Notes JAVA (importations - TPs - Activité 1)

Intro P1

Scanner s = new scanner(System.in)
Int n = s.nextInt()

20)	String	next()	It is used to get the next complete token from the scanner which is in use.
21)	BigDecimal	nextBigDecimal()	It scans the next token of the input as a BigDecimal.
22)	BigInteger	nextBigInteger()	It scans the next token of the input as a BigInteger.
23)	boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
24)	byte	nextByte()	It scans the next token of the input as a byte.
25)	double	nextDouble()	It scans the next token of the input as a double.
26)	float	nextFloat()	It scans the next token of the input as a float.
27)	int	nextInt()	It scans the next token of the input as an Int.
28)	String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
29)	long	nextLong()	It scans the next token of the input as a long.
30)	short	nextShort()	It scans the next token of the input as a short.

Packages et interfaces

Il y a deux manières d'utiliser une classe stockée dans un package :

- En utilisant le nom du package suivi du nom de la classe

```
java.util.Date now = new java.util.Date() ;
System.out.println(now) ;
```

En utilisant le mot clé **import** pour importer (inclure) le package auquel appartient la classe

```
import java.util.Date ;          // Doit être en tête du fichier !!
   // Ne permet d'importer que la classe Date de java.util
...
Date now = new Date() ;
System.out.println(now) ;

import java.util.* ;             // Permet d'importer toutes les
  classes de java.util
...
Date now = new Date() ;
System.out.println(now) ;
```

On peut généralement utiliser l'une ou l'autre de ces deux méthodes, mais il existe un cas ou l'on doit obligatoirement utiliser la première : si deux classes portant le même nom sont définies dans deux packages différents.

› ***La visibilité* friendly *est celle prise par défaut.***

Accessible aux :	méthodes de la même classe	classes dérivées dans le même package	classes du même package	classes dérivées dans un autre package	classes des autres packages
public	X	X	X	X	X
protected	X	X	X	X	
friendly	X	X	X		
private	X				

Interfaces :

LES INTERFACES

- Une interface est une déclaration permettant de décrire un ensemble de méthodes abstraites et de constantes. On peut considérer une interface comme étant une classe abstraite ne contenant que des méthodes abstraites et que des attributs final et static.

- **Syntaxe**

La syntaxe est simple et sans surprises : il suffit de fournir un corps à la méthode, et de la qualifier avec le mot-clé default

```
public interface Foo {
   public default void foo() {
          System.out.println("Default implementation of foo()");
          }
}
```

- **Une interface fonctionnelle** est une interface comprenant exactement une seule méthode abstraite.
- L'annotation **@FunctionalInterface** peut précéder la déclaration des interfaces fonctionnelles . Elle n'est pas obligatoire

Expressions lambdas

```
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
```

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
```

References_méthodes

1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

```
ContainingClass::staticMethodName
```

Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```
interface Sayable{
    void say();
}
public class MethodReference {
    public static void saySomething(){
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args) {
        // Referring static method
        Sayable sayable = MethodReference::saySomething;
        // Calling interface method
        sayable.say();
    }
}
```

Reference to an instance method :

