Java 1.8 features

Table of Content

Functional interfaces & Lambda expressions

What is an Interface?

- Interface is a fully abstraction of a class.
- All methods in an interface are "public abstract" & all variables are "public static final".
- Interface is a contract between service provider & service user.
- Interfaces gather irrelevant objects together.

Behavior parameterization

Behavior parameterization is preparing a block of code and making it available without executing it. For example:

```
interface TransactionPredicate {
        boolean test(Transaction transaction);
}
class TransactionAmountPredicate implements TransactionPredicate {
        public boolean test(Transaction transaction) {
                return transaction.getAmount() > 500 ? true : false;
        }
}
```

This block can be passed as an argument to a method. For example:

```
List<Transaction> filterTransactions(List<Transaction> transactions,
TransactionPredicate predicate) {
        for(Transaction transaction: transactions) {
          if (predicate.test(transaction)) {
                myTransactions.add(transaction);
                } }
        return myTransactions;
}
```

```
filterTransactions(transactions, new TransactionAmountPredicate());
```

Functional interface

- Any interface having a single abstract method is called Functional interface. For example Runnable, ActionListener etc.
- Java introduced a new annotation called @FunctionalInterface to mark an interface as functional interface. For example:

```
@FunctionalInterface
public interface TransactionPredicate {
    boolean test(Transaction transaction);
}
```

- Functional interface can have multiple default or static methods.
- Java provides us many pre-defined functional interfaces placed into java.util.function package.

Functional interface example

```
@FunctionalInterface
public interface Sortable {
	boolean compare(Sortable s);
	default void sortAll() {
		//code
	}
	static void compareAll() {
		//code
	}
}
```

Lambda expressions

```
parameter -> expression body
```

- Lambda expression is a concise representation of an anonymous function.
- Lambda expression does not have a name.
- Lambda expression has a list of parameters, a body, a return type & sometimes list of exceptions.
- Lambda expression can be passed as argument to a method or stored in a variable.
- Lambda expression body can optionally use ‘return’ keyword.
- Lambda expression body can have curly braces if body contains multiple statements.

Example

- With type declaration

```
MathOperation addition = (int a, int b) -> a + b;
```

- With out type declaration

```
MathOperation subtraction = (a, b) -> a - b;
```

- With return statement along with curly braces

```
MathOperation multiplication = (int a, int b) -> { return a * b; };
```

- Without return statement and without curly braces

```
MathOperation division = (int a, int b) -> a / b;
```

Examples

- With parenthesis

```
GreetingService greetService1 = message ->
System.out.println("Hello " + message);
```

- With parenthesis

```
GreetingService greetService2 = (message) ->
System.out.println("Hello " + message);
```

Quiz

Select the valid lambda expression among following:

- () -> {}

  Yes

- () -> "Welcome to Java 8"

  Yes

- () -> {return "Welcome to Java 8";}

  Yes

- (Integer i) -> return "Hello " + i;

  No

- (String s) -> {" Welcome to Java 8 ";}

  No

Predicate

```
public interface Predicate<T>{
    boolean test (T t);
}
```

```
import java.util.function.Predicate;
Predicate<String> nonEmptyStringPredicate =
        (String s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings,     nonEmptyStringPredicate);
```

Consumer

```
public interface Consumer<T>{
    void accept(T t);
}
```

```
import java.util.function.Consumer;
Consumer<Integer> consumer =
        (Integer x)->System.out.println(x);
printList(Arrays.asList(10, 15, 20, 44, 85), consumer);
```

Supplier

```
public interface Supplier<T> {
  T get();
}

import java.util.function.Supplier;
Supplier<Integer> supplier = () -> random.nextInt(100);
printGrade(supplier);
printGrade(Supplier<T> supplier) {
        Integer marks = supplier.get();
        //logic to find the grade using marks.
}
```

Function

```
public interface Function<T, R> {
        R apply(T t);
}

Function<Integer, String> function = (Integer marks)->marks > 40 ? "PASS" : "FAILED";
System.out.println("Result = " + function.apply(45));
System.out.println("Result = " + function.apply(23));
```

Primitive specializations

- Apart from generic functional interfaces like Predicate<T>, Supplier<T> etc., Java 8 also supports primitive based functional interfaces.
- If we use generic functional interfaces for primitive data then it requires autoboxing & unboxing. Due to this performance is reduced. Hence we should use primitive based functional interfaces for primitive data.
- Typical examples of primitive functional interface is IntPredicate, IntSupplier, DoubleFunction, LongConsumer etc.

