Programmation Concurrente, Réactive et Répartie

Cours N°5

Emmanuel Chailloux

Master d'Informatique
Université Pierre et Marie Curie

année 2018-2019

Plan

- Canaux synchrones en OCaml
- Compléments Java
  - processus et runtime
  - framework Executor
    - Callable
    - Future
  - λ-expression et streams

Modèle à mémoire distincte

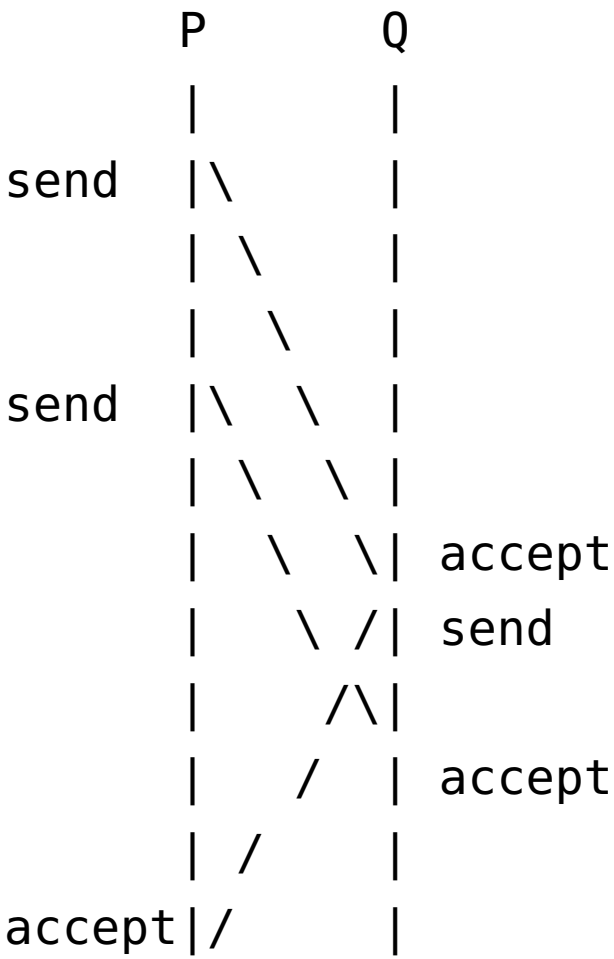
modèle à communication de messages (message passing)
2 primitives :

- “envoi un message” :
- “accepte un message”

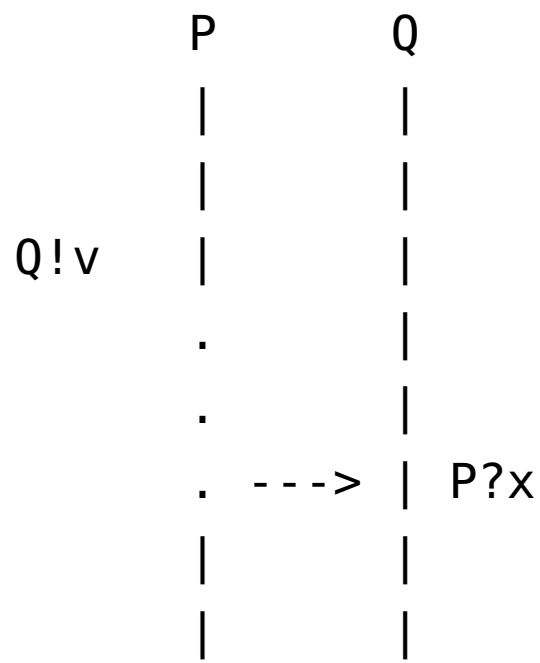
Caractéristiques

- envoi bloquant ou non
- réception bloquante ou non (*polling*)

