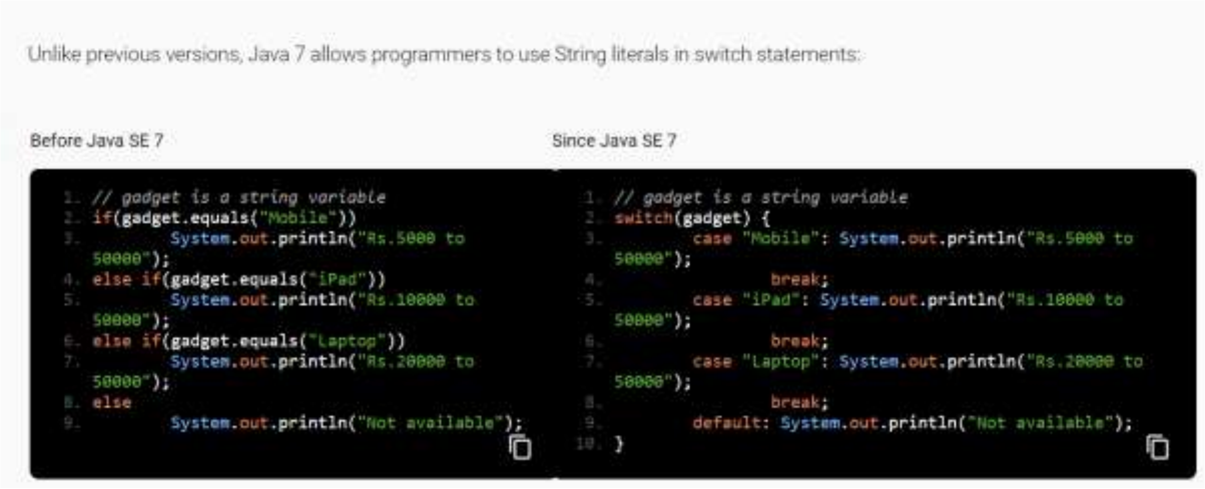
A glance at java 7 features:



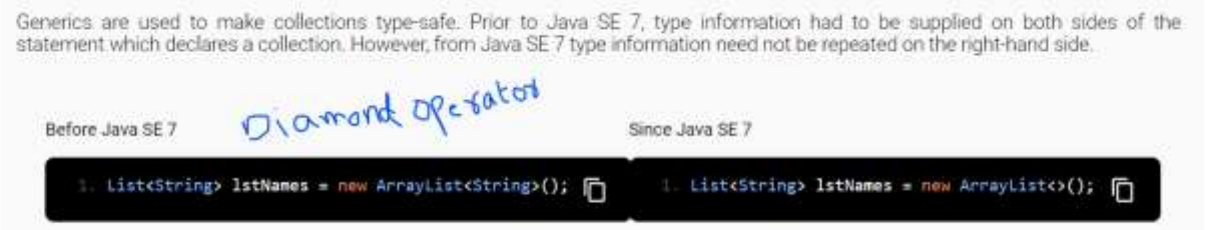
Unlike previous versions, Java 7 allows programmers to use String literals in switch statements:

Before Java SE 7

```
// gadget is a string variable
if(gadget.equals("Mobile"))
        System.out.println("Rs.5000 to 50000");
else if(gadget.equals("iPad"))
        System.out.println("Rs.10000 to 50000");
else if(gadget.equals("Laptop"))
        System.out.println("Rs.20000 to 50000");
else
        System.out.println("Not available");
```

Since Java SE 7

```
// gadget is a string variable
switch(gadget) {
        case "Mobile": System.out.println("Rs.5000 to 50000");
                break;
        case "iPad": System.out.println("Rs.10000 to 50000");
                break;
        case "Laptop": System.out.println("Rs.20000 to 50000");
                break;
        default: System.out.println("Not available");
}
```



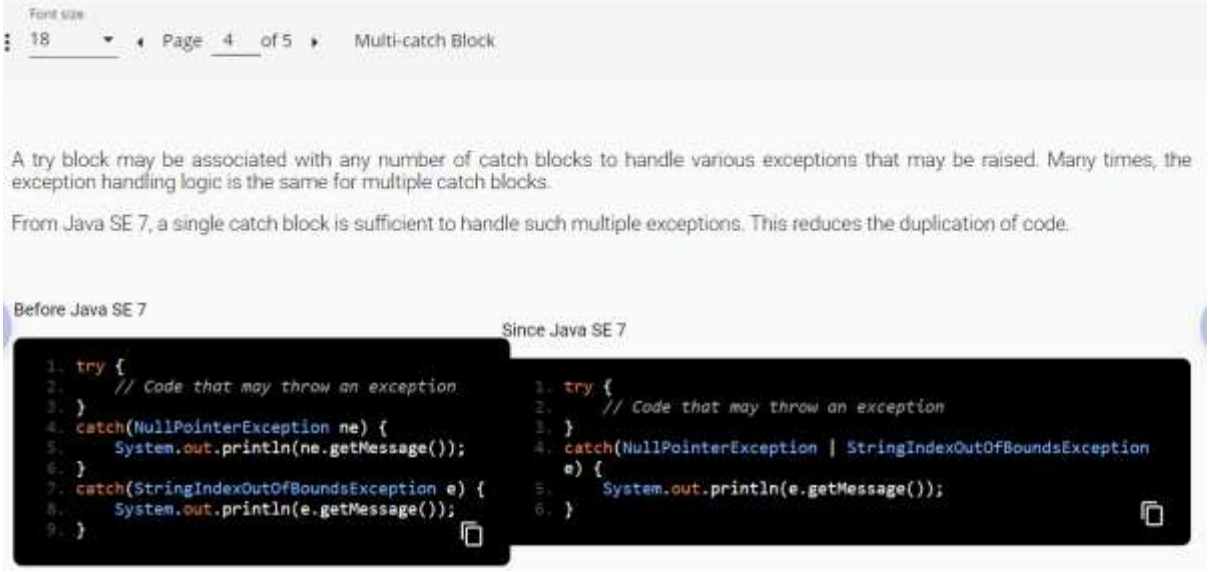
Generics are used to make collections type-safe. Prior to Java SE 7, type information had to be supplied on both sides of the statement which declares a collection. However, from Java SE 7 type information need not be repeated on the right-hand side.

Before Java SE 7 Diamond operator

```
List<String> lstNames = new ArrayList<String>();
```

Since Java SE 7

```
List<String> lstNames = new ArrayList<>();
```



A try block may be associated with any number of catch blocks to handle various exceptions that may be raised. Many times, the exception handling logic is the same for multiple catch blocks.

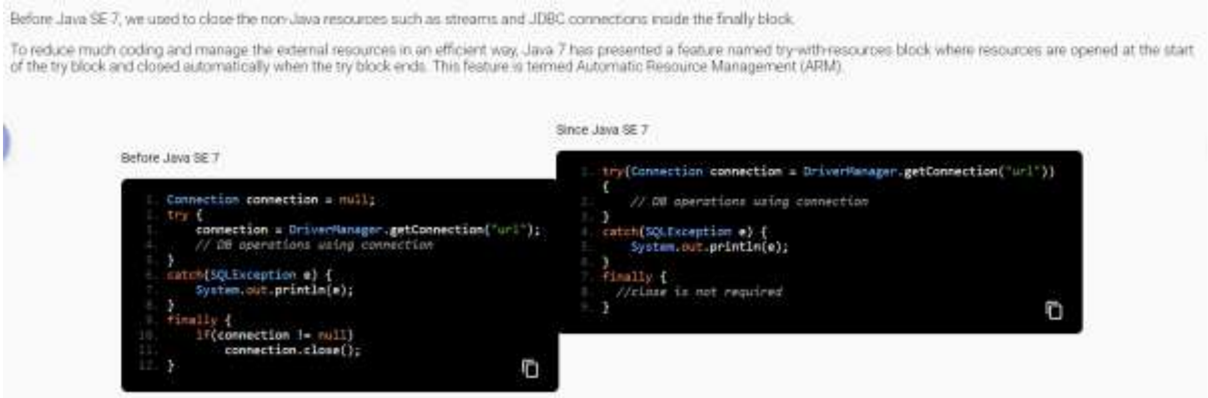
From Java SE 7, a single catch block is sufficient to handle such multiple exceptions. This reduces the duplication of code.

Before Java SE 7

```
try {
    // Code that may throw an exception
}
catch(NullPointerException ne) {
    System.out.println(ne.getMessage());
}
catch(StringIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```

Since Java SE 7

```
try {
    // Code that may throw an exception
}
catch(NullPointerException | StringIndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
```



Before Java SE 7, we used to close the non-Java resources such as streams and JDBC connections inside the finally block.

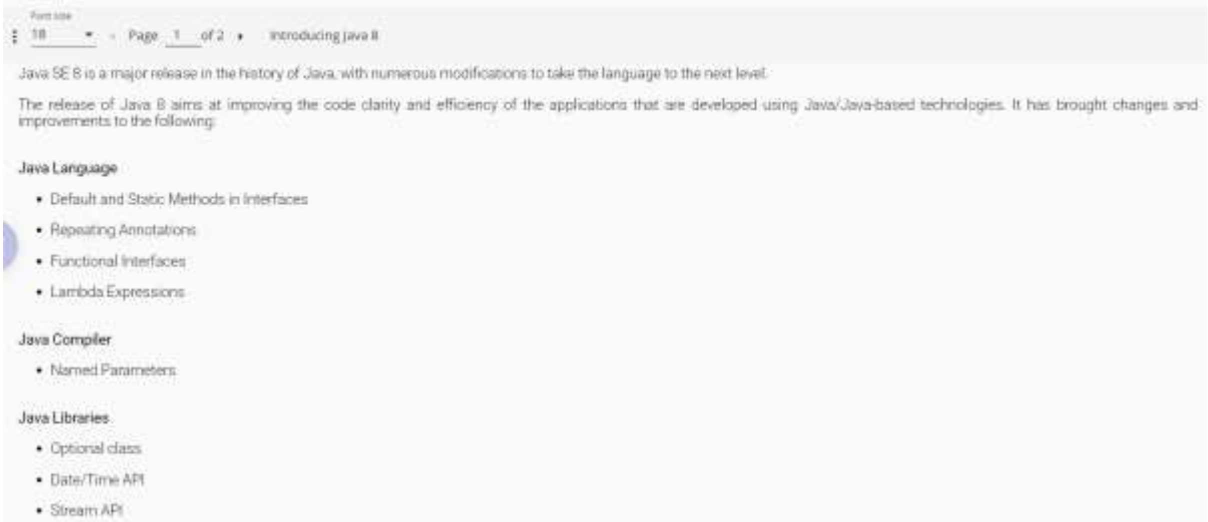
To reduce much coding and manage the external resources in an efficient way, Java 7 has presented a feature named try-with-resources block where resources are opened at the start of the try block and closed automatically when the try block ends. This feature is termed Automatic Resource Management (ARM).

Before Java SE 7

```
Connection connection = null;
try {
    connection = DriverManager.getConnection("url");
    // DB operations using connection
}
catch(SQLException e) {
    System.out.println(e);
}
finally {
    if(connection != null)
        connection.close();
}
```

Since Java SE 7

```
try(Connection connection = DriverManager.getConnection("url"))
{
    // DB operations using connection
}
catch(SQLException e) {
    System.out.println(e);
}
finally {
  //close is not required
}
```



Java SE 8 is a major release in the history of Java, with numerous modifications to take the language to the next level.

The release of Java 8 aims at improving the code clarity and efficiency of the applications that are developed using Java/Java-based technologies. It has brought changes and improvements to the following:

Java Language

- Default and Static Methods in Interfaces
- Repeating Annotations
- Functional Interfaces
- Lambda Expressions

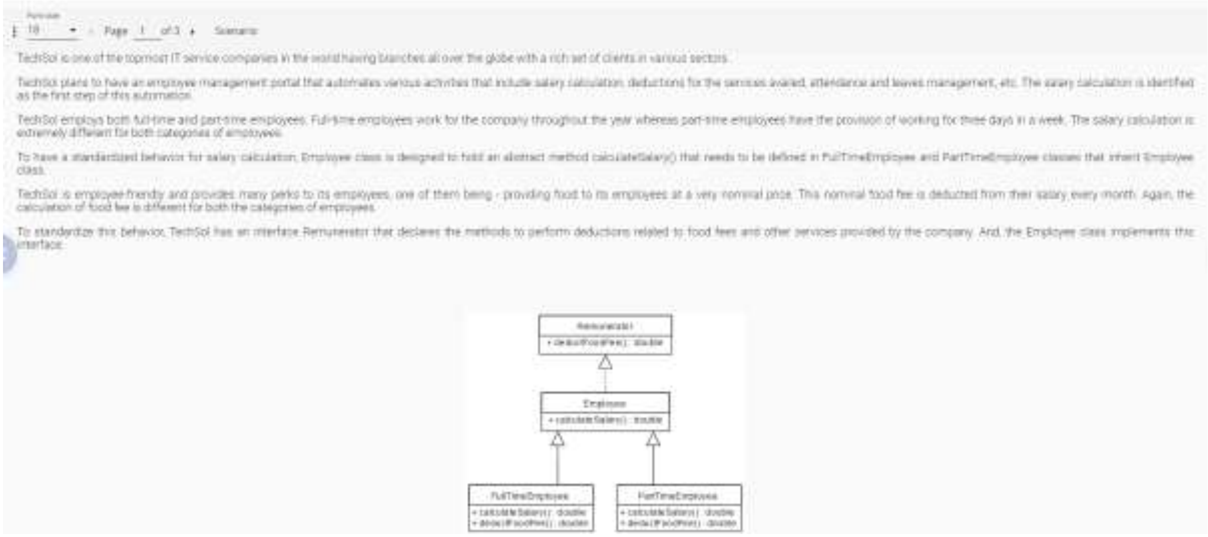
Java Compiler

- Named Parameters

Java Libraries

- Optional class
- Date/Time API
- Stream API

Default and Static Methods in Interface:



TechSol is one of the topmost IT service companies in the world having branches all over the globe with a rich set of clients in various sectors.

TechSol plans to have an employee management portal that automates various activities that include salary calculation, deductions for the services availed, attendance and leaves management, etc. The salary calculation is identified as the first step of this automation.

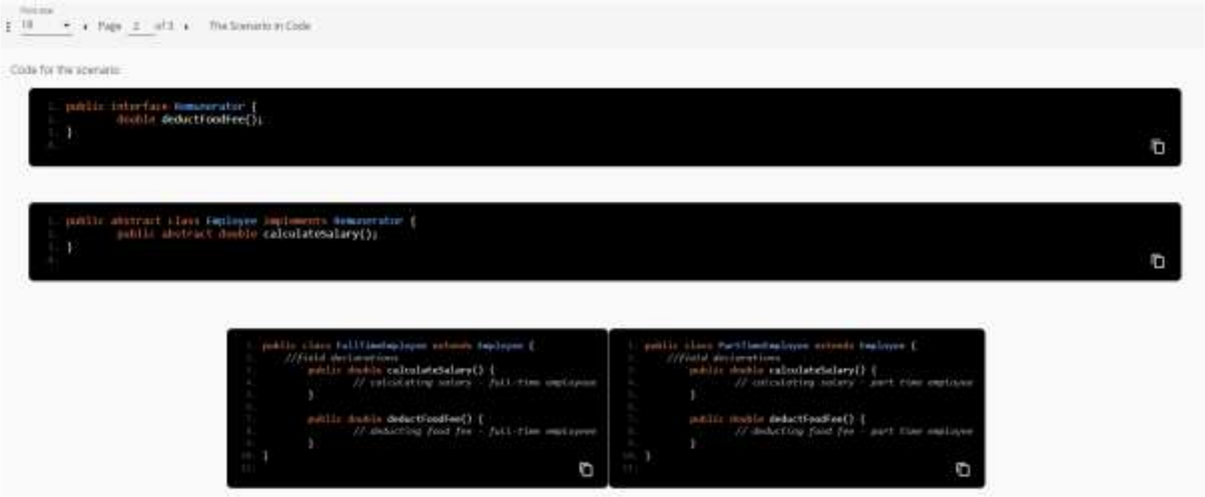
TechSol employs both full-time and part-time employees. Full-time employees work for the company throughout the year whereas part-time employees have the provision of working for three days in a week. The salary calculation is extremely different for both categories of employees.

To have a standardized behavior for salary calculation, Employee class is designed to hold an abstract method calculateSalary() that needs to be defined in FullTimeEmployee and PartTimeEmployee classes that inherit Employee class.

TechSol is employee-friendly and provides many perks to its employees, one of them being - providing food to its employees at a very nominal price. This nominal food fee is deducted from their salary every month. Again, the calculation of food fee is different for both the categories of employees.

To standardize this behavior, TechSol has an interface Remunerator that declares the methods to perform deductions related to food fees and other services provided by the company. And, the Employee class implements this interface.

Remunerator
+ deductFoodFee() : double
Employee
+ calculateSalary() : double
FullTimeEmployee
+ calculateSalary() : double
+ deductFoodFee() : double
PartTimeEmployee
+ calculateSalary() : double
+ deductFoodFee() : double



Font size
18
Page 2 of 3
The Scenario in Code

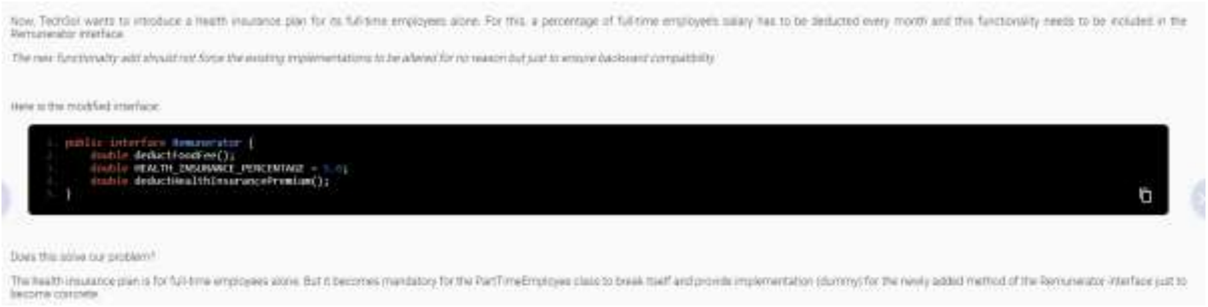
Code for the scenario:

```java
public interface Remunerator {
        double deductFoodFee();
}
```

```java
public abstract class Employee implements Remunerator {
        public abstract double calculateSalary();
}
```

```java
public class FullTimeEmployee extends Employee {
    //field declarations
        public double calculateSalary() {
                // calculating salary - full-time employee
        }

        public double deductFoodFee() {
                // deducting food fee - full-time employee
        }
}
```

```java
public class PartTimeEmployee extends Employee {
    //field declarations
        public double calculateSalary() {
                // calculating salary - part time employee
        }

        public double deductFoodFee() {
                // deducting food fee - part time employee
        }
}
```



Now, TechSol wants to introduce a health insurance plan for its full-time employees alone. For this, a percentage of full-time employee's salary has to be deducted every month and this functionality needs to be included in the Remunerator interface.

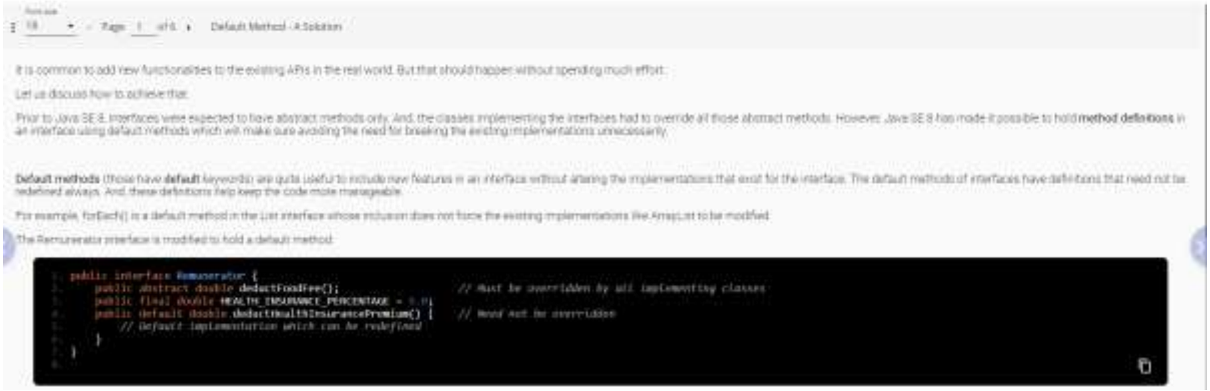
The new functionality add should not force the existing implementations to be altered for no reason but just to ensure backward compatibility.

Here is the modified interface:

```
public interface Remunerator {
    double deductFoodFee();
    double HEALTH_INSURANCE_PERCENTAGE = 5.0;
    double deductHealthInsurancePremium();
}
```

Does this solve our problem?

The health insurance plan is for full-time employees alone. But it becomes mandatory for the PartTimeEmployee class to break itself and provide implementation (dummy) for the newly added method of the Remunerator interface just to become concrete.



Font size 18 ▾ ◂ Page 1 of 6 ▸ Default Method - A Solution

It is common to add new functionalities to the existing APIs in the real world. But that should happen without spending much effort.

Let us discuss how to achieve that.

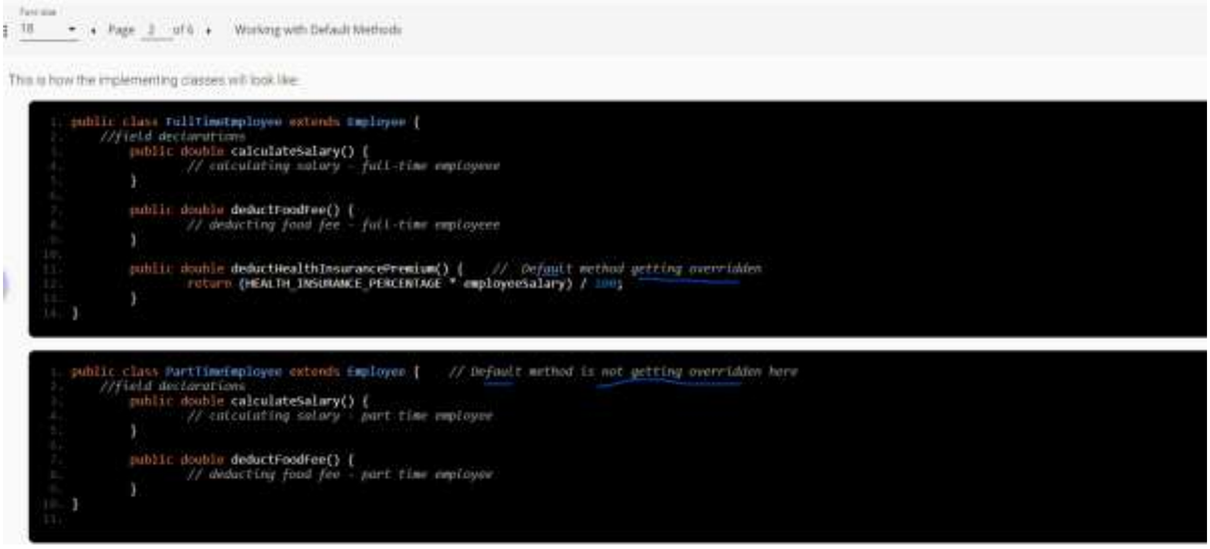
Prior to Java SE 8, interfaces were expected to have abstract methods only. And, the classes implementing the interfaces had to override all those abstract methods. However, Java SE 8 has made it possible to hold **method definitions** in an interface using default methods which will make sure avoiding the need for breaking the existing implementations unnecessarily.

Default methods (those have **default** keywords) are quite useful to include new features in an interface without altering the implementations that exist for the interface. The default methods of interfaces have definitions that need not be redefined always. And, these definitions help keep the code more manageable.

For example, forEach() is a default method in the List interface whose inclusion does not force the existing implementations like ArrayList to be modified.

The Remunerator interface is modified to hold a default method.

