CSIT121
Object Oriented Design and Programming

Lesson 9

Lambda and Stream

Lambda expression

- Represents an anonymous method
  - a shorthand notation for implementing a functional interface,
- Similar to an anonymous inner class.

Lambda expression

- Provides a clear and concise way to represent one method interface using an expression.
- Provide the implementation of an interface which has **functional interface**.
- Java lambda expression is treated as a function, so compiler does not create .class file.
- **Functional interface**.
  - An interface which has only **one abstract method** is called functional interface.

Java Lambda Expression Syntax

- Consisted of three components.
  1. **Argument-list:** It can be empty or non-empty as well.
  2. **Arrow-token:** It is used to link arguments-list and body of expression.
  3. **Body:** It contains expressions and statements for lambda expression.

(argument-list) -> {body}

Java Lambda Expression Syntax

- No Parameter Syntax

```
() -> {
        //Body of no parameter lambda
}
```

- One Parameter Syntax

```
(p1) -> {
        //Body of single parameter lambda
}
```

- Two Parameter Syntax

```
(p1,p2) -> {
        //Body of multiple parameter lambda
}
```

Java Lambda Expression Example: No Parameter

```
//functional interface
interface SayNothing {
    public String nothingToSay();
}

public class Example1 {
    public static void main(String[] args) {
        //Lambda Expression that implements Interface SayNothing
        SayNothing s = () -> {
            return "I have nothing to say.";
        };

        //call nothingToSay
        System.out.println(s.nothingToSay());
    }
}
```

Java Lambda Expression Example: With single Parameter

```
//functional interface
interface SomethingToSay {
    public String say(String something);
}

public class Example2 {
    public static void main(String[] args) {

        //Lambda Expression that implements Interface SomethingToSay
        //Lambda expression with single parameter.
        SomethingToSay s1 = (name) -> {
            return "Hello, " + name;
        };
        System.out.println(s1.say("John"));

        //Lambda Expression that implements Interface SomethingToSay
        //You can omit function parentheses
        SomethingToSay s2 = name -> {
            return "Good day, " + name;
        };
        System.out.println(s2.say("Peter"));

        //Lambda Expression that implements Interface SomethingToSay
        //can omit {} if only one line of code.
        //return is implicit in this case
        SomethingToSay s3 = name -> ("How are you "+name);
        System.out.println(s3.say("Bob"));
    }
}
```

Java Lambda Expression Example: With multiple Parameters

```
interface Adder{
    int add(int a,int b);
}

public class Example3{
    public static void main(String[] args) {

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

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

        //Lambda expression with return keyword.
        //Multiple Statements must be in {} with return
        Adder ad3=(int a,int b)->{
                        int r = a + b;
                        return r;
                    };
        System.out.println(ad3.add(100,200));
    }
}
```

Java Lambda Expression Example: With ArrayList

```
import java.util.*;
import java.util.stream.Stream;

public class Example4{
    public static void main(String[] args) {

        ArrayList<String> animals=new ArrayList<String>();
        animals.add("cat");
        animals.add("lion");
        animals.add("horse");
        animals.add("dog");

        //using lambda to iterate through arraylist
        animals.forEach(
            (animal)->System.out.println(animal)
        );
```

anonymous method

Java Lambda Expression Example: With ArrayList to filter data

Continue from previous Example4

```
System.out.println();
//using lambda to iterate through arraylist
//with condition to filter data
animals.forEach(
                        //Multiple Statements must be in {}
        (animal)-> { if(animal.length()==3) {
                        System.out.println(animal);
                    }
                   }
);
```

Java Lambda Expression Example: With ArrayList to filter date **with stream**

Continue from previous Example4

```
        System.out.println();
        //using stream and lambda to filter data
        Stream<String> filtered_data = animals.stream().filter(s -> s.length() == 3);

        filtered_data.forEach(
                s -> System.out.println(s)
        );
    }
}
```

Java Lambda Expression Example: Passing multiple statements

```
interface Talk{
    String say(String message);
}

public class Example5{
    public static void main(String[] args) {

        //Pass multiple statements in lambda expression
        //Multiple Statements must be in {} with return
        Talk person = (message)-> {
            String str1 = "I would like to say, ";
            String str2 = str1 + message;
            return str2;
        };
        System.out.println(person.say("Java is easy!"));
    }
}
```