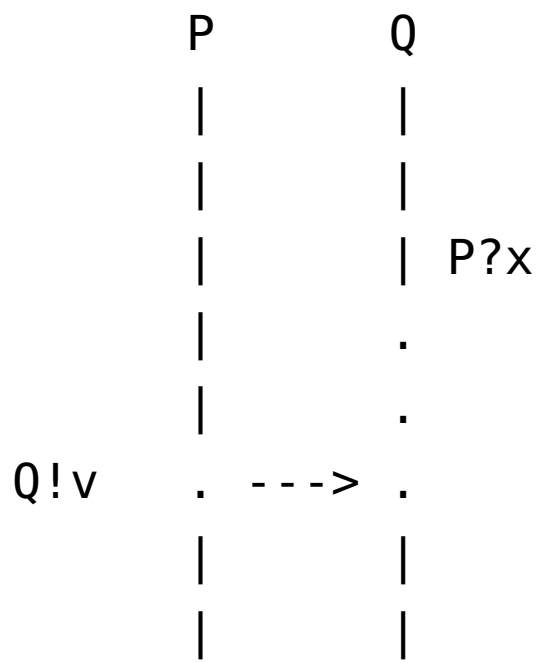
Communications asynchrones



Communications synchrones



Communications synchrones



Module Event - OCAML

- communication synchrone
- canaux fortement typés
- si synchronisation, réception bloquante ou non (*poll*)

event.mli

```
1  type 'a channel
2  val new_channel: unit -> 'a channel
3  type 'a event
4  val send: 'a channel -> 'a -> unit event
5  val receive: 'a channel -> 'a event
6  val always: 'a -> 'a event
7  val choose: 'a event list -> 'a event
8  val wrap: 'a event -> f:('a -> 'b) -> 'b event
9  val guard: (unit -> 'a event) -> 'a event
10 val sync: 'a event -> 'a
11 val select: 'a event list -> 'a
12 val poll: 'a event -> 'a option
```

Evénements, canaux et communication

- 2 types abstraits : `'a channel` et `'a event`
- `new_channel : unit -> 'a channel` : création d'un canal
- `send : 'a channel -> 'a -> unit event` : envoi une valeur `v` de type `'a` sur un canal `c` de type `'a channel`, retourne un événement dont la valeur est de type `unit` (valeur `()`)
- `receive : 'a channel -> 'a event`, retourne un événement de la valeur transmise.

`send` et `receive` ne sont pas bloquantes ! ! !

Synchronisation

- `sync : 'a event -> 'a` : fonction principale de synchronisation

  transforme un événement lié à une valeur en cette valeur.

Exemple 1 : partage de référence

```
1  let ch = Event.new_channel () ;;
2  let v = ref 0;;
3
4  let reader () = Event.sync (Event.receive ch);;
5  let writer () = Event.sync (Event.send ch ("S" ^ (string_of_int !v)));;
6
7  let loop_reader s d () =
8    for i=1 to 10 do
9      let r = reader() in
10       print_string (s ^ " " ^ r); print_newline();
11     Thread.delay  d
12   done ;;
13
14 let loop_writer d () =
15    for i=1 to 10 do incr v; writer(); Thread.delay d
16    done ;;
17
18 Thread.create (loop_reader "A" 1.1) ();;
19 Thread.create (loop_reader "B" 1.5) ();;
20 Thread.create (loop_reader "C" 1.9) ();;
21 Thread.delay 2.0;;
22 loop_writer 1. ();;
```

Exemple 1 : trace

```
% ocamlc -thread unix.cma threads.cma es1.ml
```

```
% ./a.out
C S1
A S2
B S3
C S4
A S5
B S6
C S7
A S8
B S9
C S10
```

```
% ./a.out
B S1
A S2
C S3
B S4
A S5
C S6
B S7
A S8
C S9
B S10
```

Exemple 2 : gensym (sans synchro)

```
1  type uid = UID of string Event.channel;;
2
3  let makeUidSrc () =
4    let ch = Event.new_channel () in
5    let rec loop i = begin
6       Event.send ch ("S"^(string_of_int i));
7       loop (i+1)
8    end in
9      Thread.create (fun () -> loop 0) () ;
10     UID ch
11 ;;
12
13 let getUid (UID ch) = Event.receive ch;;
```

Exemple 2 : gensym (avec synchro)

```
1  type uid = UID of string Event.channel;;
2
3  let makeUidSrc () =
4    let ch = Event.new_channel () in
5    let rec loop i = begin
6       Event.sync (Event.send ch ("S"^(string_of_int i)));
7       loop (i+1)
8    end in
9      Thread.create (fun () -> loop 0) () ;
10     UID ch
11 ;;
12
13 let getUid (UID ch) = Event.sync(Event.receive ch);;
```