IntPredicate

```
public interface IntPredicate {
		boolean test(int x);
}

IntPredicate intPredicate = (int marks)->marks > 40 ? true : false;
System.out.println("Passed? " + intPredicate.test(55));
System.out.println("Passed? " + intPredicate.test(23));
```

DoubleFunction

```
public interface DoubleFunction<R> {
        R apply(double value);
}

DoubleFunction<String> doubleFunc = (double temperature) -> 
temperature > 20 ? "HOT" : "COOL";
System.out.println("How is the weather? " + doubleFunc.apply(32.2));
System.out.println("How is the weather? " + doubleFunc.apply(8.7));
```

LongConsumer

```
public interface LongConsumer {
        void accept(long value);
}

LongConsumer longConsumer = (long marks) -> System.out.println("Marks: " +
marks);

longConsumer.accept(55);
longConsumer.accept(78);
```

Functional Interfaces Continue…

S.N.	Interface & Description
1	BiConsumer<T,U> Represents an operation that accepts two input arguments and returns no result.
2	BiFunction<T,U,R> Represents a function that accepts two arguments and produces a result.
3	BinaryOperator<T> Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
4	BiPredicate<T,U> Represents a predicate (boolean-valued function) of two arguments.
5	BooleanSupplier Represents a supplier of boolean-valued results.
6	Consumer<T> Represents an operation that accepts a single input argument and returns no result.

Functional Interfaces Continue…

7	DoubleBinaryOperator Represents an operation upon two double-valued operands and producing a double-valued result.
8	DoubleConsumer Represents an operation that accepts a single double-valued argument and returns no result.
9	DoubleFunction<R> Represents a function that accepts a double-valued argument and produces a result.
10	DoublePredicate Represents a predicate (boolean-valued function) of one double-valued argument.
11	DoubleSupplier Represents a supplier of double-valued results.
12	DoubleToIntFunction Represents a function that accepts a double-valued argument and produces an int-valued result.

Functional Interfaces Continue…

13	DoubleToLongFunction Represents a function that accepts a double-valued argument and produces a long-valued result.
14	DoubleUnaryOperator Represents an operation on a single double-valued operand that produces a double-valued result.
15	Function<T,R> Represents a function that accepts one argument and produces a result.
16	IntBinaryOperator Represents an operation upon two int-valued operands and producing an int-valued result.
17	IntConsumer Represents an operation that accepts a single int-valued argument and returns no result.

Functional Interfaces Continue…

18	IntFunction<R> Represents a function that accepts an int-valued argument and produces a result.
19	IntPredicate Represents a predicate (boolean-valued function) of one int-valued argument.
20	IntSupplier Represents a supplier of int-valued results.
21	IntToDoubleFunction Represents a function that accepts an int-valued argument and produces a double-valued result.
22	IntToLongFunction Represents a function that accepts an int-valued argument and produces a long-valued result.

Method references

Lambda expression:

```
Comparator<Transaction> comp = (Transaction t1, Transaction t2)->
t1.getLocation().compareTo(t2.getLocation());
```

Method references:

```
Comparator<Transaction> comp =
Comparator.comparing(Transaction::getLocation);
```

- Method references let you reuse existing method definitions and pass them just like lambdas.
- Method references appear more readable and feel more natural than using lambda expressions.
- Method references can be seen as shorthand for lambdas calling only a specific method.

Types of Method references

There are mainly 3 types of method references supported:

- A method reference to static method. For example Double::parseDouble, Collections::sort etc.
- A method reference to an instance method. For example String::length, Person::getName etc.
- A method reference to an instance method of an existing object. For example transaction::getAmount etc.

