run ( );  
}
```

```
public class Thread implements Runnable {  
    // un thread = un processus léger  
    .....  
    public void run ( ) {.....};  
}
```

Lancer un thread, le protocole `start()` : `unThread.start()`;

Lancer un exécutable (1)

sous-classer Thread

```
public class Activité extends Thread {  
    /* Activité est un Runnable et est un Thread par héritage */  
    public Activité (...) { ... this.start ( );} // auto lancement  
    .....  
    public void run ( ) { // description de l'activité.....};  
}
```

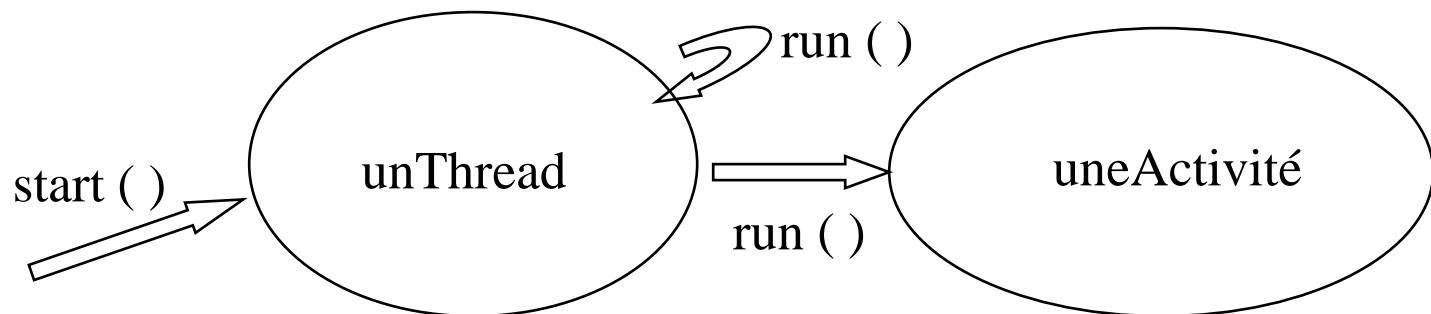
Code appelant : **new Activité();**



Lancer un exécutable (2)

implémenter Runnable

```
public class Activité implements Runnable {  
    /* Activité est un Runnable qui doit être géré par un Thread pour  
    être exécuté et lié au scheduler */  
    public Activité ( ) {(new Thread (this)).start ( ); }  
    .....  
    public void run ( ) { // description de l'activité..... };  
}
```



Sous-classer Thread ou implémenter Runnable?

Question : veut-on définir une activité parallèle à d'autres (y compris éventuellement avec partage de données)

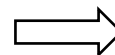
ou veut-on gérer un sous-type de processus léger (l'arrêter explicitement, le relancer, modifier sa priorité,...) ?



Programmeur d'application vs programmeur système



Définit des activités parallèles
Gère des données partagées



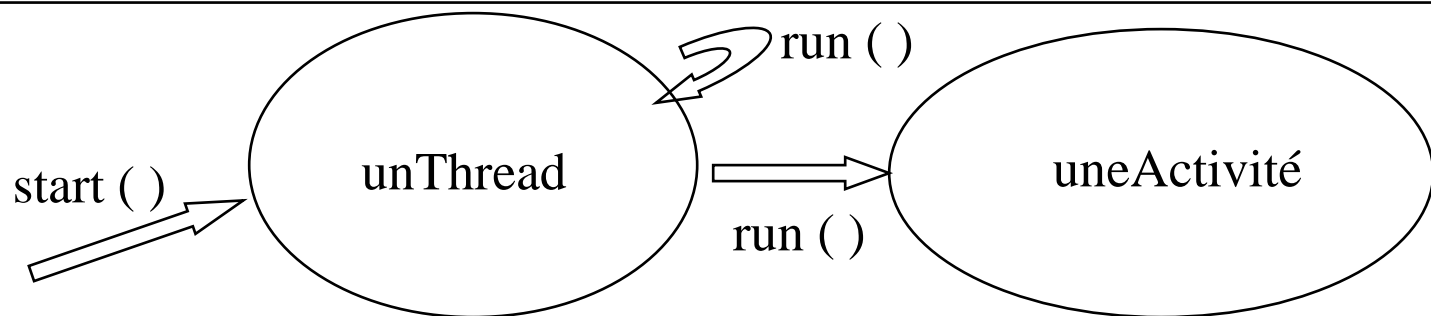
implémenter Runnable

Lancer un exécutable (3)

extraire le thread du Runnable

```
public class Activité implements Runnable {  
    /* Activité est un Runnable (explicitement qui doit être lié à un  
    Thread pour avoir sa propre pile d'exécution */  
    public Activité (...) {...} // pas d'auto-lancement  
    .....  
    public void run ( ) { // description de l'activité..... };  
}
```

Code appelant : **new Thread(new Activité()).start();**



Classe java.util.Thread

```
public class Thread implements Runnable {  
    private Runnable target;  
        public Thread( Runnable r) {this.target = r;}  
        public void start() {.....}  
        public void run ( ) { if (target != null) target.run() ;};  
}
```

*uneVoiturePopu.calculAccélération()
uneVoiturePopu.accelerer()
uneActivite.run()
unThread.run()*

pile d'exécution du thread

*Le fond de pile ne contient
pas un main()*

Attention à l'héritage !

```
public class Activité implements Runnable {  
    public Activité() {  
        new Thread(this).start();  
    }  
    public void run() { ... }  
}
```

Si nous dérivons cette classe :

```
public class ActivitéClient extends Activité {  
    private Client client;  
    public ActivitéClient (Client c) {  
        super();  
        this.client = c;  
    }  
    public void run() { this.client.getNom(); }  
}
```

le thread a démarré
alors que ***this.client***
n'est pas initialisée

risque de
NullPointerException

Lancer un thread

Ne jamais permettre de lancer un thread non initialisé

Solution 1 : interdire l'héritage

```
public final class Activité implements Runnable {  
    .....  
}
```

La classe Activité
n'est pas sous-
classable

Solution 2 : séparer constructeur et lancement

```
public class Activité implements Runnable {  
    public Activité (...) {...}  
}  
public void lancer() {new Thread(this).start();}  
}  
new Activité (...).lancer();
```

Le lancement doit
être fait
explicitement dans
le code appelant

Solution 3 : gérer le thread hors Runnable

```
new Thread(new Activité(...)).start();
```

Notion de « thread courant »

Thread courant = le thread en train de s'exécuter

Accès au thread courant : méthode statique *currentThread()* de Thread