Programme principal

```
1  let ch1 = makeUidSrc ();;
2
3  let main ti msg  () =
4    while (true) do
5      Thread.delay(ti);
6      let r = getUid ch1 in
7      print_string (msg); print_string " -- ";
8      print_string r;  print_newline();
9   done;;
10
11 Thread.create (main 1.1 "A") ();;
12
13 main 2.1 "B"  ();;
```

Trace

```
A -- S0
Src0
B -- S1
Src1
A -- S2
Src2
A -- S3
Src3
B -- S4
Src4
A -- S5
Src5
A -- S6
Src6
B -- S7
Src7
```

Polling

- `'a event -> 'a option` : version non bloquante de `sync`

  retourne `Some v` si un événement est présent, sinon `None`

Autres fonctions sur les événements

- `always : 'a -> 'a event` : crée un événement toujours prêt pour la synchronisation ;
- `wrap : 'a event -> ('a -> 'b) -> 'b event` applique une fonction sur la valeur de l'événement (fonction de post-processing)
- `wrap_abort : 'a event -> (unit -> unit) -> 'a event` applique la fonction en cas de non sélection de l'événement

Choix d'un événement dans une liste

- `choose : 'a event list -> 'a event`
- `select : 'a event list -> 'a`

```
1    let select x = sync(choose x);;
```

Exemple : accumulateur +/-

3 canaux : addCh, SubCh et readCh :

```
1  let rec accum sum =
2     print_int sum; print_newline();
3     Event.sync  (
4       Event.choose [
5          wrap (receive addCh) (fun x -> accum(sum + x));
6          wrap (receive subCh) ( fun x -> accum(sum - x));
7          wrap (send  readCh sum)  ( fun x -> accum(sum))
8        ]
9     );;
```

wrap associe des actions aux communications ! ! !

Requêtes

```
1  let clientCallEvt x =
2    wrap (send reqCh x) (fun () -> receive replyCh);;
```

Mémoire partagée synchronisée (1)

M-variable :

- une M-variable est soit vide, soit pleine
- opération take : prendre la valeur d'une M-variable si elle est pleine, bloquante sinon
- opération put : remplit une M-variable, provoque une erreur si elle est pleine

Interface

```
1 type 'a mvar
2 val mVar : unit -> 'a mvar
3 exception Put
4 val mTake : 'a mvar -> 'a Event.event
5 val mPut : 'a mvar -> 'a -> unit
```

Une M-variable est construite dans un état vide.

Mémoire partagée synchronisée (2)

```
1  type 'a mvar = MV of ('a Event.channel * 'a Event.channel
2                        * bool Event.channel);;
3
4  let mVar () =
5    let takeCh = Event.new_channel ()
6    and putCh = Event.new_channel ()
7    and ackCh = Event.new_channel () in
8    let rec empty () =
9      let x = Event.sync (Event.receive putCh) in
10       Event.sync (Event.send ackCh true);
11       full x
12   and full x =
13     Event.select
14       [Event.wrap (Event.send takeCh x) empty ;
15        Event.wrap (Event.receive putCh)
16                   (fun _ -> (Event.sync (Event.send  ackCh false); full x))]
17   in
18     ignore (Thread.create empty ());
19     MV (takeCh, putCh, ackCh) ;;
```

Mémoire partagée synchronisée (3)

```
1
2  let mTake ( mv : 'a mvar) = match mv with
3     MV (takechannel, _, _ ) -> Event.receive takechannel ;;
4
5  exception Put;;
6  let mPut mv x = match mv with
7    MV (takechannel, putchannel, ackchannel) ->
8       Event.sync (Event.send putchannel x);
9       if (Event.sync( Event.receive ackchannel)) then ()
10      else raise Put ;;
```

