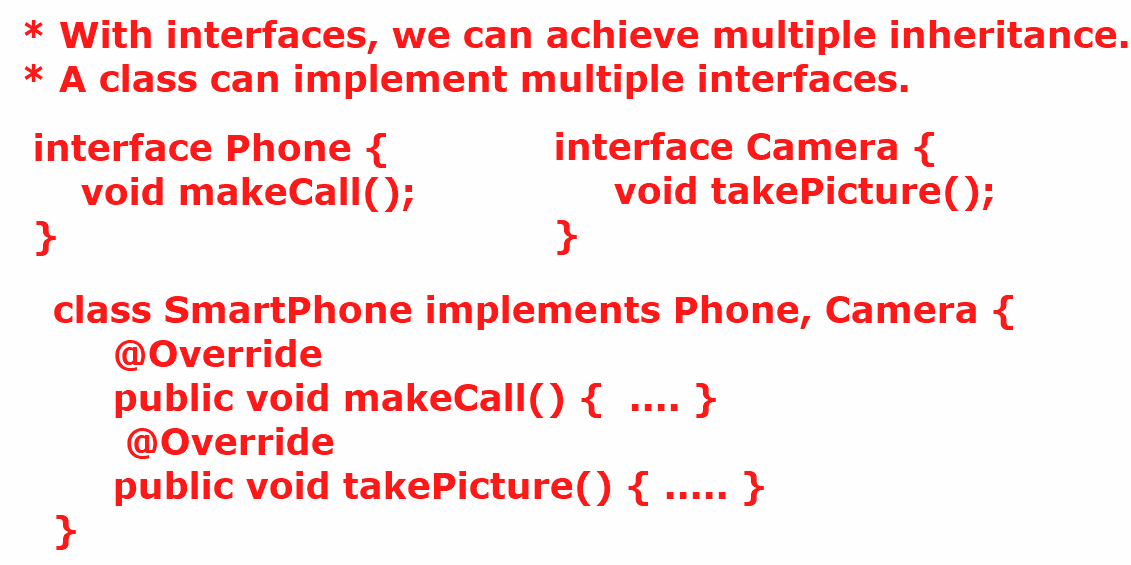
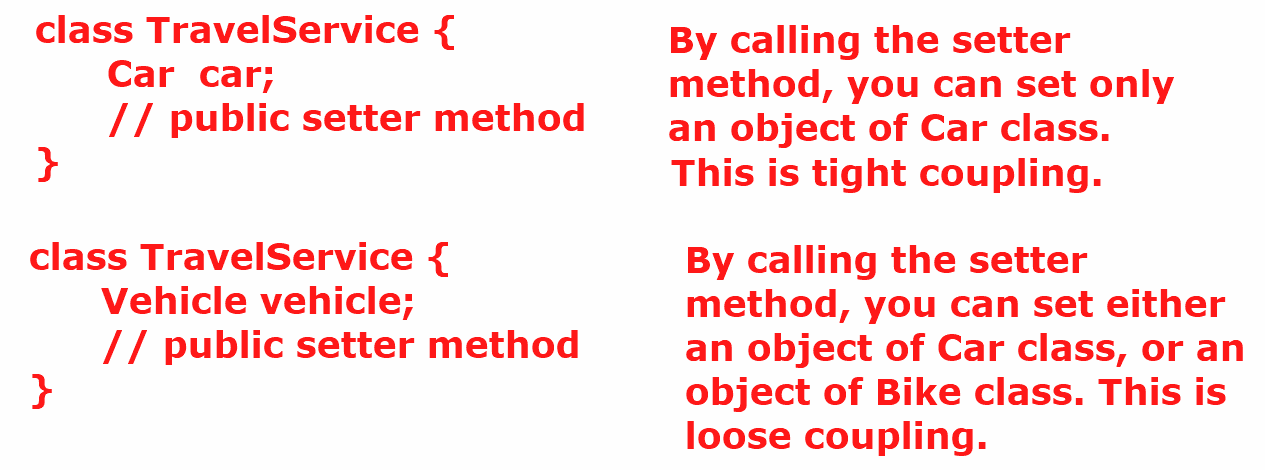


* **Here, there is an ambiguity that whose m1() should be called. This is called diamond problem.**



* **suppose, if the two interfaces have the same abstract method then the class has to override the abstract method for only once. So, there is no diamond problem in interfaces.**

**\***

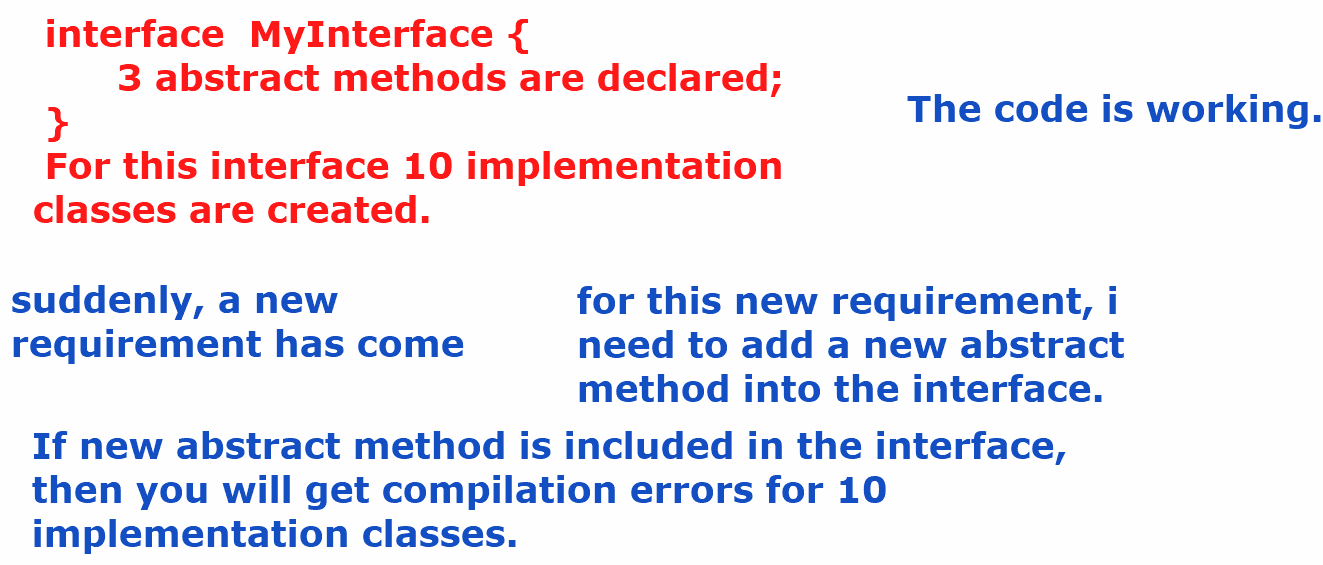
* **An interface acts as a contract for the classes who implements that interface. It means, an interface tells the behaviours that a class has to implement.**