```
public static Thread currentThread() {...}
```

Méthodes de la classe Thread concernant le thread courant :

public static void **dumpStack()** → recopie la pile d'exécution du thread courant

public static boolean **interrupted()** → test si le thread courant a été 'interrompu'

public static void **sleep**(long ms) throws InterruptedException

→ pause de *ms* millisecondes dans l'exécution du thread courant

public static void **yield()**

→ suspend l'exécution du thread courant

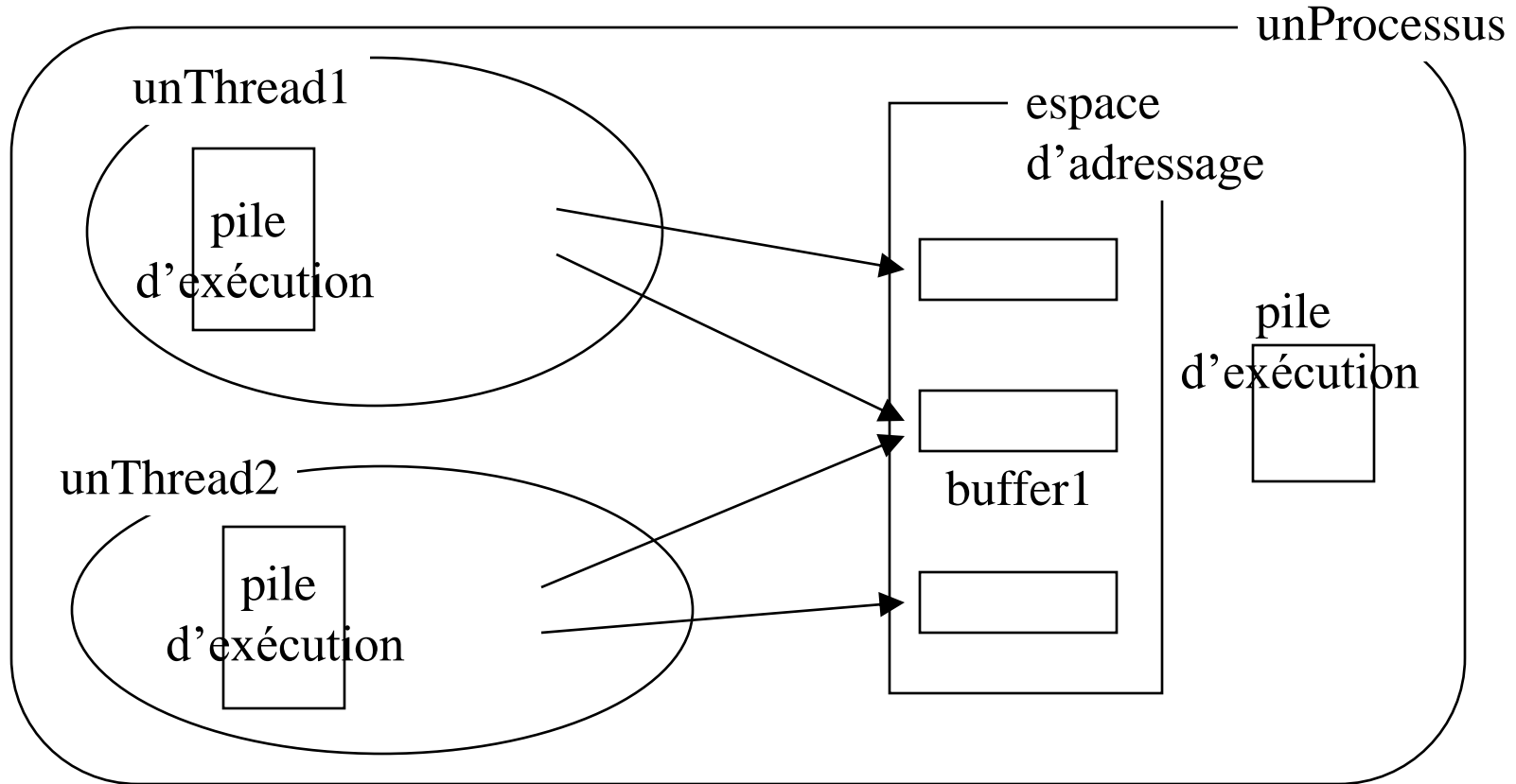
2. Concurrency - Sureté

Sureté – Thread safety

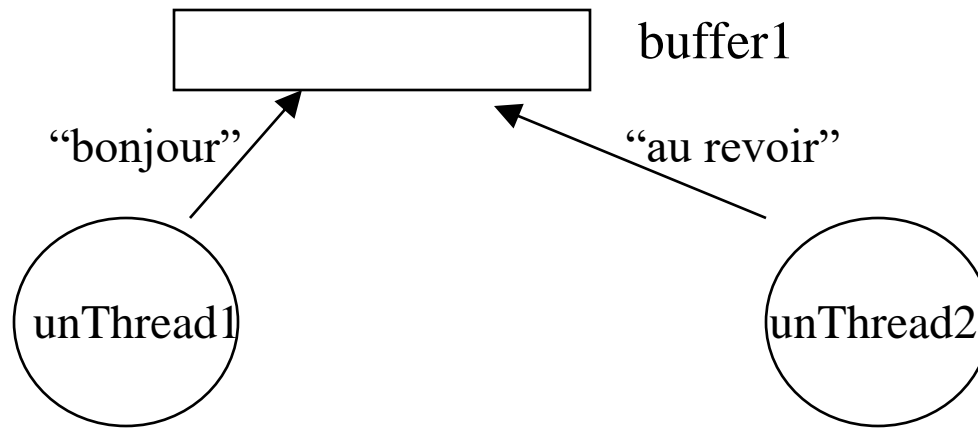
Moniteur de Hoare

Sections critiques

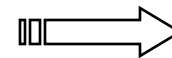
Accès concurrentiel



Accès concurrentiel (2)

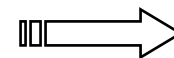


bonjour au revoir



section critiques + verrou
(threads asynchrones)

bonjour au revoir
au revoir bonjour



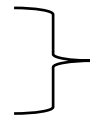
logique d'ordonnancement
des threads
(threads synchrones)

Sureté : thread safety

Thread safety : un objet peut-il supporté d'être adressé en parallèle par plusieurs threads ?

Exemple : java.util.ArrayList n'est pas thread-safe

```
private int size;  
private E[] elementData;  
public void add(E element) {  
    elementdata [size] = element ;  
    size++;  
}
```



section critique non verrouillée

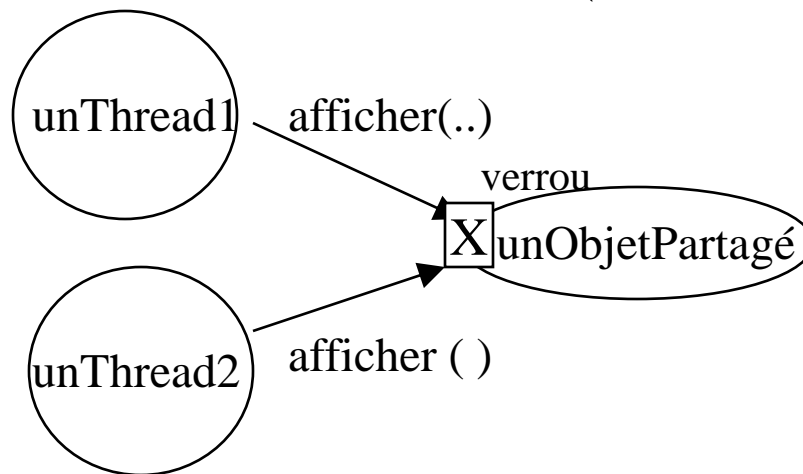
→ java.util.Vector

Section critique + verrou

Eviter “bonaujo revuoir” \Rightarrow *synchronized*

Séquentialiser l'accès aux ressources partagées lorsqu'il y a un risque.
Tout objet Java possède un verrou (moniteur de Hoare) et son accès peut donc être verrouillé.

synchronized \Rightarrow bloc de code où le parallélisme est interdit
(= section critique)



```
class ObjetPartagé {  
    synchronized void afficher (..) {....}  
    synchronized void message2 {....}  
    .....  
}
```

Exemple : une Fifo partagée

(cf le code complet sur le serveur commun)

Une file (Fifo)

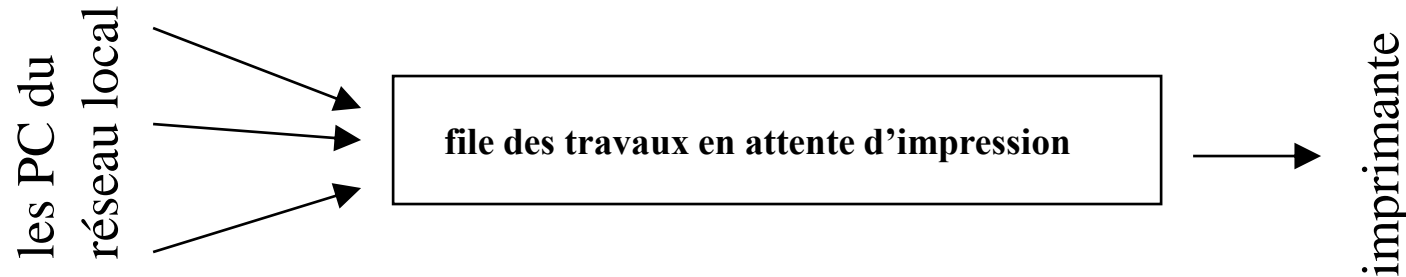
Des producteurs qui écrivent (*push*)

Des consommateurs qui lisent (*pop*)



Exemples de fifos partagées

Une fifo n-producteur 1-consommateur (spool d'imprimante)



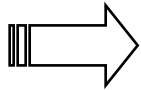
Une fifo n-producteur n-consommateurs (hot line)



Séquentialiser les méthodes d'accès *pop* et *push*

```
public class Fifo <E> {.....  
    public synchronized void push (E anObject) {...}  
    public synchronized E pop ( ) {...}  
    .....  
}
```

- sécurité garantie (thread safety)
- pas très souple ni très efficace (toutes les sections critiques sont verrouillées quand on passe dans une)



- attention aux performances :

- * faut-il verrouiller isEmpty() ?

- * réduire la taille des blocs synchronisés