Processus et runtime

Retour vers le futur: : processus systèmes

- `Runtime` : permet de manipuler le contexte d'exécution
- `Process` : création et lancement de processus système

classe Runtime

- `Runtime Runtime.getRuntime()` : retourne le contexte d'exécution
- `Process exec(String)`
  ou `Process exec(String, String[])`
  ou `Process exec(String, String[], String)` :
  - exécute une commande (avec ou sans arguments)
    (on peut aussi passer le catalogue de travail)
  - et retourne une instance de Process

classe Process

- classe abstraite
- contrôle d'un processus extérieur
- instance de retour des appels exec de `Runtime`

```
1  // lancement
2  Process myG = Runtime.getRuntime().exec("where sleep");
3
4  // attente
5  myG.waitFor();
6
7  // valeur de retour
8  myG.exitValue();
```

framework Executor

dans le but d'éviter le coût de création de thread et le coût mémoire d'un nouveau thread, Java 1.5 introduit le framework `Executor` :

- fournit un pool de threads : interface *ExecutorService*
- appel de thread avec résultat de retour : interface *Callable*
- appel asynchrone : classe *Future*

Executor et ExecutorService

2 interfaces pour lancer des calculs :

```java
public interface Executor {
    void execute(Runnable command);
}
```

```java
public interface ExecutorService extends Executor {

    // Job submission
     <V> Future<V> submit(Callable<V> job);
     Future<?> submit(Runnable job);
     Future<V> submit(Runnable job, V resut);
     // ...
     <V> List<Future<V>> invokeAll(Collection <? extedns Callable<V>> ↩
          jobs);
     <V> List<Future<V>> invokeAll(Collection <? extedns Callable<V>> ↩
          jobs,long timeout, TimeUnit unit);
}
```

Pool de threads

Création d'un pool de threads :

```
1  import java.util.concurrent.*;
2
3  public class MyExecutorService {
4    public static void main(String... args) {
5      Runnable job = new Runnable() {
6        public void run() {
7          System.out.println("Je suis dans le thread : " +
8                              Thread.currentThread().getName());
9      System.out.flush();
10       }
11     };
12     // Pool avec 4 threads
13     ExecutorService pool = Executors.newFixedThreadPool(4);
14     int max = Integer.parseInt(args[0]);
15     for (int i =0; i< max; i++) {pool.submit(job);}
16     pool.shutdown();
17     System.out.println("Je suis dans le thread : " +
18                        Thread.currentThread().getName());
19     System.out.flush();
20
21   }
22 }
```

Exécution

2 lancements de job :

```
% java MyExecutorService 2
Je suis dans le thread : pool-1-thread-1
Je suis dans le thread : pool-1-thread-2
Je suis dans le thread : main
```

10 lancements de job (pool de 4 threads) :

```
% java MyExecutorService 10
Je suis dans le thread : pool-1-thread-1
Je suis dans le thread : pool-1-thread-4
Je suis dans le thread : pool-1-thread-3
Je suis dans le thread : pool-1-thread-2
Je suis dans le thread : pool-1-thread-2
Je suis dans le thread : pool-1-thread-3
Je suis dans le thread : pool-1-thread-4
Je suis dans le thread : pool-1-thread-1
Je suis dans le thread : pool-1-thread-3
Je suis dans le thread : pool-1-thread-2
Je suis dans le thread : main
```

utilisation de plusieurs fois le même thread.

interface *Callable*

L'interface *Runnable* ne possède qu'une seule définition : `void run()`. Cette méthode ne retourne pas de résultat, ni n'indique d'échappement d'exception.

Pour cela l'interface *Callable<V>* définit une méthode `call` qui retourne un élément de type `V` ou déclenche une exception.

```
1 public interface Callable<V> {
2     V call() throws Exception;
```

Test de Callable (1)

1er exemple : on ne tient pas compte du retour de submit

```
1  public class TestCallable {  static int count = 1;
2    public static int fib(int n){if (n<2) return 1;
3                                 else return (fib(n-1)+fib(n-2));}
4    public static void main(String[] args) {
5      Callable<Integer> job = new Callable<Integer>() {
6        public Integer call() throws Exception {
7          int n = ++count;
8          Integer res = fib(n);
9          System.out.println("Je suis dans le thread " +
10           Thread.currentThread().getName() +  "res = fib("+n+") = " + res);
11         return res;
12       }
13     };
14     ExecutorService pool = Executors.newFixedThreadPool(4);
15     int max = Integer.parseInt(args[0]);
16     for (int i =0; i< max; i++) {pool.submit(job);}
17     pool.shutdown();
18     System.out.println("Je suis dans le thread " +
19                        Thread.currentThread().getName());
20     }
21 }
```