```
public interface Remunerator {
    public abstract double deductFoodFee();                    // Must be overridden by all implementing classes
    public final double HEALTH_INSURANCE_PERCENTAGE = 5.0;
    public default double deductHealthInsurancePremium() {     // Need not be overridden
        // Default implementation which can be redefined
    }
}
```

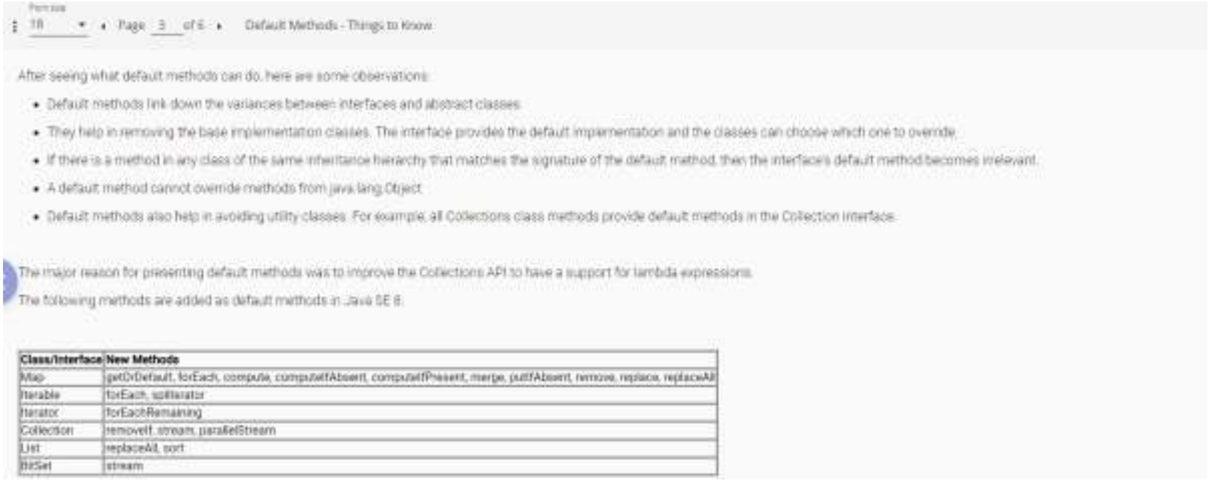


Font size 18 ▾ ◂ Page 2 of 6 ▸ Working with Default Methods

This is how the implementing classes will look like:

```
public class FullTimeEmployee extends Employee {
    //field declarations
        public double calculateSalary() {
            // calculating salary - full-time employee
        }

        public double deductFoodFee() {
            // deducting food fee - full-time employee
        }

        public double deductHealthInsurancePremium() {    // Default method getting overridden
            return (HEALTH_INSURANCE_PERCENTAGE * employeeSalary) / 100;
        }
}
```

```
public class PartTimeEmployee extends Employee {    // Default method is not getting overridden here
    //field declarations
        public double calculateSalary() {
            // calculating salary - part time employee
        }

        public double deductFoodFee() {
            // deducting food fee - part time employee
        }
}
```



Font size 18 ▾ ◂ Page 3 of 6 ▸ Default Methods - Things to Know

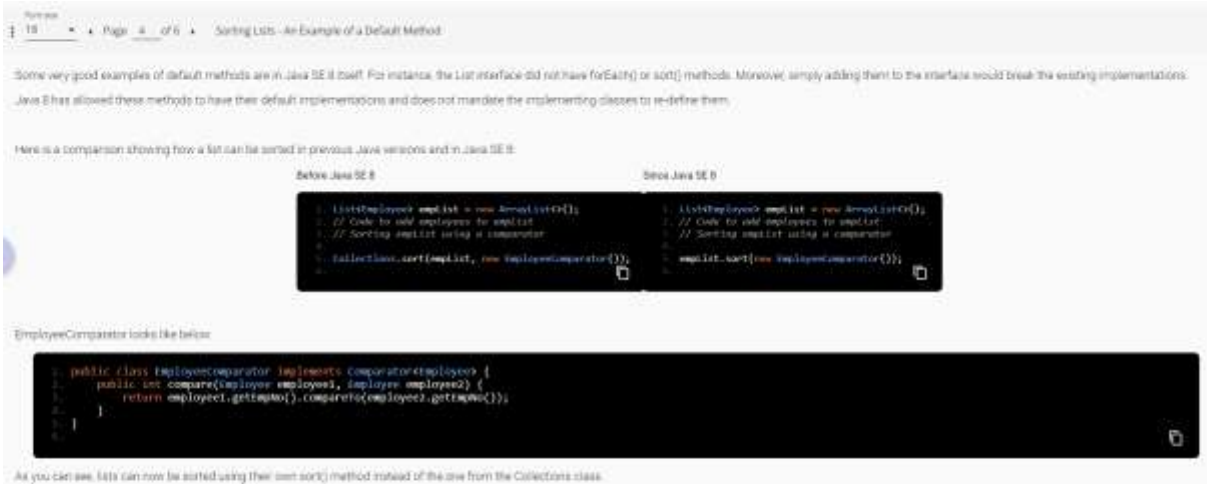
After seeing what default methods can do, here are some observations:

- Default methods link down the variances between interfaces and abstract classes.
- They help in removing the base implementation classes. The interface provides the default implementation and the classes can choose which one to override.
- If there is a method in any class of the same inheritance hierarchy that matches the signature of the default method, then the interface's default method becomes irrelevant.
- A default method cannot override methods from java.lang.Object.
- Default methods also help in avoiding utility classes. For example, all Collections class methods provide default methods in the Collection interface.

The major reason for presenting default methods was to improve the Collections API to have a support for lambda expressions.

The following methods are added as default methods in Java SE 8.

Class/Interface	New Methods
Map	getOrDefault, forEach, compute, computeIfAbsent, computeIfPresent, merge, putIfAbsent, remove, replace, replaceAll
Iterable	forEach, spliterator
Iterator	forEachRemaining
Collection	removeIf, stream, parallelStream
List	replaceAll, sort
BitSet	stream



Sorting Lists - An Example of a Default Method

Some very good examples of default methods are in Java SE 8 itself. For instance, the List interface did not have forEach() or sort() methods. Moreover, simply adding them to the interface would break the existing implementations. Java 8 has allowed these methods to have their default implementations and does not mandate the implementing classes to re-define them.

Here is a comparison showing how a list can be sorted in previous Java versions and in Java SE 8:

Before Java SE 8

```java
List<Employee> empList = new ArrayList<>();
// Code to add employees to empList
// Sorting empList using a comparator

Collections.sort(empList, new EmployeeComparator());
```

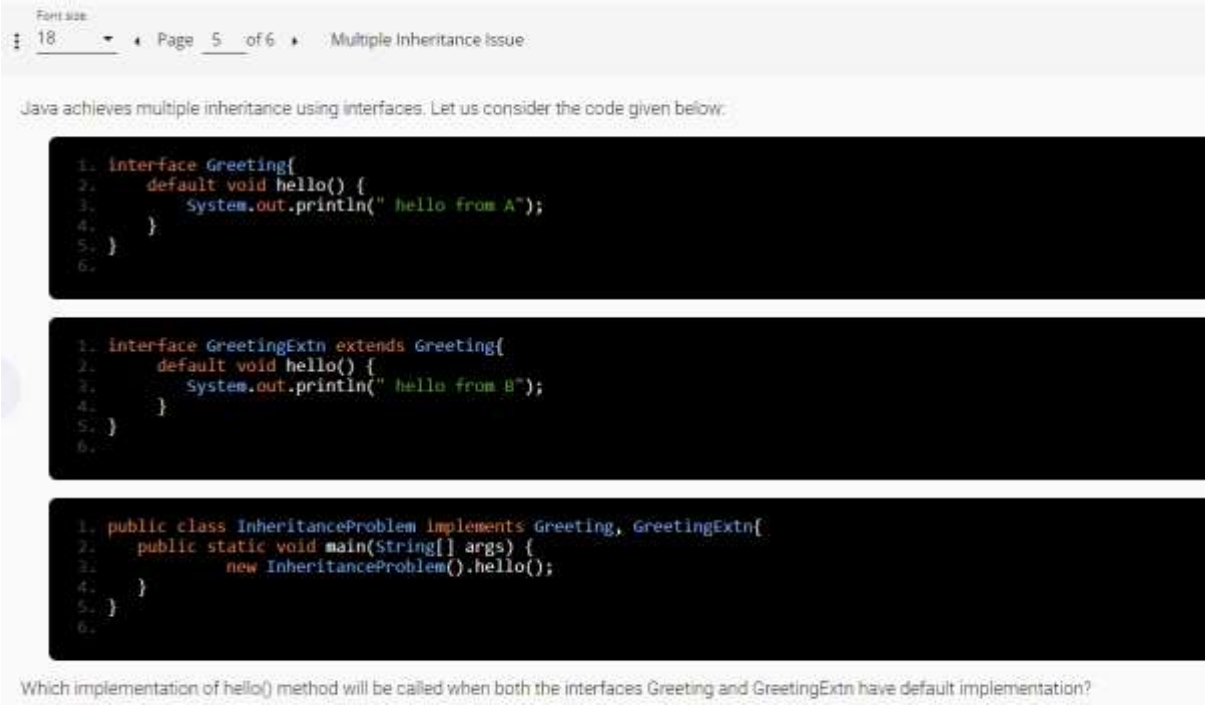
Since Java SE 8

```java
List<Employee> empList = new ArrayList<>();
// Code to add employees to empList
// Sorting empList using a comparator

empList.sort(new EmployeeComparator());
```

EmployeeComparator looks like below:

```java
public class EmployeeComparator implements Comparator<Employee> {
    public int compare(Employee employee1, Employee employee2) {
        return employee1.getEmpNo().compareTo(employee2.getEmpNo());
    }
}
```

As you can see, lists can now be sorted using their own sort() method instead of the one from the Collections class.



Multiple Inheritance Issue

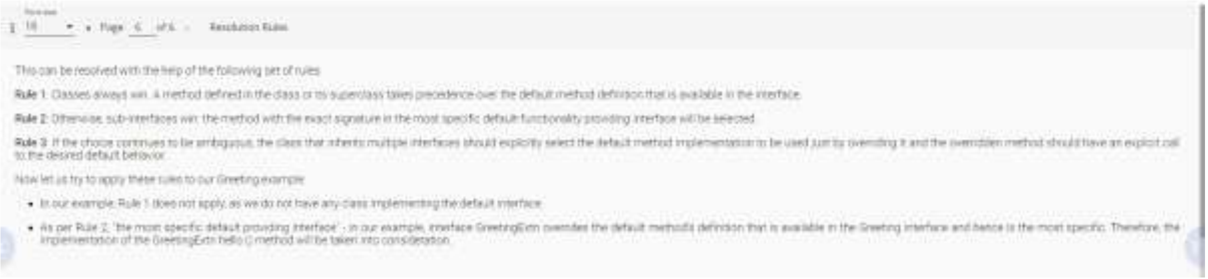
Java achieves multiple inheritance using interfaces. Let us consider the code given below:

```java
interface Greeting{
    default void hello() {
        System.out.println(" hello from A");
    }
}
```

```java
interface GreetingExtn extends Greeting{
     default void hello() {
        System.out.println(" hello from B");
     }
}
```

```java
public class InheritanceProblem implements Greeting, GreetingExtn{
   public static void main(String[] args) {
           new InheritanceProblem().hello();
   }
}
```

Which implementation of hello() method will be called when both the interfaces Greeting and GreetingExtn have default implementation?



Resolution Rules

This can be resolved with the help of the following set of rules:

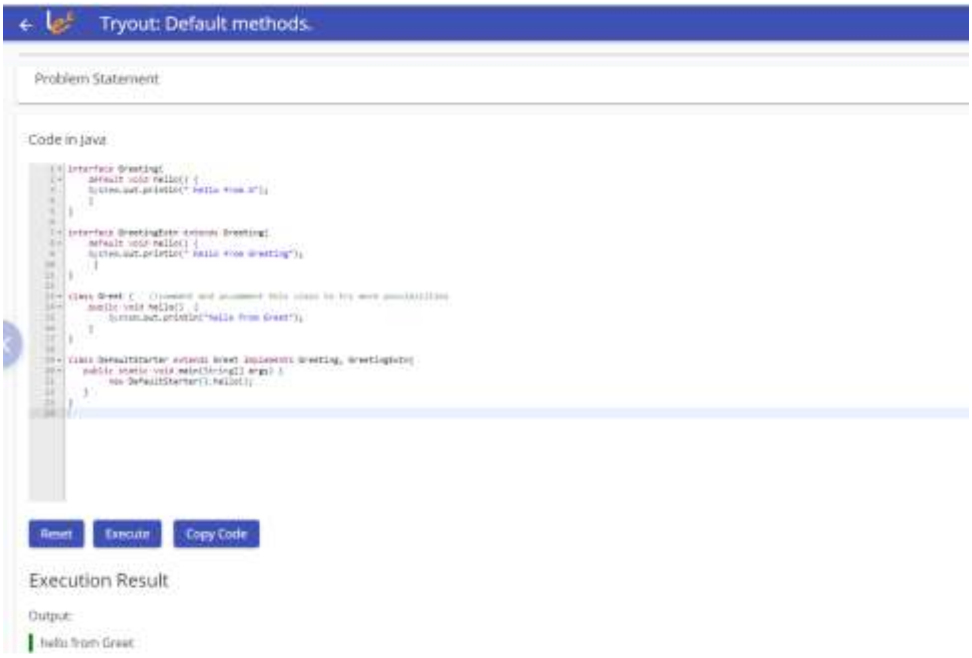
Rule 1: Classes always win. A method defined in the class or its superclass takes precedence over the default method definition that is available in the interface.

Rule 2: Otherwise, sub-interfaces win: the method with the exact signature in the most specific default functionality providing interface will be selected.

Rule 3: If the choice continues to be ambiguous, the class that inherits multiple interfaces should explicitly select the default method implementation to be used just by overriding it and the overridden method should have an explicit call to the desired default behavior.

Now let us try to apply these rules to our Greeting example:

- In our example, Rule 1 does not apply, as we do not have any class implementing the default interface.
- As per Rule 2, 'the most specific default providing interface' - in our example, interface GreetingExtn overrides the default method's definition that is available in the Greeting interface and hence is the most specific. Therefore, the implementation of the GreetingExtn hello () method will be taken into consideration.



Tryout: Default methods.

Problem Statement

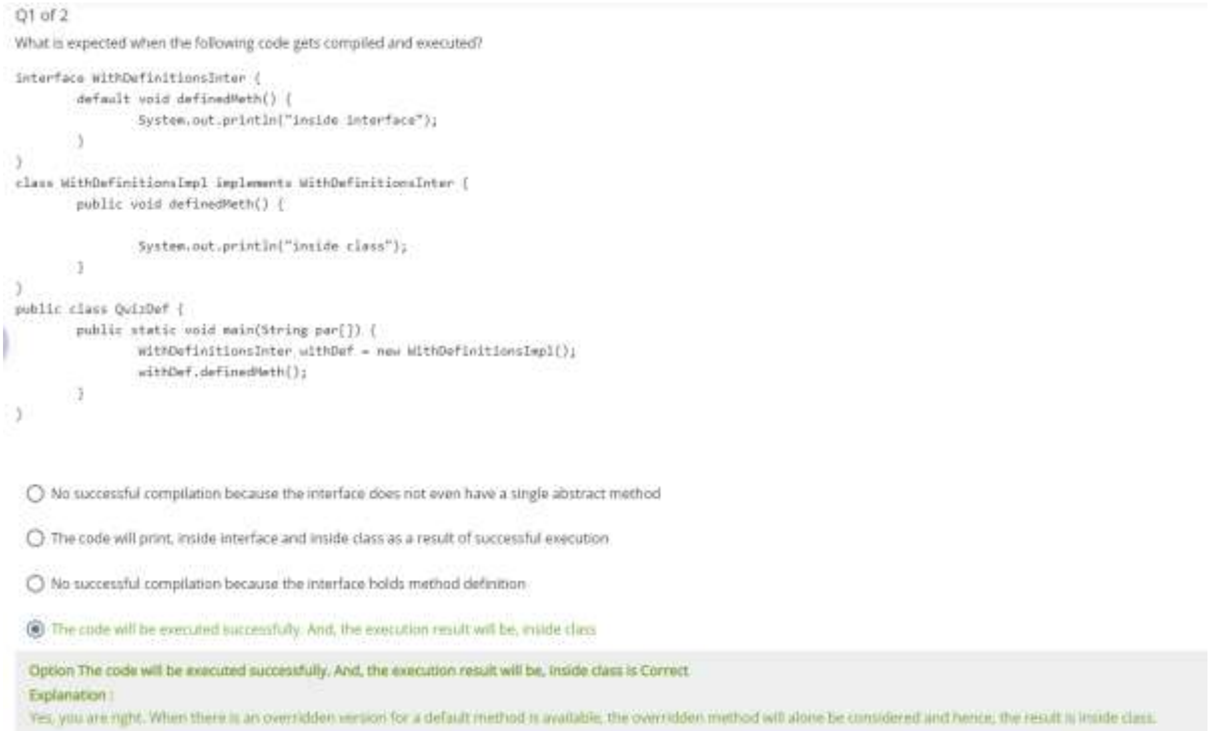
Code in Java

```
interface Greeting{
    default void hello() {
    System.out.println(" hello from X");
    }
}

interface GreetingExtn extends Greeting{
    default void hello() {
    System.out.println(" hello from Greeting");
     }
}

class Greet {   //comment and uncomment this class to try more possibilities
    public void hello()  {
        System.out.println("hello from Greet");
    }
}

class DefaultStarter extends Greet implements Greeting, GreetingExtn{
   public static void main(String[] args) {
        new DefaultStarter().hello();
   }
}
/
```

Execution Result

Output:

```
hello from Greet
```



Q1 of 2

What is expected when the following code gets compiled and executed?