```
public void pop (E anObject) {... // section non-section critique  
    synchronized (this) { // section critique.....  
    }  
    ..... // section non-section critique  
} // fin méthode pop
```

Verrou et thread courant

```
public class Activité implements Runnable {  
    private RessourcePartagée maRessource;
```

```
    public void run() {  
        ..... maRessource.action();  
    }  
}
```

```
public class RessourcePartagée {  
    public void action() {  
        synchronized(this) {  
            .....  
        }  
    }  
}
```

*maRessource.action()
uneActivite.run()
unThread.run()*

*Ici, le thread courant
prend le verrou de la
ressource partagée*

Verrou et thread courant : la fifo

```
public class Producteur implements Runnable {  
    private Fifo<Client> maFile;  
    public void run() {  
        this.maFile.push(...);  
    }  
}
```

```
public class File<E> {  
    public void push(E objet) {  
        synchronized(this) {  
            .....}  
        }  
    }  
}
```

```
..... main.....  
File<Client> uneFile = new File<Client>();  
new Producteur(uneFile).lancer(); //pile d'exec1  
new Producteur(uneFile).lancer(); //pile d'exec2
```

*maFile.push(...)
unProducteur1.run()
unThread1.run()*

pile d'exec 1

*maFile.push(...)
unProducteur2.run()
unThread2.run()*

pile d'exec 2

*Deux threads **concurrents**
pour l'exécution de push pour
uneFifo.*

*Le premier thread qui rentre
dans le bloc synchronized
verrouille uneFifo*

Verrou et moniteur

```
public class Producteur implements Runnable {  
    private Fifo<Client> maFile;  
    public void run() {  
        this.maFile.push(...);  
    }  
}
```

```
public class File<E> {  
    public void push(E objet) {  
        synchronized(this) {  
            .....  
        }  
    }  
}
```

```
..... main.....  
File<Client> uneFile = new File<Client>();  
new Producteur(uneFile).lancer();  
new Producteur(uneFile).lancer();
```

maFile.push(...)
unProducteur1.run()
unThread1.run()

pile d'exec 1

maFile.push(...)
unProducteur2.run()
unThread2.run()

pile d'exec 2

unThread1 vient de rentrer dans le bloc : le moniteur de Hoare de maFile garde la référence de unThread1 qui désormais possède le verrou jusqu'à la sortie du bloc sync : aucun autre thread ne peut exécuter ce bloc

3. Communication TCP par sockets

Communication entre machines

Sockets

Client/serveur : Serveur-Service

Communication entre machines

Bas niveau : sockets

Client/serveur : JDBC, servlets, JSP

Informatique distribuée:RMI, CORBA, EJB

Rappels réseau

Couche la plus basse : IP Internet Protocol

Couche transport : TCP, UDP

Ports d'E/S

Sockets et client/serveur

Couche transport : TCP, UDP

Définit le mode de transmission des paquets de données
2 options : Très fiable ou très rapide

TCP : Transmission Control Protocol

Assure la réexpédition des données égarées

Le récepteur émet un accusé de réception pour chaque paquet

Transmission de fichiers dont la complétude doit être garantie

UDP : User Datagram Protocol

Pas d'AR, pas d'assurance de bonne transmission

Transmission de fichiers volumineux pour lesquels un pourcentage de perte d'informations est envisageable (images, vidéos,...)

Sockets

Socket : canal logique de communication entre 2 machines

Avantages : les aspects *Système/Réseau* ne sont pas à gérer
(décomposition du message en paquets, en tête de paquet,
réexpédition en cas de pertes pour TCP, ...)

Client/serveur : une machine attend (serveur)
une machine se connecte (client)

Deux aspects :

- **le serveur attend** les demandes sur un port défini
- un client se manifeste, une socket est créée en mode point à point et **le dialogue se fait via cette socket**

Connexion - Echanges

1. Etablir la connexion : handshake

new Socket ← serverSocket.accept()

création d'un flux full-duplex

(un buffer *in*, un buffer *out* de chaque côté)

2. Protocole d'échanges envisageables :

- requête-réponse (ping-pong) half-duplex
- full duplex : 2 threads côté client

Dans ce cours, on se limite à du ping-pong (similaire à HTTP pour le web)

Applications concurrentes

Plusieurs clients peuvent être en liaison avec le serveur en même temps et sur le même port d'E/S sans risque de confusion

Multithreading coté serveur :

n clients \Rightarrow **n runnables communicants +
1 runnable à l'écoute de nouveaux clients**

Opérations élémentaires liées aux sockets :

- s'attacher à un port d'E/S
- attendre les demandes de connexion émanant des clients
- accepter les connexions sur le port défini
- se connecter à un serveur
- envoyer, recevoir des données
- clôre une connexion

java.net

Gestion des URL :

URL, URLConnection, URLEncoder, MalformedURLException

Datagrammes UDP :

DatagramPacket, DatagramSocket, MulticastSocket, ...

Socket à comportement non standard :

SocketImpl, SocketImplFactory, ...

Gestion des sockets :

InetAddress (adresse IP), ContentHandler (gestionnaire de contenu)

Socket (pour les sockets) et ServerSocket (pour les serveurs)

La classe Socket

Une instance de la classe Socket permet de :

- se connecter à un serveur
- envoyer, recevoir des données
- clôre une connexion

Se connecter : instancier Socket

Préciser l'adresse IP et le port d'E/S

new Socket(adresseIp, port);

Socket maSocket = new Socket ("192.93.28.9", 1234);

La classe Socket (2)

Envoyer/recevoir des données

Ecrire (resp. lire) dans le outputStream (resp. inputStream)

BufferedReader socketIn =

 new BufferedReader (new InputStreamReader(**maSocket.getInputStream** ());

PrintWriter socketOut = new PrintWriter (**maSocket.getOutputStream** (), true);

String line = socketIn .readLine();

socketOut.println (“OK”);

Clore une connexion **maSocket.close** ();

La classe ServerSocket

Une instance de la classe ServerSocket permet de :

- s'attacher à un port d'E/S

- attendre les demandes de connexion émanant des clients

- accepter les connexions sur le port défini

S'attacher à un port d'E/S

Lors de l'instantiation

```
ServerSocket monServeur = new ServerSocket (1234);
```

Attendre des demandes de connexion et les accepter

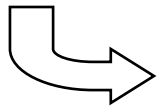
La méthode accept()

```
Socket socketCotéServeur = monServeur.accept ( );
```

Ecriture d'un serveur (1)

main() de l'application serveur :

```
int numPort = 1234;  
ServerSocket monServeur = new ServerSocket (numPort);  
while (true) {  
    Socket socketCotéServeur = monServeur.accept ( );  
    ..... dialogue avec le client .....  
    socketCotéServeur.close ( );  
    }  
}
```



un seul client à la fois !!!!

Ecriture d'un serveur (2)

**Il faut un thread de serveur (arrivée de nouveaux clients)
et un thread de service pour chaque client**

new Thread(new ServeurBrette(1234)).start();

run () de ServeurBrette :