Constructor references

- Sometimes a lambda expression does nothing but call an existing method. In such cases we can use constructor reference.
- You can create a reference to an existing constructor using its name and the keyword 'new'. For example:

Lambda expression:

```
Supplier<Transaction> supplier = ()->new Transaction();
Function<Integer, Transaction> func = ()->new Transaction(1001);
```

Constructor reference:

```
Supplier<Transaction> supplier = Transaction::new;
Function<Integer, Transaction> func = Transaction::new;
Transaction t = func.apply(1001);
```

Method reference to static method

```
public class MethodReferencesTest {
		public static void main(String[] args) {
IntPredicate predicate = MethodReferencesTest::isCool;
System.out.println("Is Cool? " + predicate.test(25));
		}

		public static boolean isCool(int temperature) {
				if (temperature < 20)
						return true;
				return false;
		}
}
```

Method reference to instance method

```
    public static void main(String[] args) {
List<Transaction> transactions = new ArrayList<Transaction>();
transactions.add(new Transaction(new Date(), 10000, "PUNE"));
transactions.add(new Transaction(new Date(), 20000, "MUMBAI"));
List<Integer> listAllAmounts = listAllAmounts(transactions, Transaction::getAmount);
    }

    private static List<Integer> listAllAmounts(List<Transaction> transactions,
Function<Transaction, Integer> f){
List<Integer> result = new ArrayList<Integer>();
transactions.forEach(transaction -> result.add(f.apply(transaction)));
        return result;
    }
```

Method reference to an existing object

```
public static void main(String[] args) {
List<Transaction> transactions = new ArrayList<Transaction>();
transactions.add(new Transaction(new Date(), 10000, "PUNE"));
transactions.add(new Transaction(new Date(), 20000, "MUMBAI"));
printTransactions(transactions, System.out::println);
}

private static void printTransactions(List<Transaction> transactions, Consumer consumer) {
transactions.forEach(transaction -> consumer.accept(transaction));
}
```

Reference to constructor

```
Function<Integer, Transaction> func = Transaction::new;

Predicate<Transaction> tranPredicate = (Transaction transaction) ->
transaction.getAmount() > 10000 ? true : false;

System.out.println("Big transaction: " + tranPredicate.test(func.apply(10000)));
```

Function<T, R> default methods

```
Function<Integer, Integer> func_1 = x -> x + 1;
Function<Integer, Integer> func_2 = x -> x * 2;
Function<Integer, Integer> func_3 = func_1.andThen(func_2);
int result = func_3.apply(1);
    //result = 4
Function<Integer, Integer> func_4 = func_1.compose(func_2);
Result = func_4.apply(1);
    //result = 3
```

Predicate<T> default methods

```
Predicate<Integer> pd_1 = (x) -> x > 50;
Predicate<Integer> pd_2 = (x) -> x < 60;
Predicate<Integer> pd_3 = pd_1.and(pd_2);
System.out.println("Result = " + pd_3.test(40));
    //Result = false
Predicate<Integer> pd_4 = pd_1.or(pd_2);
System.out.println("Result = " + pd_4.test(40));
    //Result = true
```

Date APIs

Limitations of Date APIs prior to Java 8

- In Java 1.0, the class java.util.Date does not represent a date but a point in time in millisecond precision.
- The year starts from 1900 & month starts from zero.
- If you wish to build a date 27 Jul 2015, then create Date object as follows:

```
Date date = new Date(115, 6, 27);
```

- In Java 1.1 deprecated several methods of Date class & introduced Calendar class.
- In Calendar also month starts with zero. Using Calendar & Date builds confusion.
- In order to format the date, DateFormat class was introduced. However, it is not thread safe.
- Developers started using third party date libraries.

Introduction to Java 8 Date APIs

- Java 8 introduced a package *java.time* to handle date.
- New Date API provides separate classes for handling dates, time, different timezones, duration, easy manipulation of date/time etc.
- Important classes are:
  - LocalDate
  - LocalTime
  - LocalDateTime
  - Duration
  - Period
  - TemporalAdjusters
  - DateTimeFormatter
  - ZoneId

LocalDate

```
LocalDate localDate = LocalDate.now();
LocalDate localDate = LocalDate.of(2015, 4, 27);
System.out.println(localDate); //2015-04-27
System.out.println(localDate.getDayOfMonth() + "/" + localDate.getMonth().getValue() +
"/" + localDate.getYear());
    //27/4/2015
int year = localDate.get(ChronoField.YEAR);
int month = localDate.get(ChronoField.MONTH_OF_YEAR);
int day = localDate.get(ChronoField.DAY_OF_MONTH);
```

LocalTime

```
LocalTime localTime = LocalTime.now();
LocalTime localTime = LocalTime.of(16, 27, 10);
int hour = localTime.getHour(); //16
int minute = localTime.getMinute(); //27
int second = localTime.getSecond(); //10
LocalTime time = LocalTime.parse("15:15:20");
```