```
interface WithDefinitionsInter {
        default void definedMeth() {
                System.out.println("inside interface");
        }
}
class WithDefinitionsImpl implements WithDefinitionsInter {
        public void definedMeth() {

                System.out.println("inside class");
        }
}
public class QuizDef {
        public static void main(String par[]) {
                WithDefinitionsInter withDef = new WithDefinitionsImpl();
                withDef.definedMeth();
        }
}
```

○ No successful compilation because the interface does not even have a single abstract method

○ The code will print, inside interface and inside class as a result of successful execution

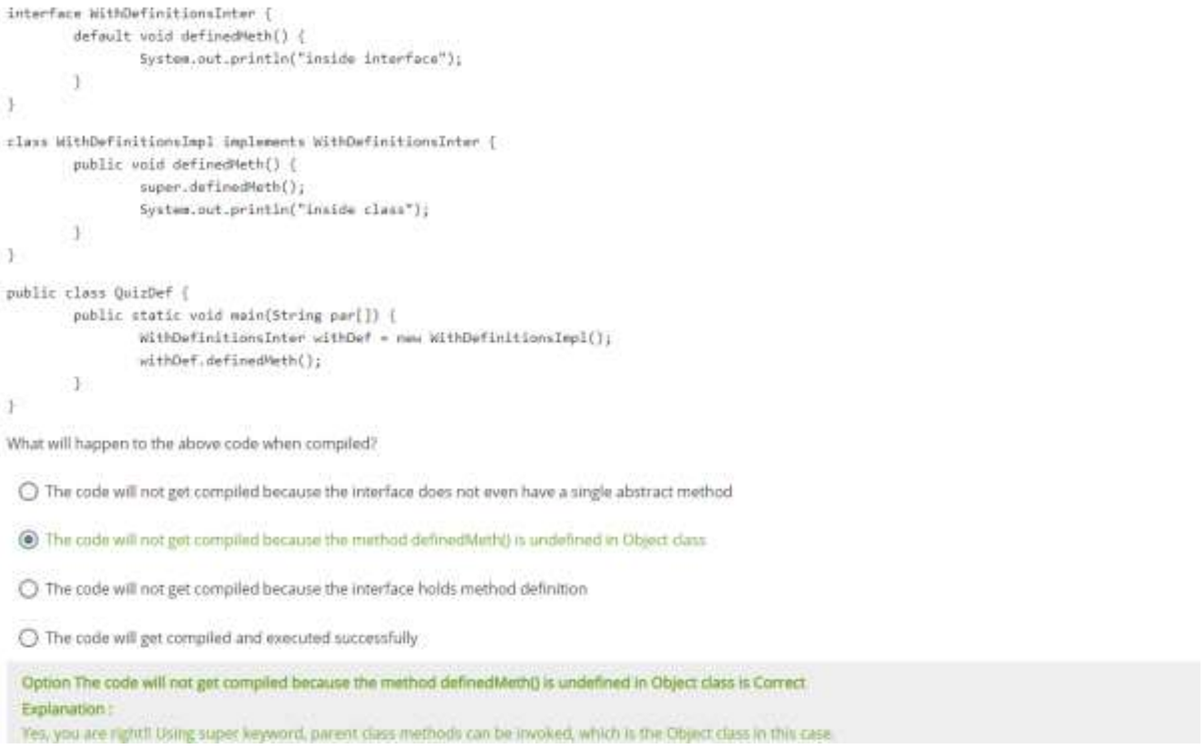
○ No successful compilation because the interface holds method definition

● The code will be executed successfully. And, the execution result will be, inside class

Option The code will be executed successfully. And, the execution result will be, inside class is Correct

Explanation :

Yes, you are right. When there is an overridden version for a default method is available, the overridden method will alone be considered and hence, the result is inside class.



```
interface WithDefinitionsInter {
        default void definedMeth() {
                System.out.println("inside interface");
        }
}

class WithDefinitionsImpl implements WithDefinitionsInter {
        public void definedMeth() {
                super.definedMeth();
                System.out.println("inside class");
        }
}

public class QuizDef {
        public static void main(String par[]) {
                WithDefinitionsInter withDef = new WithDefinitionsImpl();
                withDef.definedMeth();
        }
}
```

What will happen to the above code when compiled?

○ The code will not get compiled because the interface does not even have a single abstract method

● The code will not get compiled because the method definedMeth() is undefined in Object class

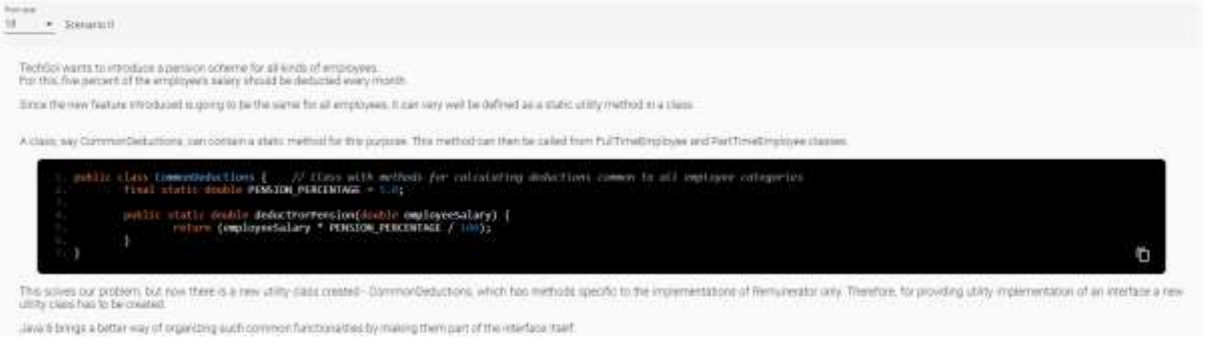
○ The code will not get compiled because the interface holds method definition

○ The code will get compiled and executed successfully

Option The code will not get compiled because the method definedMeth() is undefined in Object class is Correct

Explanation :

Yes, you are right!! Using super keyword, parent class methods can be invoked, which is the Object class in this case.



TechSol wants to introduce a pension scheme for all kinds of employees.
For this, five percent of the employee's salary should be deducted every month.

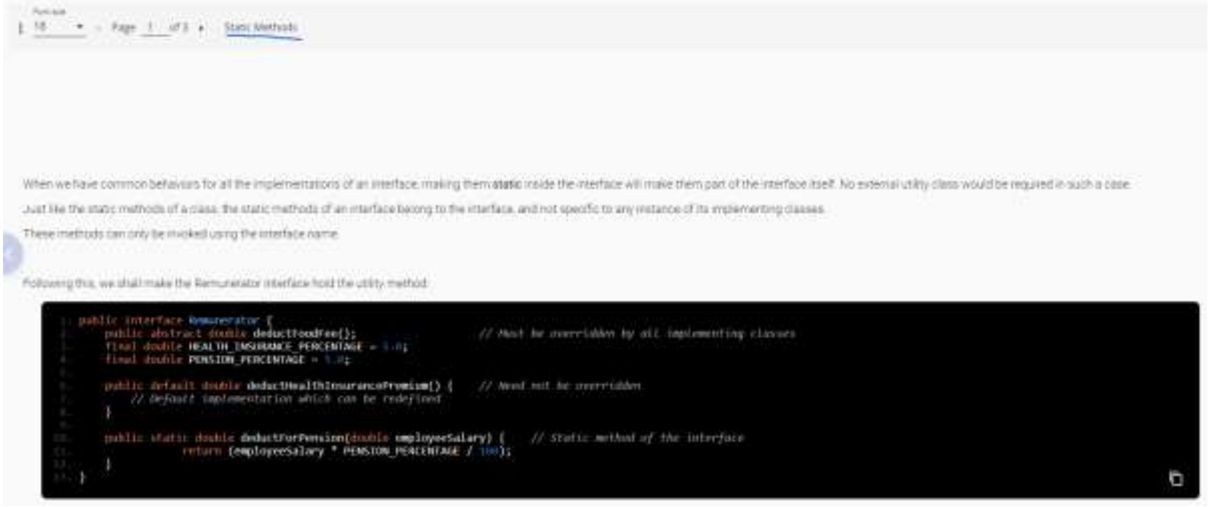
Since the new feature introduced is going to be the same for all employees, it can very well be defined as a static utility method in a class.

A class, say CommonDeductions, can contain a static method for this purpose. This method can then be called from FullTimeEmployee and PartTimeEmployee classes.

```
public class CommonDeductions {     // Class with methods for calculating deductions common to all employee categories
        final static double PENSION_PERCENTAGE = 5.0;

        public static double deductForPension(double employeeSalary) {
                return (employeeSalary * PENSION_PERCENTAGE / 100);
        }
}
```

This solves our problem, but now there is a new utility class created - CommonDeductions, which has methods specific to the implementations of Remunerator only. Therefore, for providing utility implementation of an interface a new utility class has to be created.

Java 8 brings a better way of organizing such common functionalities by making them part of the interface itself.



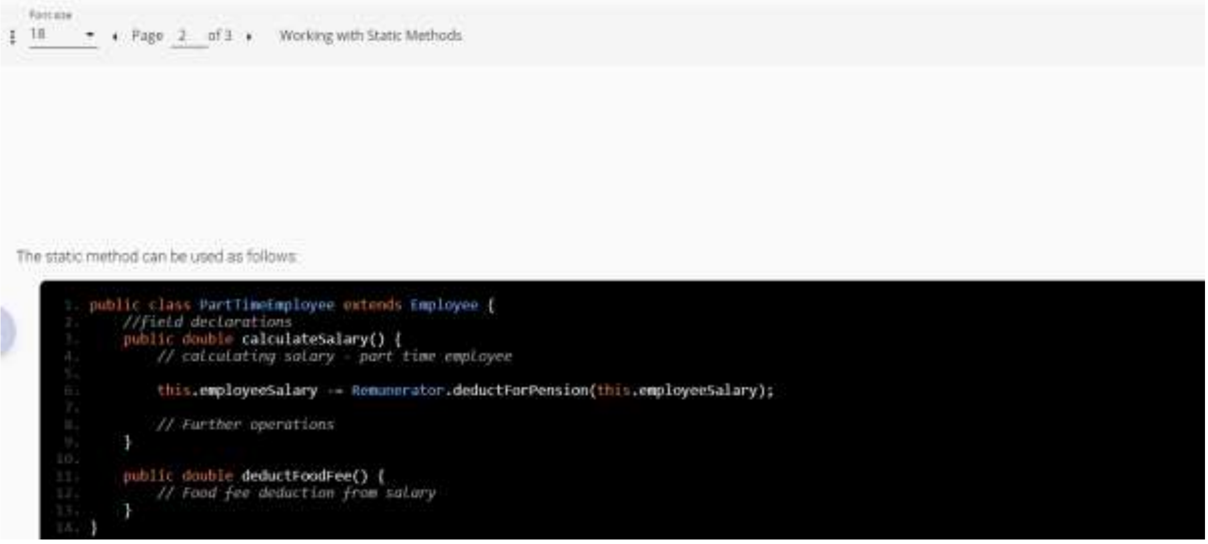
When we have common behaviors for all the implementations of an interface, making them **static** inside the interface will make them part of the interface itself. No external utility class would be required in such a case.

Just like the static methods of a class, the static methods of an interface belong to the interface, and not specific to any instance of its implementing classes.

These methods can only be invoked using the interface name.

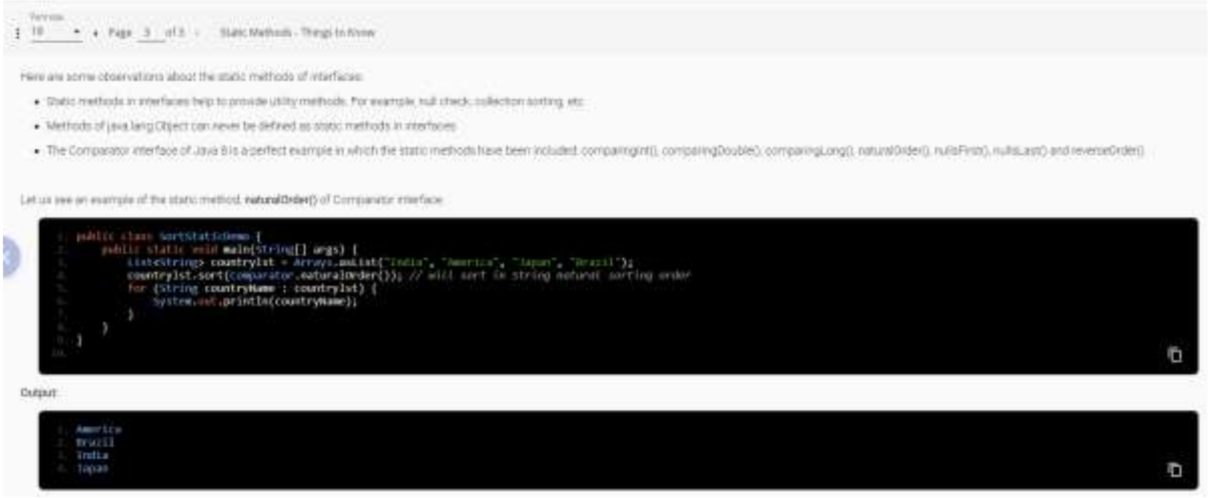
Following this, we shall make the Remunerator interface hold the utility method.

```
public interface Remunerator {
    public abstract double deductFoodFee();                    // Must be overridden by all implementing classes
    final double HEALTH_INSURANCE_PERCENTAGE = 5.0;
    final double PENSION_PERCENTAGE = 5.0;

    public default double deductHealthInsurancePremium() {    // Need not be overridden
        // Default implementation which can be redefined
    }

    public static double deductForPension(double employeeSalary) {    // Static method of the interface
            return (employeeSalary * PENSION_PERCENTAGE / 100);
    }
}
```



The static method can be used as follows:

```
public class PartTimeEmployee extends Employee {
    //field declarations
    public double calculateSalary() {
        // calculating salary - part time employee

        this.employeeSalary -= Remunerator.deductForPension(this.employeeSalary);

        // Further operations
    }

    public double deductFoodFee() {
        // Food fee deduction from salary
    }
}
```



Here are some observations about the static methods of interfaces:

- Static methods in interfaces help to provide utility methods. For example, null check, collection sorting, etc.
- Methods of java.lang.Object can never be defined as static methods in interfaces
- The Comparator interface of Java 8 is a perfect example in which the static methods have been included: comparingInt(), comparingDouble(), comparingLong(), naturalOrder(), nullsFirst(), nullsLast() and reverseOrder()

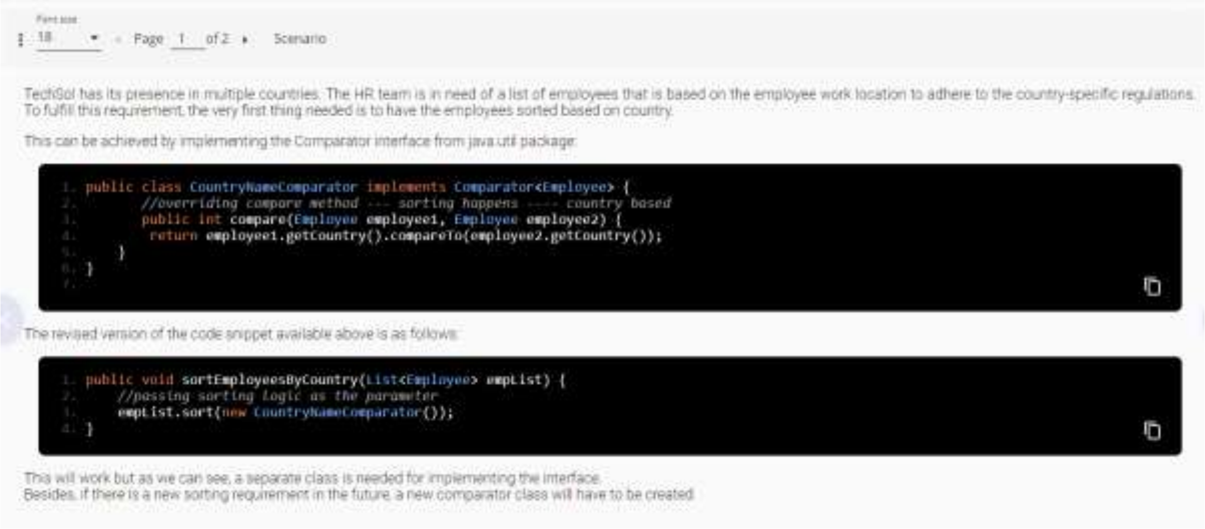
Let us see an example of the static method, **naturalOrder()** of Comparator interface:

```
public class SortStaticDemo {
    public static void main(String[] args) {
        List<String> countrylst = Arrays.asList("India", "America", "Japan", "Brazil");
        countrylst.sort(Comparator.naturalOrder()); // will sort in string natural sorting order
        for (String countryName : countrylst) {
            System.out.println(countryName);
        }
    }
}
```

Output:

```
America
Brazil
India
Japan
```

Lambda Expressions:



TechSol has its presence in multiple countries. The HR team is in need of a list of employees that is based on the employee work location to adhere to the country-specific regulations. To fulfill this requirement, the very first thing needed is to have the employees sorted based on country.

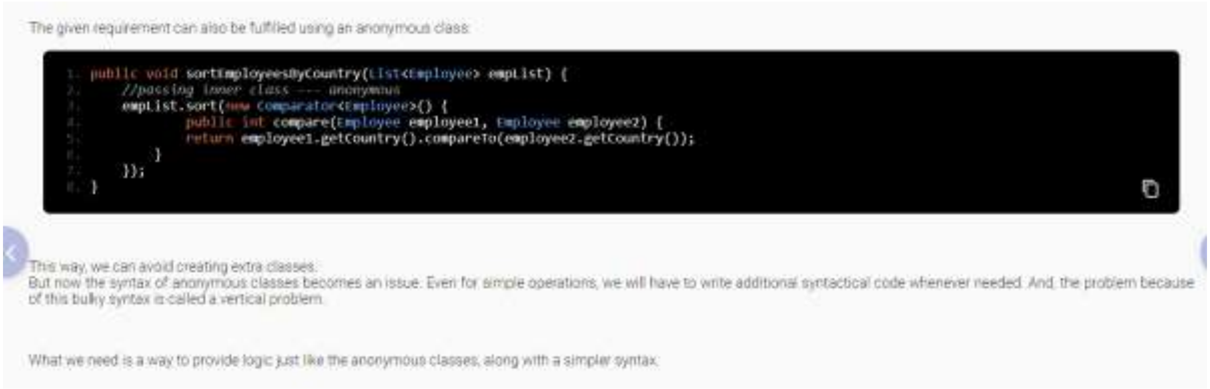
This can be achieved by implementing the Comparator interface from java.util package:

```
public class CountryNameComparator implements Comparator<Employee> {
        //overriding compare method --- sorting happens ---- country based
        public int compare(Employee employee1, Employee employee2) {
         return employee1.getCountry().compareTo(employee2.getCountry());
    }
}
```

The revised version of the code snippet available above is as follows:

```
public void sortEmployeesByCountry(List<Employee> empList) {
    //passing sorting logic as the parameter
    empList.sort(new CountryNameComparator());
}
```

This will work but as we can see, a separate class is needed for implementing the interface.
Besides, if there is a new sorting requirement in the future, a new comparator class will have to be created.

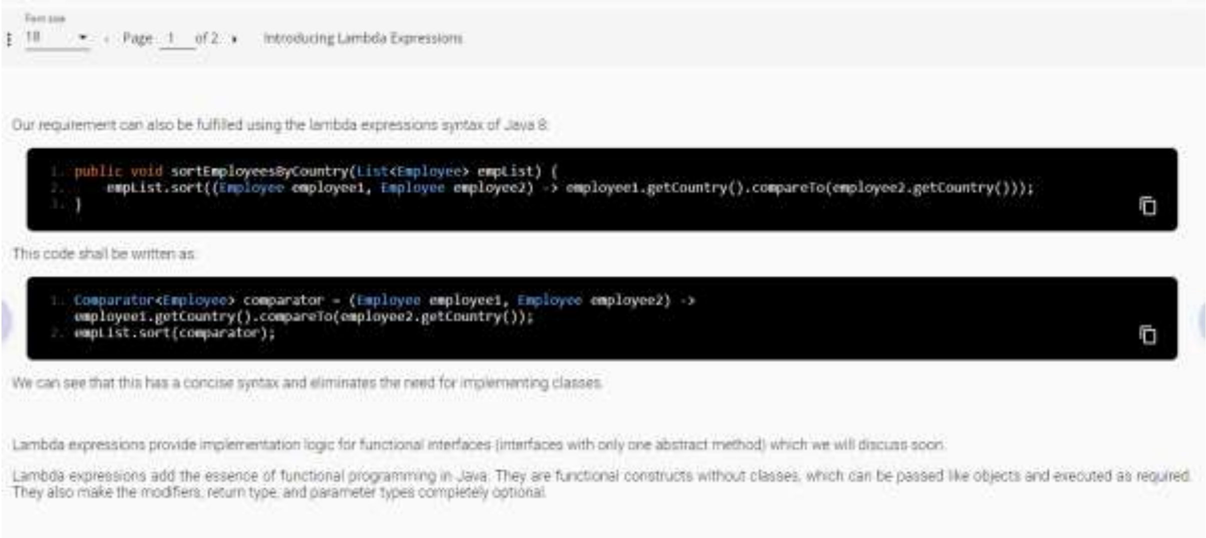


The given requirement can also be fulfilled using an anonymous class:

```
public void sortEmployeesByCountry(List<Employee> empList) {
    //passing inner class --- anonymous
    empList.sort(new Comparator<Employee>() {
            public int compare(Employee employee1, Employee employee2) {
            return employee1.getCountry().compareTo(employee2.getCountry());
        }
    });
}
```

This way, we can avoid creating extra classes.
But now the syntax of anonymous classes becomes an issue. Even for simple operations, we will have to write additional syntactical code whenever needed. And, the problem because of this bulky syntax is called a vertical problem.

What we need is a way to provide logic just like the anonymous classes, along with a simpler syntax.



Our requirement can also be fulfilled using the lambda expressions syntax of Java 8:

```
public void sortEmployeesByCountry(List<Employee> empList) {
    empList.sort((Employee employee1, Employee employee2) -> employee1.getCountry().compareTo(employee2.getCountry()));
}
```

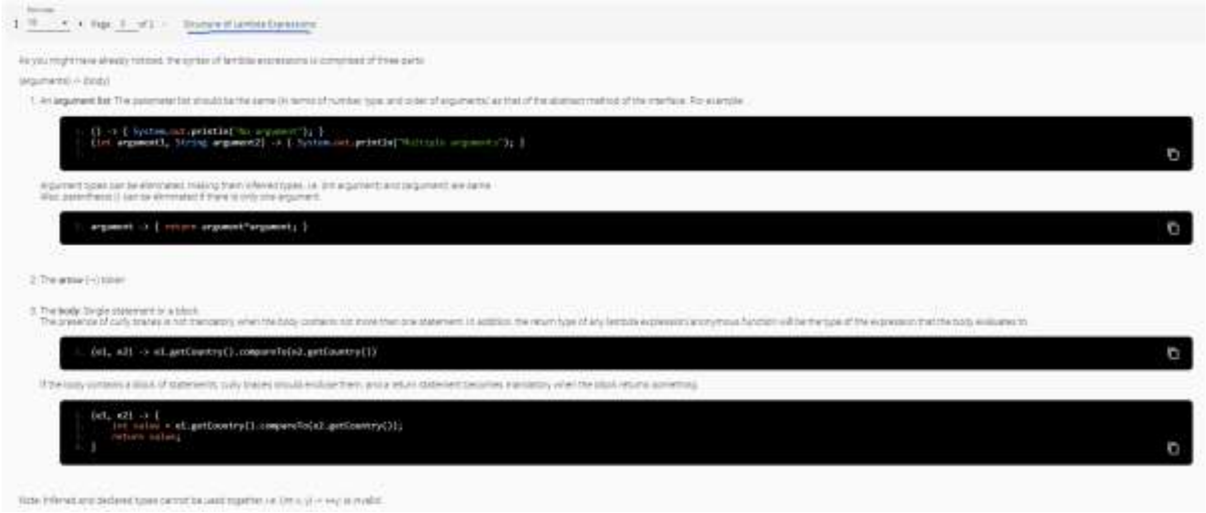
This code shall be written as:

```
Comparator<Employee> comparator = (Employee employee1, Employee employee2) ->
employee1.getCountry().compareTo(employee2.getCountry());
empList.sort(comparator);
```

We can see that this has a concise syntax and eliminates the need for implementing classes.

Lambda expressions provide implementation logic for functional interfaces (interfaces with only one abstract method) which we will discuss soon.

Lambda expressions add the essence of functional programming in Java. They are functional constructs without classes, which can be passed like objects and executed as required. They also make the modifiers, return type, and parameter types completely optional.



As you might have already noticed, the syntax of lambda expressions is comprised of three parts:

(arguments) -> (body)

1. An **argument list**: The parameter list should be the same (in terms of number, type and order of arguments) as that of the abstract method of the interface. For example:

```
() -> { System.out.println("No argument"); }
(int argument1, String argument2) -> { System.out.println("Multiple arguments"); }
```

Argument types can be eliminated, making them inferred types, i.e. (int argument) and (argument) are same.
Also, parenthesis () can be eliminated if there is only one argument.

```
argument -> { return argument*argument; }
```

2. The **arrow** (->) token

3. The **body**: Single statement or a block
The presence of curly braces is not mandatory when the body contains not more than one statement. In addition, the return type of any lambda expression/anonymous function will be the type of the expression that the body evaluates to.

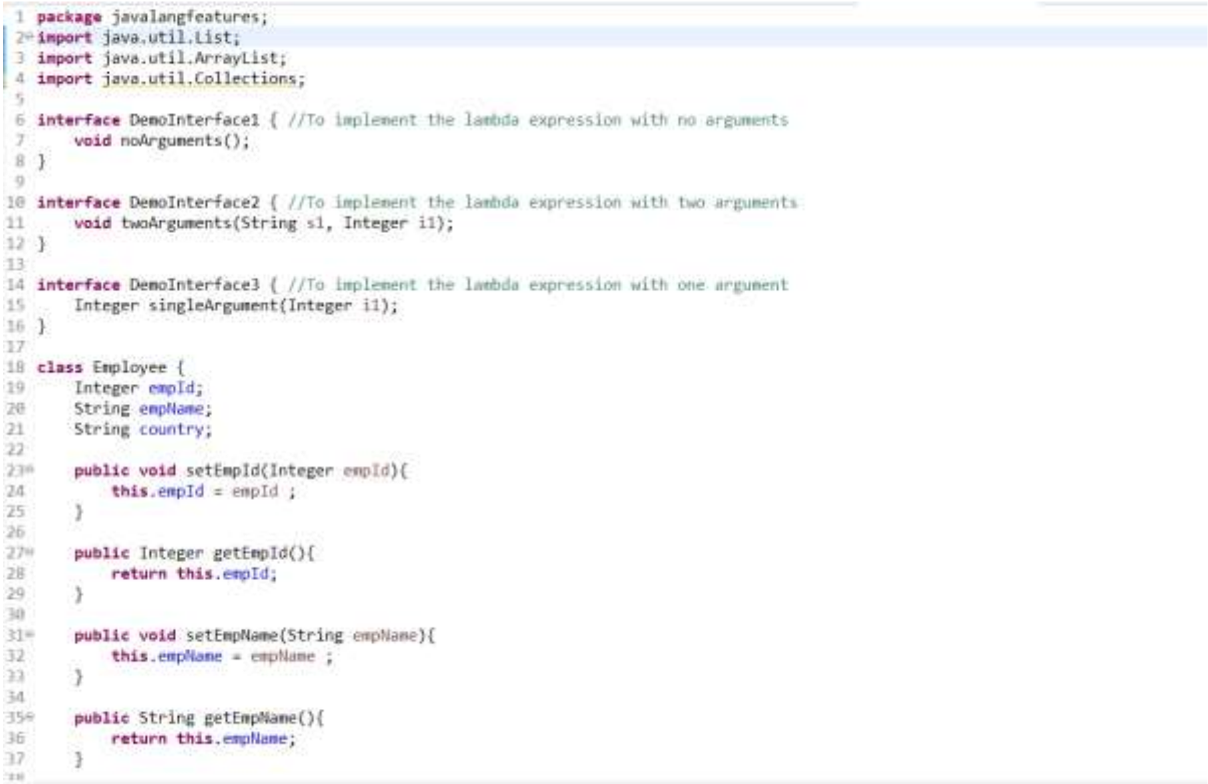
```
(e1, e2) -> e1.getCountry().compareTo(e2.getCountry())
```

If the body contains a block of statements, curly braces should enclose them, and a return statement becomes mandatory when the block returns something.

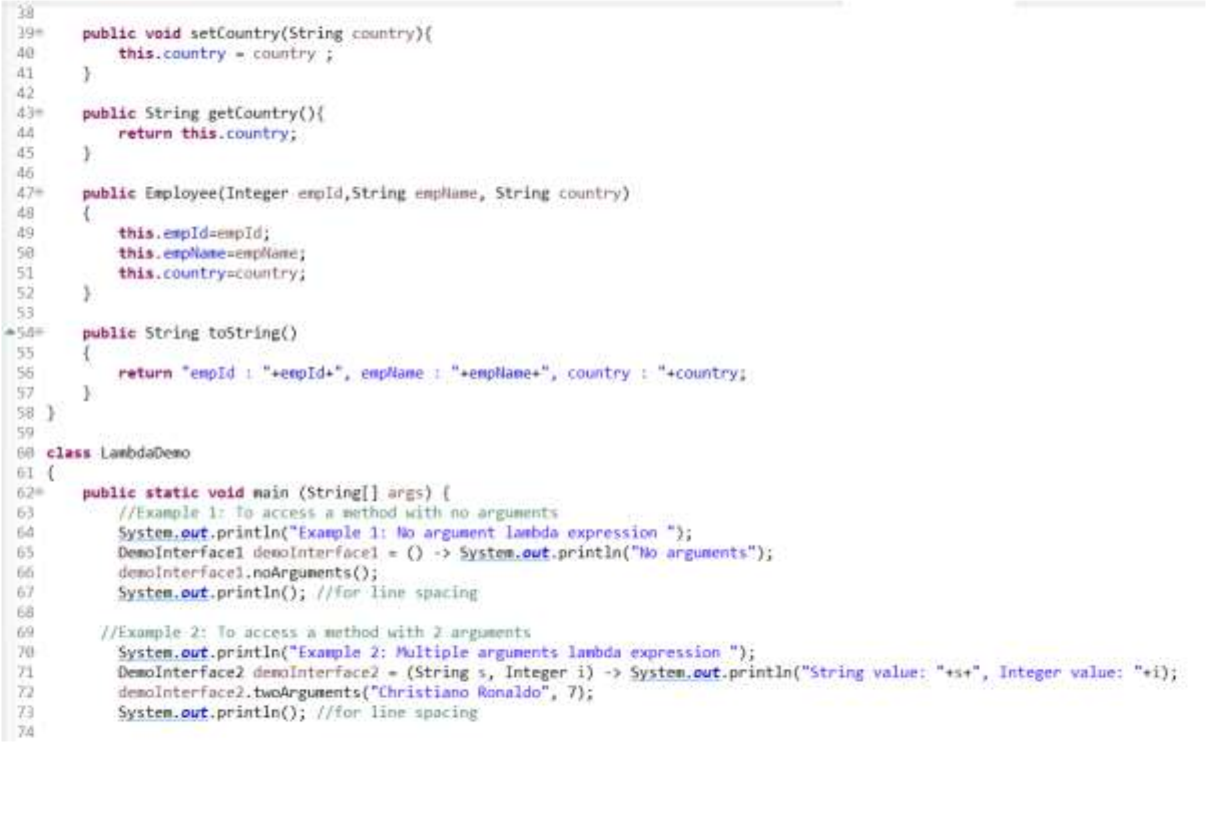
```
(e1, e2) -> {
    int value = e1.getCountry().compareTo(e2.getCountry());
    return value;
}
```

Note: Inferred and declared types cannot be used together i.e. (int x, y) -> x+y; is invalid.

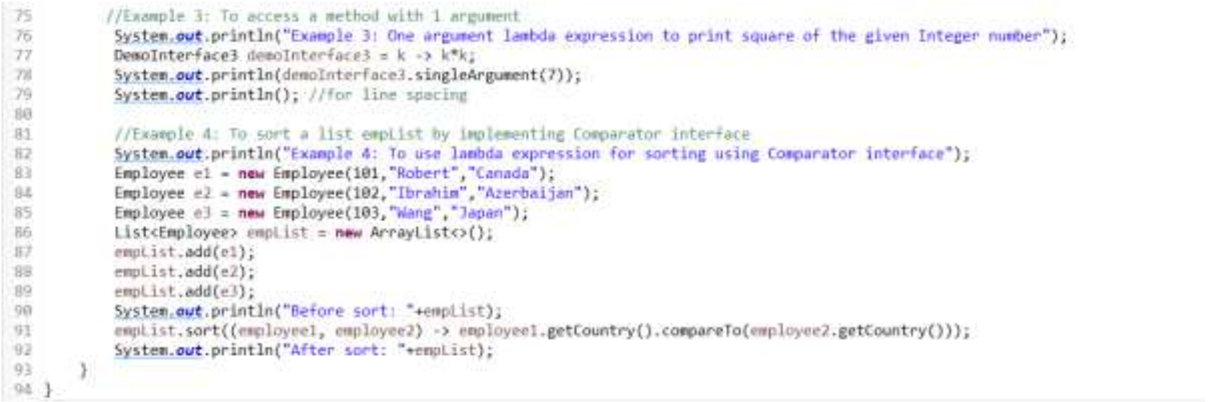
Lambda Demo:



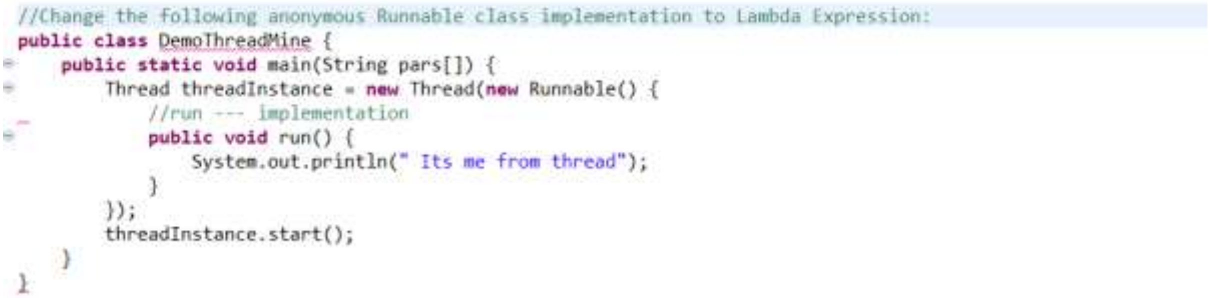
```java
package javalangfeatures;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

interface DemoInterface1 { //To implement the lambda expression with no arguments
    void noArguments();
}

interface DemoInterface2 { //To implement the lambda expression with two arguments
    void twoArguments(String s1, Integer i1);
}

interface DemoInterface3 { //To implement the lambda expression with one argument
    Integer singleArgument(Integer i1);
}

class Employee {
    Integer empId;
    String empName;
    String country;

    public void setEmpId(Integer empId){
        this.empId = empId ;
    }

    public Integer getEmpId(){
        return this.empId;
    }

    public void setEmpName(String empName){
        this.empName = empName ;
    }

    public String getEmpName(){
        return this.empName;
    }
```



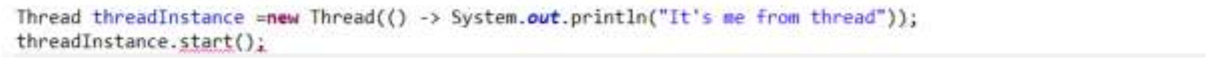
```java

    public void setCountry(String country){
        this.country = country ;
    }

    public String getCountry(){
        return this.country;
    }

    public Employee(Integer empId,String empName, String country)
    {
        this.empId=empId;
        this.empName=empName;
        this.country=country;
    }

    public String toString()
    {
        return "empId : "+empId+", empName : "+empName+", country : "+country;
    }
}

class LambdaDemo
{
    public static void main (String[] args) {
        //Example 1: To access a method with no arguments
        System.out.println("Example 1: No argument lambda expression ");
        DemoInterface1 demoInterface1 = () -> System.out.println("No arguments");
        demoInterface1.noArguments();
        System.out.println(); //for line spacing

      //Example 2: To access a method with 2 arguments
        System.out.println("Example 2: Multiple arguments lambda expression ");
        DemoInterface2 demoInterface2 = (String s, Integer i) -> System.out.println("String value: "+s+", Integer value: "+i);
        demoInterface2.twoArguments("Christiano Ronaldo", 7);
        System.out.println(); //for line spacing

```



```
        //Example 3: To access a method with 1 argument
        System.out.println("Example 3: One argument lambda expression to print square of the given Integer number");
        DemoInterface3 demoInterface3 = k -> k*k;
        System.out.println(demoInterface3.singleArgument(7));
        System.out.println(); //for line spacing

        //Example 4: To sort a list empList by implementing Comparator interface
        System.out.println("Example 4: To use lambda expression for sorting using Comparator interface");
        Employee e1 = new Employee(101,"Robert","Canada");
        Employee e2 = new Employee(102,"Ibrahim","Azerbaijan");
        Employee e3 = new Employee(103,"Wang","Japan");
        List<Employee> empList = new ArrayList<>();
        empList.add(e1);
        empList.add(e2);
        empList.add(e3);
        System.out.println("Before sort: "+empList);
        empList.sort((employee1, employee2) -> employee1.getCountry().compareTo(employee2.getCountry()));
        System.out.println("After sort: "+empList);
    }
}
```

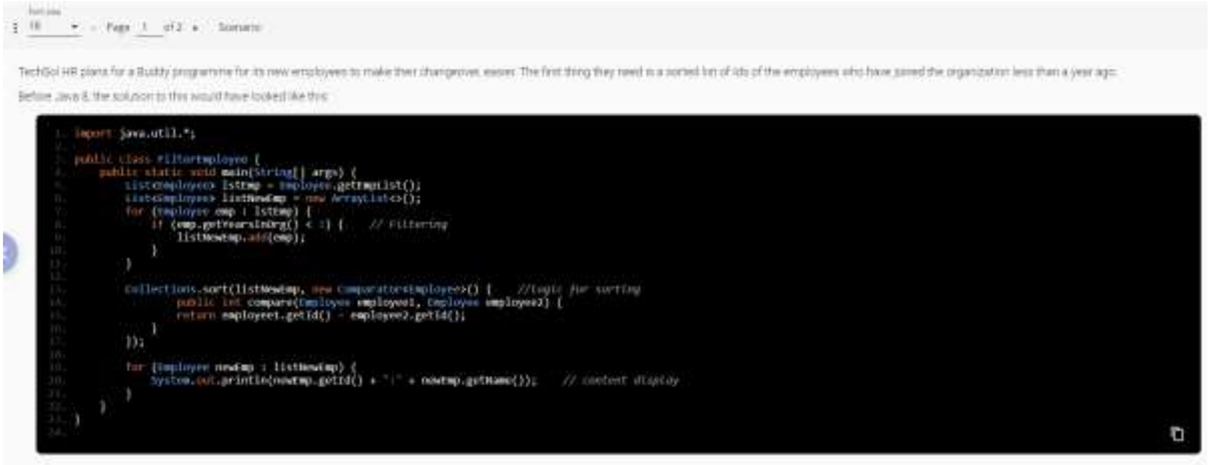


```
//Change the following anonymous Runnable class implementation to Lambda Expression:
public class DemoThreadMine {
    public static void main(String pars[]) {
        Thread threadInstance = new Thread(new Runnable() {
            //run --- implementation
            public void run() {
                System.out.println(" Its me from thread");
            }
        });
        threadInstance.start();
    }
}
```



```
Thread threadInstance =new Thread(() -> System.out.println("It's me from thread"));
threadInstance.start();
```

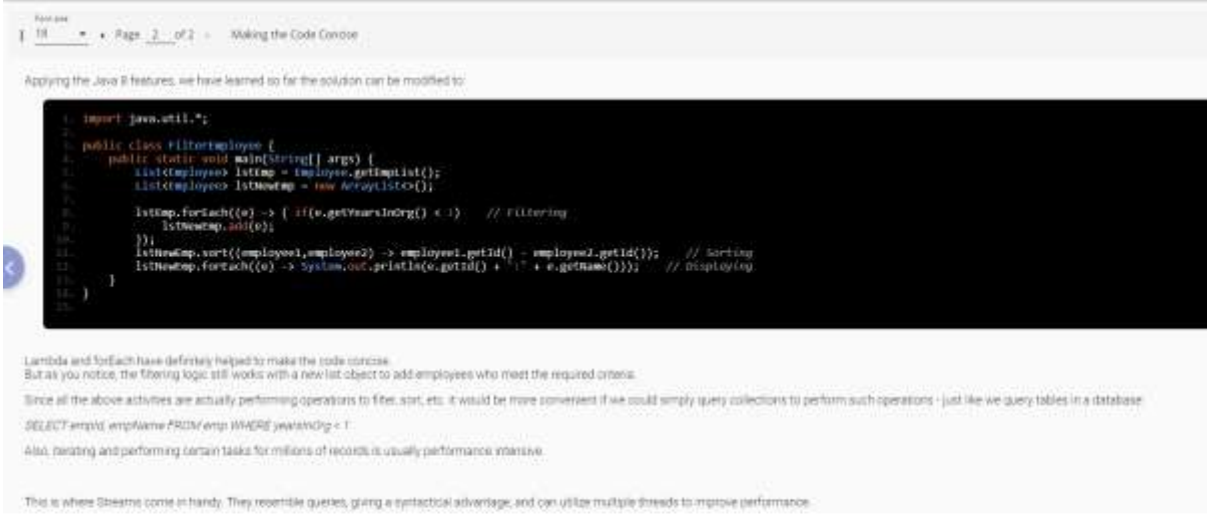
STREAMS:



TechSol HR plans for a Buddy programme for its new employees to make their changeover easier. The first thing they need is a sorted list of ids of the employees who have joined the organization less than a year ago.

Before Java 8, the solution to this would have looked like this:

```
import java.util.*;

public class FilterEmployee {
    public static void main(String[] args) {
        List<Employee> lstEmp = Employee.getEmpList();
        List<Employee> listNewEmp = new ArrayList<>();
        for (Employee emp : lstEmp) {
            if (emp.getYearsInOrg() < 1) {     // Filtering
                listNewEmp.add(emp);
            }
        }

        Collections.sort(listNewEmp, new Comparator<Employee>() {     //logic for sorting
                public int compare(Employee employee1, Employee employee2) {
                return employee1.getId() - employee2.getId();
            }
        });

        for (Employee newEmp : listNewEmp) {
            System.out.println(newEmp.getId() + ":" + newEmp.getName());     // content display
        }
    }
}
```



Applying the Java 8 features, we have learned so far the solution can be modified to:

```
import java.util.*;

public class FilterEmployee {
    public static void main(String[] args) {
        List<Employee> lstEmp = Employee.getEmpList();
        List<Employee> lstNewEmp = new ArrayList<>();

        lstEmp.forEach((e) -> { if(e.getYearsInOrg() < 1)     // Filtering
            lstNewEmp.add(e);
        });
        lstNewEmp.sort((employee1,employee2) -> employee1.getId() - employee2.getId());     // Sorting
        lstNewEmp.forEach((e) -> System.out.println(e.getId() + ":" + e.getName()));     // Displaying
    }
}
```

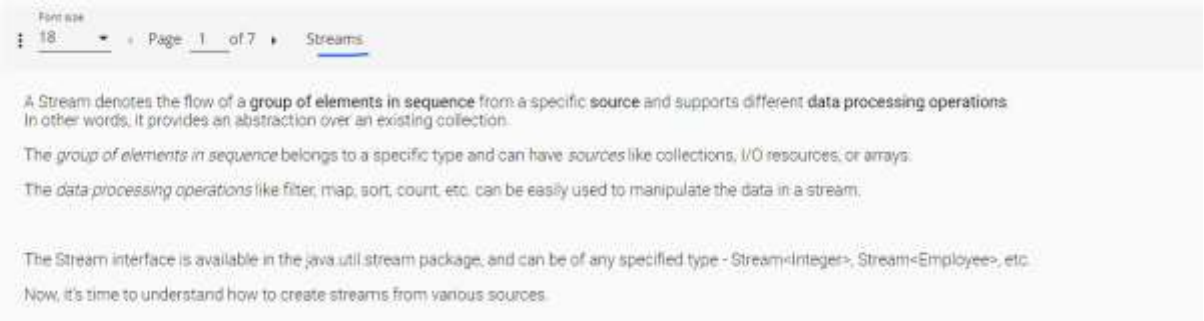
Lambda and forEach have definitely helped to make the code concise.
But as you notice, the filtering logic still works with a new list object to add employees who meet the required criteria.

Since all the above activities are actually performing operations to filter, sort, etc. it would be more convenient if we could simply query collections to perform such operations - just like we query tables in a database.

SELECT empId, empName FROM emp WHERE yearsInOrg < 1

Also, iterating and performing certain tasks for millions of records is usually performance intensive.

This is where Streams come in handy. They resemble queries, giving a syntactical advantage, and can utilize multiple threads to improve performance.



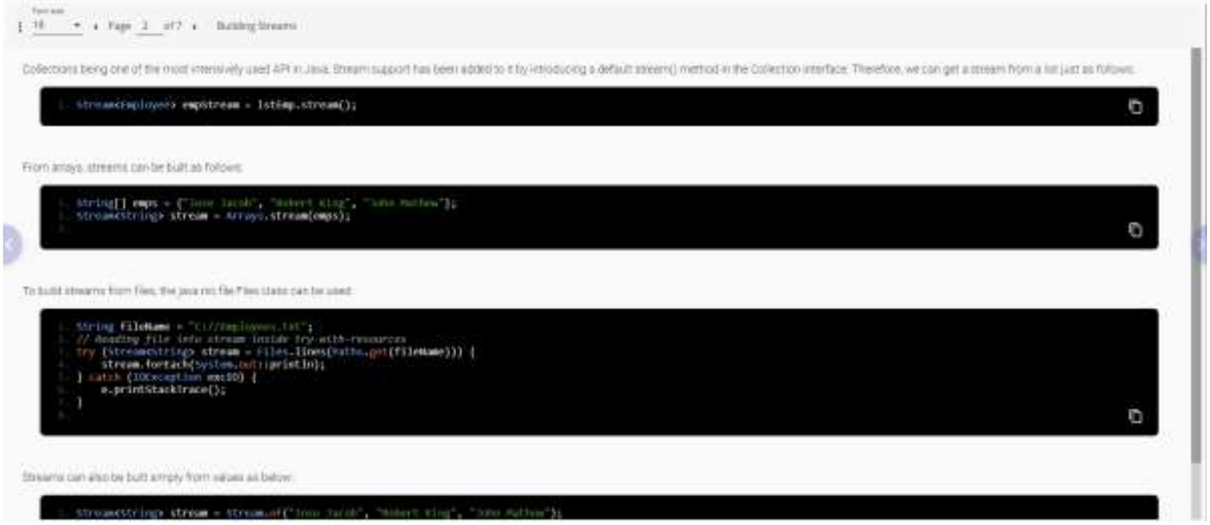
A Stream denotes the flow of a **group of elements in sequence** from a specific **source** and supports different **data processing operations**.
In other words, it provides an abstraction over an existing collection.

The *group of elements in sequence* belongs to a specific type and can have *sources* like collections, I/O resources, or arrays.

The *data processing operations* like filter, map, sort, count, etc. can be easily used to manipulate the data in a stream.

The Stream interface is available in the java.util.stream package, and can be of any specified type - Stream<Integer>, Stream<Employee>, etc.

Now, it's time to understand how to create streams from various sources.



Collections being one of the most intensively used API in Java, Stream support has been added to it by introducing a default stream() method in the Collection interface. Therefore, we can get a stream from a list just as follows:

```
Stream<Employee> empStream = lstEmp.stream();
```

From arrays, streams can be built as follows:

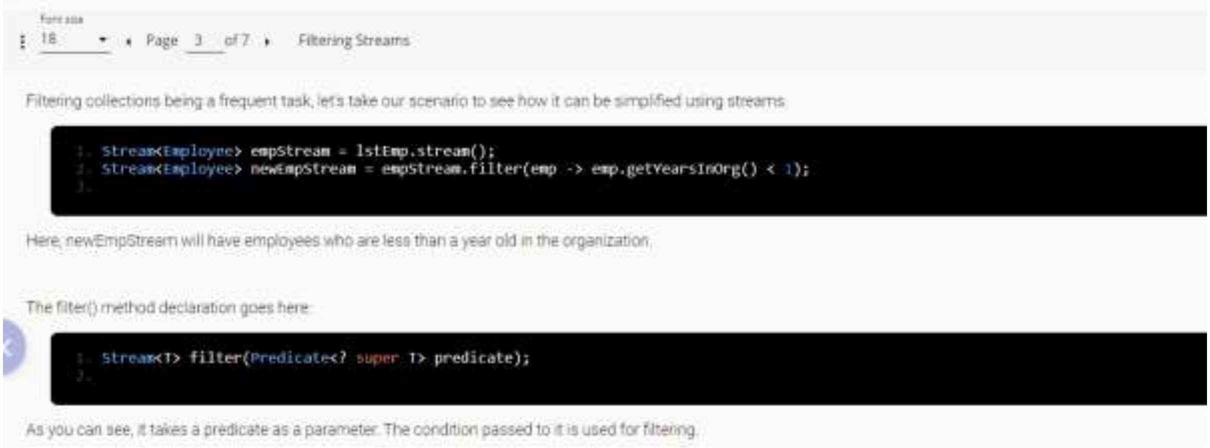
```
String[] emps = {"Jose Jacob", "Robert King", "John Mathew"};
Stream<String> stream = Arrays.stream(emps);
```

To build streams from files, the java.nio.file.Files class can be used:

```
String fileName = "C://Employees.txt";
// Reading file into stream inside try-with-resources
try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
    stream.forEach(System.out::println);
} catch (IOException exIO) {
    e.printStackTrace();
}
```

Streams can also be built simply from values as below:

```
Stream<String> stream = Stream.of("Jose Jacob", "Robert King", "John Mathew");
```



Filtering Streams

Filtering collections being a frequent task, let's take our scenario to see how it can be simplified using streams.

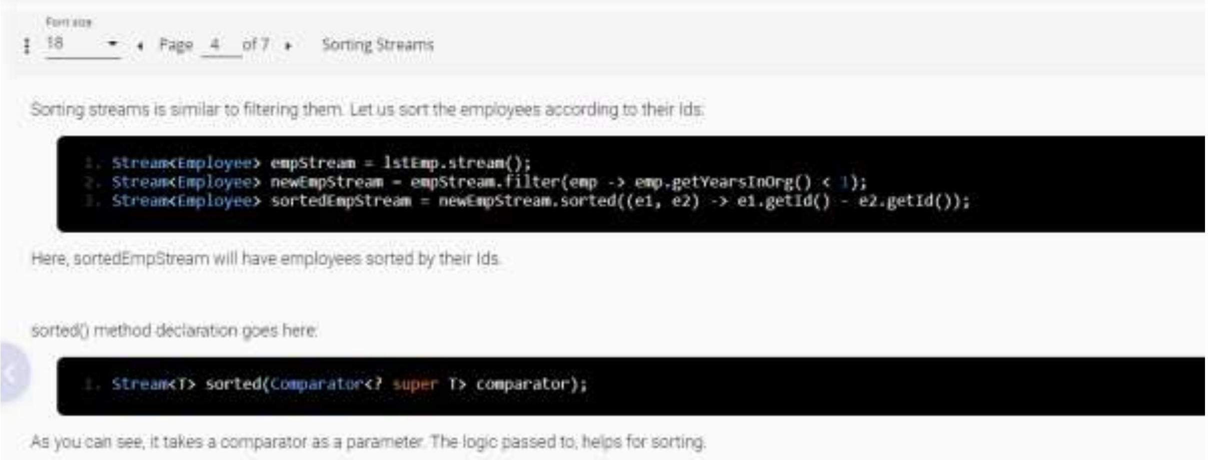
```java
Stream<Employee> empStream = lstEmp.stream();
Stream<Employee> newEmpStream = empStream.filter(emp -> emp.getYearsInOrg() < 1);

```

Here, newEmpStream will have employees who are less than a year old in the organization.

The filter() method declaration goes here:

```java
Stream<T> filter(Predicate<? super T> predicate);

```

As you can see, it takes a predicate as a parameter. The condition passed to it is used for filtering.



Sorting Streams

Sorting streams is similar to filtering them. Let us sort the employees according to their Ids:

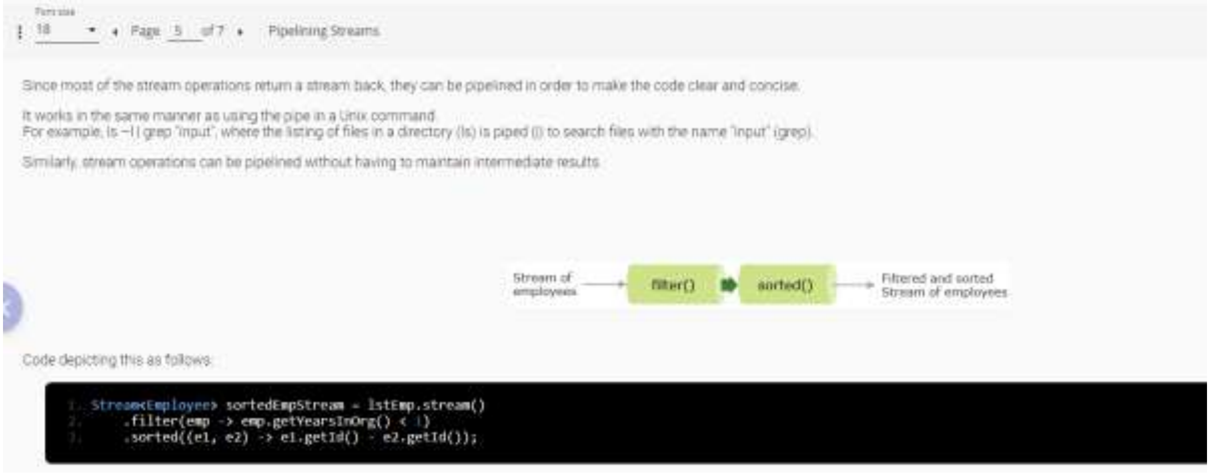
```java
Stream<Employee> empStream = lstEmp.stream();
Stream<Employee> newEmpStream = empStream.filter(emp -> emp.getYearsInOrg() < 1);
Stream<Employee> sortedEmpStream = newEmpStream.sorted((e1, e2) -> e1.getId() - e2.getId());
```

Here, sortedEmpStream will have employees sorted by their Ids.

sorted() method declaration goes here:

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

As you can see, it takes a comparator as a parameter. The logic passed to, helps for sorting.



Pipelining Streams

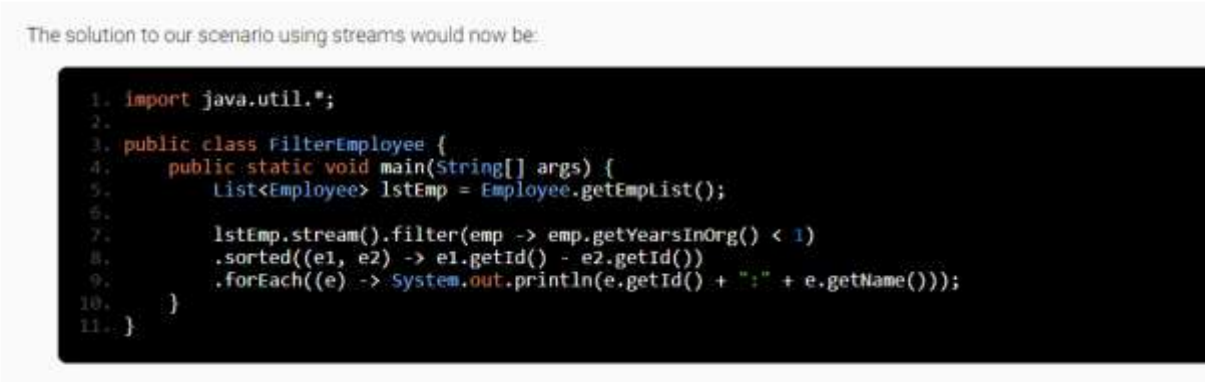
Since most of the stream operations return a stream back, they can be pipelined in order to make the code clear and concise.

It works in the same manner as using the pipe in a Unix command.
For example, ls –l | grep "input", where the listing of files in a directory (ls) is piped (|) to search files with the name "input" (grep).

Similarly, stream operations can be pipelined without having to maintain intermediate results.

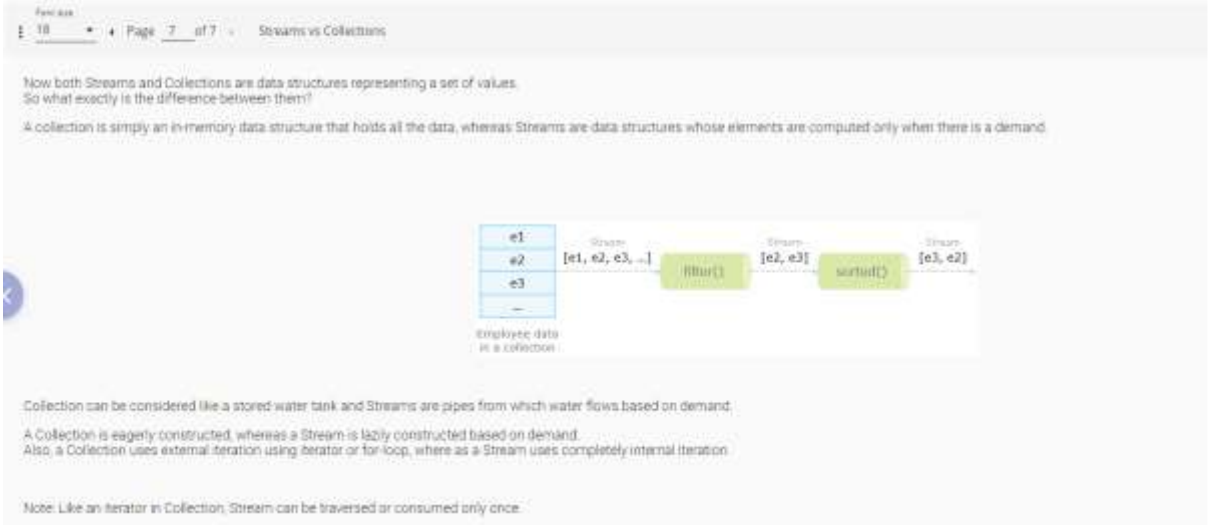
Code depicting this as follows:

```java
Stream<Employee> sortedEmpStream = lstEmp.stream()
    .filter(emp -> emp.getYearsInOrg() < 1)
    .sorted((e1, e2) -> e1.getId() - e2.getId());
```



The solution to our scenario using streams would now be:

```java
import java.util.*;

public class FilterEmployee {
    public static void main(String[] args) {
        List<Employee> lstEmp = Employee.getEmpList();