Java Lambda Expression Example: Sorting with Comparator

```java
import java.util.*;

public class Example6{
    public static void main(String[] args) {
        ArrayList<Product> list=new ArrayList<Product>();

        //Adding Products
        list.add(new Product(1,"Laptop",2500));
        list.add(new Product(3,"Keyboard",30));
        list.add(new Product(2,"Mouse",15));

        System.out.println("Sort by name...");

        Collections.sort(list,(p1,p2)->{
            //need to return an int
            return p1.getName().compareTo(p2.getName());
        });

        for(Product p:list){
            System.out.println(p.getId()+" "+p.getName()+" "+p.getPrice());
        }
```

```java
public class Product{

    private int id;
    private String name;
    private double price;

    public Product(int id, String name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public int getId() {
        return id;
    }

    public double getPrice() {
        return price;
    }
}
```

Java Lambda Expression Example: Sorting with Comparator

Continue from previous Example6

```
        System.out.println("Sort by price...");

        Collections.sort(list,(p1,p2)->{
            //need to return an int
            return (int)(p1.getPrice()-p2.getPrice());
        });

        for(Product p:list){
            System.out.println(p.getId()+" "+p.getName()+" "+p.getPrice());
        }

    }
}
```

Java Lambda Expression Example: Filtering **with Stream**

```
import java.util.*;
import java.util.stream.Stream;

public class Example7{
    public static void main(String[] args) {
        ArrayList<Product> list=new ArrayList<Product>();
        list.add(new Product(1,"Laptop",2500));
        list.add(new Product(2,"Keyboard",30));
        list.add(new Product(3,"Mouse",15));
        list.add(new Product(4,"Macbook",3500));
        list.add(new Product(5,"Router",230));
        list.add(new Product(6,"Laptop Bag",150));

        //using stream and lambda to filter data
        Stream<Product> filtered_data = list.stream().filter(p -> p.getPrice() > 2000);

        // using lambda to iterate through the stream
        filtered_data.forEach(
                product -> System.out.println(product.getName()+": "+product.getPrice())
        );
    }
}
```

Streams

- Streams are
  - objects of classes that implement interface Stream (from the package java.util.stream) or
  - one of the specialized stream interfaces for processing collections of int, long or double values.
- Together with lambdas, streams enable you to perform tasks on collections of elements—often from an array or collection object.

IntStream Operations

- IntStream (package java.util.stream)
  - A specialized stream for manipulating int values.
  - The techniques shown in this example also apply to LongStreams and DoubleStreams for long and double values, respectively.

Stream Operations

`forEach`	Performs processing on every element in a stream (e.g., display each element).
Reduction operations*—*Take all values in the stream and return a single value	
`average`	Calculates the *average* of the elements in a numeric stream.
`count`	Returns the *number of elements* in the stream.
`max`	Locates the *largest* value in a numeric stream.
`min`	Locates the *smallest* value in a numeric stream.
`reduce`	Reduces the elements of a collection to a *single value* using an associative accumulation function (e.g., a lambda that adds two elements).

IntStream Operations Examples

```
import java.util.stream.IntStream;

public class IntStreamOperation1
{
   public static void main(String[] args)
   {
      int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

      //Display original values
      System.out.print("Original values: ");
      IntStream.of(values)
               .forEach(value -> System.out.print(value+" "));
      System.out.println();

      //count, min, max, sum and average of the values
      long count = IntStream.of(values).count();
      int min = IntStream.of(values).min().getAsInt();
      int max = IntStream.of(values).max().getAsInt();
      int sum = IntStream.of(values).sum();
      double ave = IntStream.of(values).average().getAsDouble();

      System.out.println("Count:"+ count);
      System.out.println("Min:"+ min);
      System.out.println("Max:"+ max);
      System.out.println("Sum:"+ sum);
      System.out.println("Ave:"+ ave);
```

```
Original values: 3 10 6 1 4 8 2 5 9 7
Count:10
Min:1
Max:10
Sum:55
Ave:5.5
```

