Lambda Expressions

Lambda Expression

- A lambda expression is an anonymous function
- A function that doesn’t have a name and doesn’t belong to any class
- Labda Expressions are classes implementing functional interfaces
- Functional interface is the one which has only one method
- To check a functional interface, java 8 introduces new annotation @FunctionalInterface

```
@FunctionalInterface
interface Operation{
  int getResult(int a, int b);
}
```

Anonymous classes

```
interface Operation{
  int getResult(int a, int b);
}
Operation o = new Operation(){
        public getResult(int a, int b){
                return a + b;
        }
}

System.out.println(o.getResult(30,40));
```

Lambda Expression

```
interface Operation{
  int getResult(int a, int b);
}
Operation o =  (int a, int b) -> {return a + b; };
// can also be written as
Operation o = (a,b) -> a+b;

System.out.println(o.getResult(30,40));
```

Lambda Expression Defined

Argument List	Arrow Token	Body
`(int x, int y)`	`->`	`x + y`

Basic Lambda examples

```
(int x, int y) -> x + y
(x, y) -> x + y

(x, y) -> { system.out.println(x + y);}

(String s) -> s.contains("word")
s -> s.contains("word")
```

Lambda Expressions

```
() -> System.out.println(this)

 (String str) -> System.out.println(str)

 str -> System.out.println(str)

 (String s1, String s2) -> { return s2.length() - s1.length(); }

 (s1, s2) -> s2.length() - s1.length()
```

Lambda Expressions as Variables

```
interface  StringTest{
    boolean test(String a, String b);
}
```

```
void testAll(String [ ] strArr,  String  testStr, StringTest  check ){


    for(String x : strArr){
         if(check.test(x,  testStr){
              System.out.println(x);
         }
    }
}
```

```
String  [ ]   names={ ………………};
 testAll(names,  “ramana”,  (s,t)-> s.contains(t));

 StringTest tst =(a,b)-> a.equalsIgnoreCase(b);
  testAll(names,  “ramana”,   tst );
```

Example 1: Lambda with no parameter

```
Interface  Message {
     //A method with no parameter
    public String saySomeThing();
}

public class Example {

   public static void main(String args[]) {
        // lambda expression
     Message  msg = () ->   "Hello“;
        System.out.println(msg.saySomeThing());
    }
}
```

Example 2: Lambda with one parameter

```
interface  Incrementer {
     //A method with single parameter
     int increment(int a);
}

public class Example {

   public static void main(String args[]) {
        // lambda expression with single parameter num
      Incrementor inc = (num) -> num+5;      //  num -> num + 5     also fine
        System.out.println(inc.increment (22));
    }
}
```

Example 3: Lambda with two parameters

```
interface  Operator {
     //A method with  two parameters
     int operate(int a, int b);
}

public class Example {

   public static void main(String args[]) {
        // lambda expression with two parameters
    Operator add = (a, b) ->   a + b;
    Operator sub = (a, b) ->   a - b;
     System.out.println( add.operate(30,20));      //  50
     System.out.println( sub.operate(30,20));      //  10
    }
}
```

Method Reference

- Method reference is a shorthand notation of a lambda expression to call a method

- If your lambda expression is like this:

  str -> {System.out.println(str)}
- then you can replace it with a method reference like this:

  System.out::println

- The :: operator is used in method reference to separate the class or object from the method name

Example: Method Reference

```
interface Display {
      void print(String s);
}

public class Main {

     public static void main(String[] args) {

              Display d = System.out::println ;
       // Normal Lambda
      //   Display  d  =  ( str  )->  {System.out.println( str );};
             d.print("Hello World");
     }
}
```

Method Reference : Types

- Method reference to an instance method of an object – object::instanceMethod

  lambda : (args) -> objRef.method(args)

- Method reference to a static method of a class – className::staticMethod

  lambda : (args) -> className.staticMethod(args)

- Method reference to an instance method of an arbitrary object of a particular type – className::instanceMethod