        lstEmp.stream().filter(emp -> emp.getYearsInOrg() < 1)
        .sorted((e1, e2) -> e1.getId() - e2.getId())
        .forEach((e) -> System.out.println(e.getId() + ":" + e.getName()));
    }
}
```



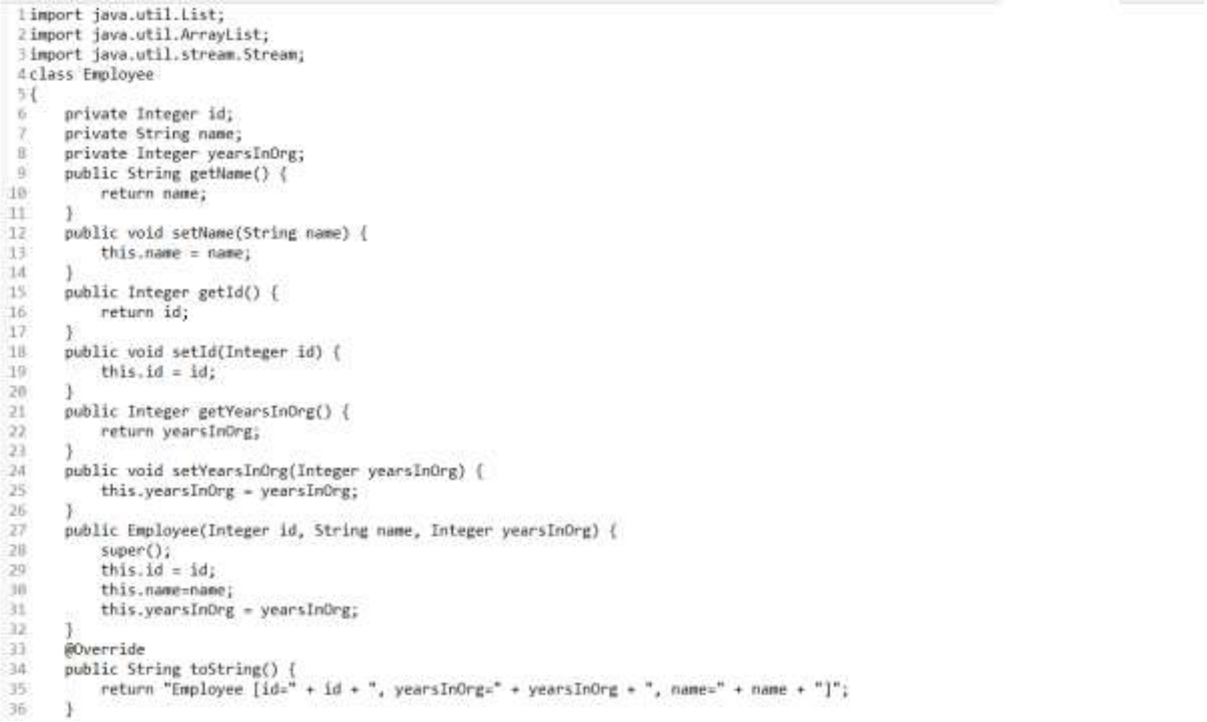
Now both Streams and Collections are data structures representing a set of values.
So what exactly is the difference between them?

A collection is simply an in-memory data structure that holds all the data, whereas Streams are data structures whose elements are computed only when there is a demand.

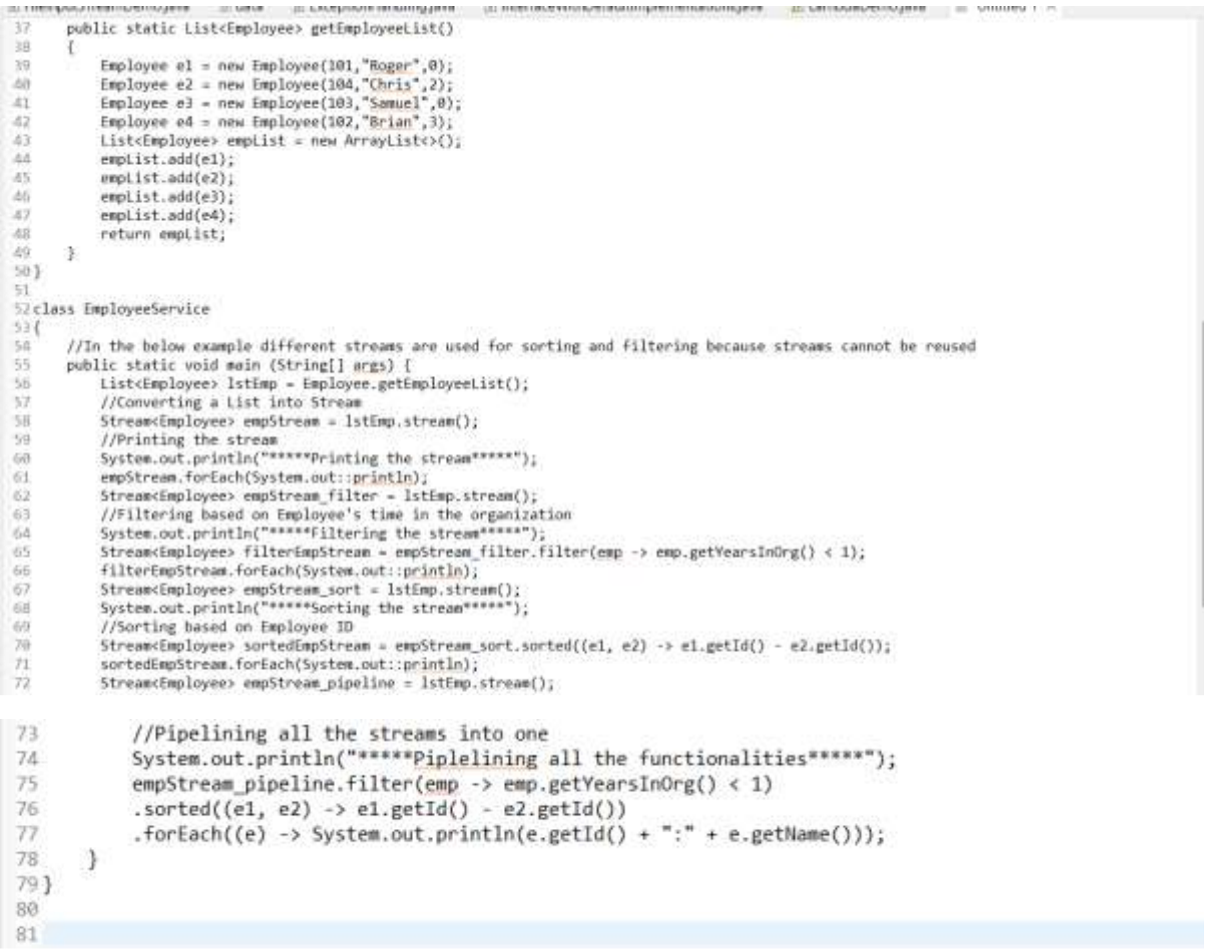
Collection can be considered like a stored water tank and Streams are pipes from which water flows based on demand.

A Collection is eagerly constructed, whereas a Stream is lazily constructed based on demand.
Also, a Collection uses external iteration using iterator or for-loop, where as a Stream uses completely internal iteration.

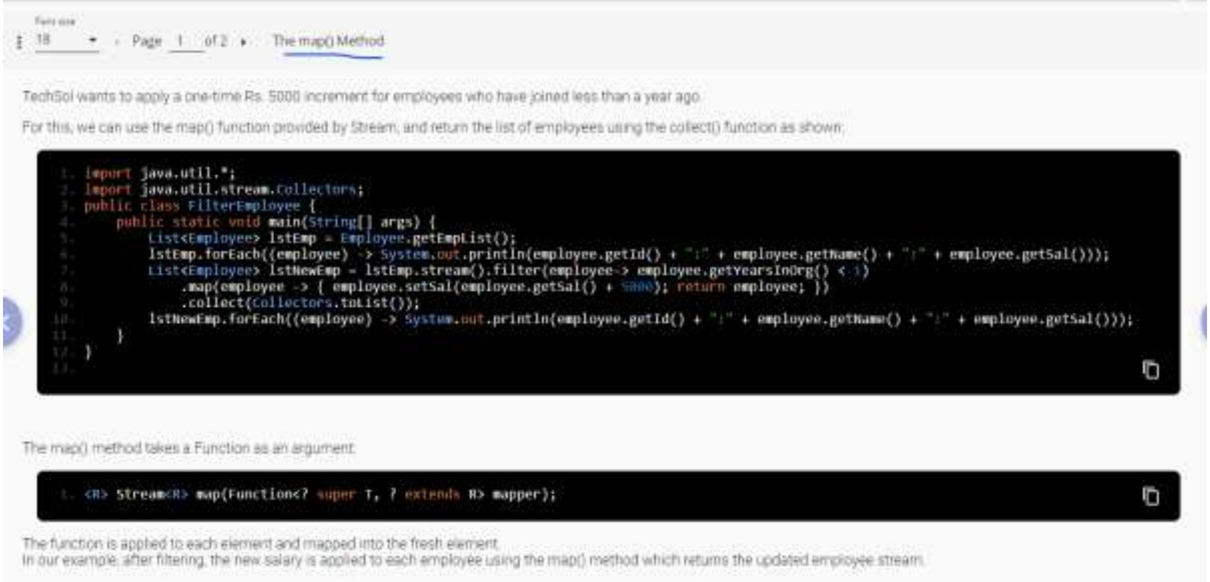
Note: Like an iterator in Collection, Stream can be traversed or consumed only once.



```java
import java.util.List;
import java.util.ArrayList;
import java.util.stream.Stream;
class Employee
{
    private Integer id;
    private String name;
    private Integer yearsInOrg;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public Integer getYearsInOrg() {
        return yearsInOrg;
    }
    public void setYearsInOrg(Integer yearsInOrg) {
        this.yearsInOrg = yearsInOrg;
    }
    public Employee(Integer id, String name, Integer yearsInOrg) {
        super();
        this.id = id;
        this.name=name;
        this.yearsInOrg = yearsInOrg;
    }
    @Override
    public String toString() {
        return "Employee [id=" + id + ", yearsInOrg=" + yearsInOrg + ", name=" + name + "]";
    }
```



```java
    public static List<Employee> getEmployeeList()
    {
        Employee e1 = new Employee(101,"Roger",0);
        Employee e2 = new Employee(104,"Chris",2);
        Employee e3 = new Employee(103,"Samuel",0);
        Employee e4 = new Employee(102,"Brian",3);
        List<Employee> empList = new ArrayList<>();
        empList.add(e1);
        empList.add(e2);
        empList.add(e3);
        empList.add(e4);
        return empList;
    }
}

class EmployeeService
{
    //In the below example different streams are used for sorting and filtering because streams cannot be reused
    public static void main (String[] args) {
        List<Employee> lstEmp = Employee.getEmployeeList();
        //Converting a List into Stream
        Stream<Employee> empStream = lstEmp.stream();
        //Printing the stream
        System.out.println("*****Printing the stream*****");
        empStream.forEach(System.out::println);
        Stream<Employee> empStream_filter = lstEmp.stream();
        //Filtering based on Employee's time in the organization
        System.out.println("*****Filtering the stream*****");
        Stream<Employee> filterEmpStream = empStream_filter.filter(emp -> emp.getYearsInOrg() < 1);
        filterEmpStream.forEach(System.out::println);
        Stream<Employee> empStream_sort = lstEmp.stream();
        System.out.println("*****Sorting the stream*****");
        //Sorting based on Employee ID
        Stream<Employee> sortedEmpStream = empStream_sort.sorted((e1, e2) -> e1.getId() - e2.getId());
        sortedEmpStream.forEach(System.out::println);
        Stream<Employee> empStream_pipeline = lstEmp.stream();
        //Pipelining all the streams into one
        System.out.println("*****Piplelining all the functionalities*****");
        empStream_pipeline.filter(emp -> emp.getYearsInOrg() < 1)
        .sorted((e1, e2) -> e1.getId() - e2.getId())
        .forEach((e) -> System.out.println(e.getId() + ":" + e.getName()));
    }
}
```



TechSol wants to apply a one-time Rs. 5000 increment for employees who have joined less than a year ago.

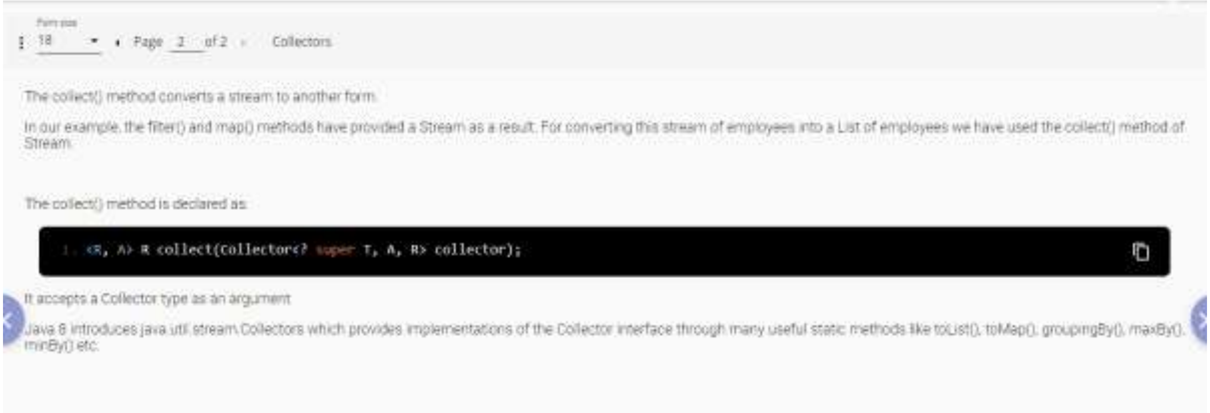
For this, we can use the map() function provided by Stream, and return the list of employees using the collect() function as shown:

```java
import java.util.*;
import java.util.stream.Collectors;
public class FilterEmployee {
    public static void main(String[] args) {
        List<Employee> lstEmp = Employee.getEmpList();
        lstEmp.forEach((employee) -> System.out.println(employee.getId() + ":" + employee.getName() + ":" + employee.getSal()));
        List<Employee> lstNewEmp = lstEmp.stream().filter(employee-> employee.getYearsInOrg() < 1)
            .map(employee -> { employee.setSal(employee.getSal() + 5000); return employee; })
            .collect(Collectors.toList());
        lstNewEmp.forEach((employee) -> System.out.println(employee.getId() + ":" + employee.getName() + ":" + employee.getSal()));
    }
}
```

The map() method takes a Function as an argument.

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

The function is applied to each element and mapped into the fresh element.
In our example, after filtering, the new salary is applied to each employee using the map() method which returns the updated employee stream.



The collect() method converts a stream to another form.

In our example, the filter() and map() methods have provided a Stream as a result. For converting this stream of employees into a List of employees we have used the collect() method of Stream.

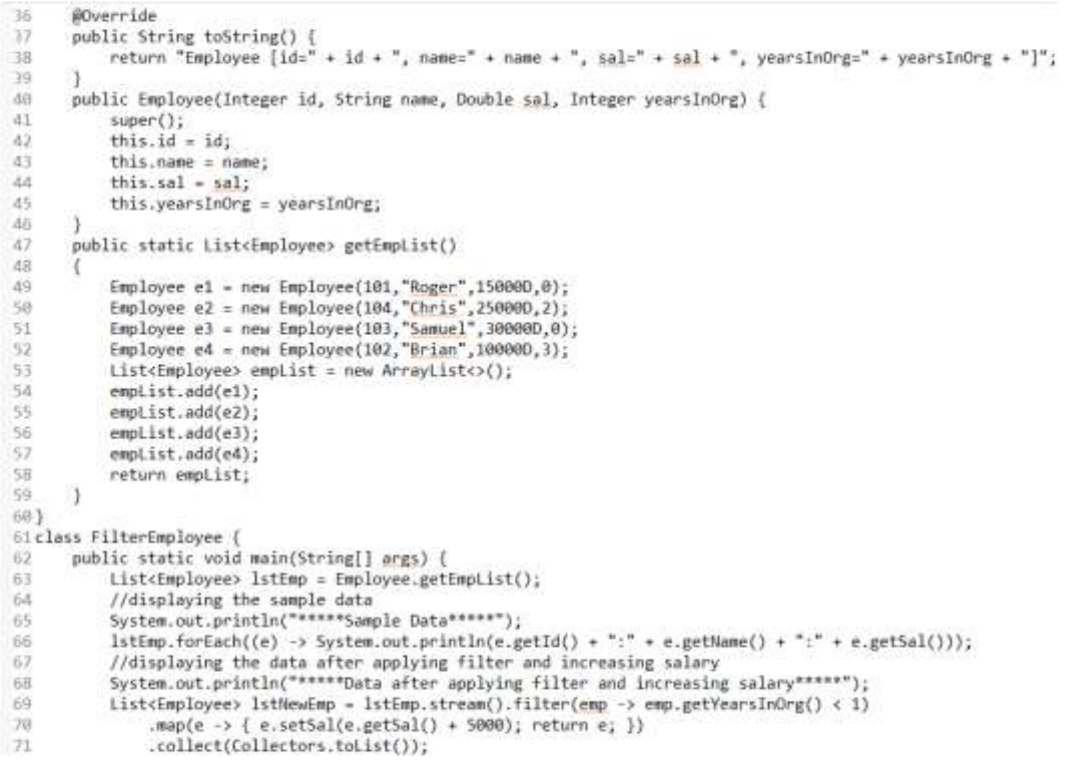
The collect() method is declared as:

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

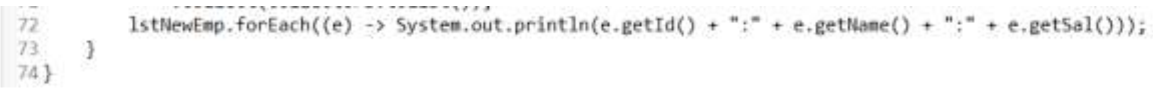
It accepts a Collector type as an argument.

Java 8 introduces java.util.stream.Collectors which provides implementations of the Collector interface through many useful static methods like toList(), toMap(), groupingBy(), maxBy(), minBy() etc.

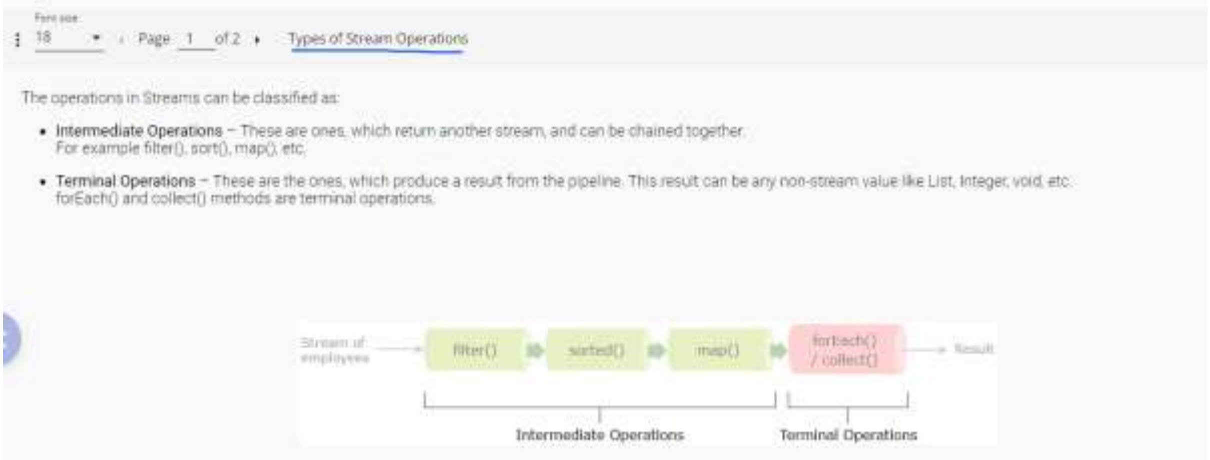
```
import java.util.List;
import java.util.ArrayList;
import java.util.stream.*;
class Employee
{
    private Integer id;
    private String name;
    private Double sal;
    private Integer yearsInOrg;

    public String getName() {
        return name;
    }
    public Double getSal() {
        return sal;
    }
    public void setSal(Double sal) {
        this.sal = sal;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public Integer getYearsInOrg() {
        return yearsInOrg;
    }
    public void setYearsInOrg(Integer yearsInOrg) {
        this.yearsInOrg = yearsInOrg;
    }

```



```
    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", sal=" + sal + ", yearsInOrg=" + yearsInOrg + "]";
    }
    public Employee(Integer id, String name, Double sal, Integer yearsInOrg) {
        super();
        this.id = id;
        this.name = name;
        this.sal = sal;
        this.yearsInOrg = yearsInOrg;
    }
    public static List<Employee> getEmpList()
    {
        Employee e1 = new Employee(101,"Roger",15000D,0);
        Employee e2 = new Employee(104,"Chris",25000D,2);
        Employee e3 = new Employee(103,"Samuel",30000D,0);
        Employee e4 = new Employee(102,"Brian",10000D,3);
        List<Employee> empList = new ArrayList<>();
        empList.add(e1);
        empList.add(e2);
        empList.add(e3);
        empList.add(e4);
        return empList;
    }
}
class FilterEmployee {
    public static void main(String[] args) {
        List<Employee> lstEmp = Employee.getEmpList();
        //displaying the sample data
        System.out.println("*****Sample Data*****");
        lstEmp.forEach((e) -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
        //displaying the data after applying filter and increasing salary
        System.out.println("*****Data after applying filter and increasing salary*****");
        List<Employee> lstNewEmp = lstEmp.stream().filter(emp -> emp.getYearsInOrg() < 1)
            .map(e -> { e.setSal(e.getSal() + 5000); return e; })
            .collect(Collectors.toList());
```

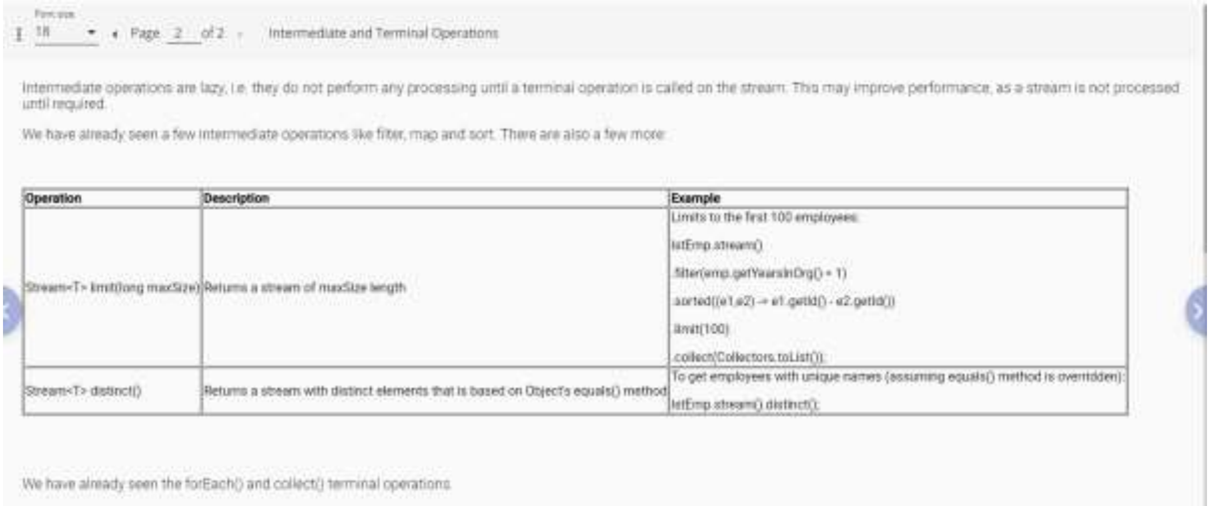