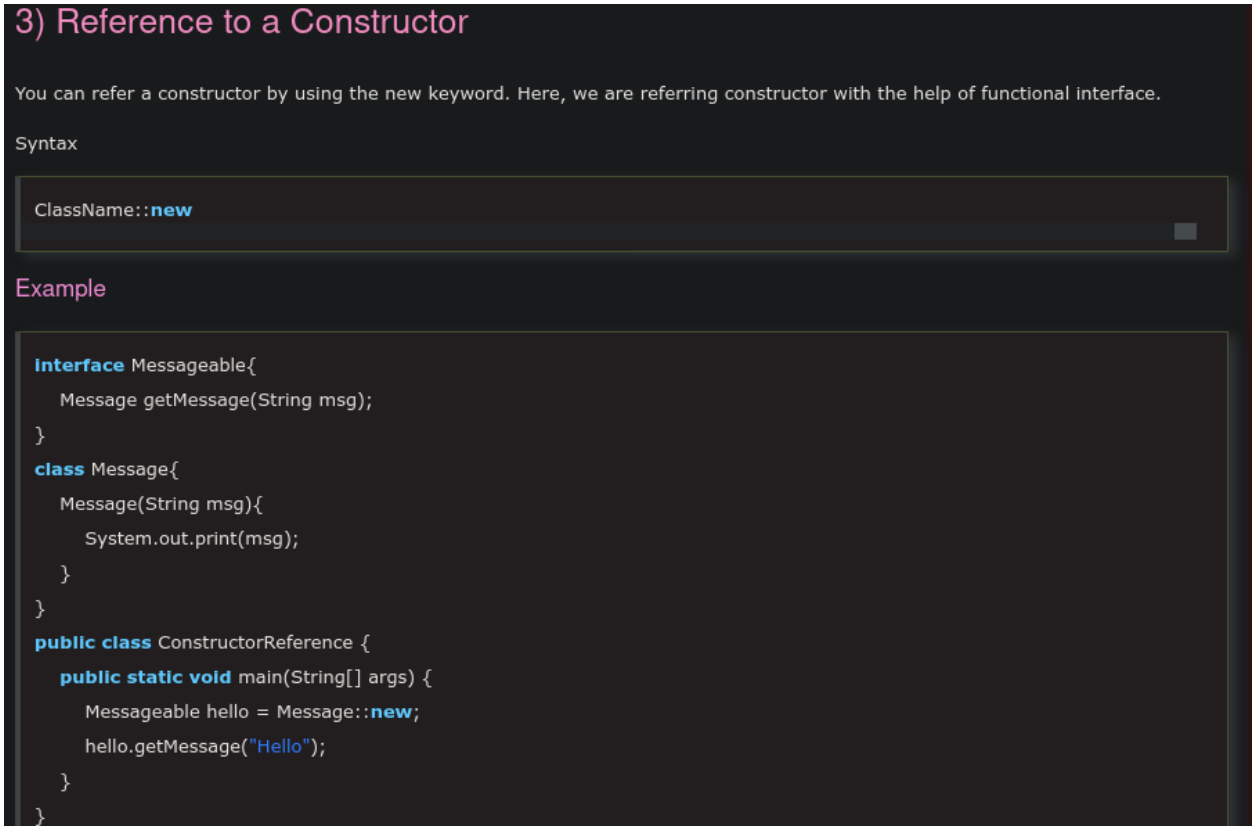
Syntax

```
containingObject::instanceMethodName
```

Example 1

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

```
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args) {
        InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
        // Referring non-static method using reference
            Sayable sayable = methodReference::saySomething;
        // Calling interface method
            sayable.say();
            // Referring non-static method using anonymous object
            Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anonymous object also
            // Calling interface method
            sayable2.say();
```



3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

```
ClassName::new
```

Example

```
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

Classes génériques

Classe<?>

Accepte n'importe quelle classe

Classe<? extends type>

Accepte n'importe quelle classe fille de type

classe <? super type>

Accepte tout les ascendants de type

Instantiation :

classe<type_generique> nom_var = new classe<> ()

Exceptions

Classe throwable :

- Throwable(String s)
- String getMessage()
- Void printStackTrace()

Error Herite de Throwable :

- Erreurs critiques non sensés etre geré par notre programme
- Le programme s'arrete automatiquement

Exception herite de Throwable :

- Doivent etre traités

Si l'exception ne va pas etre traité localement, on doit ajouter throws TypeException dans l'entete de la méthode (Ceci est recurisf)

Sinon on la traité avec :
try {}
catch (Exception e) n'importe exception{}
Catch (TypeException e){}
Finally{} (cleanup code, fermeture de fichier connexions - va etre executé meme s'il ya un return dans le catch ou try)

Custom Exception

```
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
  super(s);
 }
}
```

Classes de base

Classe object :

toString() : Retourne nom_classe@adresse_objet
equals () : retourne boolean @objet1 == @objet2
finalize () : "destructeur" invoqué avant la destruction d'un objet
clone() : instancie un nouveau objet copie de this
Il faut que la classe implémente l'interface clonable

Classe Class :

objet.getClass()-> renvoie un objet de type Classe
Class.getName() : Renvoie le nom de la classe
Class. getSuperclass() : renvoie des informations de type Class sur la super classe.
Class.toString : = getName() + “class” || “interface” selon le cas

StringBuffer :

Représente des chaines de tailles variables
On ne peut pas utiliser “+” comme avec String

StringTokenizer:

StringTokenizer st = new StringTokenizer(texte, “,:”)
Découpe le texte en des tokens à chaque “,” ou “:”

