**Lambda Stream API**

* Stream API helps in achieving functional programming and concurrency. By this we can focus on “what to do” instead of “how to do it”.
* Instead of using external iterations, we can use internal iterations with stream API(forEach).
* Values.forEach(i -> sout(i));
* We need to give consumer inerface to forEach method.
* forEach method is a default method in the interface came in java 1.8.
* we can pass method reference as object to forEach method.
* Values.forEach(System.out::println);

**public** **class** foreachimple {

**public** **static** **void** main(String[] args) {

List<Integer> values = Arrays.*asList*(1,2,3,4,5,6);

values.forEach(**new** Consumer<Integer>() {

@Override

**public** **void** accept(Integer t) {

System.***out***.println(t);

}

});

values.forEach(i -> {System.***out***.println(i);}); // by lambda

values.forEach(System.***out***::println); // by passing method reference.

values.forEach(i -> *doubleit*(i));

values.forEach(foreachimple :: *doubleit*);

}

**public** **static** **void** doubleit(Integer i ) {

System.***out***.println(i\*2);

}

}

Stream:

* Values.stream().forEach(System.out::println);
* Values.parallelStream().forEach(System.out::println);
* This will create threads to run in parallel, no of threads depends upon the on the no of cores for processor. So we don’t need to create threads explicitly.
* Stream has lot of methods to process data.
* Stream has methods like intermediate (filter(), map(). ) and Terminate (findFirst(), forEach()).
* We cannot reuse the values of stream once used.
* We cannot also use the stream for second time.
* **package** ConsumerInterfacelearn;
* **import** java.util.Arrays;
* **import** java.util.List;
* **import** java.util.function.BinaryOperator;
* **import** java.util.function.Function;
* **import** java.util.stream.Stream;
* **public** **class** streammethods {
* **public** **static** **void** main(String args[]) {
* List<Integer> values = Arrays.*asList*(1,2,3,4,5,6);
* Stream s = values.stream();
* Function<Integer, Integer> f = **new** Function<Integer, Integer>() {
* @Override
* **public** Integer apply(Integer t) {
* **return** t\*2;
* }
* };
* Stream s1 = s.map(f);
* System.***out***.println(s1);
* BinaryOperator<Integer> b = **new** BinaryOperator<Integer>() {
* @Override
* **public** Integer apply(Integer arg0, Integer arg1) {
* **return** arg0+arg1;
* }
* };
* Integer result = (Integer) s1.reduce(0,b);//initial value and binary operator object
* System.***out***.println(result); }
* }
* Map method will take the function interface
* Reduce method will take binary operator interface.
* Filter method will take predicate interface.
* findFirst will return a optional value as output. We can use “orElse” in combination with findFirst.
* Map and filter methods are lazy evaluation methods. They will execute when only they are required.

A screenshot of a cell phone

Description automatically generated