LocalDateTime

```
LocalDateTime dt1 = LocalDateTime.of(2015, Month.APRIL, 27, 16, 20, 10);
LocalDateTime dt2 = LocalDateTime.of(localDate, localTime);
LocalDateTime dt3 = localDate.atTime(13, 45, 20);
LocalDateTime dt4 = localDate.atTime(localTime);
LocalDateTime dt5 = localTime.atDate(date);

LocalDate localDate = dt1.toLocalDate();
LocalTime localTime = dt1.toLocalTime();
```

Duration

Duration class models a quantity or amount of time in terms of seconds and nanoseconds. It is used to find out duration between two dates or two time objects. For example:

```
Duration d1 = Duration.between(time1, time2);

Duration d1 = Duration.between(dateTime1, dateTime2);

long seconds = d1.getSeconds();

Duration fiveMinutes = Duration.ofMinutes(5);
```

Period

When you need to model an amount of time in terms of years, months, and days, you can use the *Period* class.

```
Period tenDays = Period.between(LocalDate.of(2014, 3, 8), LocalDate.of(2014, 3, 18));
int days = tenDays.getDays();
int months = tenDays.getMonths();
int years = tenDays.getYears();
Period tenDays = Period.ofDays(10);
Period threeWeeks = Period.ofWeeks(3);
Period twoYearsSixMonthsOneDay = Period.of(2, 6, 1);
```

TemporalAdjusters

Sometimes you need to perform complex date/time manipulations such as adjusting a date to the next Sunday, the next working day, or the last day of the month etc. Here we can use TemporalAdjusters.

```
import static java.time.temporal.TemporalAdjusters.*;

LocalDate nextSunday = currentLocalDate.with(nextOrSame(DayOfWeek.SUNDAY));

LocalDate lastDayOfMonth = currentLocalDate.with(lastDayOfMonth());
```

TemporalAdjusters continue…

TemporalAdjusters is an functional interface implemented by most of the date related classes. We can write implementation class to meet custom requirements.

```
@FunctionalInterface
public interface TemporalAdjuster {
        Temporal adjustInto(Temporal temporal);
}
```

Custom TemporalAdjusters

```
class NextWorkingDay implements TemporalAdjuster {
        public Temporal adjustInto(Temporal temporal) {
DayOfWeek dow =
DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));
int dayToAdd = 1;
if (dow == DayOfWeek.FRIDAY) { dayToAdd = 3; }
else if (dow == DayOfWeek.SATURDAY) { dayToAdd = 2; }
return temporal.plus(dayToAdd, ChronoUnit.DAYS);
        }
}
LocalDate nextWorkingDate = currentLocalDate.with(new
NextWorkingDay());
System.out.println("Next working day = " + nextWorkingDate);
```

Date formatting

- The new java.time.format package is devoted for date formatting purpose. The central class for date formatting is DateTimeFormatter.
- The java.util.DateFormat class is thread unsafe where the new DateTimeFormatter is thread safe.

```
DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd/MM/yyyy");

LocalDate date1 = LocalDate.of(2016, 4, 27);

String formattedDate = date1.format(formatter);

LocalDate date2 = LocalDate.parse(formattedDate, formatter);

LocalDate date3 = LocalDate.parse("20140318",
DateTimeFormatter.BASIC_ISO_DATE); //2014-03-18

LocalDate date4 = LocalDate.parse("2014-03-18",
DateTimeFormatter.ISO_LOCAL_DATE); //20140318
```

Localized Date formatting

```
DateTimeFormatter italianFormatter =
DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.ITALIAN);
LocalDate date3 = LocalDate.of(2014, 3, 18);
String formattedDate_2 = date3.format(italianFormatter); //18. marzo
2014

DateTimeFormatter frenchFormatter =
DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.FRENCH);
LocalDate date5 = LocalDate.of(2014, 3, 18);
String formattedDate_3 = date5.format(frenchFormatter); //18. mars 2014
```

Time Zones

- Java 8 provides a class java.time.ZoneId as a replacement of java.util.TimeZone class.
- Here is a code to find out the current time in Rome:

```
ZoneId romeZone = ZoneId.of("Europe/Rome");
LocalTime localTime_2 = LocalTime.now(romeZone);
```

Streams

What are streams?

- RDBMS

  Suppose we have an order table & we wish to find out list of orders having order price less than 5000. How do I write the query?

  ```
  SELECT * FROM ORDER WHERE PRICE < 5000
  ```

- Java

  Suppose we have an arraylist having many Order objects & we wish to find out the orders having order price less than 5000. How do I write a program?

  ```
  for(Order order: orders) {
      if (order.getPrice() < 5000)
          print(order);
  }
  ```

What are streams?