- Method reference to a constructor – Class::new

Method Reference to instance method of an object

```
interface Inter{
    void display(String str);
}
class Test {
    public void showMessage(String s){
       System.out.println(s.toUpperCase());
    }
}
```

```
public class  Test {
     public static void main(String[] args) {
           Test test = new Test();
           Inter ref = test::showMesssage;
     // Inter ref = (str) -> {test.showMessage(str);
           ref.display(“hello”);
       }
   }
```

Method Reference to static method of a class

```
interface Math {
    int operate(int x, int y);
}

class Test {
     static int add(int x, int y) {
          return x + y;
     }
    static int subtract(int x, int y) {
        return x - y;
    }
}
```

```
public class Test {

    public static void main(String[] args) {
          Math m1 = Test::add;
//  Math m1 = (a,b) -> Test.add(a,b);
          Math m2 = Test::subtract;
//  Math m2 = (a,b) -> Test.subtract(a,b);
         System.out.println(m1.operate(30, 20));
         System.out.println(m2.operate(30, 20));
    }
}
```

Method Reference to method of a particular type

- Works for lambda expression like the following:

  (obj, args) -> obj.instanceMethod(args)

- Where an instance of an object is passed, and one of its methods is executed with some optional parameter(s)

```
class Shipment {
        double price;
        Shipment(double c) {   price = c;   }
        double getCost(double weight) {
          return price * weight;
        }
}

interface Calc {
        double get(Shipment s, double weight);
}
```

```
public class Test {
        public static void main(String[] args) {
              Shipment s  =  new Shipment(30.5);
              Calc c = Shipment::getCost;
 // Calc c = (sh, w)-> sh.getCost(w) ;
              System.out.println( c.get(s, 5));
        }
}
```

Method Reference to method of a particular type…

```
public class Test {
        static double getTotal(List<Shipment> list, Calc c) {
              double total = 0;
              for (Shipment s : list) {
                   total = total + c.get(s, 5);
              }
              return total;
         }

        public static void main(String[] args) {
              Shipment s[] = { new Shipment(30.5), new Shipment(2.5), … };
              List<Shipment> list = Arrays.asList(s);
              double dd = getTotal(list, Shipment::getCost);
              System.out.println(dd);
        }
}
```

Method Reference to method of a particular type…

```
 // Array sorting example

String[ ]   names = { “Kumar", "Ramesh", “Nidhi", “John", “Ismail", “Sudhir”};
Arrays.sort(names, String::compareTo);
// same as   Arrays.sort(names,   (s,t)-> s.compareTo( t ) )
for(String  name : names) {
    System.out.println(s);
}
```

Method Reference to a constructor

- Works for lambda expression like the following:

  (args**) -> new ClassName(**args**)**

- That can be turned into the following method reference:

  ClassName::new

- The only thing this lambda expression does is to create a new object and we just reference a constructor of the class with the keyword new.
- Like in the other cases, arguments (if any) are not passed in the method reference

Method Reference to constructor

```
interface Convertor{
      Person convert (String name);
}

class Person{
   String name;
   public Person(String name) {
         this.name = name;
   }
}
```

```
public class Test {
      public static void main(String[] args) {
              String name="Ramana";
              **Convertor con = Person::new;**
     // Convertor con = (str) -> new Person(str);
              Person p = con.convert(name);
              System.out.println(p.name);
      }
  }
```

Built-in Functional Interfaces

Functional interfaces

Java 8 comes with several new functional interfaces in the package, java.util.function.

Function<T,R> - takes an object of type T and returns R.

Supplier<T> - just returns an object of type T.

Predicate<T> - returns a boolean value based on input of type T.

Consumer<T> - performs an action with given object of type T.

BiFunction - like Function but with two parameters.

