

# Systèmes concurrents & intergiciels

## Projet

Philippe Quéinnec

ENSEEIH  
Département Sciences du Numérique

15 novembre 2021

# Plan

- 1 Linda
- 2 Réalisations
- 3 Modalités

# Objectif

Réaliser un service Linda (espace partagé de tuples) :

- En mémoire partagée
- En client - serveur (mono-serveur, plusieurs clients concurrents)

Sujet, présentation et code fourni sur moodle.

## Linda (TSpaces)

- Espace partagé contenant des tuples
- Un tuple = un n-uplet de valeurs
- Opérations de dépôt, retrait, consultation
- Retrait/consultation d'après un motif
- Retrait/consultation bloquantes
- Abonnement à un événement de dépôt

# Tuple

## Tuple de valeurs

N-uplet ordonné (= liste) de valeurs de type arbitraire (y compris tuple) :

```
[ 3 4 ], [ 10 'A' true ], [ "coucou" [ 10000 false ] 23 ]
```

## Tuple motif

N-uplet ordonné de valeurs ou de **types** (classes en java) :

```
[ ?Integer ?Boolean ], [ ?Integer 'A' ?Boolean ],  
[ ?String ?Tuple 23 ]
```

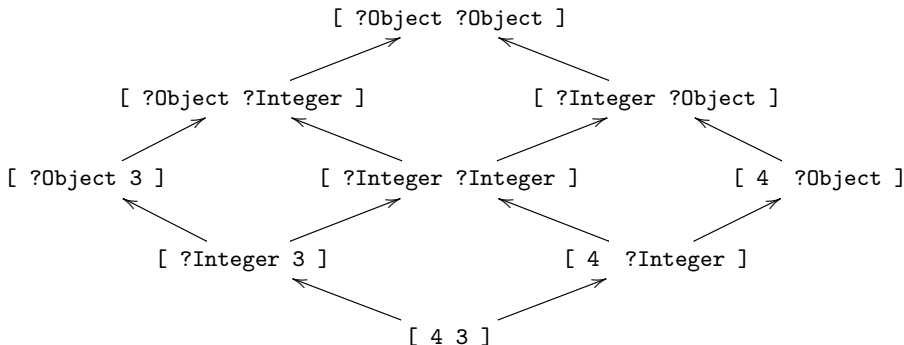
## Correspondance

Un tuple **correspond** (matches) à un tuple s'il est inclus dedans :

```
[ 3 4 ] matches [ ?Integer ?Integer ]  
[ 3 4 ] matches [ ?Integer 4 ] or [ 3 ?Integer ]  
[ 3 ?Integer ] matches [ ?Integer ?Integer ]
```

# Tuples

- ?Object est le type universel
- ?Tuple correspond à tout tuple
- On peut comparer des motifs :



# Manipulation des tuples

La classe Tuple étend List<Serializable>

Plus :

- constructeur `Tuple(...)` :  
    `new Tuple(4, "foo", Integer.class)`  
     $\Rightarrow$  [ 4 "foo" ?Integer ]
- `t.toString()` pour écrire un tuple
- `Tuple.valueOf(String)` pour construire un tuple à partir d'une chaîne
- `t.matches(motif)`  
    Renvoie vrai si t est inclus dans le motif
- `motif.contains(t)`  
     $=$  `t.matches(motif)`

# Les primitives bloquantes du modèle Linda

## Objectif

Un espace partagé contenant des tuples.

Opérations bloquantes :

- **write**(tuple) : dépose le tuple dans l'espace partagé ;
- **take**(motif) : extrait de l'espace partagé un tuple correspondant au motif précisé en paramètre ;
- **read**(motif) : recherche (sans l'extraire) dans l'espace partagé un tuple correspondant au motif fourni en paramètre.



## Exemple

```
// Attend un tuple
// [ ?Integer ?String ]
new Thread() {
    public void run() {
        Tuple m, r;
        m = new Tuple(Integer.class,
                       String.class);
        r = linda.take(m);
        System.out.println(r);
    }
}.start();
```

```
// Dépose [ 4 5 ]
// et [ 4 "foo" ]
new Thread() {
    public void run() {
        Tuple v1, v2;
        v1 = new Tuple(4, 5);
        linda.write(v1);

        v2 = new Tuple(4, "foo");
        linda.write(v2);
    }
}.start();
```

## Les primitives non bloquantes du modèle Linda

- **tryTake**(motif) : version non bloquante de take ;
- **tryRead**(motif) : version non bloquante de read ;
- **takeAll**(motif) : renvoie, en extrayant, *tous* les tuples correspondant au motif (collection vide si aucun ne correspond) ;
- **readAll**(motif) : renvoie, sans extraire, tous les tuples correspondant au motif (collection vide si aucun).

# La primitive d'abonnement du modèle Linda

```
public interface Callback {  
    /** Callback when a tuple appears. */  
    void call(Tuple t);  
}
```

**eventRegister**(mode, timing, motif, callback)

- S'abonner à l'existence/l'apparition d'un tuple correspondant au motif.
- `callback.call()` sera invoqué avec le tuple identifié.
- Le callback n'est déclenché qu'une fois, puis oublié.

# Types d'abonnement

## Mode

- `eventMode.READ` : le tuple est laissé dans l'espace
- `eventMode.TAKE` : le tuple est retiré de l'espace

## Timing

- `eventTiming.IMMEDIATE` : l'état courant est considéré : si un tuple correspondant existe déjà, le callback est déclenché.
- `eventTiming.FUTURE` : l'état courant n'est pas considéré : seuls les futurs write seront pris en compte.