```
void AfficheParMots(String texte) {
   StringTokenizer st = new  StringTokenizer(texte, ",:") ;

   while ( st.hasMoreTokens() ) {
        String mot = st.nextToken() ;
        System.out.println(mot) ;
   }
}
...
AfficheParMots("Lundi,Mardi:Mercredi;Jeudi");
// Affiche :
Lundi
Mardi
Mercredi;Jeudi
```

Collections

Vecteurs Dynamiques : ArrayList, Vector
Listes chainées : LinkedList
Ensembles ‘Set’ : HashSet - TreeSet
PriorityQueue - ArrayDecque

Interface Collection :

Int size()
Boolean Contains(element)
Boolean isempty()
Boolean add(element)
Boolean remove(element)
Iterator iterator()
Int hashCode()
Boolean equals(element)

Boolean add/remove/containsAll(collection)
Void clear()

Interface Iterator :

Boolean hasNext()
Obj next()
Void remove()

Exemple :

```
Collection c = new …
Iterator it = c.iterator()
While (it.hasNext()){
        Element = it.next()
        ….
}
```

Interface Set :

Herite de Collection
La seule différence est que les élements ne sont pas dupliqués
SortedSet hérite de Set
A = {1,2,3}

Classe TreeSet, HashSet implémentent Set et SortedSet :

Exemple :

