Syntax:

The **basic syntax of a lambda expression** is

|  |
| --- |
| either  (parameters) -> expression  or  (parameters) -> { statements; }  or  () -> expression |

Let’s see some examples as well:

|  |
| --- |
| (int a, int b) ->    a \* b                           // takes two integers and returns their multiplication  (a, b)          ->   a - b                           // takes two numbers and returns their difference  () -> 99                                             // takes no values and returns 99  (String a) -> System.out.println(a)                  // takes a string, prints its value to the console, and returns nothing  a -> 2 \* a                                       // takes a number and returns the result of doubling it  c -> { //some complex statements }               // takes a collection and do some procesing |

Let’s identify some **rules which will help us in writing lambda expression**:

1. A lambda expression can have zero, one or more parameters.
2. The type of the parameters can be explicitly declared or it can be inferred from the context.
3. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.
4. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a.
5. The body of the lambda expressions can contain zero, one or more statements.
6. If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.
7. **What is a functional interface?**
8. **Single Abstract Method interfaces** (SAM Interfaces) is not a new concept. It means **interfaces with only one single method**. In java, we already have many examples of such SAM interfaces. From java 8, they will also be **referred as functional interfaces as well**. Java 8, enforces the rule of single responsibility by marking these interfaces with a new annotation i.e. **@FunctionalInterface**.
9. For example, new definition of Runnable interface is like this:

|  |
| --- |
| @FunctionalInterface  public interface Runnable {      public abstract void run();  } |

new Thread(new Runnable() {

    @Override

    public void run() {

        System.out.println("Efficiency");

    }

}).start();

If we use the lambda expression for this task then code will be :

|  |
| --- |
| new Thread(              () ->   {                          System.out.println("My Runnable");                      }           ).start(); |

**Few examples of Lambda expressions**

I am listing out some code samples which you can read and analyze to how a lambda expression can be used in day to day programming.

**1) Iterating over a List and perform some operations**

|  |
| --- |
| List<String> pointList = new ArrayList();  pointList.add("1");  pointList.add("2");    pointList.forEach(p ->  {                              System.out.println(p);                              //Do more work                          }                   ); |

**2) Create a new runnable and pass it to thread**

|  |
| --- |
| new Thread(      () -> System.out.println("My Runnable");  ).start(); |

**3) Sorting employees objects by their name**

|  |
| --- |
| public class LambdaIntroduction {      public static void main (String[] ar){            Employee[] employees  = {                new Employee("David"),                new Employee("Naveen"),                new Employee("Alex"),                new Employee("Richard")};              System.out.println("Before Sorting Names: "+Arrays.toString(employees));            Arrays.sort(employees, Employee::nameCompare);            System.out.println("After Sorting Names "+Arrays.toString(employees));        }  }    class Employee {    String name;      Employee(String name) {      this.name = name;    }      public static int nameCompare(Employee a1, Employee a2) {      return a1.name.compareTo(a2.name);    }      public String toString() {      return name;    }  }    Output:    Before Sorting Names: [David, Naveen, Alex, Richard]  After Sorting Names [Alex, David, Naveen, Richard] |

**4) Adding an event listener to a GUI component**

|  |
| --- |
| JButton button =  new JButton("Submit");  button.addActionListener((e) -> {      System.out.println("Click event triggered !!");  }); |

## String concatenation (CSV) with join()

**import** java.time.ZoneId;

**public** **class** StringJoinDemo {

**public** **static** **void** main(String[] args){

String joined = String.*join*("/","usr","local","bin");

System.***out***.println(joined);

String ids = String.*join*("\n ", ZoneId.*getAvailableZoneIds*());

System.***out***.println(ids);

}

}

## ****New classes to represent local date and timezone****

The new classes intended to replace Date class are **LocalDate**, **LocalTime** and **LocalDateTime**.  
**LocalDate:** The LocalDate class represents a date. There is no representation of a time or time-zone.

|  |
| --- |
| LocalDate localDate = LocalDate.now();  System.out.println(localDate.toString());                //2013-05-15  System.out.println(localDate.getDayOfWeek().toString()); //WEDNESDAY  System.out.println(localDate.getDayOfMonth());           //15  System.out.println(localDate.getDayOfYear());            //135  System.out.println(localDate.isLeapYear());              //false  System.out.println(localDate.plusDays(12).toString());   //2013-05-27 |

**LocalTime:** The LocalTime class represents a time. There is no representation of a date or time-zone.

//LocalTime localTime = LocalTime.now();     //toString() in format 09:57:59.744

LocalTime localTime = LocalTime.of(12, 20);

System.out.println(localTime.toString());    //12:20

System.out.println(localTime.getHour());     //12

System.out.println(localTime.getMinute());   //20

System.out.println(localTime.getSecond());   //0

System.out.println(localTime.MIDNIGHT);      //00:00

System.out.println(localTime.NOON);          //12:00

**LocalDateTime:** The LocalDateTime class represents a date-time. There is no representation of a time-zone.

LocalDateTime localDateTime = LocalDateTime.now();

System.out.println(localDateTime.toString());      //2013-05-15T10:01:14.911

System.out.println(localDateTime.getDayOfMonth()); //15

System.out.println(localDateTime.getHour());       //10

System.out.println(localDateTime.getNano());       //911000000

**New classes to represent timestamp and duration**

For representing the specific timestamp ant any moment, the class needs to be used is Instant. The **Instant** class represents an instant in time to an accuracy of nanoseconds. Operations on an Instant include comparison to another Instant and adding or subtracting a duration.