- ## RDBMS

  Now suppose we wish to find out orders having price less than 5000 & sorted by price in ascending fashion. How do I write the query?

  ```
  SELECT * FROM ORDER
  WHERE PRICE < 5000
  ORDER BY PRICE
  ```

- ## Java

  How do I achieve the above requirement in Java?

What are streams?

We have 2 options to meet the requirement:

Write a complex code using traditional way i.e.

1. Create a separate arraylist for orders having price less than 5000.
2. Sort the order list by price.

Second option is to use java 1.8 exciting feature called 'Streams'.

```
List<Order> finalOrders = orders.stream().filter(order -> order.getPrice() < 5000).sorted(Comparator.comparing(Order::getPrice)).collect(Collectors.toList());
```

What are streams?

- RDBMS

  Now suppose we wish to find out location based minimum order price order by order location. How do I write the query?

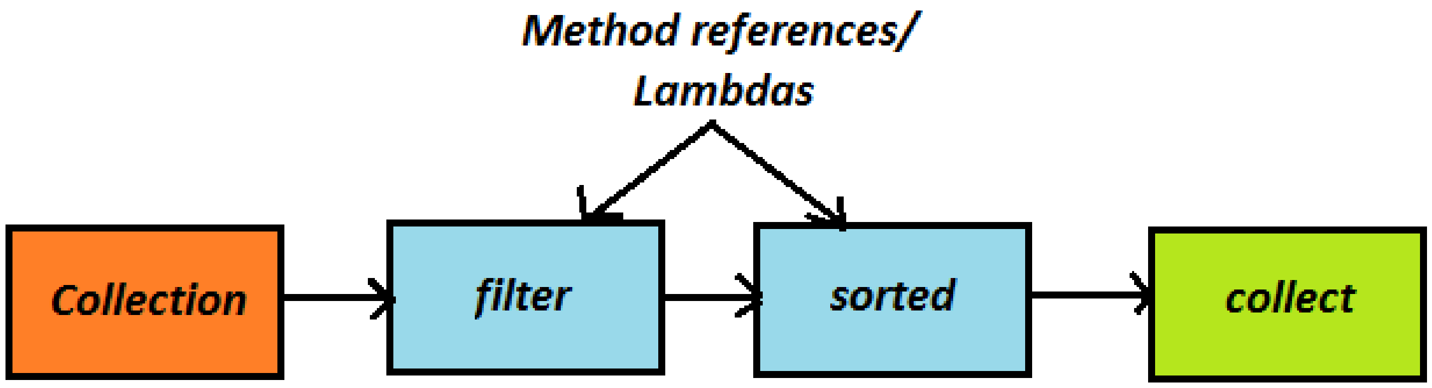
  ```
  SELECT LOCATION, MIN(PRICE) FROM ORDER
  GROUP BY LOCATION
  ORDER BY LOCATION
  ```

- Java

```
Map<String,Optional<Order>> minPriceOrderByLocation =
orders.stream().collect(Collectors.groupingBy(Order::getLocation,
Collectors.minBy(Comparator.comparing(Order::getPrice))));
```

What are streams?

- Streams is a technique to manipulate collections of data in a declarative way.
- Streams can process your collection data in parallel, without you to write any multithreaded code.



Collections vs Streams

- Collections follow supplier-driven approach where as streams follow producer-consumer approach i.e. collection is eagerly constructed & streams is lazily constructed.
- Streams are traversable only once; whereas we can travel into a collection many times.

```
Stream<String> stream = bookNameList.stream();
stream.forEach(System.out::println);
stream.forEach(System.out::println); //IllegalStateException
```

Stream can be consumed only once.

- In collection, user writes program to iterate over data. However, in streams iteration happens internally.

```
List<String> bookNameList =
books.stream().map(Book::getName).collect(toList());
```

Streams API

- Java 8 stream API defines a core interface called java.util.stream.Stream. This interface have several operations which can be divided into two types:
  - Intermediate operation: This operation that can be connected to another operation for example: filter(), map(), limit(), sorted(), distinct() etc.
  - Terminal operation: This operation closes the stream, for example: collect(), count(), forEach() etc.
- java.util.Collection interface defines two default methods stream() & parallelStream() those return Stream object. It means that any collection class that implements Collection interface, can be streamed using these two methods.