```
Set a = new HashSet();
a.add(“eqsd”);
….
// l’ajout d’un élément déja existant n’implique pas un erreur
```

a ne sera pas trié
Set b = new TreeSet(a)
b sera trié

Interface List :

Hérite de collection
Obj get(index);
Obj set(index, element)
Void add(index, element)
Obj remove(index)
Boolean addAll(index, collection)
Int indexOf(element)
Int lastIndexOf(element)
ListIterator listIterator(index("optionnel"))
List subList(fromindex,toIndex)

Interface ListIterator :

Hérite de Iterator
Permet le parcours dans les deux sens
++
Boolean hasPrevious()
Obj previous()
Void set(element)
Void add(element)
Int nextIndex()
Int previousIndex()
Exemple : Parcours à l'envers :
ListIterator it = new list.listIterator(list.size())
While (it.hasPrevious()){
….
}

Classe LinkedList :

Previous du premier element est Null
Next du dernier element est Null
Void addFirst(element)
Void removeLast(element)
Pour l'insertion au milieu on utilise l'iterator it.next() , it.add(element)

Classe ArrayList :

Plus rapide en lecture que linkedList
Moins rapide en écriture modification au milieu de la liste
Boolean contains(element)

For (type element : arrayListName)
arrayListName.forEach(callback/reference.../ *Exemple : System.out::println*)

Pour les collections génériques, la déclaration de l'iterator doit être générique aussi.