|  |
| --- |
| Instant instant = Instant.now();  System.out.println(instant.toString());                                 //2013-05-15T05:20:08.145Z  System.out.println(instant.plus(Duration.ofMillis(5000)).toString());   //2013-05-15T05:20:13.145Z  System.out.println(instant.minus(Duration.ofMillis(5000)).toString());  //2013-05-15T05:20:03.145Z  System.out.println(instant.minusSeconds(10).toString()); |

**Duration** class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

|  |
| --- |
| Duration duration = Duration.ofMillis(5000);  System.out.println(duration.toString());     //PT5S    duration = Duration.ofSeconds(60);  System.out.println(duration.toString());     //PT1M    duration = Duration.ofMinutes(10);  System.out.println(duration.toString());     //PT10M    duration = Duration.ofHours(2);  System.out.println(duration.toString());     //PT2H    duration = Duration.between(Instant.now(), Instant.now().plus(Duration.ofMinutes(10)));  System.out.println(duration.toString());  //PT10M |

**Predicate Example**

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.List;

**import** test.Employee;

**import** **static** test.EmployeePredicates.\*;

**public** **class** TestEmployeePredicates {

**public** **static** **void** main(String[] args){

Employee e1 = **new** Employee(1,23,"M","Rick","Beethovan");

Employee e2 = **new** Employee(2,13,"F","Martina","Hengis");

Employee e3 = **new** Employee(3,43,"M","Ricky","Martin");

Employee e4 = **new** Employee(4,26,"M","Jon","Lowman");

Employee e5 = **new** Employee(5,19,"F","Cristine","Maria");

Employee e6 = **new** Employee(6,15,"M","David","Feezor");

Employee e7 = **new** Employee(7,68,"F","Melissa","Roy");

Employee e8 = **new** Employee(8,79,"M","Alex","Gussin");

Employee e9 = **new** Employee(9,15,"F","Neetu","Singh");

Employee e10 = **new** Employee(10,45,"M","Naveen","Jain");

List<Employee> employees = **new** ArrayList<Employee>();

employees.addAll(Arrays.*asList*(**new** Employee[]{e1,e2,e3,e4,e5,e6,e7,e8,e9,e10}));

System.***out***.println(*filterEmployees*(employees, *isAdultMale*()));

System.***out***.println(*filterEmployees*(employees, *isAdultFemale*()));

System.***out***.println(*filterEmployees*(employees, *isAgeMoreThan*(35)));

//Employees other than above collection of "isAgeMoreThan(35)" can be get using negate()

System.***out***.println(*filterEmployees*(employees, *isAgeMoreThan*(35).negate()));

}

}

**package** test;

**public** **class** Employee {

**public** Employee(Integer id, Integer age, String gender, String fName, String lName){

**this**.id = id;

**this**.age = age;

**this**.gender = gender;

**this**.firstName = fName;

**this**.lastName = lName;

}

**private** Integer id;

**private** Integer age;

**private** String gender;

**private** String firstName;

**private** String lastName;

@Override

**public** String toString() {

**return** **this**.id.toString()+" - "+**this**.age.toString(); //To change body of generated methods, choose Tools | Templates.

}

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

**public** String getGender() {

**return** gender;

}

**public** **void** setGender(String gender) {

**this**.gender = gender;

}

**public** String getFirstName() {

**return** firstName;

}

**public** **void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **void** setLastName(String lastName) {

**this**.lastName = lastName;

}

}

**package** test;

**import** java.util.List;

**import** java.util.function.Predicate;

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

**public** **class** EmployeePredicates

{

**public** **static** Predicate<Employee> isAdultMale() {

**return** p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");

}

**public** **static** Predicate<Employee> isAdultFemale() {

**return** p -> p.getAge() > 18 && p.getGender().equalsIgnoreCase("F");

}

**public** **static** Predicate<Employee> isAgeMoreThan(Integer age) {

**return** p -> p.getAge() > age;

}

**public** **static** List<Employee> filterEmployees (List<Employee> employees, Predicate<Employee> predicate) {

**return** employees.stream().filter( predicate ).collect(Collectors.<Employee>*toList*());

}

}

**import java.util.Arrays;**

**import java.util.LinkedList;**

**import java.util.List;**

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

**public class FilterExample {**

**public static void main(String[] args) {**

**/\*List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");**

**//get count of empty string**

**int count =(int) strings.stream().filter(string -> !string.isEmpty()).count();**

**System.out.println(count);**

**//strings.stream().filter(string -> strings.)**

**Random random = new Random();**

**random.ints().limit(10).forEach(System.out::println);**

**\*/**

**List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");**

**List<String> filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.toList());**

**System.out.println("Filtered List: " + filtered);**

**String mergedString = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.joining("\n "));**

**System.out.println("Merged String: " + mergedString);**

**List<String> s=Arrays.asList("a","","b");**

**List<String> s1=new LinkedList<>();**

**int count=0;**

**for(String ss:s)**

**{**

**if(ss.isEmpty())**

**{**

**count++;**

**}**

**else**

**{**

**s1.add(ss);**

**}**

**}**

**System.out.println(s1);**

**System.out.println(count);**

**List ll = s.stream().filter(ss->!ss.isEmpty()).collect(Collectors.toList());**

**System.out.println(ll);**

**}**

**}**