IntStream Operations Examples

```
//sum of values with reduce method
int sumReduce =  IntStream.of(values).reduce(0, (x, y) -> x + y);
System.out.println("Sum Reduce:"+ sumReduce);

//sum of squares of values with reduce method
//sum of squares = 3*3+10*10+6*6+1*1....+7*7
int sumOfSquareReduce = IntStream.of(values).reduce(0, (x, y) -> x + y * y);
System.out.println("Sum of squares via reduce method:"+ sumOfSquareReduce);

//product of values with reduce method
//product = 3*10*6*1....*7
int productReduce = IntStream.of(values).reduce(1, (x, y) -> x * y);
System.out.println("Product via reduce method:"+ productReduce);
```

```
Sum Reduce:55
Sum of squares via reduce method:385
Product via reduce method:3628800
```

IntStream Operations

Streams Operations	
filter	Results in a stream containing only the elements that satisfy a condition.
sorted	Results in a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream.
map	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type) —e.g., mapping numeric values to the squares of the numeric values. The new stream has the same number of elements as the original stream.

IntStream Operations Examples

```
    // even values displayed in sorted order
    System.out.println();
    System.out.println("Even values displayed in sorted order: ");
    IntStream.of(values)
             .filter(value -> value % 2 == 0)
             .sorted()
             .forEach(value -> System.out.println(value));
    System.out.println();

    // odd values multiplied by 10 and displayed in sorted order
    System.out.println("Odd values multiplied by 10 displayed in sorted order: ");
    IntStream.of(values)
             .filter(value -> value % 2 != 0)
             .map(value -> value * 10)
             .sorted()
             .forEach(value -> System.out.println(value));
  }
}
```

```
Even values displayed in sorted order:
2
4
6
8
10

Odd values multiplied by 10 displayed in sorted order:
10
30
50
70
90
```

Arrays and Streams

- Array's stream method can be used to create a Stream from an array of objects.
- Performs filtering and sorting on a Stream<Integer>, using the same techniques

Arrays and Streams Operations Examples

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

public class ArraysAndStreams1 {

    public static void main(String[] args) {

        //Note: the type is Integer not int
        Integer[] values = { 2, 9, 5, 0, 3, 7, 1, 4, 8, 6 };

        // display original values
        List<Integer> original = Arrays.asList(values);
        System.out.println("Original values: " + original);

        // sort values in ascending order with streams
        List<Integer> sorted = Arrays.stream(values)
                                     .sorted()
                                     .collect(Collectors.toList());
        System.out.println("Sorted values: "+sorted);

        // values greater than 4
        List<Integer> greaterThan4 = Arrays.stream(values)
                                           .filter(value -> value > 4)
                                           .collect(Collectors.toList());
        System.out.println("Values greater than 4: "+greaterThan4);

        // filter values greater than 4 then sort the results
        List<Integer> greaterThan4Sorted = Arrays.stream(values)
                                                 .filter(value -> value > 4)
                                                 .sorted()
                                                 .collect(Collectors.toList());
        System.out.printf("Sorted values greater than 4: "+greaterThan4Sorted);
    }
}
```

```
Original values: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]
Sorted values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Values greater than 4: [9, 5, 7, 8, 6]
Sorted values greater than 4: [5, 6, 7, 8, 9]
```

Stream<String> Operations

- Next, we see how to use the same stream operations but on a Stream<String>.
- In addition, we demonstrate case-insensitive sorting and sorting in descending order.

Stream<String> Operations Example

```
Original strings:
[Red, orange, Yellow, green, Blue, indigo, Violet]
strings in uppercase: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
strings greater than m sorted ascending: [orange, Red, Violet, Yellow]
strings greater than m sorted descending: [Yellow, Violet, Red, orange]
```

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

public class ArraysAndStreams2
{
   public static void main(String[] args)
   {
      String[] strings =
         {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};

      // display original strings
      List<String> original = Arrays.asList(strings);
      System.out.println("Original strings: ");
      System.out.println(original);

      // strings in uppercase
      List<String> uppercase = Arrays.stream(strings)
                                     .map(String::toUpperCase)
                                     .collect(Collectors.toList());
      System.out.println("strings in uppercase: "+uppercase);

      // strings greater than "m" (case insensitive) sorted ascending
      List<String> greaterThanMsortedAsc =  Arrays.stream(strings)
              .filter(s -> s.compareToIgnoreCase("m") > 0)
              .sorted(String.CASE_INSENSITIVE_ORDER)
              .collect(Collectors.toList());
      System.out.println("strings greater than m sorted ascending: "+greaterThanMsortedAsc);

      // strings greater than "m" (case insensitive) sorted descending
      List<String> greaterThanMsortedDes = Arrays.stream(strings)
                                                 .filter(s -> s.compareToIgnoreCase("m") > 0)
                                                 .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
                                                 .collect(Collectors.toList());

      System.out.println("strings greater than m sorted descending: "+greaterThanMsortedDes);
   }
}
```

