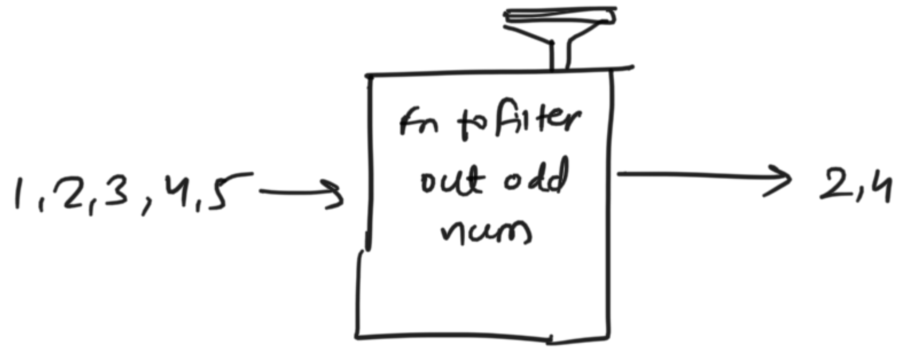
Java: Lambdas and Streams

★ Lambdas and Streams

- Functional programming: Allow you to write powerful code in a concise way
- commonly used since java 8
- Streams allow to do complex things with data manipulation

★ Functional Programming (FP)

- forget about classes, objects etc. and focus on functions
- Functions in FP give same output everytime you give same input.



- Functions can have functions as input.
- Functions can output other functions
- Lambdas and streams allow you to do functional style programming in java.

★ Lambda

- implements functional interfaces

functional interface - an interface that contains only one abstract method
- Runnable, Comparable etc are functional interfaces
- aka Single Abstract Method Interfaces
SAM interfaces

@FunctionalInterface Annotation : used to ensure that the functional interface cant have more than one abstract method.

```
@FunctionalInterface
 interface Square {
     int calculate(int x);
 }

 class Test {
       PSUM(Str args[])
       {
          int a = 5;
```

```
        int a =5;
        Square s = (int x) -> x * x;

        int ans = s.calculate(a);
        sop(ans)      —// 25
      }
  }
```

eg 2: _interface

```
Calculator calcualte = (int x, int y) -> {

     // if there are more statements use { } to write lambda expr.
     Random random = new Random()
     int randomnumber = random.nextInt(bound: 50);
     return x * y + randomnumber
  }
calcullator. Calculate (1, 2);
```

★ But you do not need to create an interface at all
Java has ready made interfaces that can be used
java.util.function contains useful interfaces

eg: _inbuilt interface from jawa.util.function.

```
IntBinaryCalculator  calculator = (x, y) -> {
       Random . . .
        . - - -
       .return  x * y + randomnumber;
    }
calculator.applyAsInt(1, 2);
```

★ Streams API

- Used together with lambdas, streams allow you to write concise, powerful code
- Commonly used since Java 8
- used to process collection of objects
- it is not a data structure. instead takes I/p from collections, Arrays or I/O channels.

lets say you have to do all of this operations on country list
- Capitalize everything
- Filter out countries beginning with c
- Sort countries in alphabetical order
- Print result to console

Without streams it will be a long code for each step individually.

With streams api you can chain methods together

```
country.stream()
    .map (s -> s.toUppercase())
    .filter (s -> !s.startsWith("C"))
    .sorted()
    .forEach (s -> SOP(s));
```

Intermediate operation

fn being passed into a method as argument, commonly done in functional programming

forEach has to go in end, terminal operation

This would not change country list at all, because Streams are immutable

Intermediate operations all return a stream as result while terminal ones return something else so they go at end of chain

★ Intermediate Operations

1. map : used to return a Stream consisting of the results of applying the given function to the elements of this stream.

```
number = Arrays.asList (2,3,4,5);
number.stream.map (x -> x * x).collect(Collectors.toList())
o/p - [4,9,16,25]
```

2. filter: used to select elements as per the predicate passed as argument.

```
names = ['Reflection', 'Collection', 'Stream']
names.stream.filter( s -> s.startsWith ('S')).collect();
o/p - [Stream]
```

3. sorted -: used to sort the stream

```
names.stream().sorted().collect(Collectors.toList());
```

4. limit : used to reduce size of stream. eg- random.ints().limit(10).forEach(SOP());

★ Terminal Operations.

1. collect : used to return the result of intermediate operations performed on the stream

2. forEach: used to iterate through every element of the stream

```
number.stream().map(x -> x * x).forEach (y -> SOP(y));
```

3. reduce: used to reduce elements of a stream to a single value.

-It takes BinaryOperator as a parameter.

```
number = [2,3,4,5]
number.stream().filter(x->x%2==0).reduce
                          (0,(ans,i) -> ans+i);
```

0 is initial value and i is added to it

O/p → 6   (2+4)

Important Points:

- A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
- Stream is used to compute elements as per the pipelined methods without altering the original value of the object.
- Streams can be used only once and you cant call more methods on them after the first time.

How to create a stream-

* Empty Stream

```
Stream<String> streamEmpty = Stream.empty();
```

(We often use empty method upon creation to avoid returning null for streams with no element)

* Stream of collection.

```
Collection<String> c = Arrays.asList('a','b','c');
Stream<String> streamofCollection = c.stream();
```

* Stream of Array.

```
  arr = ["a","b","c"]
Stream<String>  stream1 = Arrays.stream(arr);
 ——"——          stream2 = Arrays.stream(arr,1,3);  end
                                      startindex
                                      (inclusive).
        o/p → ['b','c']
```

* Stream.of()

```
Stream<String> lettersstream = Stream.of("a","b","c");
```

* Stream.builder()

```
Stream<String> streamBuilder = Stream.<String>builder().add('a').
                                        ↑                  build();
```

the desired type should be additionaly specified in right part of the statement otherwise the build method will create an instance of Stream<Object>

* Stream.generate().

resulting stream is infinite, so the developer should specify the desired size or generate() method will work until it reaches memory limit.

```
Stream<String> streamGenerated =
                Stream.generate(() -> "element").limit(10);
// seq. of ten strings with value 'element'.
```

* Stream.iterate()

```
Stream<Integer> strIterate = Stream.iterate(40, n->n+2).limit(20);
```

40 → first parameter. when creating every following element In above example the second element will be 42.