```
ServerSocket monServeur = new ServerSocket (this.port);  
while (true) {  
    Socket socketCotéServeur = monServeur.accept ( );  
    new Thread(  
        new ServiceBrette (socketCotéServeur, ...autres infos...)).start();  
}
```

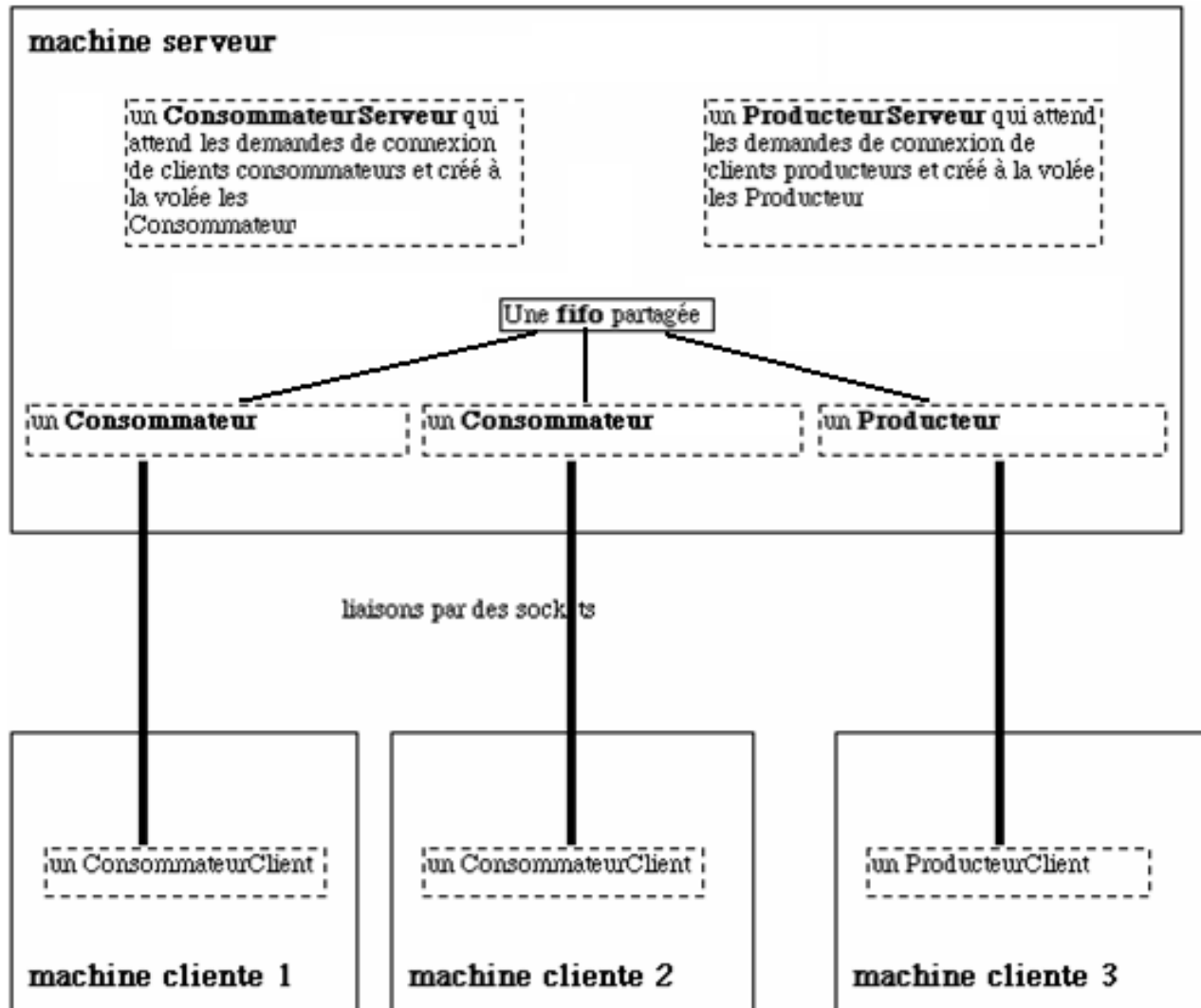
***On pourra utiliser la technique interrupt()-InterruptedException ou
close()-IOException pour fermer le serveur***

Ecriture d'un serveur (2)

avec une classe de service implémentant Runnable :

```
class ServiceBrette implements Runnable {  
    private Socket socket;  
    .... constructeur classique ....  
    public void run ( ) {  
        BufferedReader sIn = ...  
        PrintWriter sOut = ...  
        ..... échanges avec le client .....  
        ..... algorithmique du service .....  
        this.socket.close();  
    }  
}
```

Exemple : un serveur de Fifo



Un serveur de fifo (*main*)

//création des ressources partagées

Fifo fifo = new Fifo <Data> (...);

// transmission des ressources partagées aux services

Producteur.setFifo(fifo);

Consommateur.setFifo(fifo);

// lancement des serveurs

new Thread(new ServeurProd(portProd)).start();

new Thread(new ServeurConso(portConso)).start();

Un serveur de fifo (service)

```
class Consommateur implements Runnable {  
    private static Fifo laFifo; // setter static pour l'initialisation  
    private Socket socket;  
    Consommateur(Socket s) {  
        this.socket = s;  
    }  
    public void run ( ) { // il faudrait ajouter les try/catch  
        PrintWriter socketOut = .....  
        while (true)  
            socketOut.println (laFifo.pop ( ));  
    }  
    // finalize() est invoquée par le garbage collector avant destruction de l'objet  
    // deprecated since 1.9 mais la solution alternative est complexe  
    // voir Cleaner et Phantomreference  
    protected void finalize() throws Throwable {  
        socket.close( );}  
}
```

Protocole de communication

Phase 1 : établir la communication (handshake)

new Socket(..) **code client** <--> monServeur.accept() **run() du serveur**

Phase 2 : communiquer via les buffers

suite du **code client** <--> **run() du service**

établir un *protocole* d'échange synchrone (type requête - réponse)

- ordre des échanges (qui envoie la première requête)
- sémantique des échanges (

Phase 3 : fermer la communication

close() **code client** <--> close() **run() du service**

(le close() du service est essentiel mais ne dispense pas du finalize())

4. Les classes comme objets

java.lang.Class<T>

La classe `java.lang.Class`

Pendant l'exécution, chaque classe est représentée par une instance de `java.lang.Class`

- Cette instance est unique pour une classe donnée
- Elle est créée automatiquement lors du chargement-initialisation de la classe

• 3 façons d'obtenir cet objet : `Class<?> classe =`

La méthode `getClass()`

La syntaxe `.class`

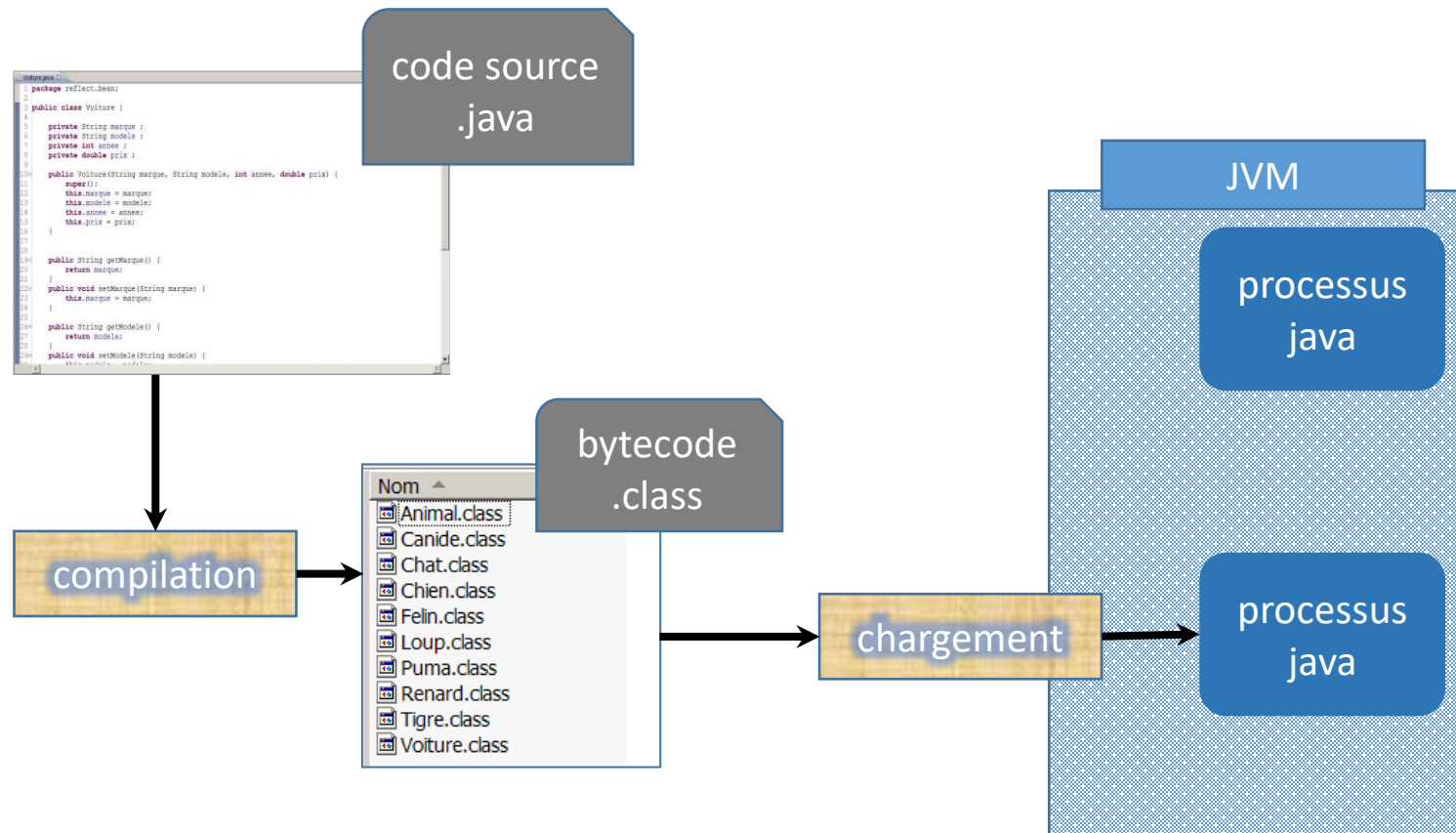
La méthode `Class.forName(String)`

`(new ArrayList()).getClass()`

`java.util.ArrayList.class`

`Class.forName(«java.util.ArrayList»)`

Cycle de vie des classes



Chargement d'une classe (`java.lang.ClassLoader`)

Chaque classe est chargée et initialisée UNE seule fois

Statique : toutes les classes nécessaires au processus sont chargées au lancement du processus (C++)

Dynamique en Java : lors de sa première apparition dans l'exécution du processus

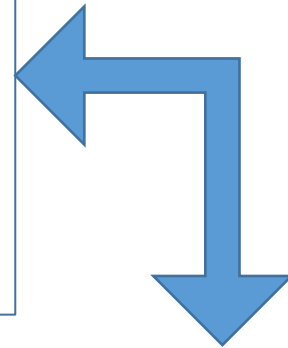
- implicite : `Client c = new Client(...);`
- explicite : `Class.forName(aString);` (`ClassNotFoundException`)

Initialisation d'une classe

