Stream API & Lambda Expression in Java

Problems faced before implementing Stream API

- Re implementing the operations using loops over and over again.

SELECT max(salary), employee_id, employee_name FROM Employee

The above SQL expression automatically returns the maximum salaried employee's details, without doing any computation on the developer's end. Using collections framework in Java, a developer has to use loops and make repeated checks. Why can’t we do something similar with collections?

- **Efficiency**

As multi-core processors are available at ease, a Java developer has to write parallel code processing that can be pretty error-prone.

Stream API – Java 8

- New abstraction called Stream that lets you process data in a declarative way.

- Streams can leverage multi-core architectures without you having to write a single line of multithread code.

Stream API Implementation compared with Java 7

Java 7

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
  if(t.getType() == Transaction.GROCERY){
    groceryTransactions.add(t);
  }
}
Collections.sort(groceryTransactions, new Comparator(){
  public int compare(Transaction t1, Transaction t2){
    return t2.getValue().compareTo(t1.getValue());
  }
});


List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
  transactionsIds.add(t.getId());
}
```

Java 8

```
List<Integer> transactionsIds =
    transactions.stream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

Parallel Stream

- Streams API will internally decompose your query to leverage the multiple cores on your computer.

```
List<Integer> transactionsIds =
    transactions.parallelStream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

What is Stream API?

- A sequence of elements from a source that supports aggregate operations.

Sequence of elements − A stream provides an interface to a sequenced set of values of a specific element type. However, streams don’t actually store elements; they are computed on demand.

Source − Streams consume from a data-providing source such as collections, arrays, or I/O resources.

Aggregate operations − Stream supports aggregate operations like **filter, map, limit, reduce, find, match**, and so on.

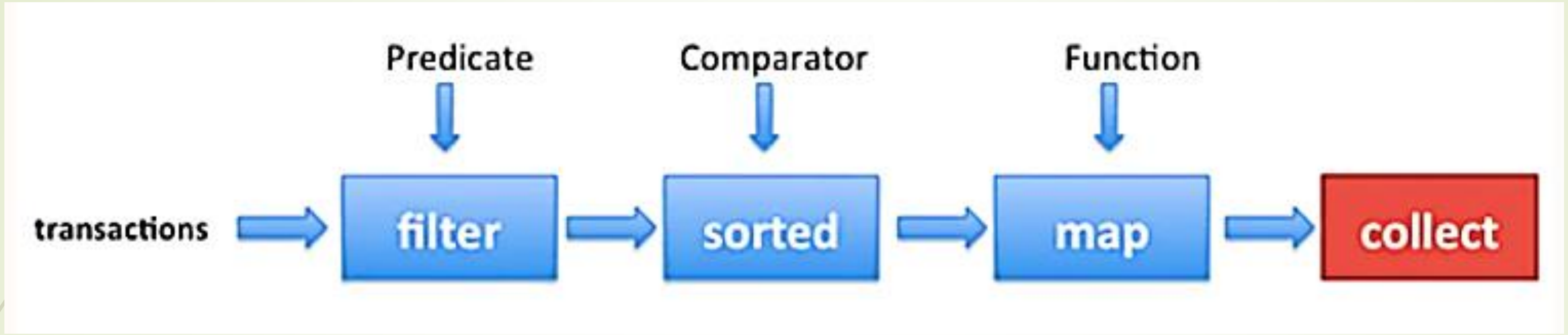
- **Pipelining**

Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline.

- **Internal iteration**

In contrast to collections, which are iterated explicitly stream operations do the iteration behind the scenes for you.

Implementation of Stream API



```
List<Integer> transactionsIds =
    transactions.parallelStream()
            .filter(t -> t.getType() == Transaction.GROCERY) //to filter elements given a predicate
            .sorted(comparing(Transaction::getValue).reversed()) //to sort the elements given a comparator
            .map(Transaction::getId) //to extract information
            .collect(toList()); //something that is not a Stream; here, a List
```

Intermediate Operations	Terminal Operations
Stream operations that can be connected are called intermediate operations.	Operations that close a stream pipeline are called terminal operations.
filter()- takes a predicate as an argument and returns a stream including all elements that match the given predicate	collect() - to receive elements from a steam and store them in a collection
map() - produces one output value of a different type 'X' for each input value of type 'Y'.	reduce() - performs a reduction on the elements of the stream with the given function
distinct()- Returns a stream with unique elements	toArray() - convert stream to array
sorted() - to sort a stream of elements	findFirst() - return first element from stream
peek() - returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream	forEach() - iterating over all elements of a stream and perform some operation on each of them
limit(N)- returns first N elements in the encounter order	min() - to select the smallest element in the stream
skip(n)- skips the first n elements in the encounter order	max() - method to select the largest element in the stream

Collection vs Stream

Collections	Stream
About data	About computations
An in-memory data structure, which holds all the values that the data structure currently has. Every element in the collection has to be computed before it can be added to the collection	A stream is a conceptually fixed data structure in which elements are computed on demand

External iteration
 	
`List<String> transactionIds = new ArrayList<>();`
 	
`for(Transaction t: transactions){`
`    transactionIds.add(t.getId());`
`}`
	Internal iteration
	
	`List<Integer> transactionIds =`
	`transactions.stream().map(Transaction::getId)`
	`.collect(toList());`

Summary about Stream API

- A data source (such as a collection) on which to perform a query
- A chain of intermediate operations, which form a stream pipeline
- One terminal operation, which executes the stream pipeline and produces a result

Lambda Expression

- Provides a clear and concise way to represent one method interface using an expression
- Helps to iterate, filter and extract data from collection
- Don't need to define the method again for providing the implementation. Here, we just write the implementation code.
- Java lambda expression is treated as a function, so compiler does not create .Class file.

Why we need lambda expression?

- To provide the implementation of Functional interface (An interface which has only one abstract method).
- Less coding.

How to implement lambda expression?

- (argument-list) -> {body}

Argument-list: It can be empty or non-empty as well.

Arrow-token: It is used to link arguments-list and body of expression.

Body: It contains expressions and statements for lambda expression.

Without Lamda

```
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();
    }
}
```

With Lamda

```
@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();
    }
}
```

Lambda Expression with multiple parameters

```
interface Addable{
    int add(int a,int b);
}
public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

Thank you