Exécution

```
% java TestCallable 2
Je suis dans le thread pool-1-thread-1res = fib(2) = 2
Je suis dans le thread pool-1-thread-2res = fib(3) = 3
Je suis dans le thread main
```

```
% java TestCallable 10
Je suis dans le thread pool-1-thread-1res = fib(2) = 2
Je suis dans le thread pool-1-thread-4res = fib(5) = 8
Je suis dans le thread pool-1-thread-2res = fib(3) = 3
Je suis dans le thread pool-1-thread-3res = fib(4) = 5
Je suis dans le thread pool-1-thread-2res = fib(8) = 34
Je suis dans le thread pool-1-thread-4res = fib(7) = 21
Je suis dans le thread pool-1-thread-1res = fib(6) = 13
Je suis dans le thread main
Je suis dans le thread pool-1-thread-4res = fib(11) = 144
Je suis dans le thread pool-1-thread-2res = fib(10) = 89
Je suis dans le thread pool-1-thread-3res = fib(9) = 55
```

classe *Future*

```
1  public Future<V> submit(Callable<V> job);
```

Les méthodes de l'interface Future<V> permettent de récupérer le résultat (get()), tester sa disponibilité (isDone()), ou d'annuler son calcul (cancel()).

```
1  public interface Future<V> {
2      public V get();
3      public V get(long timeout, TimeUnit unit);
4      public boolean isDone();
5      public boolean cancel(boolean mayInterruptIfRunning);
6      public boolean isCancelled();
7  }
```

La méthode get est bloquante si le calcul n'est pas fini par le pool de threads. La méthode `isDone` permet de vérifier avant l'appel de get que le résultat est disponible.

Test de Callable avec Future (2)

2ème exemple en tenant compte du résultat de `submit` :

```
1   ExecutorService pool = Executors.newFixedThreadPool(4);
2   int max = Integer.parseInt(args[0]);
3   Future<Integer>[] results = new Future[max];
4   for (int i =0; i< max; i++) {results[i]=pool.submit(job);}
5   pool.shutdown();
6   System.out.println("Je suis dans le thread " +
7                      Thread.currentThread().getName());
8   try {
9     for (int i =0; i< max; i++) {
10      System.out.println("Je suis dans le thread " +
11        Thread.currentThread().getName() + " res = " + results[i].get());
12    }}
13    catch (Exception e){}
14
15 }
16 }
```

Exécution

```
% java TestCallableFuture 7
Je suis dans le thread pool-1-thread-2res = fib(3) = 3
Je suis dans le thread main
Je suis dans le thread pool-1-thread-1res = fib(2) = 2
Je suis dans le thread pool-1-thread-3res = fib(4) = 5
Je suis dans le thread pool-1-thread-4res = fib(5) = 8
Je suis dans le thread pool-1-thread-3res = fib(8) = 34
Je suis dans le thread pool-1-thread-1res = fib(7) = 21
Je suis dans le thread main res = 2
Je suis dans le thread pool-1-thread-2res = fib(6) = 13
Je suis dans le thread main res = 3
Je suis dans le thread main res = 5
Je suis dans le thread main res = 8
Je suis dans le thread main res = 13
Je suis dans le thread main res = 21
Je suis dans le thread main res = 34
```

Dans une véritable application, on évitera les attentes bloquantes.

Complétion de services

La classe *CompletionService* encapsule un `ExecutorService` pour surveiller la progression des différents calculs qui ont été soumis. La méthode take renvoie les résultats au fur et à mesure qu'ils sont disponibles. Elle est bloquante.

```
1   ExecutorService pool = Executors.newFixedThreadPool(4);
2   CompletionService<Integer> completion = new ExecutorCompletionService<↩
        Integer>(pool);
3   int max = Integer.parseInt(args[0]);
4   for (int i =0; i< max; i++) {completion.submit(job);}
5   pool.shutdown();
6   System.out.println("Je suis dans le thread " +
7                      Thread.currentThread().getName());
8   Future<Integer> f;
9   try {
10    for (int i =0; i< max; i++) {
11      f = completion.take();
12      System.out.println("Je suis dans le thread " +
13        Thread.currentThread().getName() + " res = " + f.get());
14    }}
```