**How many types of interfaces in Java?**

**Ans: 1. Normal interface ( > 1 abstract method)**

**2. Marker interface ( empty)**

**3. Functional interface. ( single abstract method)**

****

* **The solution is reach out all the 10 implementation classes and override the new abstract method there.**
* **what if there are 100’s of implementation classes for an interface?**
* **Reaching out to each and every implementation class to override a new abstract method will be a time consuming task and also error-prone.**
* **what is the solution provided in Java 8?**
* **Ans: default and static methods in interface.**
* **Before Java 8, an interface can have public static final variables and public abstract methods.**
* **From Java 8+, an interface can have public static final variables, public abstract methods, default methods, static methods and private methods also.**
* public interface MyInterface {  
    
   public static final int *MAX\_USERS* = 100;  
    
   public abstract void m1();  
    
   public abstract void m2();  
    
   public default String sayHello() {  
   return "Hello";  
   }  
   public static void m4() {  
   System.*out*.println("static method");  
   }  
   private void m5() {  
   System.*out*.println("private method");  
   }  
    
  }
* **In an interface, if you have added a default method then its implementation classes can use that method as it is or they can override that method, if required.**

**IoTDevice.java**

public interface IoTDevice {  
 void connect();  
 void disconnect();  
 default String sendHeartBeat(String id) {  
 return "Device : "+ id + " is Active";  
 }  
}

**SmartLight.java**

public class SmartLight implements IoTDevice {  
 @Override  
 public void connect() {  
 System.*out*.println("SmartLight connected");  
 }  
 @Override  
 public void disconnect() {  
 System.*out*.println("SmartLight disconnected");  
 }  
  
}

**DoorSensor.java**

public class DoorSensor implements IoTDevice {  
 @Override  
 public void connect() {  
 System.*out*.println("Door sensor connected");  
 }  
 @Override  
 public void disconnect() {  
 System.*out*.println("Door sensor disconnected");  
 }  
}

**Main.java**

public class Main {  
 public static void main(String[] args) {  
 IoTDevice smartLight = new SmartLight();  
 smartLight.connect();  
 smartLight.disconnect();  
 System.*out*.println(smartLight.sendHeartBeat("sm001"));  
  
 IoTDevice doorSensor = new DoorSensor();  
 doorSensor.connect();  
 doorSensor.disconnect();  
 System.*out*.println(doorSensor.sendHeartBeat("ds001"));  
 }  
}

**output:**

**SmartLight connected**

**SmartLight disconnected**

**Device : sm001 is Active**

**Door sensor connected**

**Door sensor disconnected**

**Device : ds001 is Active**

* **In an interface, if you have added a static method, then the implementation classes of that interface can not override that method. Because the static method belongs to interface itself.**

**IoTDevice.java**

public interface IoTDevice {  
 void connect();  
  
 void disconnect();  
  
 default String sendHeartBeat(String id) {  
 return "Device : "+ id + " is Active";  
 }  
  
 static void isValidDevice(String id) {  
 if (id.startsWith("sm") || id.startsWith("ds"))  
 System.*out*.println(id + " : valid device");  
 else  
 System.*out*.println(id + " : invalid device");  
  
 }  
}

* **Suppose, if there two interfaces added with same default method and a class implements both the interfaces then how to resolve ambiguity.**
* **Answer: override the default method in the class then make a call to the default methods of the interfaces**

**with syntax, InterfaceName.super.methodName();**

**example:**

**MyInter1.java**

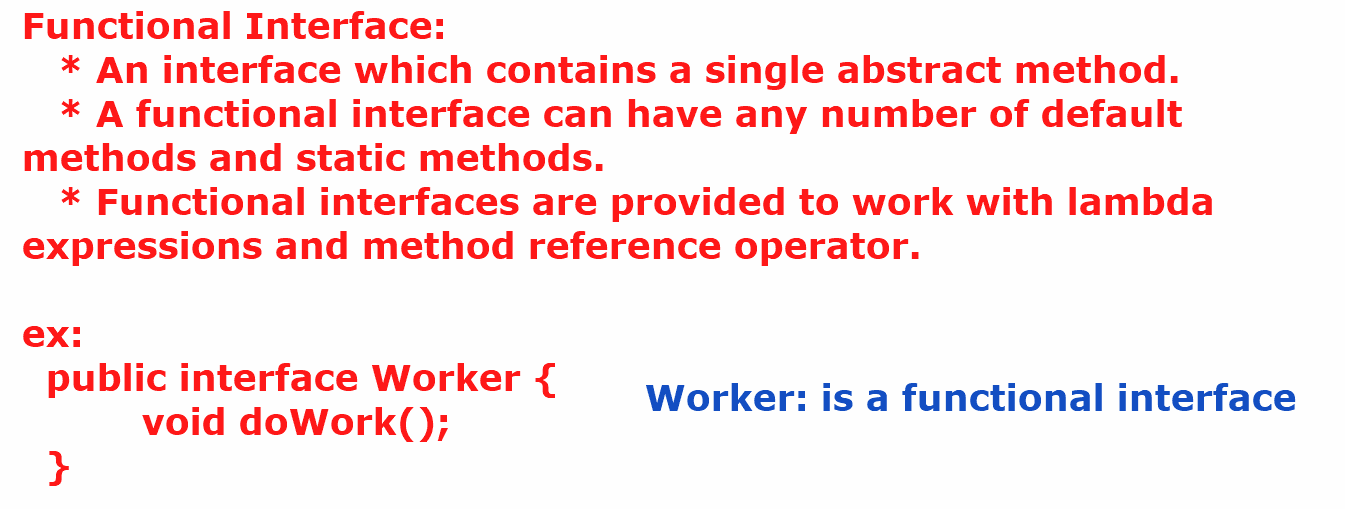
public interface MyInter1 {  
 default void m1() {  
 System.*out*.println("MyInter1 m1() called");  
 }  
}

