Streams:

To perform operations on the cached data, rather getting the data again from the db and it also increase the performance of the application.

Code:

```
import java.util.*;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class StreamFunction {
 public static void main(String[] args) {
 List<Employee> employeeList = Arrays.asList(
 new Employee(1, "Virat", "Cricket", 10.32),
 new Employee(2, "Dhoni", "Cricket", 50.43),
 new Employee(3, "Sangeeta", "Actor", 60.89),
 new Employee(4, "Sweety", "Actor", 90.29),
 new Employee(5, "Ronaldo", "FootBall", 40.71),
 new Employee(6, "Messi", "FootBall", 30.93)
 );

 System.out.println("Original List " + employeeList);
 // Filter & Collect
 List<Employee> startsWithAList = employeeList.stream().filter(emp ->
emp.getName().startsWith("S")).collect(Collectors.toList());
 System.out.println("FilteredList " + startsWithAList + " \n");

 // Predicate & ForEach
 Predicate<Employee> pred = p -> p.getDept().equalsIgnoreCase("cricket") ||
p.getName().contains("o");
 employeeList.stream().filter(pred).forEach(System.out::println);

 // Map & Reduce
 Double totalSal = employeeList.stream().map(sal -> sal.getSalary() /
2).reduce(10.0, (s1, s2) -> s1 + s2);
 System.out.println("\nTotal Salary : " + totalSal.intValue());
```

```
// Limit & Skip
employeeList.stream().limit(5).skip(3).map(sal -> sal.getSalary() *
2).forEach(System.out::println);

//Summary Statistics
IntStream intStream = IntStream.of(10, 12, 14, 16, 18, 20);
IntSummaryStatistics sumStats = intStream.summaryStatistics();
System.out.println("Summary Stats " + sumStats);

//FlatMap
List<Integer> list1 = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);
List<List<Integer>> finalList = Arrays.asList(list1, list2);
System.out.println("Org List " + finalList);
System.out.println("FlatMap List " + finalList.stream().flatMap(fm ->
fm.stream()).collect(Collectors.toList()));

//Grouping By & Partitioning By & Counting
Map<String, List<Employee>> groupby =
employeeList.stream().collect(Collectors.groupingBy(des -> des.getDept()));
Map<Boolean, List<Employee>> partitionby =
employeeList.stream().collect(Collectors.partitioningBy(id -> id.getId() % 2 ==
0));
System.out.println("Grouping By " + groupby + "\n" + "Partitioning By " +
partitionby);
Map<String, Long> freqCount =
employeeList.stream().collect(Collectors.groupingBy(des -> des.getDept(),
Collectors.counting()));
System.out.println("Frequency Count " + freqCount);

// Distinct & Count
long count = employeeList.stream().filter(e -> e.getName().length() ==
5).distinct().count();
System.out.println("Count " + count);

//Sort
List<Employee> l4 = employeeList.stream().sorted((s1, s2) -> (int) (s1.getSalary()
```

```
- s2.getSalary())).collect(Collectors.toList());

 Predicate<Employee> p1 = e -> e.getName().startsWith("S");
 Predicate<Employee> p2 = e -> e.getId() < 2;
 Predicate<Employee> p3 = e -> e.getDept().contains("oo");

 //AnyMatch & AllMatch & NoneMatch
 boolean bool1 = employeeList.stream().allMatch(p1);
 boolean bool2 = employeeList.stream().anyMatch(p2);
 boolean bool3 = employeeList.stream().noneMatch(p3);
 System.out.println(" allMatch -> " + bool1 + " anyMatch -> " + bool2 + "
noneMatch -> " + bool3);

 //FindAny & FindFirst
 List<String> list = Arrays.asList("A", "B", "C", "D");
 Optional<String> result = list.stream().findAny();
 Optional<String> result1 = list.stream().findFirst();
 System.out.println("findAny " + result + "\nfindFirst " + result1);

 //Min & Max For Integer
 List<Integer> listOfIntegers = Arrays.asList(1, 2, 3, 4, 56, 7, 89, 10);
 Integer max = listOfIntegers.stream().mapToInt(v ->
v).max().orElseThrow(NoSuchElementException::new);
 Integer min = listOfIntegers.stream().mapToInt(v ->
v).min().orElseThrow(NoSuchElementException::new);
 System.out.println("Max " + max + "\t Min " + min);

 //Min & Max For Sorting & Method Reference
 Optional<Employee> minEmp =
employeeList.stream().min(Comparator.comparing(Employee::getSalary));
 System.out.println(minEmp.get());
 Optional<Employee> maxEmp =
employeeList.stream().max(Comparator.comparing(Employee::getSalary));
 System.out.println(maxEmp.get());

 //Stream.of & toArray & foreachordered
 Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
 Object[] array = stream.toArray();
```

```
for (Object val : array) {
if (val instanceof Integer)
System.out.print(val + "\t");
}

System.out.println();
List<String> list6 = Arrays.asList("AB", "DE", "FG", "YZ");
list6.stream().forEachOrdered(System.out::println);
}
}
```