```
public class Texte {  
    private final static String texte = "hello";  
    public static String affiche() {  
        return texte;  
    }  
}
```

bloc static

```
public class Texte {  
    private final static String texte;  
    static {  
        texte = "hello";  
    }  
    public static String affiche() {  
        return texte;  
    }  
}
```



Initialisation de constantes par lecture fichier

```
public class Data {  
    private final static String nomFichier ;  
    public static int SOLDE_MAX ;  
  
    static {  
        nomFichier = "c:\\file\\fichier.txt";  
        try {  
            FileInputStream f = new FileInputStream(nomFichier);  
            SOLDE_MAX = f.read(); // IOException  
        } catch (IOException e) {  
            SOLDE_MAX = 0;  
            // e.printStackTrace(); ???  
        }  
    }  
}
```

Singleton thread-safe

```
public class Singleton {
    static { // le bloc static est thread-safe
        instance = new Singleton(.....);
    }
    private static Singleton instance;
    private Singleton(.....) {
        // initialisation des variables d'instance si nécessaire.....
    }
    public static Singleton getInstance() {
        // le test <if (instance==null)> est inutile et n'est pas thread-safe
        // à moins d'un synchronized(Singleton.class) --> baisse permanente de
        performance
        return instance;
    }
    // méthodes d'instances.....
    .....
}
```

RTTI : Run-Time Type Identification

Java maintient l'Identification de Type au Run-Time
sur tous les objets

Permet de connaître le **type dynamique** d'une référence
Pas nécessairement le même que le **type statique**

Permet d'implémenter la liaison dynamique

Type statique	Type dynamique
<code>ObjetGraphique objGraph = new Cercle();</code>	
<code>objGraph.draw();</code>	

Compilation : type statique (référence déclarée)

Exécution : type dynamique (objet instancié)

Classe et type

- Un objet de la classe **Class** est paramétré par le type qu'elle représente :

```
Class<String> c1 = String.class;
```

- Polymorphisme : **Class<? extends TypeStatique>**

```
Class<? extends ObjetGraphique> classe = objGraph.getClass();
```

```
// class Enseignant extends Personne
```

```
Personne p = new Enseignant("JF Brette");
```

```
Class<? extends Personne> classe = p.getClass();
```

```
classe == Personne.class → false
```

```
classe == Enseignant.class → true
```

- Cas le plus général : **Class<?>** (classe, interface, type de base... même void (Void))

```
public class EncapsuleClasse {  
    // juste une classe qui encapsule un attribut Class<?>  
    private Class<?> laClass;  
    public EncapsuleClasse(Class<?> laClass) {  
        this.laClass = laClass;  
    }  
    @Override  
    public String toString() {  
        return "EncapsuleClasse " + this.laClass;  
    }  
}
```

Quelques exemples d'exécution

```
System.out.println(new EncapsuleClasse(String.class));  
System.out.println(new EncapsuleClasse(int.class));  
System.out.println(new EncapsuleClasse(Runnable.class));  
System.out.println(new EncapsuleClasse(void.class));
```

Résultats affichés

```
EncapsuleClasse class java.lang.String  
EncapsuleClasse int  
EncapsuleClasse interface java.lang.Runnable  
EncapsuleClasse void
```

Instancier sans new : méthode newInstance() de Class

```
public T newInstance()  
    throws InstantiationException, IllegalAccessException
```

IllegalAccessException la classe ou son constructeur vide n'est pas accessible

InstantiationException – la classe n'est pas instanciable

ExceptionInInitializerError – l'initialisation a échoué

SecurityException – liée au SecurityManager

```
Class<Personne> cl = Personne.class;  
Personne p = cl.newInstance();
```

```
Class< ? extends Personne> cl = Enseignant.class ;  
Personne [] personnes = new Personne[10] ;  
personnes [0] = cl.newInstance () ;
```

```

public class EncapsuleRunnable {
    // une classe qui encapsule un attribut d'une classe qui implémente Runnable
    public EncapsuleRunnable(Class<? extends Runnable> laClass) {
        this.laClassRunnable = laClass;
    }

    private Class<? extends Runnable> laClassRunnable;

    public void lancer() throws InstantiationException, IllegalAccessException {
        new Thread(this.laClassRunnable.newInstance()).start();
    }
    @Override
    public String toString() {
        return "Test [laClass=" + laClassRunnable + "]";
    }
}

```

Quelques exemples d'exécution

```

//new EncapsuleRunnable(String.class); -> erreur de compilation
System.out.println(new EncapsuleRunnable(objetRunnable.Producteur.class));
System.out.println(new EncapsuleRunnable(objetRunnable.Activite.class));
new EncapsuleRunnable(objetRunnable.Producteur.class).lancer();

```

Résultats affichés

```

Test [laClass=class objetRunnable.Producteur]
Test [laClass=interface objetRunnable.Activite]
Dans le run() du producteur

```

Un serveur découplé du service

Le code de la classe serveur du td « serveur inversion » est lié au nom de la classe de service :

```
class Serveur implements Runnable {  
.....  
public void run() {  
    try {  
        while(true)  
            new ServiceInversion(listen_socket.accept()).lancer();  
    }  
    .....  
}
```

Changement de service → code du serveur à modifier !

Solution : ajouter un paramètre au serveur, la classe du service

```
new Serveur(serveurinverse.ServiceInversion.class, DEFAULT_PORT).lancer();
```

(voir la suite en tp)

Une factory plus évolutive

```
import java.util.Collection; // couplage à l'interface - normal

import java.util.ArrayList; // couplage aux implémentations possibles
import java.util.Vector;    // → pas évolutif et lié à creeCollection()

public class FabriqueCollection {

    public static Collection creeCollection(String typeDeLObjet) throws Exception {
        switch (typeDeLObjet) {
            case "java.util.ArrayList" : return new ArrayList();
            case "java.util.Vector" : return new Vector() ;
            //etc
        }
        throw new Exception (typeDeLObjet+" non géré par la fabrique");
    } // fin creeCollection

    public static Collection creeCollectionParIntrospection(String typeDeLObjet)
        throws Exception {
        try {
            Class<?> classe = Class.forName(typeDeLObjet);
            // tester que cette classe implémente Collection
            return (Collection) classe.newInstance();
        } catch (Exception e) {
            throw new Exception (typeDeLObjet+" non géré par la fabrique");
        }
    } // fin creeCollectionParIntrospection
}

(test sur Eclipse)
```

5. Performance Vivacité

Attente en mode veille

wait()-notifyAll()

Vivacité et dead-locks (risques)

Performance : thread actif/en veille

- Contexte : un thread t1 attend un état particulier de la ressource
- Attente active : t1, en testant l'état de la ressource de façon répétée, empêche celle-ci d'être modifiée
- t1 consomme du temps processeur sans succès et ralenti/bloque les threads qui pourraient le satisfaire!

actif → veille

Lorsque le contexte (l'état de la ressource partagée) est défavorable, le thread qui possède le verrou le libère et est mis en veille (`wait()` envoyé à la ressource verrouillée). Le moniteur de la ressource garde un lien vers ce thread mis en veille.

Le scheduler ne lui donne plus de temps processeur, sa pile d'exécution est arrêtée.