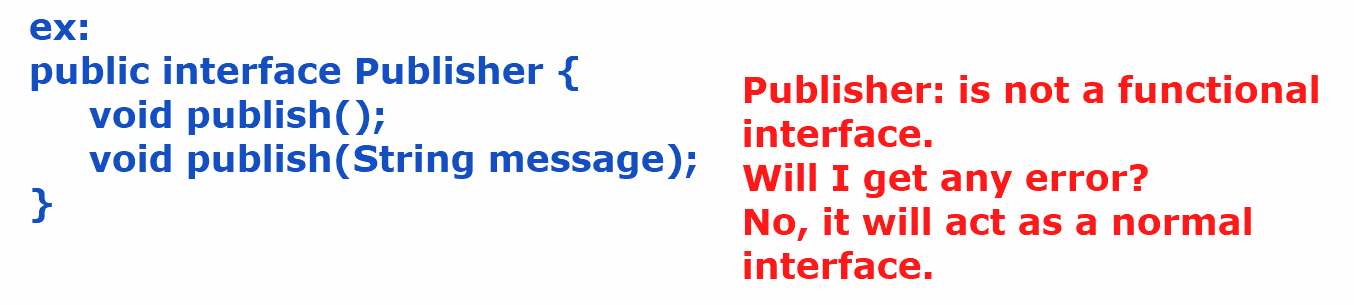
**MyInter2.java**

public interface MyInter2 {  
 default void m1() {  
 System.*out*.println("MyInter2 m1() called");  
 }  
}

**MyClass.java**

public class MyClass implements MyInter1, MyInter2 {  
 public void m3() {  
 System.*out*.println("MyInter3 m3() called");  
 m1();  
 }  
  
 @Override  
 public void m1() {  
 MyInter1.super.m1();  
 MyInter2.super.m1();  
 }  
}

****

****

* **If you want to get a compile-time error, when an interface contains more than one abstract method then add @FunctionalInterface annotation before that interface.**

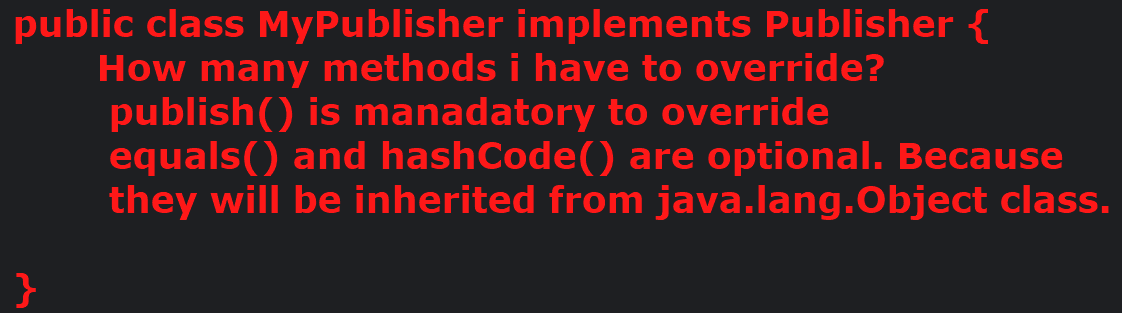
**ex:**

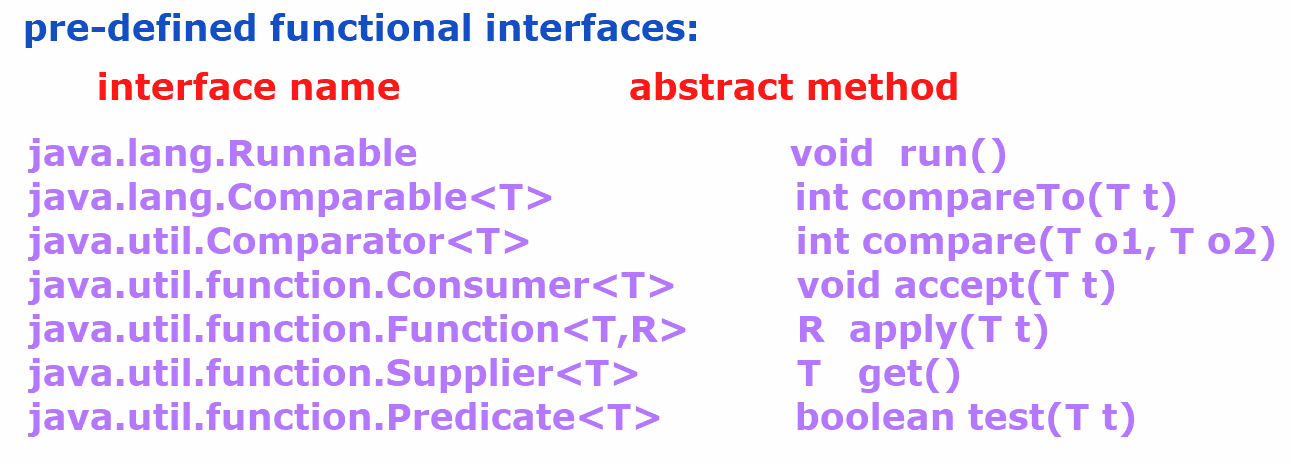
@FunctionalInterface  
public interface Publisher {  
 void publish();  
  
 default void publish(String message) {  
 System.*out*.println(message);  
 }  
   
 default void publish(String message, int number) {  
 System.*out*.println(message+ " " + number);  
 }  
  