Stream<String> Operations Example

Lambda	
String::toUpperCase	Method reference for an instance method of a class. Creates a one parameter lambda that invokes the instance method on the lambda's argument and returns the method's result.

Stream<Employee>

- Class Employee represents an employee with
  - first name, last name, salary and department
  - and provides methods for manipulating these values.

```
public class Employee {
    private String firstName;
    private String lastName;
    private double salary;
    private String department;

    public Employee(String firstName, String lastName,
                    double salary, String department) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.salary = salary;
        this.department = department;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public double getSalary() {
        return salary;
    }

    public void setDepartment(String department) {
        this.department = department;
    }

    public String getDepartment() {
        return department;
    }

    public String getName() {
        return firstName + " " + lastName;
    }

    public String toString() {
        return firstName + " " + lastName + " $" + salary + " " + department;
    }
}
```

Stream<Employee>

- The following examples demonstrates various lambda and stream capabilities using a Stream<Employee>.

Stream<Employee> Operation examples

```
Complete Employee list:
Adrain Tan $5000.0 IT
Bob Lim $7600.0 IT
Charles Goh $3587.5 Sales
Darren Yap $4700.77 Marketing
```

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

public class ProcessingEmployees
{
   public static void main(String[] args)
   {
      // initialize array of Employees
      Employee[] employees = {
         new Employee("Adrain", "Tan", 5000, "IT"),
         new Employee("Bob", "Lim", 7600, "IT"),
         new Employee("Charles", "Goh", 3587.5, "Sales"),
         new Employee("Darren", "Yap", 4700.77, "Marketing")};

      // get List view of the Employees
      List<Employee> list = Arrays.asList(employees);

      // display all Employees
      System.out.println("Complete Employee list:");
      list.stream().forEach(System.out::println);
```

Stream<Employee> Operation examples

- Creates a Stream<Employee>, then uses Stream method forEach to display each Employee's String representation.
- The instance method reference System.out::println implicitly calls class Employee's toString method to get the String representation and print the returned String.

Lambda	
System.out::println	Method reference for an instance method that should be called on a specific object. Creates a one-parameter lambda that invokes the instance method on the specified object—passing the lambda's argument to the instance method—and returns the method's result.

Stream<Employee> Operation examples

```
System.out.println();
// Predicate that returns true for salaries in the range $4000-$6000
Predicate<Employee> fourToSixThousand =
    e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);

// Display Employees with salaries in the range $4000-$6000
// sorted into ascending order by salary
System.out.println("Employees earning $4000-$6000 per month sorted by salary:");
list.stream()
    .filter(fourToSixThousand)
    .sorted(Comparator.comparing(Employee::getSalary))
    .forEach(System.out::println);
```

```
Employees earning $4000-$6000 per month sorted by salary:
Darren Yap $4700.77 Marketing
Adrain Tan $5000.0 IT
```

Stream<Employee> Operation examples

```
// Predicate that returns true for salaries in the range $4
Predicate<Employee> fourToSixThousand =
   e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);
```

- Predicate<Employee> is defined with a lambda. Defining lambdas in this manner enables you to reuse them multiple times.