```
class RessourcePartagée {
    synchronized ... methode1( ) {.....
        while (conditionNonRemplie( )) {
            // on endort le thread qui possède le verrou
            this.wait( );
        }
        // suite du travail au réveil
        .....
    }
}
```

Effet de *objet.wait()*

- Le thread *thread* qui possède le verrou de *objet* le lâche; *objet* n'est plus verrouillé
- *thread* est ajouté à la liste des threads en veille de cet *objet*
- *thread* est mis en veille (le scheduler, qui teste l'état actif d'un thread lors du « tourniquet », ne lui donnera plus de temps processeur)

veille → actif

Lorsque l'état de la ressource est modifié, on réveille (notifyAll() envoyé à la ressource) tous les threads en veille de la ressource et, lorsque le scheduler lui donnera la main, l'un d'entre eux pourra reprendre le verrou et poursuivre son travail en dépilant wait().

```
class RessourcePartagée {  
    .....  
    synchronized ... methode2 ( ) {.....  
        // on a changé l'état de l'objet  
        // on réveille le(s) thread(s) qui attendent le verrou  
        this.notifyAll( );  
    }
```

**Attention : wait() et notifyAll() sont envoyées à une ressource verrouillée
sinon → IllegalMonitorStateException**

Exemple 1 : réception requête, attente (active) de la réponse

```
public class Service implements Runnable {  
  
    private static Fifo fifo; // avec un set static  
  
    // Requete possède un attribut reponse avec getter et setter  
    private Requete laRequete;  
  
    public void run() {  
        ... création d'un objet Requête (réception socket de données...)  
        fifo.push(laRequete); // des threads traiteurs en aval de cette fifo  
        // attente active de la réponse  
        while (this.laRequete.getReponse() == null) ;  
        .... envoi socket de la réponse  
    }  
}
```

Attente (en veille) de la réponse

```
public class Service implements Runnable {  
    private static Fifo fifo; // avec un set static  
    // Requete possède un attribut reponse avec getter et setter  
    private Requete laRequete;  
    public void run() {  
        ... création d'un objet Requête (réception socket de données...)  
        fifo.push(laRequete); // des threads traiteurs en aval de cette fifo  
        // attente en veille de la réponse  
        synchronized(larequete) {  
            while (laRequete.getReponse() == null)  
                laRequete.wait();  
            // le thread qui traitera cette requête utilisera le setter de réponse et le  
            fera suivre d'un laRequete.notifyAll();  
        }  
        .... envoi socket de la réponse  
    }  
}
```


Exemple 2 : lecteur/écrivain

Partage de document

Contexte : plusieurs écrivains, plusieurs lecteurs d'un même document

Politique d'accès : soit 1 seul écrivain, soit plusieurs lecteurs

Demande de lecture : mise en attente si écrivain en cours

Demande d'écriture : mise en attente si écrivain ou lecteurs en cours

Fin de lecture ou d'écriture : réveil des lecteurs et écrivains en attente

Lecteurs et écrivains sont des Runnable partageant un Document

Document : code pour le lecteur

```
private boolean ecrivainEnCours;
private int lecteursEnCours;

public void lecture(...) throws InterruptedException{
    synchronized(this) { while (ecrivainEnCours) {wait();}
                        lecteursEnCours++
    }
    // lecture hors section critique
    // d'autres lecteurs peuvent rejoindre....
    .....
    synchronized(this) {lecteursEnCours--;
                        this.notifyAll();
    }
} // fin lecture
```

Document : code pour l'écrivain

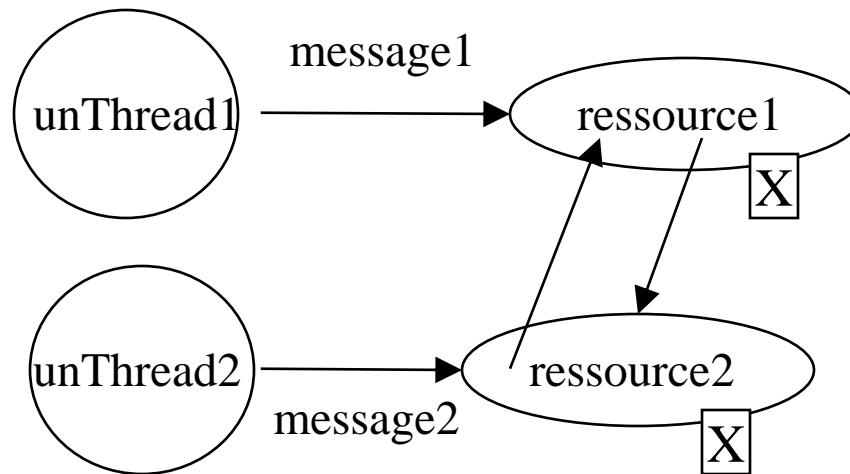
```
private boolean ecrivainEnCours;  
private int lecteursEnCours;
```

```
public void écriture (...) throws InterruptedException {  
    synchronized(this) { while (ecrivainEnCours || lecteursEnCours > 0)  
                           {wait(); }  
                           ecrivainEnCours = true ;  
    }  
    // écriture hors section critique  
    .....  
    synchronized(this) {ecrivainEnCours = false ;  
                        this.notifyAll();  
    }  
} // fin écriture
```

Vivacité : éviter les blocages

Verrou : risque de blocage (verrous mortels ou dead locks)

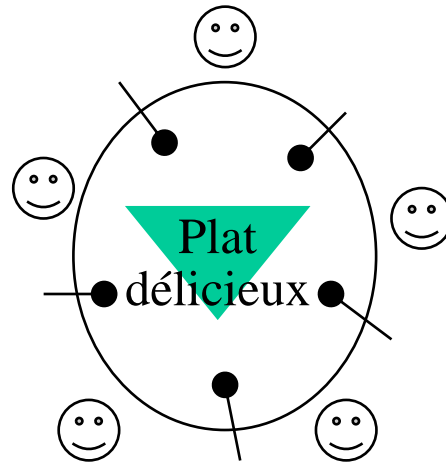
Cas le plus fréquent : interblocage à 2



Aucun moyen de sortir d'un blocage : risque majeur !

Blocage : diner des philosophes

5 philosophes + table ronde + 5 cuillères + plat délicieux au milieu
Pour manger : 2 cuillères (droite et gauche)



Chaque philosophe prend sa cuillère droite → interblocage

Blocage simple : ordre du verrouillage

```
public class LeftRightDeadlock {  
    private final Object left = new Object();  
    private final Object right = new Object();  
    public void leftRight() {  
        synchronized(left) {  
            synchronized(right) {  
                action();  
            }  
        } // fin leftRight  
    public void rightLeft() {  
        synchronized(right) {  
            synchronized(left) {  
                autreAction();  
            }  
        } // fin rightLeft  
    }  
    ... suite de la classe
```

**Ne prenez jamais des verrous
dans un ordre différent !**



Blocage plus complexe : ordre du verrouillage