}

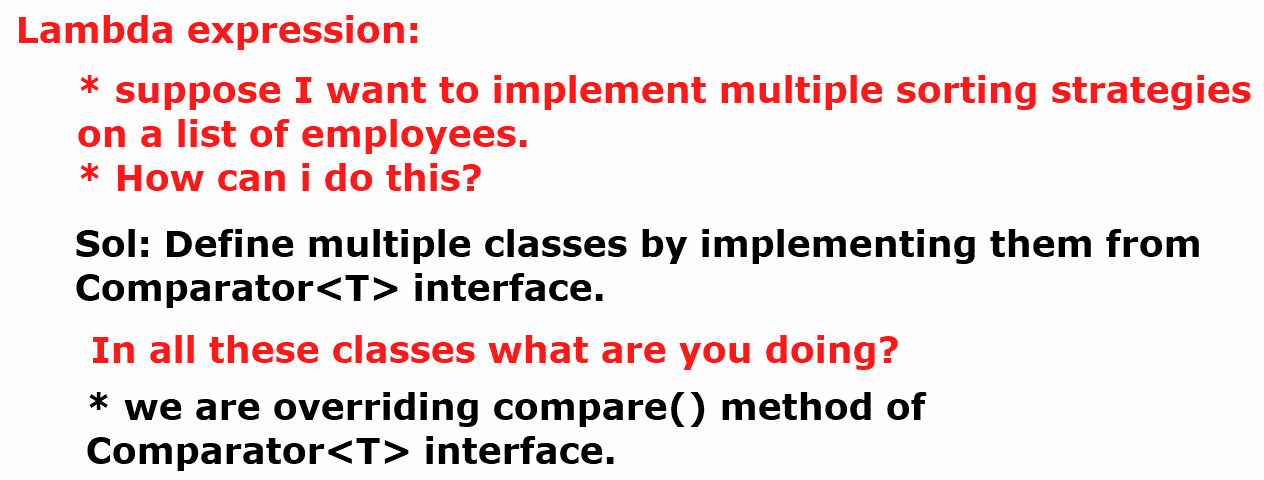
* **In a functional inteface, if we add any java.lang.Object class method as an abstract method, but still it will be a fucntional interface only.**