## Exemple d'abonnement

```
class MyCallback implements Callback {  
    public void call(Tuple t) {  
        System.out.println("Got " + t);  
    }  
}
```

...

```
Tuple motif = new Tuple(Integer.class, String.class);  
Callback cb = new MyCallback();  
linda.eventRegister(eventMode.READ, eventTiming.FUTURE,  
                    motif, cb);
```

## Exemple d'abonnement permanent

```
Tuple motif = new Tuple(Integer.class, String.class);
...

class MyCallback implements Callback {
    public void call(Tuple t) {
        System.out.println("Got " + t);
        linda.eventRegister(eventMode.TAKE, eventTiming.IMMEDIATE,
                             motif, this);
    }
}

...
Callback cb = new MyCallback();
linda.eventRegister(eventMode.TAKE, eventTiming.IMMEDIATE,
                    motif, cb);
```

## Callback asynchrone

Attention : un callback est invoqué lors d'un write, dans le contexte du thread déposateur  $\Rightarrow$  il ne doit jamais se bloquer !

**Callback asynchrone** : s'exécute dans son propre thread.

```
...  
Tuple motif;  
motif = new Tuple(Integer.class, String.class);  
Callback cb = new AsynchronousCallback(new MyCallback());  
linda.eventRegister(..., ..., motif, cb);
```

## Spécification libérale

- Choix arbitraire quand plusieurs tuples correspondent à un motif d'un take
  - Choix arbitraire quand plusieurs take sont en attente et qu'un dépôt pourrait en débloquent plusieurs
  - Sélection arbitraire quand des read et un take sont en attente et qu'un dépôt peut les débloquent :  
outre take, tous les read sont débloquentés, ou aucun, ou seulement certains
  - Choix arbitraire entre déblocage d'un take et déclenchement d'un callback (en mode TAKE) concerné par un même tuple
- « Arbitraire » ne signifie pas aléatoire, mais non spécifié : vous pouvez choisir une stratégie bien identifiée.



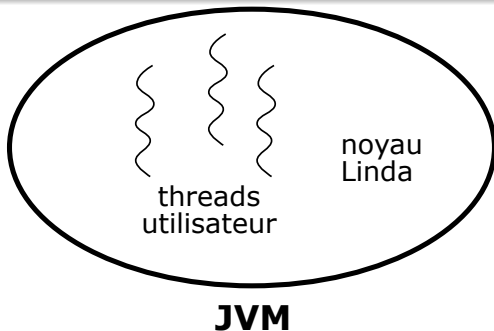
# Plan

- 1 Linda
- 2 Réalisations**
- 3 Modalités

# Réalisations

- ❶ Version en mémoire partagée
- ❷ Version client / mono-serveur
- ❸ Applications :
  - Calcul des nombres premiers
  - Recherche de valeur par distance

## Version en mémoire partagée

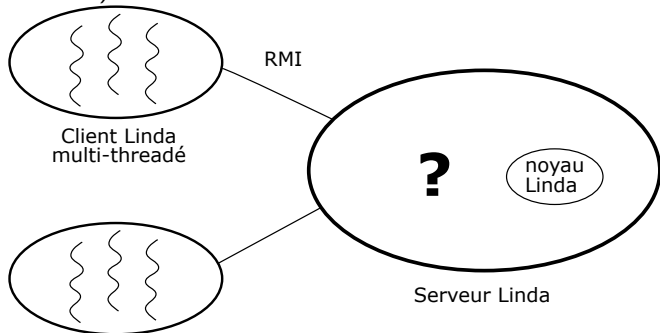


Plusieurs activités (threads), dans la même machine virtuelle Java.

- ❶ Écrire write, read, take
- ❷ Ajouter eventRegister, refactoring de read et take
- ❸ Compléter les opérations non bloquantes

## Version client / mono-serveur

Un serveur stockant l'espace de tuples, un ou des clients distants accédant au serveur. Chaque client peut lui-même être concurrent (multi-threadé).



- Réutiliser le noyau linda centralisé, inchangé (si possible)
- La difficulté est dans les callbacks

# Applications

## Calcul des nombres premiers

- Version séquentielle du crible d'Eratosthène
- Versions parallèles

## Recherche approximative

- Recherche d'un mot dans une liste par distance
- 1 manager + 1 chercheur (fourni)
- Extensions à faire :
  - plusieurs chercheurs
  - plusieurs requêtes simultanées
  - création/termination dynamique de chercheurs ;
  - ...

# Plan

- 1 Linda
- 2 Réalisations
- 3 Modalités**

# Modalités

Cf page moodle

<https://moodle-n7.inp-toulouse.fr/course/view.php?id=1974>

- Projet exécuté *en trinôme*
- Séances de suivi et d'assistance, présence obligatoire pour faire le point
- Rendu intermédiaire avant Noël
- Rendu final le **16 janvier 2022** sur moodle
- Séance de restitution la semaine du 17 janvier

## Modalités (2)

- Squelettes et code de base disponible sur moodle
- **Respectez l'API** pour les implantations du noyau : interfaces Linda, Callback, Tuple, etc ;  
constructeur classe `linda.shm.CentralizedLinda` ;  
constructeur classe `linda.server.LindaClient`.
- La partie application est libre.
- Les tests fournis sont clairement insuffisants : plan de tests à définir et à rendre.

**Les exemples fournis doivent compiler et s'exécuter correctement sans que vous y touchiez le moindre caractère** (sauf pour changer linda centralisé / client-serveur).