Output:

Original List [Employee{id=1, name='Virat', dept='Cricket', salary=10.32}, Employee{id=2, name='Dhoni', dept='Cricket', salary=50.43}, Employee{id=3, name='Sangeeta', dept='Actor', salary=60.89}, Employee{id=4, name='Sweety', dept='Actor', salary=90.29}, Employee{id=5, name='Ronaldo', dept='FootBall', salary=40.71}, Employee{id=6, name='Messi', dept='FootBall', salary=30.93}]

FilteredList [Employee{id=3, name='Sangeeta', dept='Actor', salary=60.89}, Employee{id=4, name='Sweety', dept='Actor', salary=90.29}]

Employee{id=1, name='Virat', dept='Cricket', salary=10.32}

Employee{id=2, name='Dhoni', dept='Cricket', salary=50.43}

Employee{id=5, name='Ronaldo', dept='FootBall', salary=40.71}

Total Salary : 151

180.58

81.42

Summary Stats IntSummaryStatistics{count=6, sum=90, min=10, average=15.000000, max=20}

Org List [[1, 2, 3, 4, 5], [10, 20, 30, 40, 50]]

FlatMap List [1, 2, 3, 4, 5, 10, 20, 30, 40, 50]

Grouping By {Cricket=[Employee{id=1, name='Virat', dept='Cricket', salary=10.32}, Employee{id=2, name='Dhoni', dept='Cricket', salary=50.43}], Actor=[Employee{id=3, name='Sangeeta', dept='Actor', salary=60.89}, Employee{id=4, name='Sweety', dept='Actor',

salary=90.29}], FootBall=[Employee{id=5, name='Ronaldo', dept='FootBall', salary=40.71}, Employee{id=6, name='Messi', dept='FootBall', salary=30.93}]}

Partitioning By {false=[Employee{id=1, name='Virat', dept='Cricket', salary=10.32}, Employee{id=3, name='Sangeeta', dept='Actor', salary=60.89}, Employee{id=5, name='Ronaldo', dept='FootBall', salary=40.71}], true=[Employee{id=2, name='Dhoni', dept='Cricket', salary=50.43}, Employee{id=4, name='Sweety', dept='Actor', salary=90.29}, Employee{id=6, name='Messi', dept='FootBall', salary=30.93}]}

Frequency Count {Cricket=2, Actor=2, FootBall=2}

Count 3

allMatch -> false anyMatch -> true noneMatch -> false

findAny Optional[A]

findFirst Optional[A]

Max 89 Min 1

Employee{id=1, name='Virat', dept='Cricket', salary=10.32}

Employee{id=4, name='Sweety', dept='Actor', salary=90.29}

1 2 3 4 5

AB

DE

FG

YZ

Optional Class:

It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

Code:

```
import java.util.Optional;
```

```
public class OptionalClass {
 int data = 0;

 OptionalClass(int data) {
 this.data = data;
 }

 public static void main(String[] args) {
 OptionalClass obj1 = new OptionalClass(20);
 OptionalClass obj2 = new OptionalClass(50);

 Optional<OptionalClass> op1 = Optional.ofNullable(obj1);
 Optional<OptionalClass> op2 = Optional.ofNullable(obj2);

 OptionalClass ob1 = op1.orElse(new OptionalClass(0));
 OptionalClass ob2 = op2.orElse(new OptionalClass(0));

 System.out.println(ob1.data + ob2.data);
 }
}
```

Output:

70

Method Reference:

Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference

Types of Method References

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

Code:

```
public interface IStaticMethodRef {
 public void say();
}


public interface IInstanceMethodRef {
 public void say();
}


public class Method_Ref {

 Method_Ref() {
 System.out.println("In the constructor ref");
 }

 public static void staticMethod() {
 System.out.println("Static Method for ref");
 }

 public static void instMethod() {
 System.out.println("Instance Method for ref");
 }

 public static void main(String[] args) {
 IStaticMethodRef iStaticMethodRef = Method_Ref::staticMethod;
 iStaticMethodRef.say();

 IInstanceMethodRef iInstanceMethodRef = Method_Ref::instMethod;
 iInstanceMethodRef.say();

 IInstanceMethodRef iInstanceMethodRefCons = Method_Ref::new;
 iInstanceMethodRefCons.say();
 }
}
```

Output:

Static Method for ref

Instance Method for ref

In the constructor ref

Local Date and Local Time:

LocalDateTime is an immutable date-time object that represents a date-time, often viewed as year-month-day-hour-minute-second. Other date and time fields, such as day-of-year, day-of-week and week-of-year, can also be accessed. Time is represented to nanosecond precision. For example, the value "2nd October 2007 at 13:45.30.123456789" can be stored in a LocalDateTime.

