// Lambda //method reference  
// list.forEach(System.out::println);  
for(int i = 0; i<persons.size(); i++){  
 System.out.println(persons.get(i));  
  
}  
/\*for(Person p : persons){  
 System.out.println(p);  
}\*/  
persons.forEach(x -> System.out.println(x));

persons.forEach(System.out::println);  
  
// 1-Filter (param: Predicate)   
List<Person> filteredList1 = persons.stream()  
 .filter(p -> App.*getAge*(p.getBirthDate()) >= 18)  
 .collect(Collectors.*toList*());  
  
//App.printList(filteredList1);   
  
// 2-Map (param: Function)  
Function<String, String> coderFunction = name -> "Coder " + name;  
List<String> filteredList2 = persons.stream()  
 //.filter(p -> App.getAge(p.getBirthDate()) >= 18)  
 //.map(p -> App.getAge(p.getBirthDate()))  
 //.map(p -> "Coder " + p.getName())  
 //.map(p-> p.getName())  
 .map(Person::getName)  
 .map(coderFunction)  
 .collect(Collectors.*toList*());  
//App.printList(filteredList2);   
  
// 3-Sorted (param: Comparator)  
Comparator<Person> byNameAsc = (Person o1, Person o2) -> o1.getName().compareTo(o2.getName());  
Comparator<Person> byNameDesc = (Person o1, Person o2) -> o2.getName().compareTo(o1.getName());  
Comparator<Person> byBirthDate = (Person o1, Person o2) -> o1.getBirthDate().compareTo(o2.getBirthDate());  
  
List<Person> filteredList3 = persons.stream()  
 .sorted(byBirthDate)  
 .collect(Collectors.*toList*());  
//App.printList(filteredList3);   
  
// 4-Match (param: Predicate)  
Predicate<Person> startsWithPredicate = person -> person.getName().startsWith("J");  
// anyMatch : No evalua *todo el stream, termina en la coincidencia*boolean rpta1 = persons.stream()  
 .anyMatch(startsWithPredicate);   
// allMatch : Evalua *todo el stream bajo la condicion*boolean rpta2 = persons.stream()  
 .allMatch(startsWithPredicate);   
  
// noneMatch : Evalua *todo el stream bajo la condicion*boolean rpta3 = persons.stream()  
 .noneMatch(startsWithPredicate);   
  
// 5-Limit/Skip  
int pageNumber = 1;  
int pageSize = 2;  
List<Person> filteredList4 = persons.stream()  
 .skip(pageNumber \* pageSize)  
 .limit(pageSize)  
 .collect(Collectors.*toList*());  
//App.printList(filteredList4);  
  
// 6-Collectors  
// GroupBy  
Map<String, List<Product>> collect1 = products.stream()  
 .filter(p -> p.getPrice() > 20)  
 .collect(Collectors.*groupingBy*(Product::getName));  
//System.out.println(collect1);  
// Counting  
Map<String, Long> collect2 = products.stream()  
 .collect(Collectors.*groupingBy*(  
 Product::getName, Collectors.*counting*()  
 )  
 );  
//System.out.println(collect2);  
//Agrupando por nombre producto y sumando  
Map<String, Double> collect3 = products.stream()  
 .collect(Collectors.*groupingBy*(  
 Product::getName,   
 Collectors.*summingDouble*(Product::getPrice)  
 )  
 );  
//System.out.println(collect3);  
//Obteniendo suma y resumen  
DoubleSummaryStatistics statistics = products.stream()  
 .collect(Collectors.*summarizingDouble*(Product::getPrice));  
//System.out.println(statistics);  
  
//7-reduce   
Optional<Double> sum = products.stream()  
 .map(Product::getPrice)  
 .reduce(Double::*sum*);  
 //.reduce((a,b) -> a+b)  
System.*out*.println(sum.get());

map con filter

System.*out*.println("Map nombres + coder");  
List<String> stringList = persons.stream()  
 .map(p->p.getName().concat(" Coder"))  
 .collect(Collectors.*toList*());  
App.*printList*(stringList);  
System.*out*.println("Map Edades");  
List<Integer> integerList = persons.stream()  
 .filter(p->App.*getAge*(p.getBirthDate())>=18)  
 .map(p -> App.*getAge*(p.getBirthDate()))  
 .collect(Collectors.*toList*());  
App.*printList*(integerList);

Crear una function para utilizar en varios lugares:

Function<String , String> coderFunction = name -> name.concat(" Coder");  
List<String> stringList = persons.stream()  
 //.map(p->p.getName().concat(" Coder"))  
 .map(p->p.getName())  
 .map(coderFunction)  
 .collect(Collectors.*toList*());

Para que funcione lo de arriba, debo transformar con el map el objeto persona en tipo string

.map(p->p.getName()) y luego recién puedo pasar la función

.map(p->p.getName())  
.map(Person::getName)

¡Estas dos expresiones son iguales la misma cosa!!!!