BiConsumer - like Consumer but with two parameters

It also comes with several corresponding interfaces for primitive types, such as:

IntConsumer

IntFunction<R>

IntPredicate

IntSupplier

Predicate

```
package java.util.function;


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

```
 Predicate<Employee> validEmp =  t -> t.getSalary() >10000;


Employee emp =  ….;
 if(validEmp.test(emp){
    System.out.println(“Valid”);
```

Consumer

```
package java.util.function;
 public interface Consumer<T> {
        public void accept(T t);
}
```

```
Consumer<Sales> saleTxn = t ->    System.out.println(
                                          "Id: " + t.getTxnId()
                                       + " Buyer: " + t.getBuyer().getName());
 Sales s = ……;
 saleTxn.accept(s);
```

Supplier

```
 package java.util.function;
 public interface Supplier<T> {
    public T get();
}
```

```
  Supplier<Employee>  s1 =  ( )  ->    new Employee(100, ….  );
 Supplier<Employee>  s2 =  ( )  ->    new Employee(200, ….  );

  System.out.println( s1.get().getName());

Employee e1 = s2.get();
```

Function

```
package java.util.function;
 public interface Function<T,R> {
     public R apply(T t);
}
```

```
Function<Sales, String>  saleFn= t ->    t.getBuyer().getName();
Sales s = ……;
String name = saleFn.apply(s);
System.out.println(name);
```

Primitive Interfaces

- Primitive versions of all main interfaces
  - Will see these a lot in method calls
- Return a primitive
  - Example: `ToDoubleFunction`
- Consume a primitive
  - Example: `DoubleFunction`
- Why have these?
  - Avoids auto-boxing and unboxing

Return a Primitive Type

```
 package java.util.function;

 public interface ToDoubleFunction<T> {
     public double applyAsDouble(T t);
 }

 public ToIntFunction<T> {
    public int applyAsInt(T t);
}
```

Process a Primitive Type

```
package java.util.function;
 public interface DoubleFunction<R> {
    public R apply(double value);
 }

 public interface IntFunction<R> {
   public R apply(int value);
 }
```

Binary Types

```
package java.util.function;
 public interface BiPredicate<T, U> {
    public boolean test(T t, U u);
 }
 public interface BiFunction<T, U, R> {
    public R apply(T t, U u);
 }
 public interface BiConsumer<T, U> {
    public void accept(T t, U u);
 }
```

Collections – forEach()

- In Java 8 both Collection and Map provide forEach() method

- Collection:

  void   forEach(Consumer<? super T> action)

- Map

  void forEach(BiConsumer<? super K, ? super V> action)

forEach () example

```
List<String> list = Arrays.asList("one","two","three","four");

list.forEach(System.out::println);
list.forEach( s->{ System.out.println(s.length());});

Map <Integer, Emp> map=new TreeMap<>();

map.put(100, new Emp(100,"ramana", 5000));
map.put(200, new Emp(200,"kishore", 15000));
map.put(300, new Emp(300,"ramana", 8000));
map.put(400, new Emp(400,"neeraj", 7500));

map.forEach( (k,v)-> {System.out.println(v);});
 map.forEach( (k,v)-> {System.out.println(v.getSalary());});
```

Streams

- Java 8 brings new abilities to work with Collections in the form of a brand new Stream API

- Supports more functional approach to handle collections

- The Stream API offers easy **filtering**, **counting**, and **mapping** of collections, as well as different ways to get slices and subsets of information out of them

- Stream API allows shorter and more elegant code for working with collections

Streams Sample

```
class Book {
  String name;
  int year;
  Author author;
}
```

```
class Author {
  String name;
  int countOfBooks;
}
```

```
List<Book>  books= …..
```

Requirement : show all the author names of books published after 1995

normal

```
for (Book book : books) {
   if (book.author != null && book.year > 1995){
       System.out.println(book.author.name);
   }
}
```

<u>Using streams</u>

```
books.stream()
        .filter(book -> book.getYear() > 1995)
        .map(Book::getAuthor)
        .filter(Objects::nonNull)
        .map(Author::getName)
        .forEach(System.out::println);
```

How to work with Streams

1. Create a stream

2. Perform **intermediate operations** on the initial stream to transform it into another stream and so on

   There can be more than one intermediate operation

3. Perform **terminal operation** on the final stream to get the result. In the above example, the count() operation is terminal operation.

How to create stream

- Using stream() method of collections