```
public void transferMoney(Compte débité, Compte crédité, int montant)
    throws CompteInsuffisantException {
    synchronized (débité) {
        synchronized (crédité) {
            if (débité.crédit() < montant)
                throw new CompteInsuffisantException (débité, montant);
            débité.retrait(montant);
            crédité.dépot(montant);
        }
    }
}
```

transferMoney(compte1, compte2)



unThread1

transferMoney(compte2, compte1)

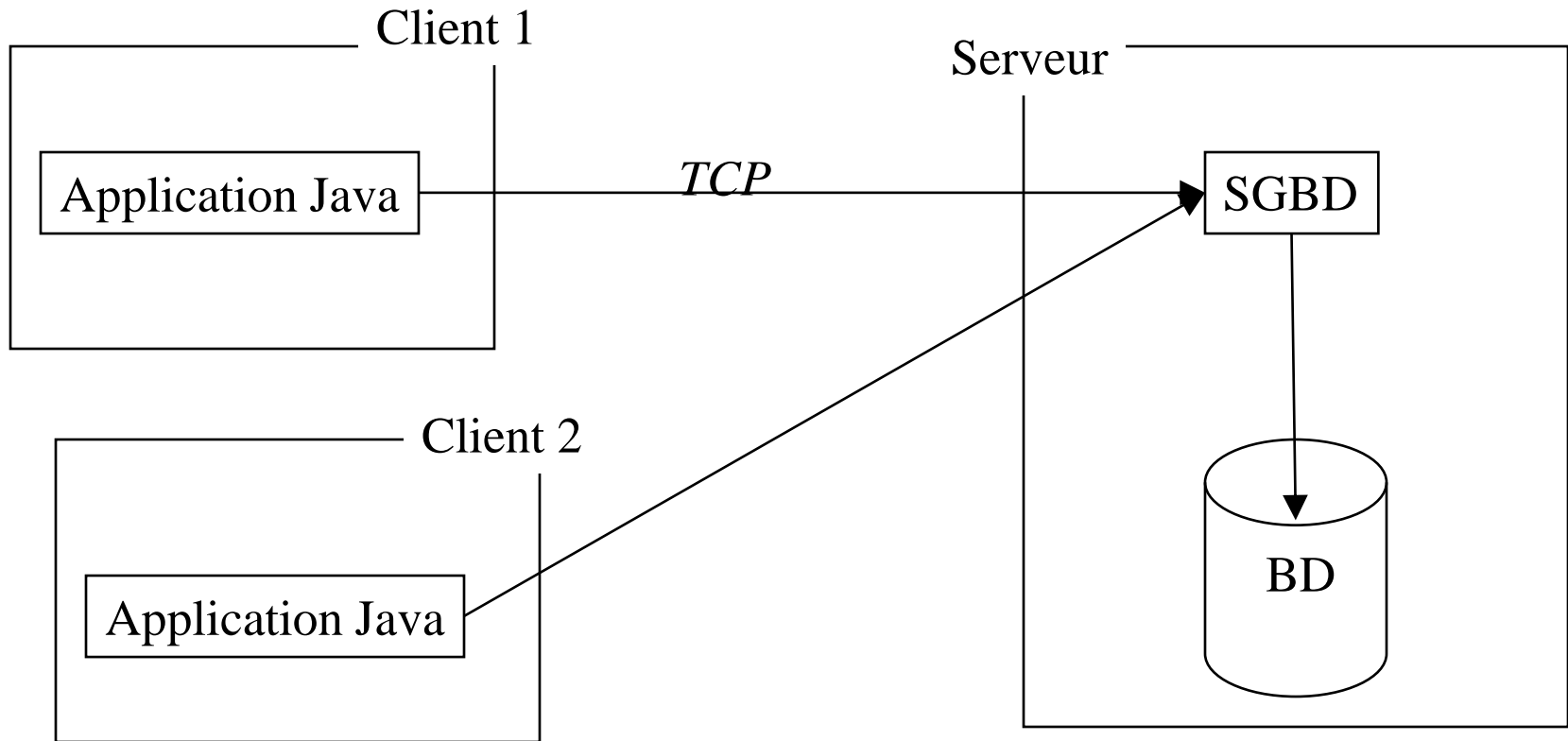


unThread2

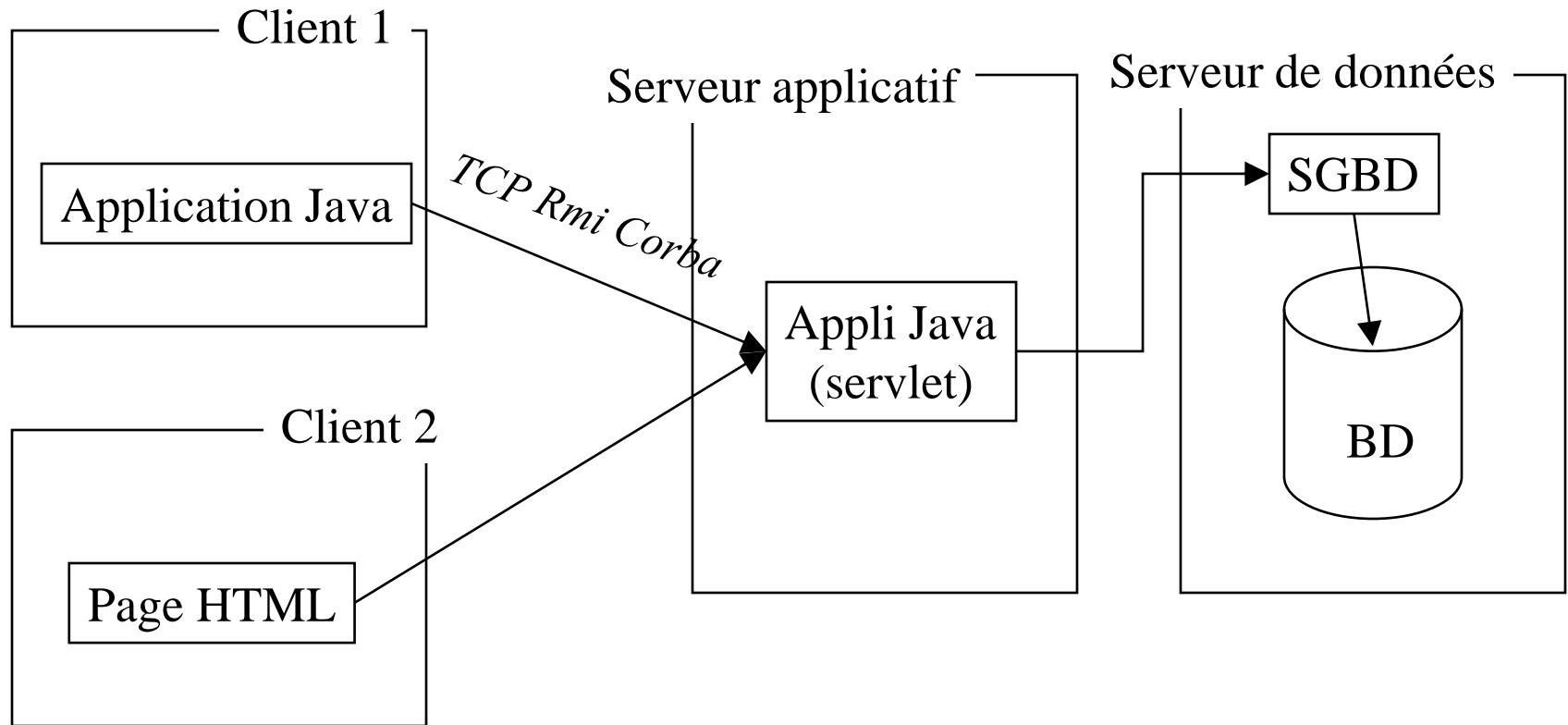
6. Accès aux données persistantes

Interrogation d'une BD relationnelle
JDBC

Architecture Client/serveur 2 couches (ou 2-tier)



Architecture Client/serveur 3 couches (ou 3-tier)



Comparaison 2-tier 3-tier

Avantage 2-tier :

plus simple à mettre en œuvre

pas besoin de serveur d'applications

Avantages 3-tier :

indépendance accrue client/données

migration des données, sécurité

**répartition du code entre client (couche graphique) et
application serveur (contrôles)**

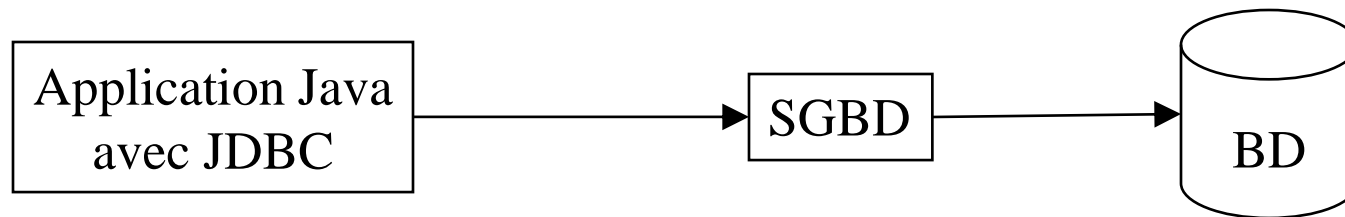
client plus léger, sécurité

échanges avec le client multi-technologies

sockets, RMI, CORBA

JDBC : Java Database Connectivity

**Un ensemble de protocoles d'interrogation d'une BD relationnelle
indépendant du SGBD**

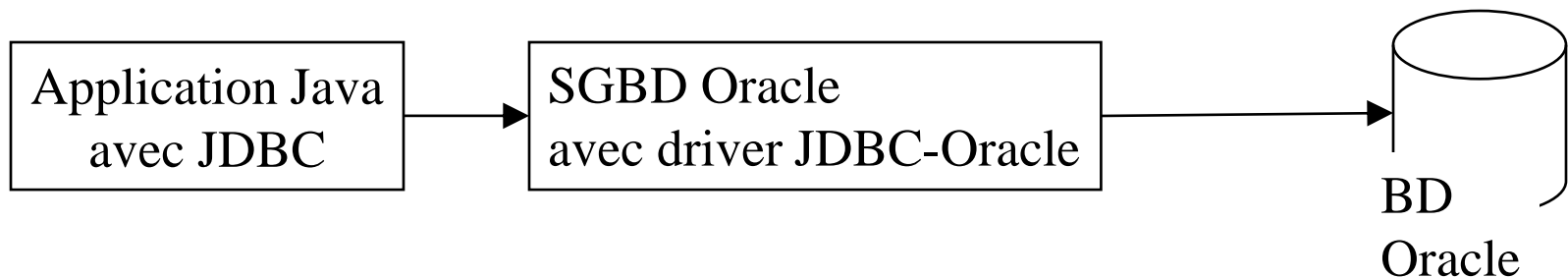


package java.sql : 8 interfaces, 6 classes, 3 classes d'exceptions

Drivers (pilotes) JDBC

Chaque SGBD utilise un pilote (driver) qui lui est propre et qui convertit les requêtes JDBC dans le langage natif du SGBD

Ces drivers (dits *drivers JDBC*) existent maintenant pour tous les principaux constructeurs (Oracle, Sybase, Informix, ...)



package java.sql (classes)

DriverManager (DriverInfo, DrivePropertyInfo)

gestionnaire de driver (et classes associées)

Date, Time, Timestamp

sous-classes de java.util.Date pour une adéquation avec les types SQL de mêmes noms

SQLException, SQLWarning, DataTruncation

erreurs de conversion de type

package java.sql (interfaces)

Statement (PreparedStatement, CallableStatement)

requête statique (requête dynamique, procédure stockée)

hiérarchie : Statement <-- PreparedStatement <-- CallableStatement

Connection

une connexion à la BD

ResultSet

résultat d'une requête

Driver

un driver JDBC

DatabaseMetaData, ResultSetMetaData

infos sur la structure (tables de la BD, colonnes, ...) de la BD ou du résultat

Mise en œuvre de JDBC

1. Initialisation du travail

Enregistrer auprès du DriverManager le driver JDBC

Etablir la connexion à la BD

2. Interrogation de la BD

Créer une zone de description de requête

Exécuter la requête

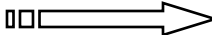
Traiter les données résultant de la requête

3. Terminaison du travail

Fermer les différents espaces ouverts par ce qui précède

Initialisation du travail

Enregistrer auprès du DriverManager le driver JDBC

Class.forName (nomDuDriver)  charge le driver *nomDuDriver*

Exemples : `Class.forName (“oracle.jdbc.OracleDriver”)`

quand une classe de driver est chargée, elle crée une instance d’elle-même et s’enregistre auprès du DriverManager (dans son bloc static)

Remarque : dans les JRE récents, toutes les classes et bibliothèques associées à un lancement de programme sont chargées automatiquement, donc `Class.forName` n’est pas obligatoire.
Il le reste pour charger dynamiquement un driver non prévu au départ.

Initialisation du travail (2)

Etablir la connexion à la BD

Connection connect = DriverManager.getConnection (url, user, password)

⇒ prend le premier driver (parmi ceux enregistrés) qui permette de se connecter à la base décrite par les paramètres
renvoie une Connection

url :

jdbc:<sous-protocole>:<nomBD>;<param=valeur>....

Exemple pour une connexion via odbc avec le driver JdbcOdbc :

jdbc:odbc:nom_base_donnée

avec le nom déclaré dans le panneau de configuration odbc de la machine locale

Interrogation de la BD

Créer une zone de description de requête : les interfaces Statement, PreparedStatement et CallableStatement