Exécution

```
% java TestCallableFutureTake 8
Je suis dans le thread pool-1-thread-2 res = fib(3) = 3
Je suis dans le thread main
Je suis dans le thread pool-1-thread-1 res = fib(2) = 2
Je suis dans le thread main res = 3
Je suis dans le thread pool-1-thread-3 res = fib(4) = 5
Je suis dans le thread pool-1-thread-4 res = fib(5) = 8
Je suis dans le thread pool-1-thread-3 res = fib(8) = 34
Je suis dans le thread main res = 2
Je suis dans le thread pool-1-thread-1 res = fib(7) = 21
Je suis dans le thread pool-1-thread-2 res = fib(6) = 13
Je suis dans le thread main res = 5
Je suis dans le thread pool-1-thread-4 res = fib(9) = 55
Je suis dans le thread main res = 8
Je suis dans le thread main res = 34
Je suis dans le thread main res = 21
Je suis dans le thread main res = 13
Je suis dans le thread main res = 55
```

ExecutorService : autres méthodes

- attente

```
boolean awaitTermination(long timeout, TimeUnit unit)
 Blocks until all tasks have completed execution after a shutdown request, or the
 timeout occurs, or the current thread is interrupted, whichever happens first.
boolean  isShutdown()
Returns true if this executor has been shut down.
boolean  isTerminated()
Returns true if all tasks have completed following shut down.
void  shutdown()
 Initiates an orderly shutdown in which previously submitted tasks are executed,
 but no new tasks will be accepted.
List<Runnable>  shutdownNow()
 Attempts to stop all actively executing tasks, halts the processing of waiting
 tasks, and returns a list of the tasks that were awaiting execution.
```

- invocation

```
<T> T  invokeAny(Collection<? extends Callable<T>> tasks)
 Executes the given tasks, returning the result of one that has completed
 successfully (i.e., without throwing an exception), if any do.
```

`invokeAny` est bloquante. Les différents *Callable* doivent retourner un élément du même type.

Introduction des Lambda-expressions

Pour répondre à ces inconvénients, Java 1.8 introduit les traits de la programmation fonctionnelle :

- des lambda-expressions (notées λ-expressions)
- passage de paramètres fonctionnels
- composition de calculs
- retour d'une valeur fonctionnelle d'une fonction ou d'une méthode
- typage des lambda-expressions via des interfaces fonctionnelles le tout dans le cadre typé à la Java :
  - interface fonctionnelle
  - API Stream

Création d'une valeur fonctionnelle

Syntaxe:

```
(parametres) -> corps_de_la_fonction
```

- si le corps de la fonction est une expression
- si le corps de la fonction est une suite d'instructions :
  les encadrer par des accolades, et utiliser l'instruction `return`

```
1  (n) -> n +1                                              // fonction successeur
2  (a,b) -> a * b                                         // fonction multiplication
3  (Etudiant et) ->  (et.getAge() >= 16)                 // ve'rifier si un e'tudiant
4                      && (et.getAge() <= 23)             //  a  entre 16 et 23 ans
```

```
1  (Etudiant et) -> { int age = et.getAge();
2    System.out.println(age);
3    return (age >= 16) && (age <= 23); }
```

Comme pour les classes locales ou anonymes, les lambda-expressions peuvent capturer des variables.

Interface fonctionnelle

On parle d'interface fonctionnelle pour les interfaces ne contenant qu'une seule déclaration de méthode. De nombreuses interfaces fonctionnelles sont définies dans `java.util.function`.

```
1 public interface Predicate<T> {
2   boolean test (T t) ;
3 }
4 public interface Function<T,R> {
5   R apply(T t) ;
6 }
```

On associera une telle interface comme type d'une λ-expression, qui se devra de respecter la signature de la méthode déclarée. Il n'y a pas de type fonctionnel directement manipulable.

```
1 Predicate<Etudiant> estJeune =
2   (Etudiant et) ->
3     (et.getAge() >= 16) && (et.getAge() <= 23) ;
```

Une λ-expression peut être considérée comme une implantation anonyme d'une telle interface. Elle se doit d'être compatible au niveau des types (paramètres formels et type de retour)).