Stream operations

- filter(Predicate p)
- distinct()
- limit(long maxSize)
- skip(long n)
- map(Function mapper)
- flatMap(Function Mapper)
- allMatch(Predicate p)
- anyMatch(Predicate p)
- noneMatch(Predicate p)

Stream operations…

- findAny()
- findFirst()
- sorted(Comparator c)
- reduce()
- forEach(Consumer c)
- collect(Collector c)
- count()
- iterate()

filter(Predicate p)

The filter() operation takes as argument a predicate (a function returning a boolean) and returns a stream including all elements that match the predicate. For example:

Find all failed transactions-

```
List<Transaction> failedTransactions = transactions.stream()
.filter(Transaction::isFailed)
.collect(Collectors.toList());
```

distinct()

The distinct() operation returns a stream with unique elements (according to the implementation of the hashCode and equals methods of the objects produced by the stream).

```
List<Transaction> failedTransactions = transactions.stream()
.filter(Transaction::isFailed)
.distinct()
.collect(Collectors.toList());
```

limit(long maxSize)

The limit() operation returns another stream that is not longer than maxsize.

```
List<Transaction> failedTransactions = transactions.stream()
.filter(Transaction::isFailed)
.limit(5)
.collect(Collectors.toList());
```

skip(long n)

The skip() operation returns a stream that discards the first n elements.

```
List<Transaction> failedTransactions = transactions.stream()
.filter(Transaction::isFailed)
.skip(5)
.collect(Collectors.toList());
```

map(Function mapper)

The map() operation allows us to select specific information from objects. For example, in SQL you can select a particular column from a table.

```
List<String> transactionIdList = transactions.stream()
.map(Transaction::getId)
.collect(Collectors.toList());
```

flatMap(Function Mapper)

The flatMap() operation is a combination of a map & a flat operation. This means you first apply map function and than flattens the result.

```
Stream<List<Integer>> stream = Stream.of(Arrays.asList(1, 2, 3), Arrays.asList(1, 12, 30), Arrays.asList(11, 2, 13));
List<Integer> flatIntList =
stream.flatMap(List::stream)
.collect(Collectors.toList());  // 1, 2, 3, 1, 12, 30, 11, 2, 13
```

allMatch(Predicate p)

The allMatch() operation checks whether all the elements of the stream match the given predicate.

```
boolean isHot = temteratures.stream()
.allMatch(t -> t.getTemperature() > 40);
```

anyMatch(Predicate p)

The anyMatch() operation checks at least one element of the stream match the given predicate.

```
boolean isHot = temteratures.stream()
.anyMatch(t -> t.getTemperature() > 40);
```

noneMatch(Predicate p)

The noneMatch() is opposite to allMatch() operation. The noneMatch() checks whether no element in the stream match the given predicate.

```
boolean isHot = temteratures.stream()
.noneMatch(t -> t.getTemperature() > 40);
```

findAny()

The findAny() method returns an arbitrary element of the current stream.

```
Optional<Transaction> opTransaction =
transactions.stream()
.filter(t -> t.getPrice() > 10000)
.findAny();
```

findFirst()

The findFirst() operation is similar to findAny() method. It always returns the first element of the current stream.

```
Optional<Transaction> opTransaction =
transactions.stream()
.filter(t -> t.getPrice() > 10000)
.findFirst();
```

sorted(Comparator c)

The sorted() operation sorts your stream in ascending order. For example:

```
List<Order> matchingOrders =
orders.stream()
.filter(order -> order.getPrice() < 200)
.sorted(Comparator.comparing(Order::getPrice))
.collect(Collectors.toList());
```

reduce()

We use aggregate methods like SUM(), MAX(), MIN() etc. in SQL. The similar aggregation is possible using reduce() operation. Thus, reduce() operation combines elements of a stream to express more complicated queries. For example:

```
int sumOfAllNumbers = numbers.stream()
.reduce(0, Integer::sum); // where '0' is an initial value of sumOfAllNumbers.
Optional<Integer> maxNumber = numbers.stream().reduce(Integer::max);
```

forEach(Consumer c)

The forEach() is a terminal operation that returns void and applies a lambda to each element of the stream.

```
transactions.stream().forEach(System.out::println);
```

collect(Collector c)