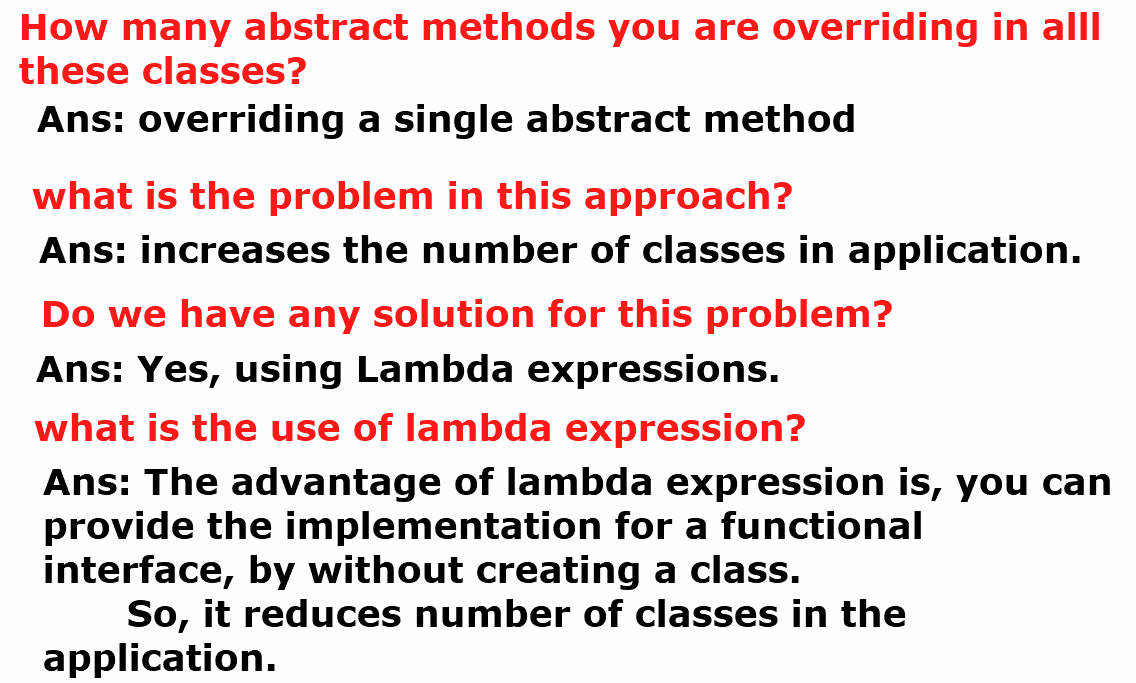
**ex:**

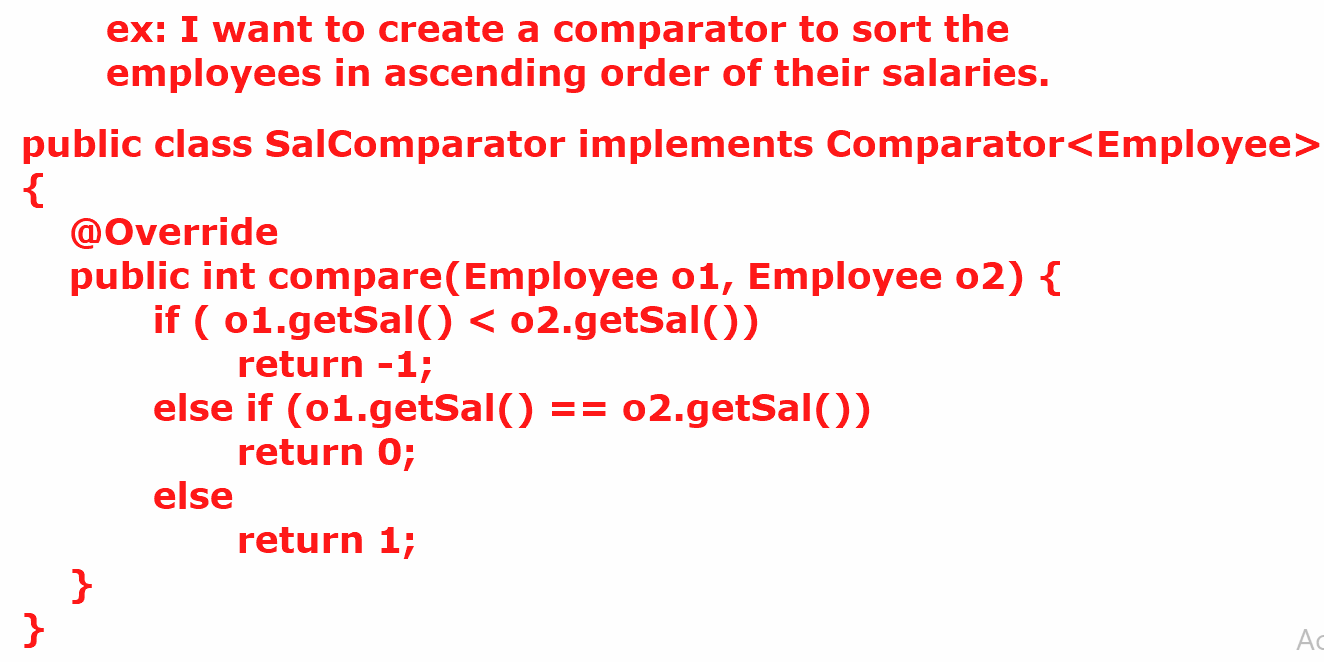
@FunctionalInterface  
public interface Publisher {  
 void publish();  
 boolean equals(Object obj);  
 int hashCode();  
}