Interface Map :

Object put (obj key, obj value)
Object get(obj key)
Object remove(obj key)
Boolean containsKey(obj key)
Boolean containsValue(obj value)
Int size()

Interface Entry :
Obj getKey()
getValue()
setValue(obj value)

Algorithmes manipulant les collections :

sort(list)
sort(list, Comparator comp)
shuffle(list)
reverse(list)
fill(list,element)
copy(listsrc,listdest)
Min,max…

Interface Queue : fifo

Boolean offer(element) ajout
Obj pull() renvoie et supprime
Obj peek renvoie

Classe PriorityQueue :

Interface Comparator :

Int compareTo(obj) ->
Object faisant l'appel est plus petit : <0
== : =0
Sinon >0

Implémentation d'un comparator dans une classe :

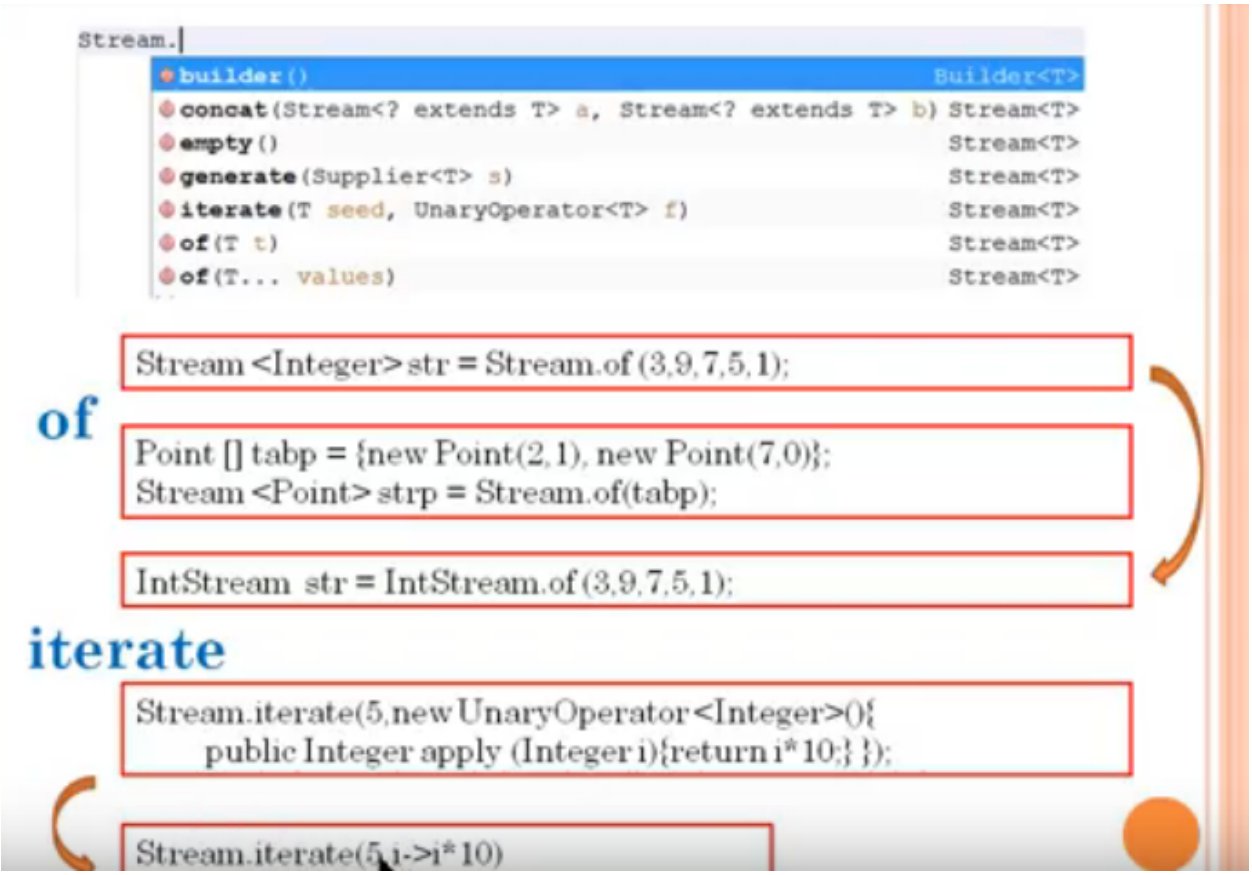
```
public static Comparator<Employee> SalaryComparator = new Comparator<Employee>() {
    @Override
    public int compare(Employee e1, Employee e2) {
        return (int) (e1.getSalary() - e2.getSalary()); });
```

Interface Deque :

add/get/removeFirst/Last()
offer/peek/poll

API Stream

Generation de stream :

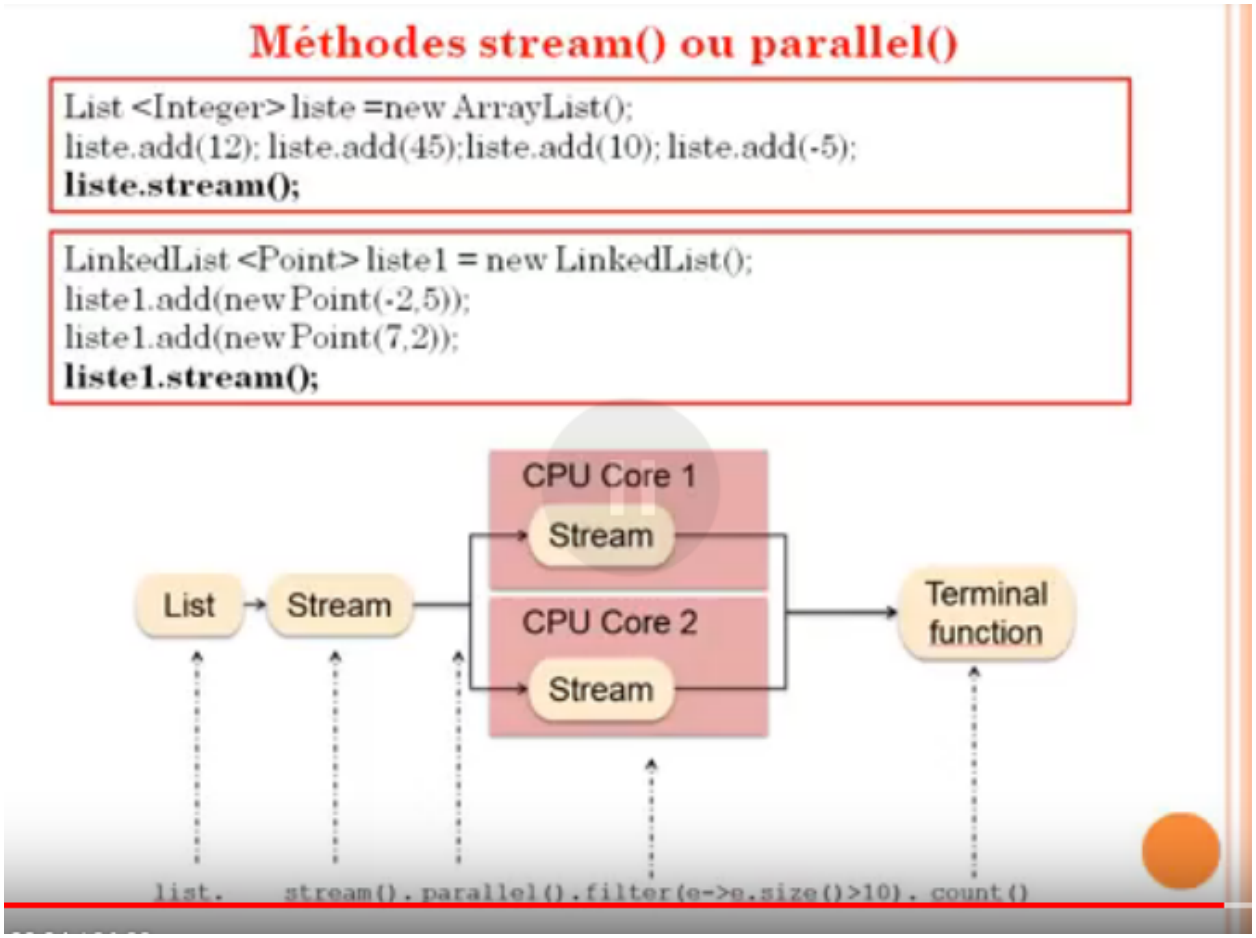


generate

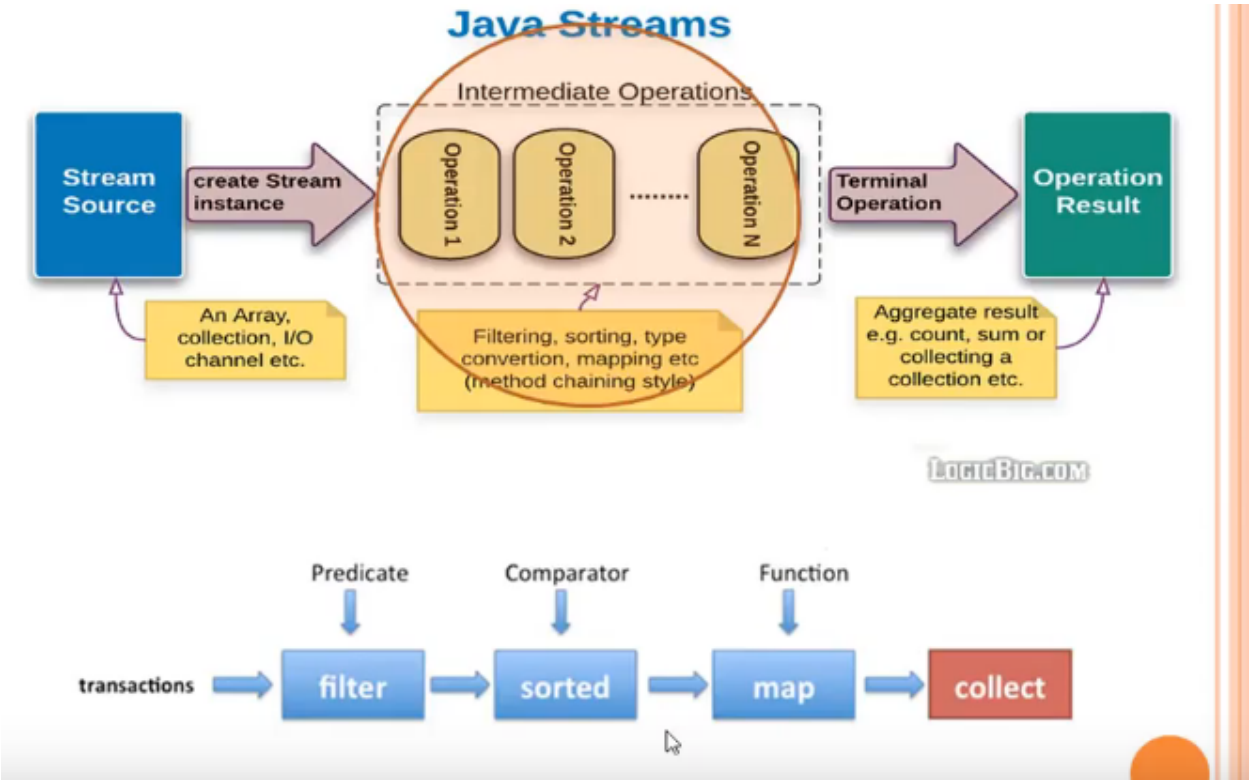
```
Stream.generate(()->"bonjour "); // Stream infini de String (bonjour)
Stream.generate(Math::random); // Stream infini
Stream.generate(Math::random).limit(8);
```