The collector() is a terminal operation & it converts a stream into another form like List, Map etc. We passed Collector instance as operation parameter. The Collector instance can be obtained using different static methods from Collectors class. For example:

```
List<Order> myOrders = orders.stream()
.filter(order -> order.getPrice() < 200)
.collect(Collectors.toList());
```

count()

The count() operation counts total number of elements in a stream.

```
long lowPriceOrderCount =
orders.stream().filter(order -> order.getPrice() < 200)
.count();
```

iterate()

The iterate() operation is used to iterate over the loop & perform some business logic in every iteration. It takes 2 arguments, an initial value and a lambda (of type Unary-Operator<T>).

```
Stream.iterate(2, n -> n * n)
.limit(5)
.forEach(System.out::println); //2, 4, 16, 256, 65536
```

Numeric Streams

Suppose we want to find out total price of all transactions.

```
int totalTransactionPrice = transactions.stream()
.map(Transaction::getPrice)
.reduce(0, Integer::sum);
```

The above stream operations will work successfully. However, there is a overhead of boxing. Behind the scene each Integer needs to be unboxed to a primitive before performing summation. In order to improve the performance, we should use primitive based streams instead of generic streams.

Numeric Streams

Java 8 provides us 3 primitive based streams:

- IntStream
- DoubleStream
- LongStream

Now, let us find out total price of all transactions using primitive streams.

```
int totalTransactionPrice = transactions.stream()
.mapToInt(Transaction::getPrice)
.sum();
```

Collectors

Collectors are used to convert elements of a stream into custom formats like List, Map etc.

```
List<Order> myOrders = orders.stream()
.filter(order -> order.getPrice() < 200)
.collect(Collectors.toList());
```

In the above example, we are converting all orders from Order stream into List<Order>. Sometimes we require to reduce (aggregate) the stream. Here we should use Collectors class. Consider the following requirements:

- Group a list of transactions by currency to obtain the sum of the values of all transactions with that currency (returning a Map<Currency, Integer>).
- Partition a list of transactions into two groups: expensive and not expensive (returning a Map<Boolean, List<Transaction>>)

Predefined collectors

Java 8 defines several predefined collectors. These collectors offer three main functionalities:

- Reducing and summarizing stream elements to a single value
- Grouping elements
- Partitioning elements

Reducing and summarizing

```
import static java.util.stream.Collectors.*;

long totalTransactionCount = transactions.stream().collect(counting());

Comparator<Order> orderPriceComparator =
Comparator.comparingInt(Order::getPrice);
Optional<Order> maxPriceOrder =
orders.stream().collect(maxBy(orderPriceComparator));

int totalOrderPrice = orders.stream().collect(summingInt(Order::getPrice));

String orderTitles = orders.stream().map(Order::getTitle).collect(joining(",
"));
```

Grouping

Single-level grouping:

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
transactions.stream()
.collect(groupingBy(Transaction::getCurrency));
```

Multilevel grouping:

```
Map<Currency, Map<String, List<Transaction>>>
transactionsByCurrenciesAndLocation =
transactions.stream().collect(groupingBy(Transaction::getCurrency,
groupingBy(Transaction::getLocation) ));
```

Subgrouping:

```
Map<Transaction.Currency, Long> currencyCount =
menu.stream().collect(groupingBy(Transaction::getCurrency,
counting()));
```

Partitioning

Partitioning is a special case of grouping: having a predicate, called a *partitioning function*, as a classification function.

```
Map<Boolean, List<Order>> partitionedOrders =
orders.stream().collect(partitioningBy(Order::isOpen));
List<Order> openOrders = partionedOrders.get(true);
```

Parallel Streams

A parallel stream is a stream that splits its elements into multiple chunks, processing each chunk with a different thread.

Sequential stream:

```
Stream.iterate(1, i -> i + 1).limit(5).reduce(Integer::sum);
```

Parallel stream:

```
Stream.iterate(1, i -> i + 1)
.limit(5)
.parallel()
.reduce(Integer::sum);
```

Decision between Sequential stream & Parallel stream

- Use parallel stream if you have at least one thousand elements.
- We should never parallel stream for operations like limit() & findFirst(). Note that parallel streams are not always faster than sequential stream.
- We can use parallel stream for findAny() operation.
- Take into account how well the data structure underlying the stream decomposes. For instance, an ArrayList can be split much more efficiently than a LinkedList. So we can use parallel stream for ArrayList but not for LinkedList.

Thank you!!