****

****

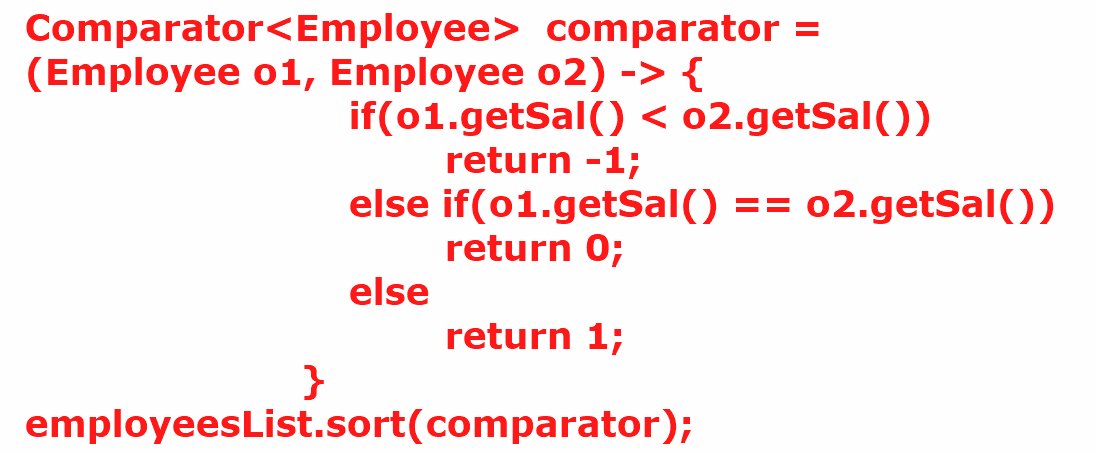
****

****

****

**employeesList.sort(new SalComparator());**

**How can I sort the list of employees by using lambda expression, without a creating a comparator class.**

****

* **To write a lambda expression, the syntax is**

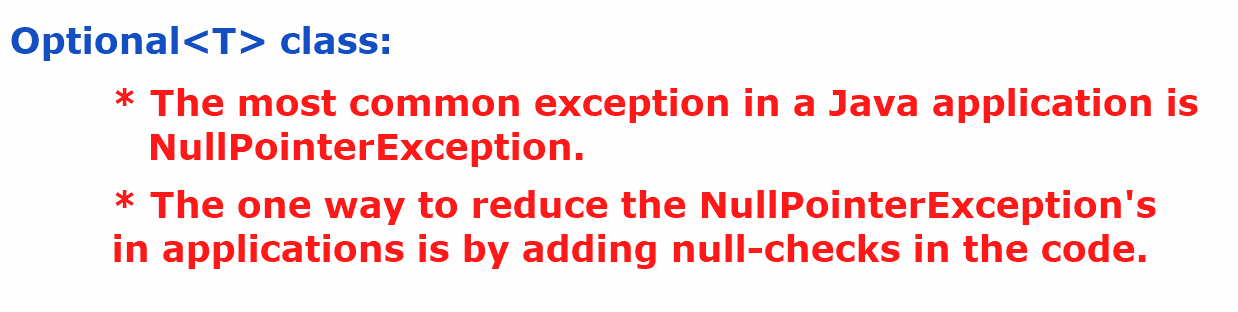
**(arguments) -> body;**

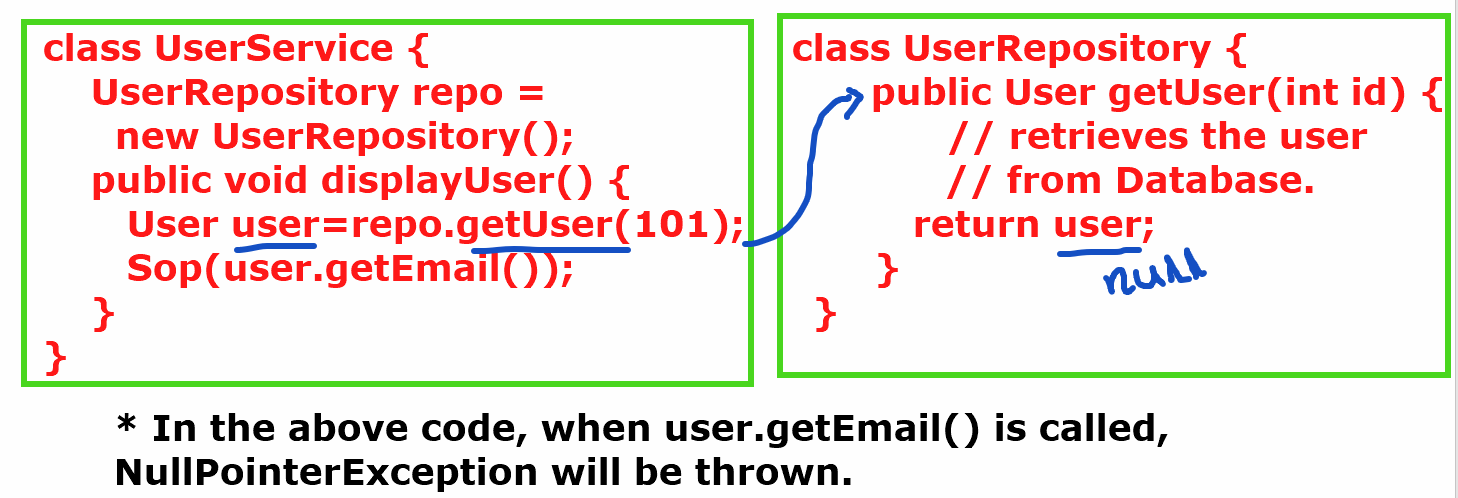
* **The arguments in this lambda expression should match with the arguments of the abstract method of the functional interface.**
* **The body of the lambda expression should match with the return type of the abstract method.**
* **If the number of arguments in lambda expression are zero or more than one then the arguments must be enclosed with in the parenthesis.**
* **If there is a single argument then parenthesis is optional.**
* **Data type for the arguments is optional. But don’t specify the type for one, and not for the others. It will generate compile-time error.**

**example on how to use lambda expression:**

@FunctionalInterface  
public interface Formatter {  
 String format(String s1, String s2);  
}

public class Main {  
 public static void main(String[] args) {  
  
 // Formatter fr = (s1,s2) -> s1.concat(s2) ;  
 *doWork*((s1,s2) -> s1.concat(s2));  
 *doWork*((s1,s2) -> s1.toUpperCase().concat(s2.toUpperCase()));  
  
 }  
 private static void doWork(Formatter formatter) {  
 System.*out*.println(formatter.format("Spring", "Boot"));  
  
 }  
  
}

****

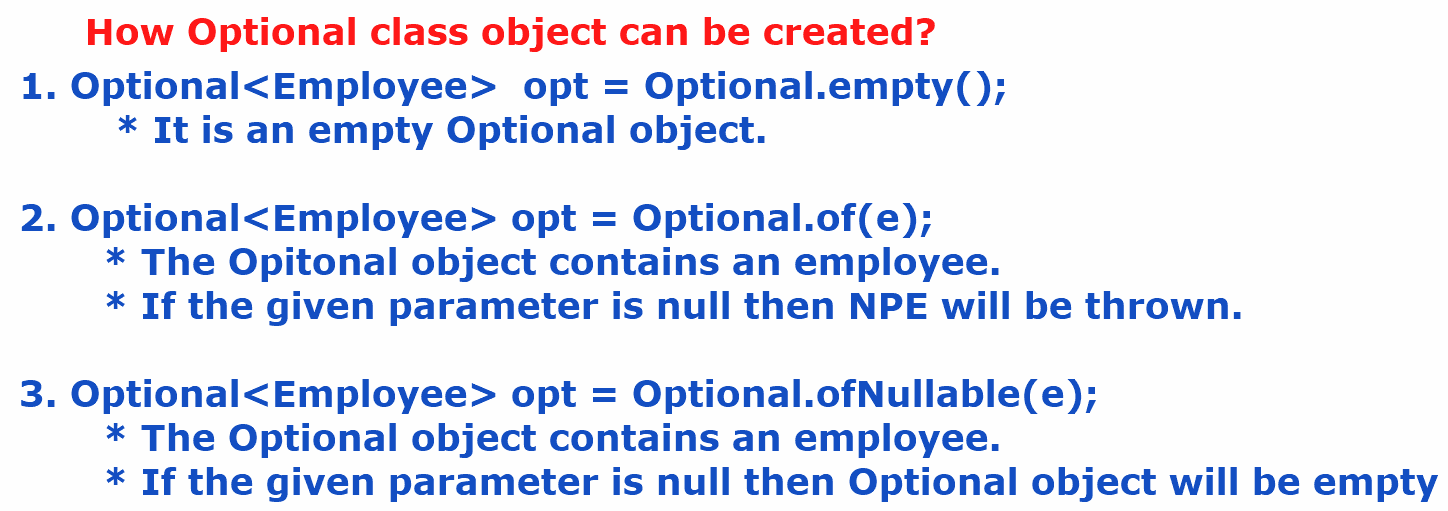
****

****

* **If more null-checks are added to the code, then the code becomes more complex and it kills the readability of the code.**
* **So, Optional class is provided in Java8, to reduce the null-checks and also to reduce the NullPointerExceptions.**