```
.filter(fourToSixThousand)
.sorted(Comparator.comparing(Employee::getSalary))
.forEach(System.out::println);
```

- Sorts by salary the Employees that remain in the stream.
- To specify a Comparator for salaries,
  - use the Comparator interface's static method comparing.
  - The method reference Employee::getSalary that's passed as an argument is converted by the compiler into an object that implements the Function interface.
  - This Function is used to extract a value from an object in the stream for use in comparisons.
  - Method comparing returns a Comparator object that calls getSalary on each of two Employee objects, then returns a negative value if the firstEmployee's salary is less than the second, 0 if they're equal

Stream<Employee> Operation examples

```
System.out.println();
// Display first Employee with salary in the range $4000-$6000
Employee emp = list.stream()
                   .filter(fourToSixThousand)
                   .findFirst()
                   .get();

System.out.println("First employee who earns $4000-$6000:"+emp);
```

```
First employee who earns $4000-$6000:Adrain Tan $5000.0 IT
```

Stream<Employee> Operation examples

```
.findFirst()
```

- findFirst—an operation that processes the stream pipeline and terminates processing as soon as the first object from the stream pipeline is found.

Stream search operation	
findFirst	Finds the first stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.

Stream<Employee> Operation examples

```
System.out.println();
// Functions for getting first and last names from an Employee
Function<Employee, String> byFirstName = Employee::getFirstName;
Function<Employee, String> byLastName = Employee::getLastName;

// Comparator for comparing Employees by first name then last name
Comparator<Employee> lastThenFirst =
   Comparator.comparing(byLastName).thenComparing(byFirstName);

// sort employees by last name, then first name
System.out.println("Employees in ascending order by last name then first:");
list.stream()
    .sorted(lastThenFirst)
    .forEach(System.out::println);
```

```
Employees in descending order by last name then first:
Darren Yap $4700.77 Marketing
Adrain Tan $5000.0 IT
Bob Lim $7600.0 IT
Charles Goh $3587.5 Sales
```

Stream<Employee> Operation examples

```
// Functions for getting first and last names from an Employee
Function<Employee, String> byFirstName = Employee::getFirstName;
Function<Employee, String> byLastName = Employee::getLastName;
```

- First, use Functions to create a Comparator that first compares two Employees by last name, then compares them by first name.

Stream<Employee> Operation examples

```
Comparator<Employee> lastThenFirst =
    Comparator.comparing(byLastName).thenComparing(byFirstName);
```

- Then use Comparator method *comparing* to create a Comparator that calls Function byLastName on an Employee to get its last name.
- On the resulting Comparator, call Comparator method *thenComparing* to create a Comparator that first compares Employees by last name and, if the last names are equal, then compares them by first name.

Stream<Employee> Operation examples

```
// sort employees by last name, then first name
System.out.println("Employees in ascending order by last name then first:");
list.stream()
    .sorted(lastThenFirst)
    .forEach(System.out::println);
```

- Finally use the lastThenFirst Comparator to sort the Employees in ascending order, then display the results.

Stream<Employee> Operation examples

```
System.out.println();
// sort employees in descending order by last name, then first name
System.out.println("Employees in descending order by last name then first:");
list.stream()
    .sorted(lastThenFirst.reversed())
    .forEach(System.out::println);
```

```
Employees in descending order by last name then first:
Darren Yap $4700.77 Marketing
Adrain Tan $5000.0 IT
Bob Lim $7600.0 IT
Charles Goh $3587.5 Sales
```

Reuse the Comparator, call its reversed method to indicate that the Employees should be sorted in descending order by last name, then first name.

Stream<Employee> Operation examples

```
System.out.println();
// display unique employee last names sorted
System.out.println("Unique employee last names:");
list.stream()
      .map(Employee::getLastName)
      .distinct()
      .sorted()
      .forEach(System.out::println);
```

```
Unique employee last names:
Goh
Lim
Tan
Yap
```

Stream<Employee> Operation examples

```
.map(Employee::getLastName)
```

- Maps the Employees to their last names using the instance method reference
- Employee::getName as method map's Function argument.
- The result is a Stream<String>.

Stream operation	
map	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type)—e.g., mapping numeric values to the squares of the numeric values. The new stream has the same number of elements as the original stream.

Stream<Employee> Operation examples

```
.distinct()
```

- Calls Stream method distinct on the Stream<String> to eliminate any duplicate String objects in a Stream<String>.

Stream operation	
distinct	Results in a stream containing only the unique elements.

Stream<Employee> Operation examples

```
.sorted()
```

- Sorts the unique last names.

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

- Finally, performs a forEach operation that processes the stream
- pipeline and outputs the unique last names in sorted order.

Stream<Employee> Operation examples

```
System.out.println();
// display only first and last names
System.out.println("Employee names in order by last name then first name:");
list.stream()
    .sorted(lastThenFirst)
    .map(Employee::getName)
    .forEach(System.out::println);
```

```
Employee names in order by last name then first name:
Charles Goh
Bob Lim
Adrain Tan
Darren Yap
```

Stream<Employee> Operation examples

```
Comparator<Employee> lastThenFirst =
   Comparator.comparing(byLastName).thenComparing(byFirstName);

t.stream()
 .sorted(lastThenFirst)
```

- Reuse lastThenFirst comparator created previously to sort the stream.