```
Stream<Customer> stream =  list.stream();
```

- BufferedReader has lines() method which returns stream

```
BufferedReader br = new BufferedReader(new FileReader(…));
br.lines().forEach(System.out::println);
```

- Stream.of() method to create a stream out of array

```
Stream<String> tream = Stream.of(“one”,”two”,”three”);
stream.map(String::toUpperCase).forEach(System.out::println);
```

Types of Operations

- Intermediate
  - `filter()  map()  peek()`

- Terminal
  - `forEach() count() sum() average()  min() max() collect()`

- Terminal short-circuit
  - `findFirst()  findAny()  anyMatch() allMatch()  noneMatch()`

Extracting Data with Map

- Used to transform data into different type

```
map(Function<? super T,? extends R> mapper)
```

- A map takes one `Function` as an argument.
  - A `Function` takes one generic and returns something else.
- Primitive versions of `map`

```
mapToInt()    which returns IntStream
mapToLong()   which returns LongStream
mapToDouble() which returns DoubleStream
```

map() example

```
List<Employee> empList  =  - - - -

empList.stream()
        .map(Emp::getName)
        .forEach(  name -> System.out.println(name) );  // display all names

empList.stream()
                .map(Emp::getName)
                .map(String::toUpperCase)
                .forEach(System.out::println);  // display all names in uppercase

 double totalSalary = empList.stream().mapToDouble(Emp::getSalary).sum();
  System.out.println(totalSalary);     // display total salary
```

Taking a Peek

```
peek(Consumer<? super T> action)
```

- The peek method performs the operation specified by the lambda expression and returns the elements to the stream
- Great for printing intermediate results or to take any other action without disturbing the stream flow

peek() example

```
List<Employee> empList  =  - - - -



 double totalSalary =
             empList.stream()
             .peek(s -> System.out.println(s.getName())
             .mapToDouble(Emp::getSalary).sum();
```

Search Methods: Overview

`Optional<T> findFirst()`

- Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty

`boolean allMatch(Predicate)`

- Returns `true` if all the elements meet the criteria

`boolean noneMatch(predicate)`

- Returns `true` if none of the elements meet the criteria

- All of the above are short-circuit terminal operations.

Optional Class

- **`Optional<T>`**
  - A container object that may or may not contain a non-null value
  - If a value is present, **`isPresent()`** returns true.
  - **`get()`** returns the value.
  - `Part of` **`java.util`** `package`

- Optional primitives
  - **`OptionalDouble OptionalInt OptionalLong`**

Search methods: example

```
 List<Employee> empList  =  - - - -

 Optional<Emp> data = empList.stream()
            .filter(e -> e.getSalary() > 10000)
            .findFirst();
 System.out.println(data.get());

if (empList.stream().allMatch(e -> e.getSalary() > 10000))
           System.out.println("all elements match");

if (empList.stream().noneMatch(e -> e.getSalary() > 100000))
          System.out.println("No employee gets more than 100000");
```

Search Methods

- Nondeterministic search methods
  - Used for nondeterministic cases. In effect, situations where parallel is more effective.
  - Results may vary between invocations

`Optional<T> findAny()`

- Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty
- Results may vary when performed in parallel

`boolean anyMatch(Predicate)`

- Returns true if any elements meet the criteria
- Results may vary when performed in parallel.

Stream Data Methods

`long count()`

- Returns the count of elements in this stream

`Optional<T> max(Comparator<? super T> comparator)`

- Returns the maximum element of this stream according to the provided `Comparator`

`Optional<T> min(Comparator<? super T> comparator)`

- Returns the minimum element of this stream according to the provided `Comparator`

Data methods: example

```
List<Employee> empList  =  - - - -

 long count = empList.stream()
           .filter(e -> e.getSalary() > 10000).
           count();
System.out.println(count);