Code:

```
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.util.Set;

public class LocalTimeDate {

 public static void main(String[] args) {

 LocalDateTime currentTime = LocalDateTime.now();
 System.out.println("\nCurrent Time is "+currentTime);

 LocalDate localDate = currentTime.toLocalDate();

 DateTimeFormatter format = DateTimeFormatter.ofPattern("MM/dd/yy");
 String strDate = localDate.format(format);
 System.out.println("\nModified local date format: "+strDate);
 LocalTime localTime = currentTime.toLocalTime();

 System.out.println("\nCurrent Date "+localDate);

 int month = localDate.getMonthValue();
 int day = localDate.getDayOfMonth();
```

```
int year = localDate.getYear();
System.out.println("\nDate is "+day+"-"+month+"-"+year);

int hr = currentTime.getHour();
int min = currentTime.getMinute();
int sec = currentTime.getSecond();
System.out.println("\nCurrent Time is "+hr+":"+min+":"+sec);

LocalTime lt1 = LocalTime.now(ZoneId.of("Singapore"));
LocalTime lt2 = LocalTime.now(ZoneId.of("America/Panama"));

System.out.println("Current Time is Singapore is "+lt1);
System.out.println("Current Time is America/Panama is "+lt2);

Set s = ZoneId.getAvailableZoneIds();
System.out.println("List of zones...");

s.forEach(System.out::println);
 }
}
```

Output:

Current Time is 2023-12-30T20:18:14.181255600

Modified local date format: 12/30/23

Current Date 2023-12-30

Date is 30-12-2023

Current Time is 20:18:14

Current Time is Singapore is 22:48:14.197342900

Current Time is America/Panama is 09:48:14.197342900

List of zones...

Asia/Aden

America/Cuiaba

Etc/GMT+9

Etc/GMT+8

Africa/Nairobi

America/Marigot

Asia/Aqtau

Pacific/Kwajalein

America/El_Salvador

Asia/Pontianak

Africa/Cairo

Pacific/Pago_Pago

Africa/Mbabane

Asia/Kuching

Pacific/Honolulu

Pacific/Rarotonga

America/Guatemala

Australia/Hobart

Asia/Tehran

WET

Europe/Astrakhan

Africa/Juba

America/Campo_Grande

America/Belem

Etc/Greenwich

Asia/Saigon

America/Ensenada

Pacific/Midway

America/Jujuy

Africa/Timbuktu

America/Bahia

America/Goose_Bay

Europe/Athens

US/Pacific

Europe/Monaco

Nashorn JavaScript Engine:

We can execute JavaScript code at Java Virtual Machine. Nashorn is introduced in JDK 8 to replace existing JavaScript engine i.e. Rhino. Nashorn is far better than Rhino in term of performance.

Code:

```
import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
import java.io.FileNotFoundException;
import java.io.FileReader;

public class NashornDemo {

 public static void main(String[] args) throws ScriptException,
NoSuchMethodException, FileNotFoundException {
 ScriptEngineManager mgr = new ScriptEngineManager();
 ScriptEngine engine = mgr.getEngineByName("Nashorn");

 engine.eval(new
FileReader("C:\\Users\\Wissen\\IdeaProjects\\Java8\\src\\Test.js"));
```

```
 Invocable inv = (Invocable) engine;

 inv.invokeFunction("test");
 System.out.println(inv.invokeFunction("sum",10,20));
 }
}
```

Test.js:

```
for(int i=0;i<10;++i){
 print("From java script: "+i)
}

function test(){
 print("From the test().. in Javascript...")
}

function sum(x,y){
 return x+y;
}
```

Output:

From java script : 0

From java script : 1

From java script : 2

From java script : 3

From java script : 4

From java script : 5

From java script : 6

From java script : 7

From java script : 8

From java script : 9

From the test() in Javascript...

30

Default & Static Method In Interface:

We can execute JavaScript code at Java Virtual Machine. Nashorn is introduced in JDK 8 to replace existing JavaScript engine i.e. Rhino. Nashorn is far better than Rhino in term of performance.

Code:

```
public interface DefaultStaticExampleInterface {

 void doShow();
 default void show() {
 System.out.println("In Java 8- default method - DefaultStaticExampleInterface");
 }
 static void display() {
 System.out.println("In DefaultStaticExampleInterface I");
 }
}

public class DefaultStaticExampleClass implements
DefaultStaticExampleInterface {

 public static void main(String args[]) {

 DefaultStaticExampleClass dsed = new DefaultStaticExampleClass();
 dsed.doShow();
 // Call interface static method on Interface
 DefaultStaticExampleInterface.display();
 DefaultStaticExampleClass defaultStaticExampleClass = new
DefaultStaticExampleClass();

 // Call default method on Class
 defaultStaticExampleClass.show();
 }
```

```
 @Override
 public void doShow() {
 System.out.println("Do Show ");
 }
}
```

Output:

Do Show

In DefaultStaticExampleInterface I

In Java 8- default method – DefaultStaticExampleInterface