**what is Optional class object?**

**Ans: Optional class object is a container object, which may or may not contain non-null value.**

****

**Employee.java**

package com.example.model;  
  
public class Employee {  
 private int empno;  
 private String ename;  
 private double sal;  
 public Employee(int empno, String ename, double sal) {  
 this.empno = empno;  
 this.ename = ename;  
 this.sal = sal;  
 }  
 public int getEmpno() {  
 return empno;  
 }  
 public void setEmpno(int empno) {  
 this.empno = empno;  
 }  
 public String getEname() {  
 return ename;  
 }  
 public void setEname(String ename) {  
 this.ename = ename;  
 }  
 public double getSal() {  
 return sal;  
 }  
 public void setSal(double sal) {  
 this.sal = sal;  
 }  
 @Override  
 public String toString() {  
 return "Employee{" + "empno=" + empno + ", ename=" + ename + ", sal=" + sal + '}';  
 }  
}

**EmployeeRepository.java**

package com.example.repository;  
  
import com.example.model.Employee;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Optional;  
  
public class EmployeeRepository {  
 List<Employee> employees = new ArrayList<>();  
 public EmployeeRepository() {  
 employees.add(new Employee(7189,"King",4000));  
 employees.add(new Employee(7276,"Turner",5000));  
 employees.add(new Employee(7788,"Scott",6000));  
 }  
  
 public Optional<Employee> getEmployeeById(int empno) {  
 for (Employee employee : employees) {  
 if (employee.getEmpno() == empno) {  
 return Optional.*of*(employee);  
 }  
 }  
 return Optional.*empty*();  
 }  
}

**EmployeeService.java**

package com.example.service;  
  
import com.example.model.Employee;  
import com.example.repository.EmployeeRepository;  
  
import java.util.Optional;  
  
public class EmployeeService {  
 EmployeeRepository repo = new EmployeeRepository();  
  
 public void displayEmployee(int empno) {  
 Optional<Employee> opt = repo.getEmployeeById(empno);  
 // opt.ifPresent(e -> System.out.println(e.toString()));  
 if (opt.isPresent()) {  
 Employee emp = opt.get();  
 System.*out*.println(emp.toString());  
 }  
 else {  
 System.*out*.println("No employee with such id exists");  
 }  
 }  
}

**Main.java**

package com.example.main;  
  
import com.example.service.EmployeeService;  
  
public class Main {  
 public static void main(String[] args) {  
 EmployeeService service = new EmployeeService();  
 service.displayEmployee(7788);  
 }  
  
}

Methods of Optional<T> class:

--------------------------

1. isPresent() : If value is present in the Optional object then returns true. Otherwise, returns false.

2. get() : Returns the value present in the Optional object.

ex:

Optional<String> opt = Optional.ofNullable(“Hello World”);

if(opt.isPresent()) {

String str = opt.get();

System.out.println(str);

}

3. ifPresent(Consumer<T> consumer) : Executes the given action, if the value is present in the Optional object. Otherwise, do’s nothing.

. This method accepts Consumer object as parameter.

. Consumer is a functional interface, so we can pass lambda expression.

ex:

Optional<String> opt = Optional.ofNullable(“Hello World”);

opt.ifPresent( str -> System.out.println(str) );

4. orElse(T another): if the Optional contains value, it returns that value. If it is empty then returns the another value provided as parameter.

ex1:

Optional<String> opt = Optional.ofNullable(“Hello world”);

String result = opt.orElse(“Bye world”);

System.out.println(result); //output: Hello world

ex2:

Optional<String> opt = Optional.empty();

String result = opt.orElse(“Bye world”);

System.out.println(result); //output: Bye world

5. orElseThrow(): If the Optional contains a value, then returns that value. If it is empty, then throws NoSuchElementException.

ex1:

Optional<String> opt = Optional.ofNullable(“Hello world”);

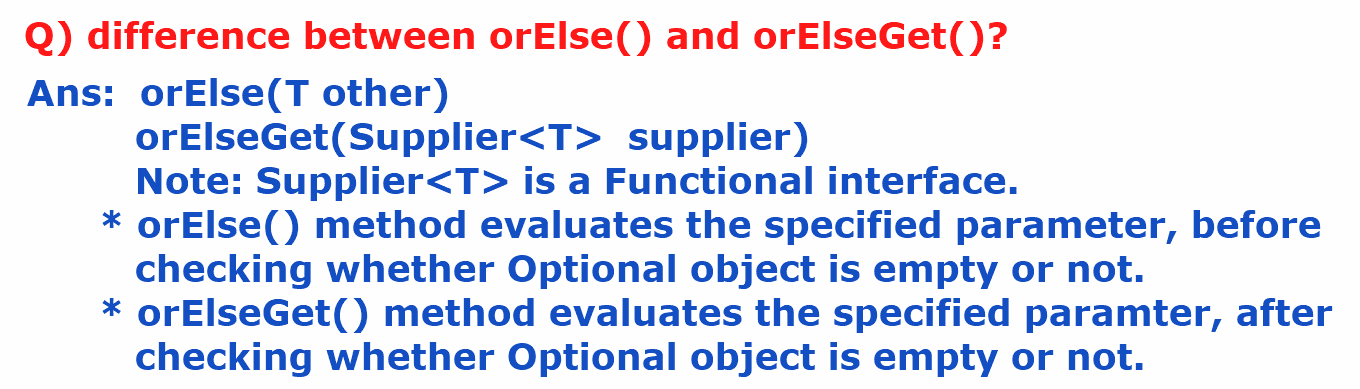
String result = opt.orElseThrow();

System.out.println(result); //output: Hello world

ex2:

Optional<String> opt = Optional.empty();