```
.map(Employee::getName)
```

- Map the Employees to Strings with Employee instance method getName.

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

- Display the sorted names using forEach operation

Stream<Employee> Operation examples

```
System.out.println();
// group Employees by department
System.out.println("Employees by department:");
Map<String, List<Employee>> groupedByDepartment =
        list.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment));

groupedByDepartment.forEach(
   (department, employeesInDepartment) ->
   {
      System.out.println(department);
      employeesInDepartment.forEach(
         employee -> System.out.println(employee)
      );
   }
);
```

```
Employees by department:
Sales
Charles Goh $3587.5 Sales
IT
Adrain Tan $5000.0 IT
Bob Lim $7600.0 IT
```

Stream<Employee> Operation examples

```
System.out.println();
// group Employees by department
System.out.println("Employees by department:");
Map<String, List<Employee>> groupedByDepartment =
        list.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment));

groupedByDepartment.forEach(
   (department, employeesInDepartment) ->
   {
      System.out.println(department);
      employeesInDepartment.forEach(
         employee -> System.out.println(employee)
      );
   }
);
```

```
Employees by department:
Sales
Charles Goh $3587.5 Sales
IT
Adrain Tan $5000.0 IT
Bob Lim $7600.0 IT
Marketing
Darren Yap $4700.77 Marketing
```

Stream<Employee> Operation examples

```
Map<String, List<Employee>> groupedByDepartment =
        list.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment));
```

- Use the Collectors static method groupingBy, to group the Employee by department.
- It returns a Map with the String (Key) and List<Employee> (Value).
- The String (Key) is the department and List<Employee> (Value) consists of all the Employees object belonging to that department.

Stream<Employee> Operation examples

```
groupedByDepartment.forEach(
   (department, employeesInDepartment) ->
   {
      System.out.println(department);
      employeesInDepartment.forEach(
         employee -> System.out.println(employee)
      );
   }
);
```

- Map method forEach performs an operation on each of the Map's key–value pairs.
- Key is the department; Value is the List<Employee>.
- In the above case, it prints out the department (Key).
- And the all the Employee objects (Value) in the List.

Stream<Employee> Operation examples

```
System.out.println();
// sum of Employee salaries with DoubleStream sum method
double sumStream =  list.stream()
                  .mapToDouble(Employee::getSalary)
                  .sum();
System.out.println("Sum of Employees' salaries (via sum method): "+sumStream);
```

```
Sum of Employees' salaries (via sum method): 20888.27
```

- mapToDouble() function maps objects to double values and returns a DoubleStream.
- Then apply sum() function on the DoubleStream.

Stream<Employee> Operation examples

```
System.out.println();
// calculate sum of Employee salaries with Stream reduce method
double sumReduce =    list.stream()
                          .mapToDouble(Employee::getSalary)
                          .reduce(0, (value1, value2) -> value1 + value2);
System.out.println("Sum of Employees' salaries (via reduce method): "+sumReduce);
```

```
Sum of Employees' salaries (via reduce method): 20888.27
```

- mapToDouble() function maps objects to double values and returns a DoubleStream.
- Then use reduce() operation to sum up all the doubles in the stream.

Stream<Employee> Operation examples

```
System.out.println();
// average of Employee salaries with DoubleStream average method
double aveStream = list.stream()
                       .mapToDouble(Employee::getSalary)
                       .average()
                       .getAsDouble();
System.out.printf("Average of Employees' salaries:"+aveStream);
```

```
Average of Employees' salaries:5222.0675
```

- mapToDouble() function maps objects to double values and returns a DoubleStream.
- Then apply the average() function on the DoubleStream.