```
Statement req1 = connect.createStatement ( ) ;
```

```
aString = "SELECT .... requête avec arguments dynamiques ...."
```

```
PreparedStatement req2 = connect.prepareStatement ( aString) ;
```

```
aString = ".... nom d'une procédure stockée sur le SGBD...."
```

```
CallableStatement req3 = connect. prepareCall ( aString) ;
```

Interrogation de la BD (2)

Exécuter la requête

public ResultSet executeQuery (...) pour les requêtes (SELECT) qui retournent une table (tuples résultants)

public int executeUpdate (..) pour les requêtes (INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE) qui font une maj et retourne un entier (nombre de tuples traités)

public boolean execute () pour les procédures stockées (en général)

```
ResultSet res = req1.executeQuery
```

```
    (“SELECT nom, prenom FROM clients “ +
```

```
        “WHERE nom = ‘Brette’ ORDER BY prenom”);
```

```
int nb = req1.executeUpdate (“INSERT into dept(DEPT) VALUES (06)”);
```

Interrogation de la BD (3)

Traiter les données résultant de la requête (accès aux tuples)

TYPE_FORWARD_ONLY (par défaut)

parcours séquentiel tuple par tuple par le biais d'un curseur

public boolean next () fait avancer le curseur d'un tuple, renvoie false si le curseur est allé au-delà du dernier tuple

Il faut faire un premier `res.next()` pour être sur le premier tuple

while (res.next ()) { Traitement d'un tuple }

TYPE_SCROLL dans la Statement ou PreparedStatement

retour arrière possible **previous ()**

déplacement absolu **absolute (int row), first (), last ()**

déplacement relatif **relative (int row)**

Interrogation de la BD (4)

Traiter les données résultant de la requête (accès aux colonnes)

Les colonnes sont accessibles par leur numéro ou leur nom

Cet accès est séquentiel (sauf avec TYPE_SCROLL)

L'accès aux valeurs se fait par des méthodes getXxx (...)

où Xxx est le type de données

getInt, getBytes, getBoolean, getDate,

String unPrenom = res.getString (2);

String unPrenom = res.getString ("prenom");

res.isNull () permet de tester si la dernière colonne lue (avec getXxx) a une valeur nulle

Correspondance des types

CHAR	String
NUMERIC DECIMAL	java.math.BigDecimal
BINARY	byte []
BIT	boolean
INTEGER	int
BIGINT	long
REAL	float
DOUBLE, FLOAT	double
DATE	java.sql.Date
TIME	java.sql.Time
.....

Terminaison du travail

Fermer les différents espaces ouverts par ce qui précède
(sinon, le garbage collector s'en occupe mais c'est moins efficace)

```
connect.close ( );  
statement.close ( );  
res.close ( );
```


Accès aux méta données du ResultSet

ResultSetMetaData

informations sur les types de données du ResultSet

--> *getMetaData () de ResultSet*

(nombre de colonnes, nom d'une colonne, largeur d'affichage d'une colonne, nom de la table,...)

```
ResultSet res = req1.executeQuery ("SELECT * FROM client");
```

```
ResultSetMetaData rsmd = rs.getMetaData ( );
```

```
// affichage de tous les noms des colonnes
```

```
int nbCol = rsmd.getColumnCount ( );
```

```
for (int i=1;i<=nbCol;i++)
```

```
    System.out.println(rsmd.getColumnName(i));
```

Requêtes précompilées avec arguments

PreparedStatement (extends Statement)

argument dynamiques --> ?

On donne dynamiquement les arguments par les méthodes

setXxx(numeroArg, valeurArg)

```
PreparedStatement ps = connect.prepareStatement  
("UPDATE emp SET salary = ? WHERE name = ?");
```

```
for (int i= 0; i<nbUpdates; i++)  
{ps.setFloat (1, updates[i].getSalary ());  
ps.setString (2, updates[i].getName ());  
ps.executeUpdate ( );  
}
```

Validation avec Commit

Par défaut mode auto-commit : un commit est effectué après chaque ordre SQL

Passage en mode manuel : `connect.setAutoCommit (false);`

Commit manuel : `connect.commit ();`

Annulation en cours de transaction : `connect.rollback ();`