```
import java.util.function.UnaryOperator;
import java.util.stream.IntStream;
import java.util.stream.Stream;
```

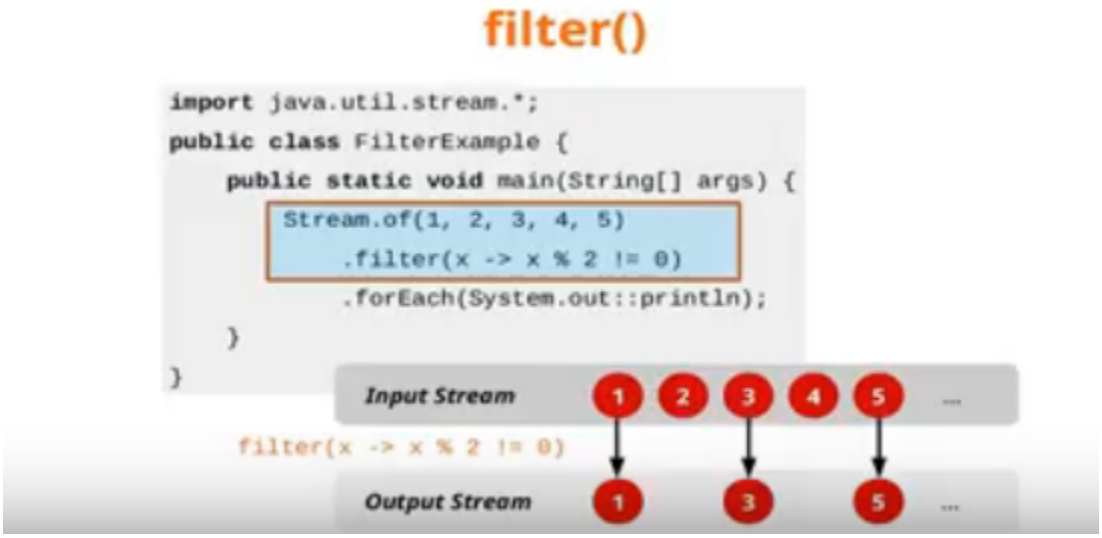
```
// spécifique IntStream
IntStream.range(20,23); // suite : 20,21,22
IntStream.rangeClosed(20,23); //suite : 20,21,22,23
```



Méthodes intermédiaires :



Filter :



Map : (prend en paramétre “new Function<TypeEntree,TypeSortie> {TypsSortie apply(TypeEntree){}”)

Stream