Ordenando con un comparador

// 3-Sorted (param: Comparator)  
System.*out*.println("Ordenando con comparator");  
Comparator<Person> byNameAsc=(Person per1,Person per2)->per1.getName().compareTo(per2.getName());  
List<Person> personList1 = persons.stream()  
 .sorted(byNameAsc)  
 .collect(Collectors.*toList*());  
App.*printList*(personList1);

Creando comparator

Comparator<Person> byNameAsc=(Person per1,Person per2)->per1.getName().compareTo(per2.getName());

Usando el comparator para ordenar

List<Person> personList1 = persons.stream()  
 .sorted(byNameAsc)  
 .collect(Collectors.*toList*());  
App.*printList*(personList1);

Para hacerlo de forma descendente cambiamos el comparator

Comparator<Person> byNameDesc=(Person per1,Person per2)->per2.getName().compareTo(per1.getName());

Cambia el nombre y el orden de los elementos :

per2.getName().compareTo(per1.getName());

4-Match (param: Predicate)  
anyMatch : No evalua todo el stream, termina en la

boolean resp1 = persons.stream()  
 .anyMatch(p -> p.getName().startsWith("J"));  
System.*out*.println(resp1);

Me retorna un Boolean en el caso que exista un elemento que coincida con lo que se busca pero se detiene al encontrar el primero con esa característica, no recorre todo el stream.

allMatch : Evalua todo el stream bajo la condición

boolean resp2 = persons.stream()  
 .allMatch(p->p.getName().startsWith("J"));  
System.*out*.println(resp2);

Al no coincidir todos los elementos del stream con la condición retorna false.

noneMatch : Evalua todo el stream bajo la condicion

aqui estoy diciendo que ninguno coincida con la J retorna false, para que de true ninguno de los elementos debe empezar con la letra J.

boolean resp3 = persons.stream()  
 .noneMatch(p->p.getName().startsWith("J"));

Creo un predicado para evitar repetir código:

Predicate<Person> startsWithPredicate= per->per.getName().startsWith("j");

Limit/Skip

// 5-Limit/Skip  
System.*out*.println("Limit");  
  
List<Person> filterPersonList4 = persons.stream()  
 .limit(2) //limita la captura al 1 y 2 indice.  
 .collect(Collectors.*toList*());  
  
App.*printList*(filterPersonList4);  
  
System.*out*.println("skip");  
  
List<Person> filterPersonList5 = persons.stream()  
 .skip(2) //te captura de la posicion 2 en adelante o sea retorna 3,4 y 5 en este caso  
 .collect(Collectors.*toList*());  
App.*printList*(filterPersonList5);  
//especie de paginacion  
System.*out*.println("Convinar Limit y skip");  
int pageNumber=2;  
int pageSize= 2;  
List<Person> filterPersonList6 = persons.stream()  
 .skip(pageNumber\*pageSize)//donde inicia  
 .limit(pageSize)//cual es el limite  
 .collect(Collectors.*toList*());  
App.*printList*(filterPersonList6);

Collectors-GroupBy retorna un map<TipoDeValorQueAgrupa,Lista<Objeto>>

En el ejemplo Map<Double,List<Product>> como es un map puedo utilizar values()

// 6-Collectors  
// GroupBy  
System.*out*.println("Collectors-GroupBy");  
Map<Double, List<Product>> collect1 = products.stream()  
 .filter(p -> p.getPrice() > 20)  
 .collect(Collectors.*groupingBy*(Product::getPrice));  
  
System.*out*.println(collect1);  
System.*out*.println("Collectors-GroupBy ejemplo 2");  
Map<String, List<Product>> collect2 = products.stream()  
 .filter(p -> p.getPrice() > 20)  
 .collect(Collectors.*groupingBy*(Product::getName));  
System.*out*.println(collect2.values());

Counting

¡¡¡Permite contar cuantas veces se repite un producto por su nombre!!!

// Counting  
Map<String, Long> collect3 = products.stream()  
 .collect(Collectors.*groupingBy*(  
 Product::getName,  
 Collectors.*counting*()  
 )  
 );  
  
System.*out*.println(collect3);

Agrupar por nombre de producto y sumar

Usamos el Collectors.summigDouble()

//Agrupando por nombre producto y sumando  
System.*out*.println("Agrupando por nombre producto y sumando");  
Map<String,Double> collect4= products.stream()  
 .collect(Collectors.*groupingBy*(  
 Product::getName,  
 Collectors.*summingDouble*(Product::getPrice))  
 );  
  
System.*out*.println(collect4);

Obteniendo suma y resumen

Cuando uso Collect(Collectors.summarizingDouble()) eso me retorna un objeto DoubleSumaryStatistics con los siguientes valores DoubleSummaryStatistics{count=4, sum=91,000000, min=15,000000, average=22,750000, max=35,500000} a los cuales se puede acceder con .getSum() o .getMax, .getMin()

.getSum, getAverange()/\*promedio\*/

//Obteniendo suma y resumen  
System.*out*.println("Obteniendo suma y resumen");  
DoubleSummaryStatistics statistics = products.stream()  
 .collect(Collectors.*summarizingDouble*(Product::getPrice)  
 );  
  
  
System.*out*.println(statistics.getSum());

Reduce me permite sacar la suma, el mínimo, el máximo etc y me retorna un opcional al que puedo llamar con .get();

//7-reduce  
System.*out*.println("reduce");  
Optional<Double> suma = products.stream()  
 .map(Product::getPrice)  
 .reduce(Double::*sum*);  
  
System.*out*.println(suma.get());