Référencement de méthode

Si on désire utiliser une méthode statique existante à la place d'une fonction, il est alors possible de la référencer par la notation `::`.
les références sur les méthodes d'instance ou de classe pour un objet ou une classe donnés, utilisent la même notation.

```
String::valueOf     x -> String.valueOf(x)
Object::toString    x -> x.toString()
x::toString        () -> x.toString()
ArrayList::new     () -> new ArrayList<>()
```

Utilisation d'une valeur fonctionnelle

- fonction réflexe pour l'interface graphique, ici l'interface `MouseListener` ne possède qu'un seul champ : `void mousePressed(MouseEvent e)`

```
1  MouseListener clic = (MouseEvent e) -> {
2    int x = e.getX();
3    int y = e.getY();
4    System.out.println("x = " + x + " y = "+y) ;
5    getGraphics().drawString("salut le monde", x, y)
6   } ;
7  addMouseListener(clic) ;
```

- et de manière plus concise :

```
1  MouseListener clic =
2    (e) -> getGraphics().drawString("salut le
3    monde",e.getX(),e.getY());
4  addMouseListener(clic) ;
```

Composition de calculs et de l'ordre supérieur

On peut ainsi définir de nouveaux itérateurs basé sur les fonctions :

- `map` qui applique une fonction aux éléments d'une liste et retourne la liste des résultats
- `filter` qui retourne les éléments d'une liste qui vérifient un prédicat
- `compose` qui compose deux fonctions g et f et calcule $g(f(x))$

```
1 public <T, R> List<R> map(List<T> l, Function<T, R> f) ;
2 public <T> List<T> filter(List<T> l, Predicate <T>pred) ;
3 public <R,S,T> T compose(Function <S,T> g, Function<R,S> f, <R> x) ;
```

```
1 public <T>  List<T> filter(List<T> l, Predicate<T> pred) {
2   List<T> r = new ArrayList<T>();
3   for (T e : l) {    if (pred.test(e)) r.add(e);   }
4   return r;
5 }
```

```
1 Predicate<Etudiant> estJeune =
2   (et) -> ((et.getAge() >=16) && (et.getAge() <= 23)) ;
3 List<Etudiant> le2 = filter(le,estJeune);
```

Un exemple complet d'utilisation de λ-expressions

On cherche à récupérer la liste des noms des étudiants jeunes d'une liste d'étudiants, sans tenir compte des étudiants avec un nom vide.

```
1  public List <String> getNomsEtudiantsJeunes(List<Etudiant> etudiants) {
2    List<Etudiant> l = filter(etudiants, (et) ->
3              (et.getAge() >= 16) &&  (et.getAge() <= 23)) ;
4    List<Etudiant> l2 = map(l, (p) -> p.getNom()) ;
5    List<Etudiant> l3 = filter(l2, (n) -> (! (n.getNom().equals(""))));
6    return l3.reduce("", (r,n) -> r + "," + n);
7  }
```

La classe `Etudiant` possède les méthodes `getAge` et `getNom`.
Les collections possèdent des méthodes utilisant des λ-expressions comme `map`, `filter` et `reduce`.

Api Stream

L'Api `Stream` pour les flux facilite la composition fonctionnelle de calculs, et permet de les paralléliser facilement (*stateless*).
Quelques signatures de méthodes :

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
Stream<T> filter(Predicate<? super T> predicate)
void forEach(Consumer<? super T> consumer)
T reduce(T identity, BinaryOperator<T> accumulator)
...
```

```
1  public String getNomsEtudiantsJeunes(List<Etudiant> etudiants) {
2    return etudiants                                    // tous les etudiants
3      .stream()                                         // dans un   flux
4      .parallel()                                        //  parallele
5      .filter((Etudiant et) ->                          //  filtrer les jeunes
6              (et.getAge() >= 16) && (et.getAge() <= 23))
7      .map(p -> p.getNom())                            // recuperer  les noms
8      .filter(n -> !(n.getNom().equals("")))           // sauf les ""
9      .reduce("", (res, n) -> res + ", " + n)         // les concatener tous
10 }
```