Optional<Emp>  e =
              empList.stream().max((x,y)->(int)(x.getSalary()-y.getSalary()));
 System.out.println( e.get()  );
```

Performing Calculations

`OptionalDouble average()`

- Returns an optional describing the arithmetic mean of elements of this stream
- Returns an empty optional if this stream is empty

`int/long/double sum()`

- Returns the sum of elements in this stream
- Return type depends on the stream

- Methods are found in primitive streams:
  - **`DoubleStream`, `IntStream`, `LongStream`**

Calculations : example

```
List<Employee> empList  =  - - - -

 OptionalDouble  opt  = empList.stream()
           .mapToDouble(Emp::getSalary)
           .average();
 System.out.println(opt.getAsDouble());

double   totSalary =
              empList.stream().mapToDouble(Emp::getSalary).sum();
 System.out.println(totSalary);
```

Sorting

`Stream<T> sorted()`

- Returns a stream consisting of the elements sorted according to natural order

`Stream<T> sorted(Comparator<? super T> comparator)`

- Returns a stream consisting of the elements sorted according to the `Comparator`

Sorting : example

```
List<Employee> empList  =  - - - -

 empList.stream()
           .mapToDouble(Emp::getSalary)
           .sorted()
           .forEach(System.out::println);

 empList.stream()
           .sorted(   (x,y) -> (int)( x.getSalary()  -  y.getSalary()  )  )
           .forEach(System.out::println);
```

Comparator Updates

`comparing(Function<? super T,? extends U> keyExtractor)`

- Allows to specify any field for comparison based on a method reference or lambda
- Primitive versions of the Function also supported

`thenComparing(Comparator<? super T> other)`

- Specify additional fields for sorting

`reversed()`

- Reverse the sort order by appending to the method chain.

Comparator : example

```
 List<Employee> empList  =  - - - -

 empList.stream()
     .sorted( Comparator.comparing(Emp::getSalary) )
     .forEach(System.out::println);

empList.stream()
     .sorted( Comparator.comparing(Emp::getDepartment)
                        .thenComparing(Emp::getName );
     .forEach(System.out::println);

Optional<Emp>  e =  empList.stream().max( Emp::getSalary );
 System.out.println( e.get()  );
```

Saving Data from a Stream

`collect(Collector<? super T,A,R> collector)`

- Allows to save the result of a stream to a new data structure
- Relies on the `Collectors` class
- Examples
  - `stream().collect(Collectors.toList());`
  - `stream().collect(Collectors.toMap());`

Collectors Class

averagingDouble(ToDoubleFunction<? super T> mapper)

- Produces the arithmetic mean of a double-valued function applied to the input elements

groupingBy(Function<? super T,? extends K> classifier)

- A "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a map

joining()

- Concatenates the input elements into a String, in encounter order

partitioningBy(Predicate<? super T> predicate)

- Partitions the input elements according to a Predicate

Collect : example

```
List<Employee> empList  =  - - - -

 List<Emp> sortedList =empList.stream()
                                          .sorted( Comparator.comparing(Emp::getSalary))
                                          .collect(Collectors.toList());
  sortedList.forEach(System.out::println);

  double avgSalary = empList.stream()
                                          .collect(Collectors.averagingDouble(Emp::getSalary));

  double totSalary = empList.stream()
                                          .collect(Collectors.summingDouble(Emp::getSalary));
```

Grouping & Partitioning : example

```
List<Employee> empList  =  - - - -

String names = empList.stream().map(Emp::getName).collect(Collectors.joining(", "));
System.out.println(names);

Map<Integer, List<Emp>> byDept  = empList.stream()
                                      .collect(Collectors.groupingBy(Emp::getDeptNo));

Map<Integer, Double> totalByDept = empList.stream()
                                          .collect(Collectors.groupingBy(Emp::getDeptNo,
                                                   Collectors.summingDouble(Emp::getSalary)));

Map<Boolean, List<Emp>> salGroup =   empList.stream()
                                      .collect(Collectors.partitioningBy(e->e.getSalary()>10000));
```