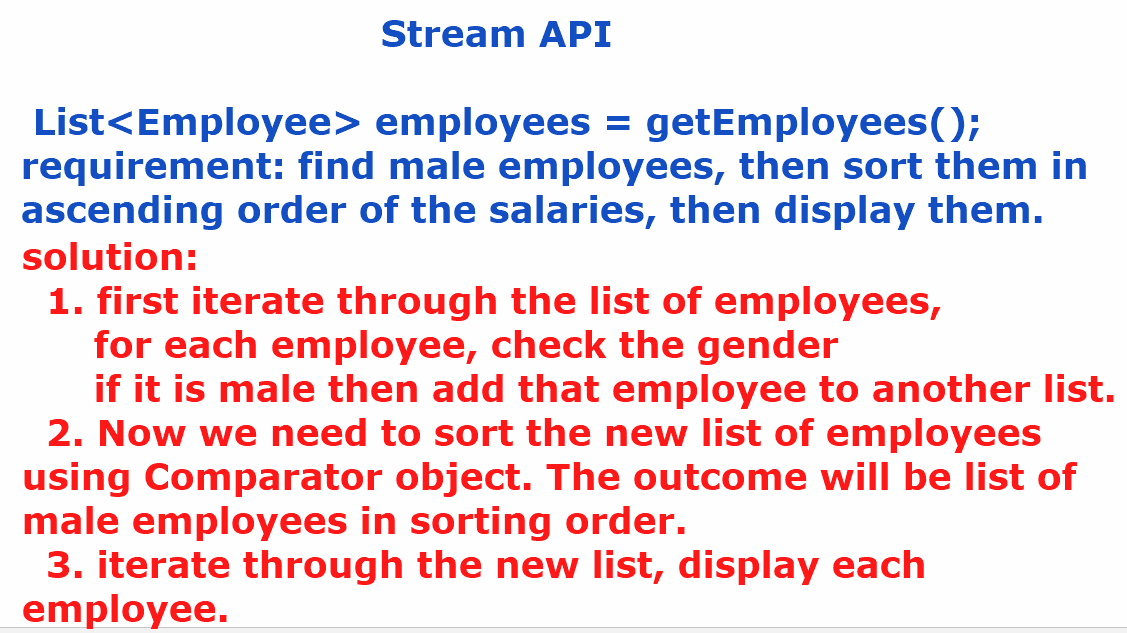
String result = opt.orElseThrow(); // throws NoSuchElementException

****

**ex:**

**Employee emp = opt.orElse(new Employee(0, null, 0.0));**

**Employee emp = opt.orElseGet( () -> new Employee(0, null, 0.0));**

****

* **Before Java8, to perform any processing operations on a collection object, we have to iterate through the collection object, with an Iterator or for each loop and we should define some if conditions to process them.**
* **This leads to more amount of code. So, the code is hard to understand and increase the complexity.**
* **In Java, Collection and Stream objects are used for different purposes.**
* **Collection object is used to store the elements, and a Stream object is used to perform processing operations on the elements.**

**creating a Stream object:**

1. we can a Stream for a Collection object.

Stream<Employee> stream = employeesList.stream();

2. we can create a Stream for an array.

Stream<Integer> stream = Arrays.stream(arr);

3. we can create a Stream object for raw values.

Stream<String> stream = Stream.of(“Tom”, “john”, “Jack”);

4. we can create an empty stream object.

Stream<Void> stream = Stream.empty();

* A stream operations are two types.

1.intermediate operations

2.terminal operations.

\* intermediate operation means, it transforms a stream to another stream.

\* terminal operation means, which produces the end result.

example:

Stream<Employee> stream1 = employeesList.stream();

Stream<Employee> stream2 = stream1.filter( e -> e.getSal() > 1000);

long count = stream2.count();

* Here, filter() is an intermediate operation and count() is a terminal operation.

public class EmployeeService {  
  
 public static List<Employee> getEmployees(){  
 List<Employee> employees = new ArrayList<>();  
 employees.add(new Employee(7078, "Blake", 5000, "Male", 10));  
 employees.add(new Employee(7156, "Turner", 9000, "Male", 20));  
 employees.add(new Employee(7267, "Sophia", 6000, "Female", 10));  
 employees.add(new Employee(7299, "King", 3000, "Male", 30));  
 employees.add(new Employee(7435, "Emma", 4000, "Female", 20));  
 employees.add(new Employee(7180, "Mary", 5000, "Female", 30));  
 employees.add(new Employee(7788, "Scott", 8000, "Male", 10));  
 employees.add(new Employee(7923, "Allen", 7000, "Male", 20));  
 employees.add(new Employee(7798, "Kathey", 5000, "Female", 10));  
 return employees;  
 }  
  
 // Get employees with sal > given salary  
 public List<Employee> getEmployeesBySalGreaterThan(double sal) {  
 /\*  
 Stream<Employee> empStream = getEmployees().stream();  
 Stream<Employee> filterdStream = empStream.filter( (e) -> e.getSal() > sal);  
 List<Employee> filterdEmployees = filterdStream.toList();  
 return filterdEmployees;  
 \*/  
 return *getEmployees*()  
 .stream()  
 .filter(e -> e.getSal() > sal)  
 .toList();  
 }  
  
 // Get the names of employees  
 public List<String> getEmployeesNames(){  
 return *getEmployees*()  
 .stream()  
 .map(Employee::getEname)  
 .toList();  
 }  
  
 // Get the female employees  
 public List<Employee> getFemaleEmployees() {  
 return *getEmployees*()  
 .stream()  
 .filter(e -> e.getGender().equalsIgnoreCase("FEMALE"))  
 .toList();  
 }  
  
  
  
}

public class Main {  
 public static void main(String[] args) {  
 EmployeeService employeeService = new EmployeeService();  
 /\*  
 List<Employee> employees = employeeService.getEmployeesBySalGreaterThan(9000);  
 employees.forEach(System.out::println);  
 \*/  
  
 /\*  
 List<String> employeeNames = employeeService.getEmployeesNames();  
 employeeNames.forEach(System.out::println);  
 \*/  
  
 List<Employee> femaleEmployees = employeeService.getFemaleEmployees();  
 femaleEmployees.forEach(System.*out*::println);  
 }  
}