```
LinkedList <Point> liste1 = new LinkedList();
liste1.add(new Point(-2,5));
liste1.add(new Point(7,2));
liste1.stream();
liste1.stream().map(p-> p.getX()+p.getY());

// La méthode map() fait correspondre à chaque point un entier (somme
// des coordonnées ici) donc elle reçoit
//en entrée un stream<Point> et on aura en sortie un stream d'entiers
```

Sorted :

sorted

```
Integer[] tab = {2,15,-5,-2,3,1,45};
Stream.of(tab).sorted(); // en ordre naturel
Stream.of(tab).sorted(Comparator.reverseOrder());
```

```
comparing(Function<? super T, ? extends U> keyExtractor)                                    Comparator<T>
comparing(Function<? super T, ? extends U> keyExtractor, Comparator<? super U> keyComparator) Comparator<T>
comparingDouble(ToDoubleFunction<? super T> keyExtractor)                                   Comparator<T>
comparingInt(ToIntFunction<? super T> keyExtractor)                                         Comparator<T>
comparingLong(ToLongFunction<? super T> keyExtractor)                                       Comparator<T>
naturalOrder()                                                                              Comparator<T>
nullsFirst(Comparator<? super T> comparator)                                                Comparator<T>
nullsLast(Comparator<? super T> comparator)                                                 Comparator<T>
reverseOrder()                                                                              Comparator<T>
class
30  Comparator.
```

Pour éviter ce genre d'erreur, on aimerait mieux modéliser la sortie ***pas de résultat.*** On a ajouté la classe Optional dans Java 8 :
un Optional peut soit contenir une valeur (non null)**, soit représenter l'absence de valeur.**

```
Optional<Person> findByLastName(List<Person> persons, String name) {
for (Person person : persons) {
if (person.getLastName().equals(name)) {
return Optional.of(person);}
}
return Optional.empty();
}
```

Collect :

```
List<Commande> mesCommandes = ... ;
List<Client> mesClients = mesCommandes.stream() .map( c -> c.getClient() ) .collect( Collectors.toList() );
```

```
List<Commande> mesCommandes = ... ;
List<Client> mesClients = mesCommandes.stream() .map( c -> c.getClient() ) .distinct() .collect( Collectors.toList() );
```

Common Collectors

- Collectors.toList()
- Collectors.toSet()
- Colloctors.groupingBy()
- Collectors.joining()

```
// Convert elements to strings and concatenate them, separated by commas
String joined = things.stream()
                      .map(Object::toString)
                      .collect(Collectors.joining(", "));

// Compute sum of salaries of employee
int total = employees.stream()
                     .collect(Collectors.summingInt(Employee::getSalary)));

// Group employees by department
Map<Department, List<Employee>> byDept
    = employees.stream()
               .collect(Collectors.groupingBy(Employee::getDepartment));

// Compute sum of salaries by department
Map<Department, Integer> totalByDept
    = employees.stream()
               .collect(Collectors.groupingBy(Employee::getDepartment,
                                              Collectors.summingInt(Employee::getSalary)));

// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing =
    students.stream()
            .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

Les terminals :

```
    public static void main(String[] args) {
// max, min, sum, average sur un IntStream
 int[] tab = {2,15,-5,-2,3,1,45};
OptionalInt max = IntStream.of(tab).max() ;
System.out.println(max);
OptionalInt min = IntStream.of(tab).min() ;
System.out.println(min);
int somme = IntStream.of(tab).sum();
 System.out.println(somme);
 int somme1 = IntStream.of(tab).filter(e->e<0).sum() ;
System.out.println(somme1);
OptionalDouble moyenne = IntStream.of(tab).average();
 System.out.println(moyenne);
OptionalDouble moyenne1 = IntStream.of(tab).filter(e->e<0).average();
System.out.println(moyenne1);
 Integer[] tabobj = {2,15,-3,2,-5,34,23,4,-8,12};
 Optional <Integer> maxobj = Stream.of(tabobj).max(Comparator.naturalOrder());
 if(maxobj.isPresent()) System.out.println("max des positifs = "+maxobj.get());
```

Optional et comparateurs :

```
    public static void main(String[] args) {
Personne [] tab = {new Personne("Arnauld","Jean",2001),new Personne("Rougier","Michel",1987), ne
List <Personne> liste = Arrays.asList(tab);
OptionalInt anneejeune = liste.stream().mapToInt(p->p.getDate()).max();
//System.out.println(anneejeune);
if(anneejeune.isPresent()) System.out.println("le plus jeune est né  "+anneejeune.getAsInt());
else System.out.println(" liste vide");
// avec comparateur
Optional <Personne> personneJeune = liste.stream().max(Comparator.comparing(Personne::getDate));
if(personneJeune.isPresent()){
   Personne pj = personneJeune.get();
   System.out.println("le plus jeune est "+pj.getNom()+" "+pj.getPrenom()+"  "+ pj.getDate());}
else System.out.println("---- liste vide ----");
    }
```