```
        lstNewEmp.forEach((e) -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
    }
}
```



The operations in Streams can be classified as:

- **Intermediate Operations** – These are ones, which return another stream, and can be chained together.
  For example filter(), sort(), map(), etc.
- **Terminal Operations** – These are the ones, which produce a result from the pipeline. This result can be any non-stream value like List, Integer, void, etc.
  forEach() and collect() methods are terminal operations.



Intermediate and Terminal Operations

Intermediate operations are lazy, i.e. they do not perform any processing until a terminal operation is called on the stream. This may improve performance, as a stream is not processed until required.

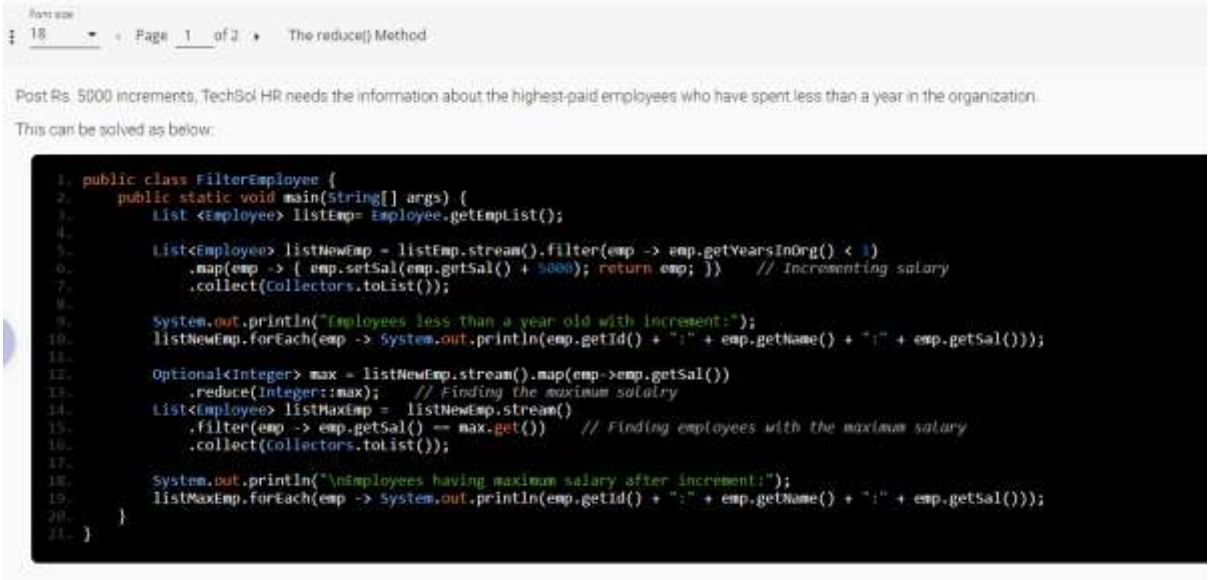
We have already seen a few intermediate operations like filter, map and sort. There are also a few more:

Operation	Description	Example

Stream<T> limit(long maxSize)	Returns a stream of maxSize length	Limits to the first 100 employees:
		listEmp.stream()
		.filter(emp.getYearsInOrg() > 1)
		.sorted((e1,e2) -> e1.getId() - e2.getId())
		.limit(100)
		.collect(Collectors.toList());

Stream<T> distinct()	Returns a stream with distinct elements that is based on Object's equals() method	To get employees with unique names (assuming equals() method is overridden): listEmp.stream().distinct();

We have already seen the forEach() and collect() terminal operations.

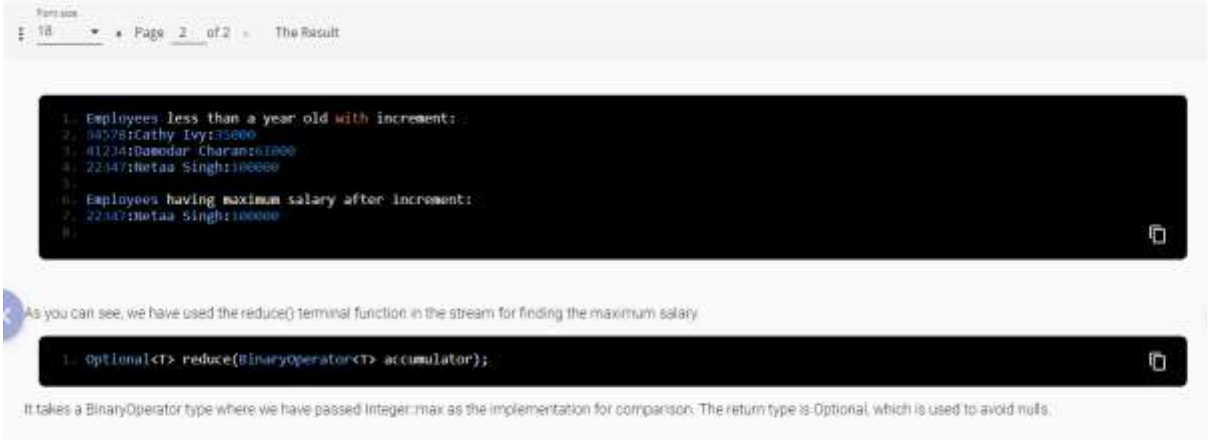


The reduce() Method

Post Rs. 5000 increments, TechSol HR needs the information about the highest-paid employees who have spent less than a year in the organization.

This can be solved as below:

```java
public class FilterEmployee {
    public static void main(String[] args) {
        List <Employee> listEmp= Employee.getEmpList();

        List<Employee> listNewEmp = listEmp.stream().filter(emp -> emp.getYearsInOrg() < 1)
            .map(emp -> { emp.setSal(emp.getSal() + 5000); return emp; })    // Incrementing salary
            .collect(Collectors.toList());

        System.out.println("Employees less than a year old with increment:");
        listNewEmp.forEach(emp -> System.out.println(emp.getId() + ":" + emp.getName() + ":" + emp.getSal()));

        Optional<Integer> max = listNewEmp.stream().map(emp->emp.getSal())
            .reduce(Integer::max);    // Finding the maximum salalry
        List<Employee> listMaxEmp =  listNewEmp.stream()
            .filter(emp -> emp.getSal() == max.get())    // Finding employees with the maximum salary
            .collect(Collectors.toList());

        System.out.println("\nEmployees having maximum salary after increment:");
        listMaxEmp.forEach(emp -> System.out.println(emp.getId() + ":" + emp.getName() + ":" + emp.getSal()));
    }
}
```



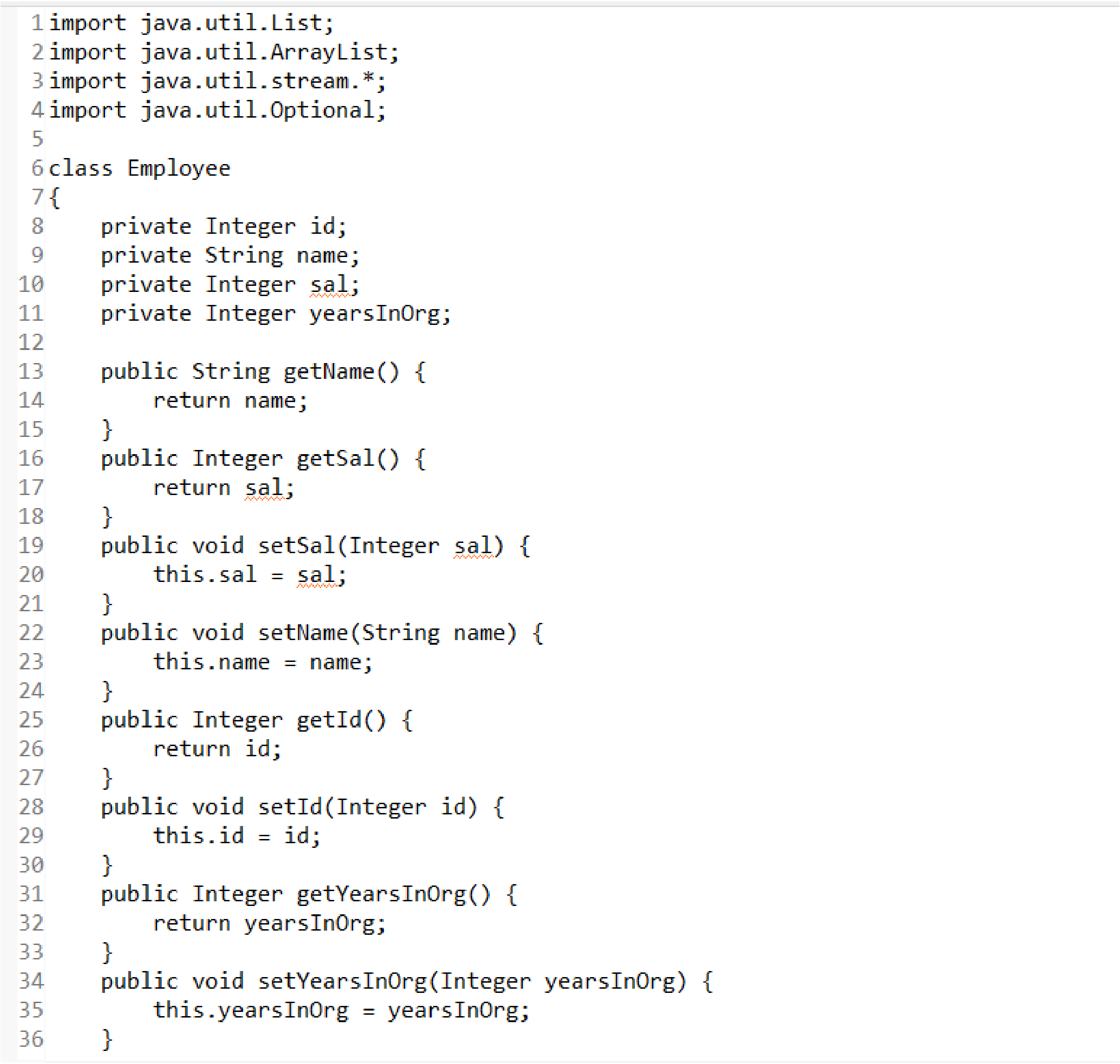
The Result

```
Employees less than a year old with increment:
34578:Cathy Ivy:35000
41234:Damodar Charan:61000
22347:Netaa Singh:100000

Employees having maximum salary after increment:
22347:Netaa Singh:100000

```

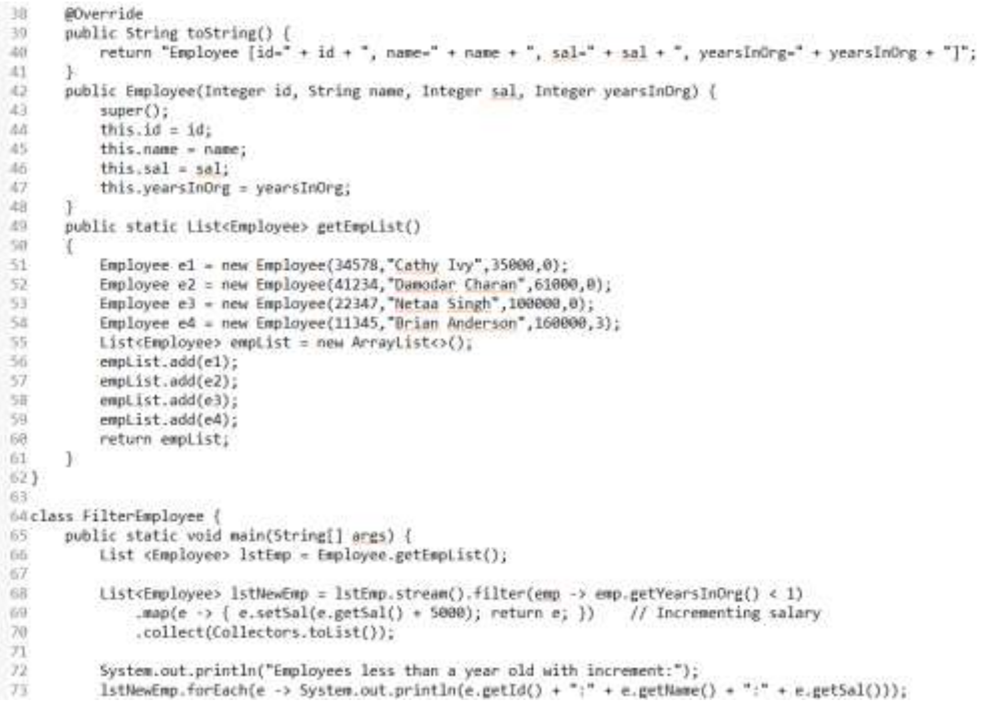
As you can see, we have used the reduce() terminal function in the stream for finding the maximum salary.

```java
Optional<T> reduce(BinaryOperator<T> accumulator);
```

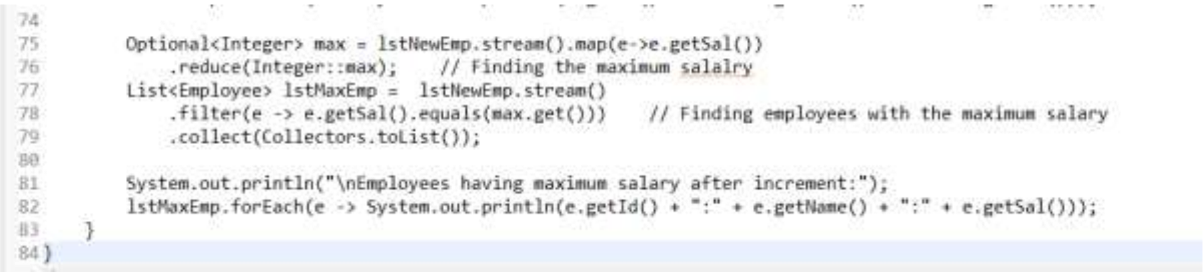
It takes a BinaryOperator type where we have passed Integer::max as the implementation for comparison. The return type is Optional, which is used to avoid nulls.



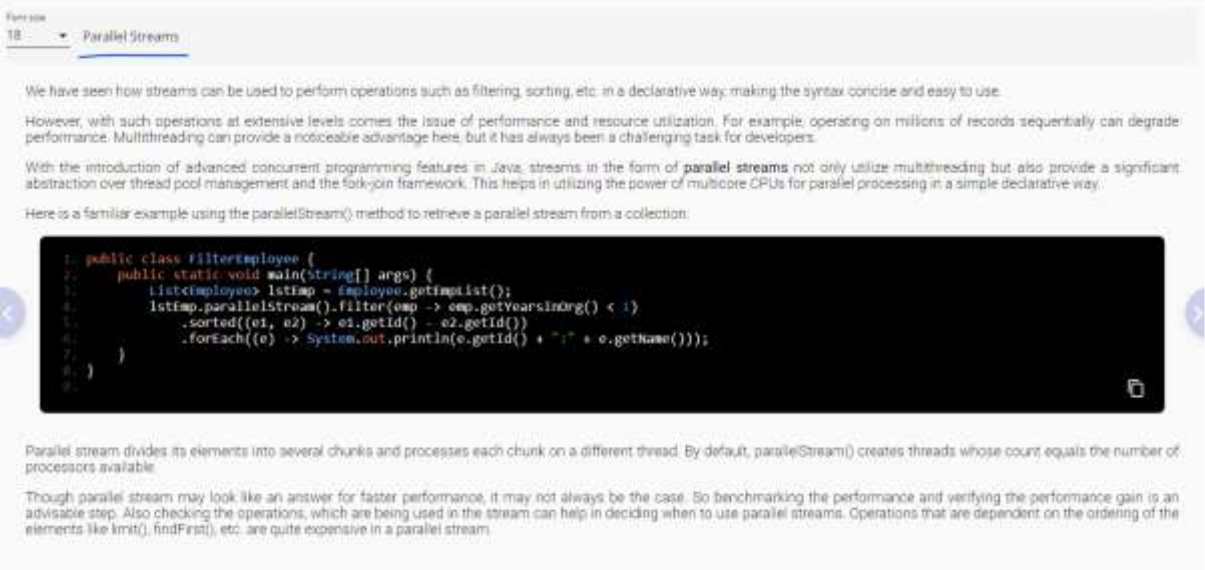
```java
import java.util.List;
import java.util.ArrayList;
import java.util.stream.*;
import java.util.Optional;

class Employee
{
    private Integer id;
    private String name;
    private Integer sal;
    private Integer yearsInOrg;

    public String getName() {
        return name;
    }
    public Integer getSal() {
        return sal;
    }
    public void setSal(Integer sal) {
        this.sal = sal;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public Integer getYearsInOrg() {
        return yearsInOrg;
    }
    public void setYearsInOrg(Integer yearsInOrg) {
        this.yearsInOrg = yearsInOrg;
    }
```



```java
    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", sal=" + sal + ", yearsInOrg=" + yearsInOrg + "]";
    }
    public Employee(Integer id, String name, Integer sal, Integer yearsInOrg) {
        super();
        this.id = id;
        this.name = name;
        this.sal = sal;
        this.yearsInOrg = yearsInOrg;
    }
    public static List<Employee> getEmpList()
    {
        Employee e1 = new Employee(34578,"Cathy Ivy",35000,0);
        Employee e2 = new Employee(41234,"Damodar Charan",61000,0);
        Employee e3 = new Employee(22347,"Netaa Singh",100000,0);
        Employee e4 = new Employee(11345,"Brian Anderson",160000,3);
        List<Employee> empList = new ArrayList<>();
        empList.add(e1);
        empList.add(e2);
        empList.add(e3);
        empList.add(e4);
        return empList;
    }
}

class FilterEmployee {
    public static void main(String[] args) {
        List <Employee> lstEmp = Employee.getEmpList();

        List<Employee> lstNewEmp = lstEmp.stream().filter(emp -> emp.getYearsInOrg() < 1)
            .map(e -> { e.setSal(e.getSal() + 5000); return e; })    // Incrementing salary
            .collect(Collectors.toList());

        System.out.println("Employees less than a year old with increment:");
        lstNewEmp.forEach(e -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
```



```java

        Optional<Integer> max = lstNewEmp.stream().map(e->e.getSal())
            .reduce(Integer::max);    // Finding the maximum salalry
        List<Employee> lstMaxEmp =  lstNewEmp.stream()
            .filter(e -> e.getSal().equals(max.get()))    // Finding employees with the maximum salary
            .collect(Collectors.toList());

        System.out.println("\nEmployees having maximum salary after increment:");
        lstMaxEmp.forEach(e -> System.out.println(e.getId() + ":" + e.getName() + ":" + e.getSal()));
    }
}
```



We have seen how streams can be used to perform operations such as filtering, sorting, etc. in a declarative way, making the syntax concise and easy to use.

However, with such operations at extensive levels comes the issue of performance and resource utilization. For example, operating on millions of records sequentially can degrade performance. Multithreading can provide a noticeable advantage here, but it has always been a challenging task for developers.

With the introduction of advanced concurrent programming features in Java, streams in the form of **parallel streams** not only utilize multithreading but also provide a significant abstraction over thread pool management and the fork-join framework. This helps in utilizing the power of multicore CPUs for parallel processing in a simple declarative way.

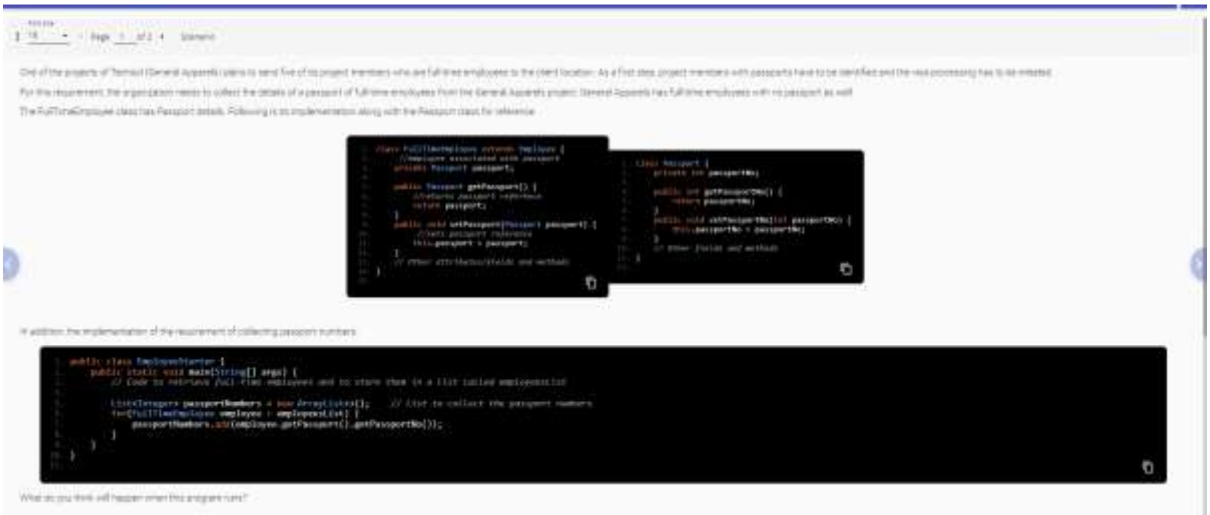
Here is a familiar example using the parallelStream() method to retrieve a parallel stream from a collection:

```java
public class FilterEmployee {
    public static void main(String[] args) {
        List<Employee> lstEmp = Employee.getEmpList();
        lstEmp.parallelStream().filter(emp -> emp.getYearsInOrg() < 1)
            .sorted((e1, e2) -> e1.getId() - e2.getId())
            .forEach((e) -> System.out.println(e.getId() + ":" + e.getName()));
    }
}
```

Parallel stream divides its elements into several chunks and processes each chunk on a different thread. By default, parallelStream() creates threads whose count equals the number of processors available.

Though parallel stream may look like an answer for faster performance, it may not always be the case. So benchmarking the performance and verifying the performance gain is an advisable step. Also checking the operations, which are being used in the stream can help in deciding when to use parallel streams. Operations that are dependent on the ordering of the elements like limit(), findFirst(), etc. are quite expensive in a parallel stream.

Optional Class:



One of the projects of [illegible] (General Apparels) plans to send five of its project members who are full-time employees to the client location. As a first step, project members with passports have to be identified and the visa processing has to be initiated.

For this requirement, the organization needs to collect the details of a passport of full-time employees from the General Apparels project. General Apparels has full-time employees with no passport as well.

The FullTimeEmployee class has Passport details. Following is its implementation along with the Passport class for reference.

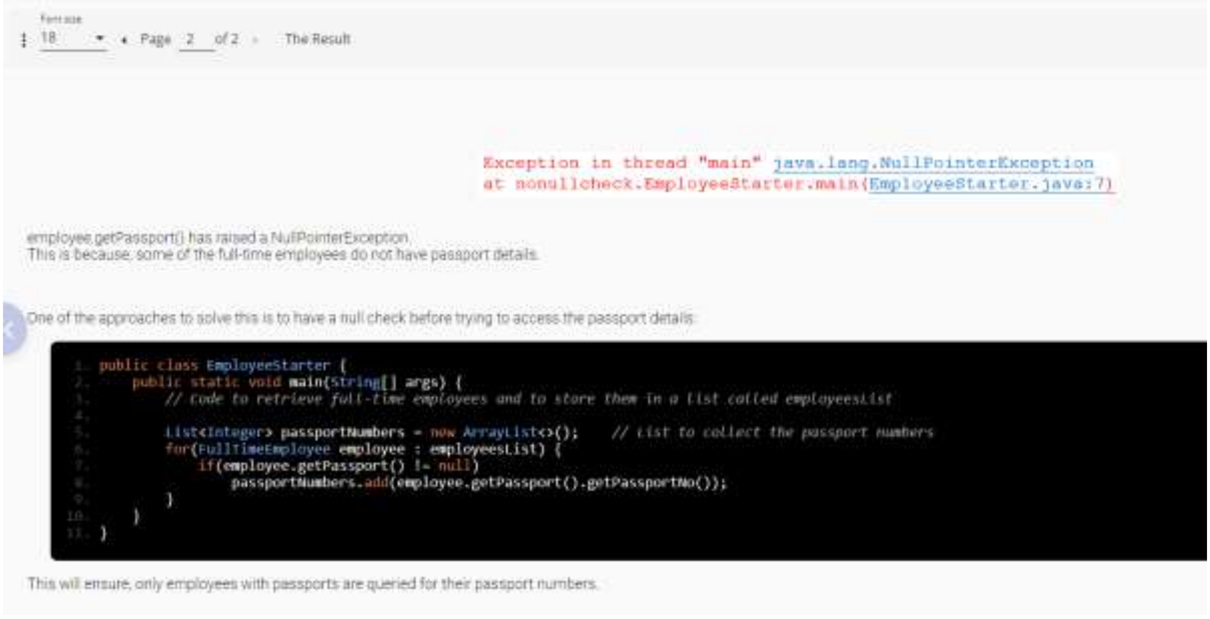
```java
class FullTimeEmployee extends Employee {
    //employee associated with passport
    private Passport passport;

    public Passport getPassport() {
        //returns passport reference
        return passport;
    }
    public void setPassport(Passport passport) {
        //sets passport reference
        this.passport = passport;
    }
    // Other attributes/fields and methods
}
```

```java
class Passport {
    private int passportNo;

    public int getPassportNo() {
        return passportNo;
    }
    public void setPassportNo(int passportNo) {
        this.passportNo = passportNo;
    }
    // Other fields and methods
}
```

In addition, the implementation of the requirement of collecting passport numbers

```java
public class EmployeeStarter {
    public static void main(String[] args) {
        // Code to retrieve full-time employees and to store them in a list called employeesList

        List<Integer> passportNumbers = new ArrayList<>();    // List to collect the passport numbers
        for(FullTimeEmployee employee : employeesList) {
            passportNumbers.add(employee.getPassport().getPassportNo());
        }
    }
}
```

What do you think will happen when this program runs?



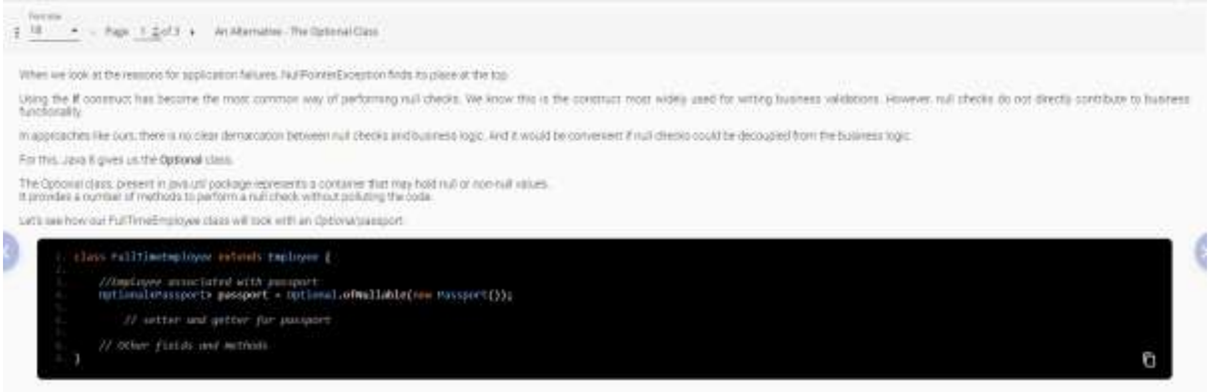
```
Exception in thread "main" java.lang.NullPointerException
at nonullcheck.EmployeeStarter.main(EmployeeStarter.java:7)
```

employee.getPassport() has raised a NullPointerException.
This is because, some of the full-time employees do not have passport details.

One of the approaches to solve this is to have a null check before trying to access the passport details:

```java
public class EmployeeStarter {
    public static void main(String[] args) {
        // Code to retrieve full-time employees and to store them in a list called employeesList

        List<Integer> passportNumbers = new ArrayList<>();    // List to collect the passport numbers
        for(FullTimeEmployee employee : employeesList) {
            if(employee.getPassport() != null)
                passportNumbers.add(employee.getPassport().getPassportNo());
        }
    }
}
```

This will ensure, only employees with passports are queried for their passport numbers.



When we look at the reasons for application failures, NullPointerException finds its place at the top.

Using the if construct has become the most common way of performing null checks. We know this is the construct most widely used for writing business validations. However, null checks do not directly contribute to business functionality.

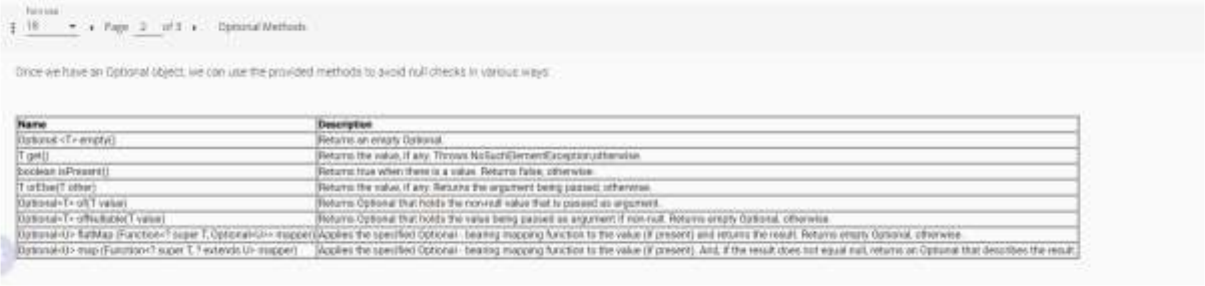
In approaches like ours, there is no clear demarcation between null checks and business logic. And it would be convenient if null checks could be decoupled from the business logic.

For this, Java 8 gives us the **Optional** class.

The Optional class, present in java.util package represents a container that may hold null or non-null values.
It provides a number of methods to perform a null check without polluting the code.

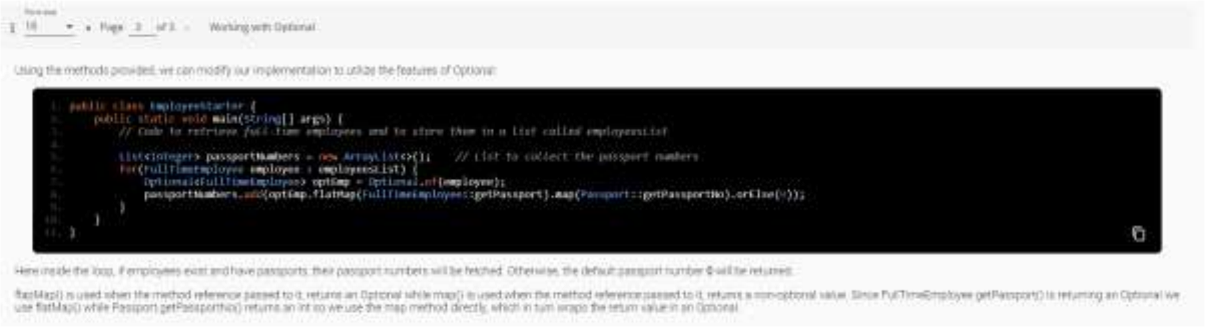
Let's see how our FullTimeEmployee class will look with an Optional<passport>

```java
class FullTimeEmployee extends Employee {

    //Employee associated with passport
    Optional<Passport> passport = Optional.ofNullable(new Passport());

        // setter and getter for passport

    // Other fields and methods
}
```



Once we have an Optional object, we can use the provided methods to avoid null checks in various ways:

Name	Description
Optional <T> empty()	Returns an empty Optional.
T get()	Returns the value, if any. Throws NoSuchElementException,otherwise.
boolean isPresent()	Returns true when there is a value. Returns false, otherwise.
T orElse(T other)	Returns the value, if any. Returns the argument being passed, otherwise.
Optional<T> of(T value)	Returns Optional that holds the non-null value that is passed as argument.
Optional<T> ofNullable(T value)	Returns Optional that holds the value being passed as argument if non-null. Returns empty Optional, otherwise.
Optional<U> flatMap (Function<? super T, Optional<U>> mapper)	Applies the specified Optional - bearing mapping function to the value (if present) and returns the result. Returns empty Optional, otherwise.
Optional<U> map (Function<? super T, ? extends U> mapper)	Applies the specified Optional - bearing mapping function to the value (if present). And, if the result does not equal null, returns an Optional that describes the result.

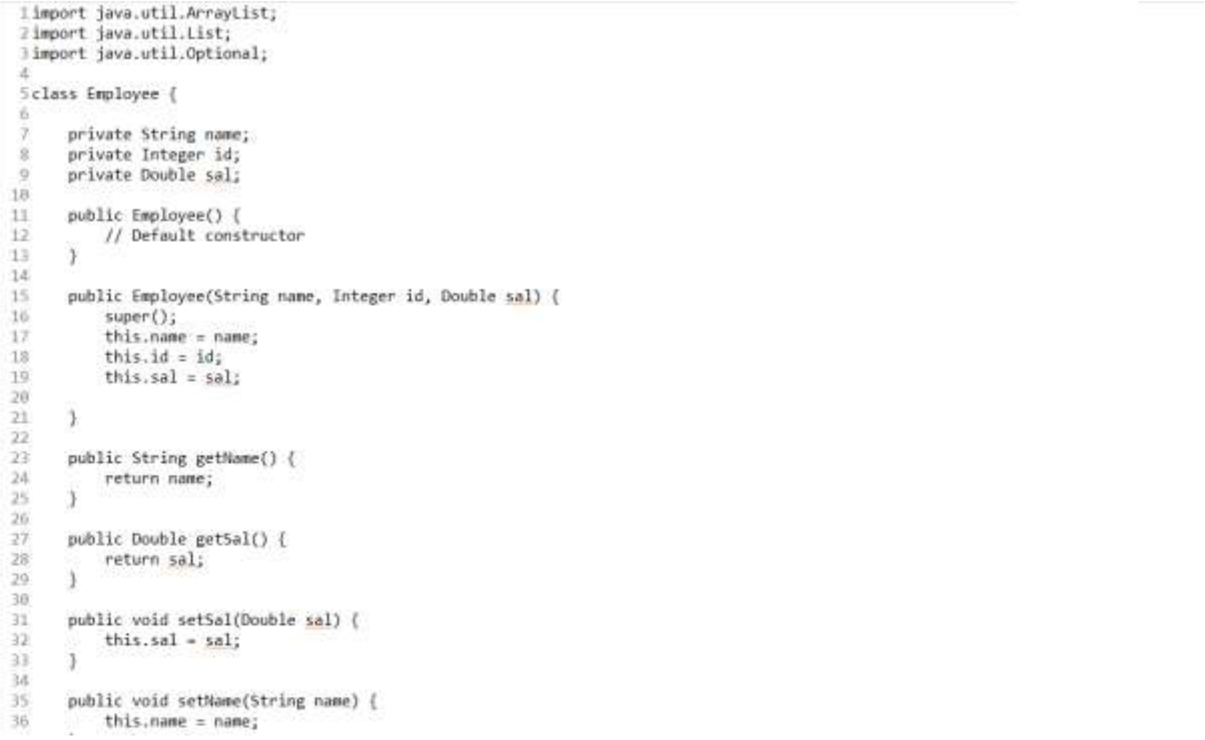


Using the methods provided, we can modify our implementation to utilize the features of Optional:

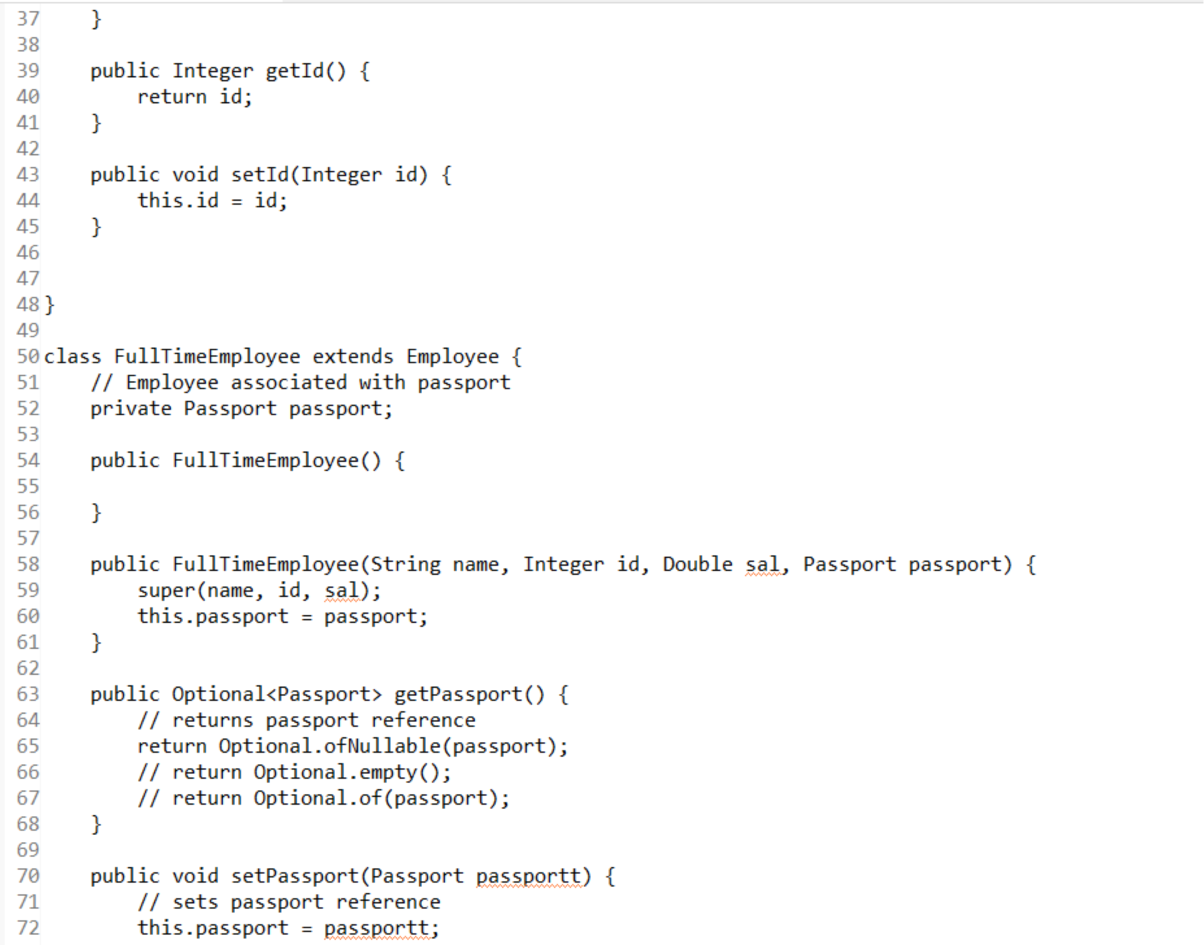
```
public class EmployeeStarter {
    public static void main(String[] args) {
        // Code to retrieve full-time employees and to store them in a list called employeesList

        List<Integer> passportNumbers = new ArrayList<>();    // List to collect the passport numbers
        for(FullTimeEmployee employee : employeesList) {
            Optional<FullTimeEmployee> optEmp = Optional.of(employee);
            passportNumbers.add(optEmp.flatMap(FullTimeEmployee::getPassport).map(Passport::getPassportNo).orElse(0));
        }
    }
}
```

Here inside the loop, if employees exist and have passports, their passport numbers will be fetched. Otherwise, the default passport number 0 will be returned.

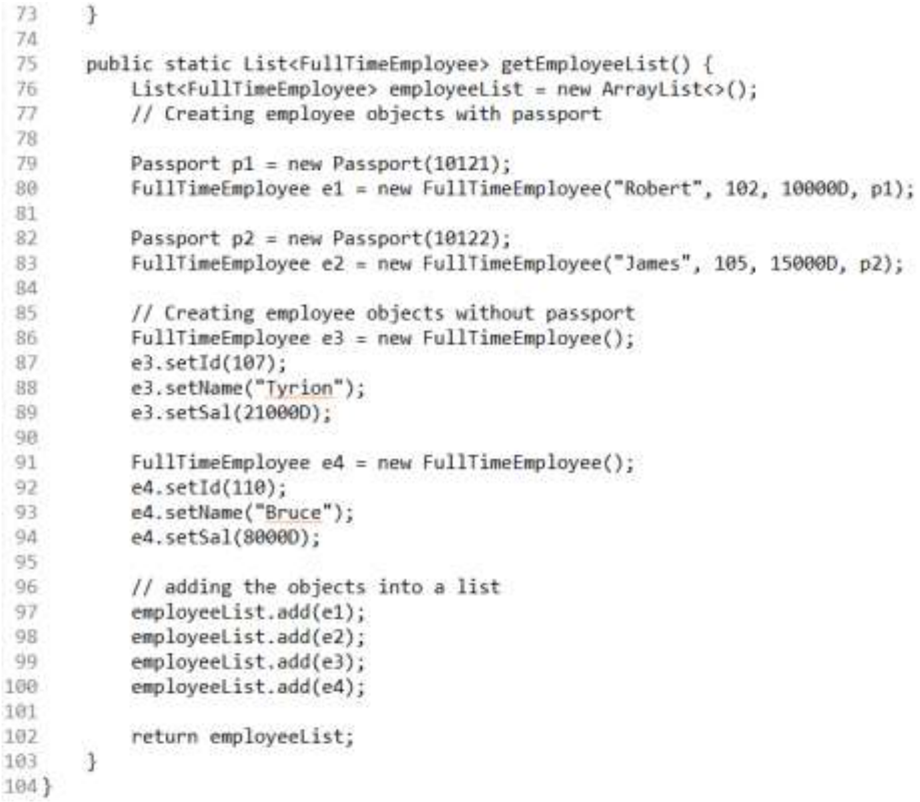
flatMap() is used when the method reference passed to it, returns an Optional while map() is used when the method reference passed to it, returns a non-optional value. Since FullTimeEmployee.getPassport() is returning an Optional we use flatMap() while Passport.getPassportNo() returns an int so we use the map method directly, which in turn wraps the return value in an Optional.



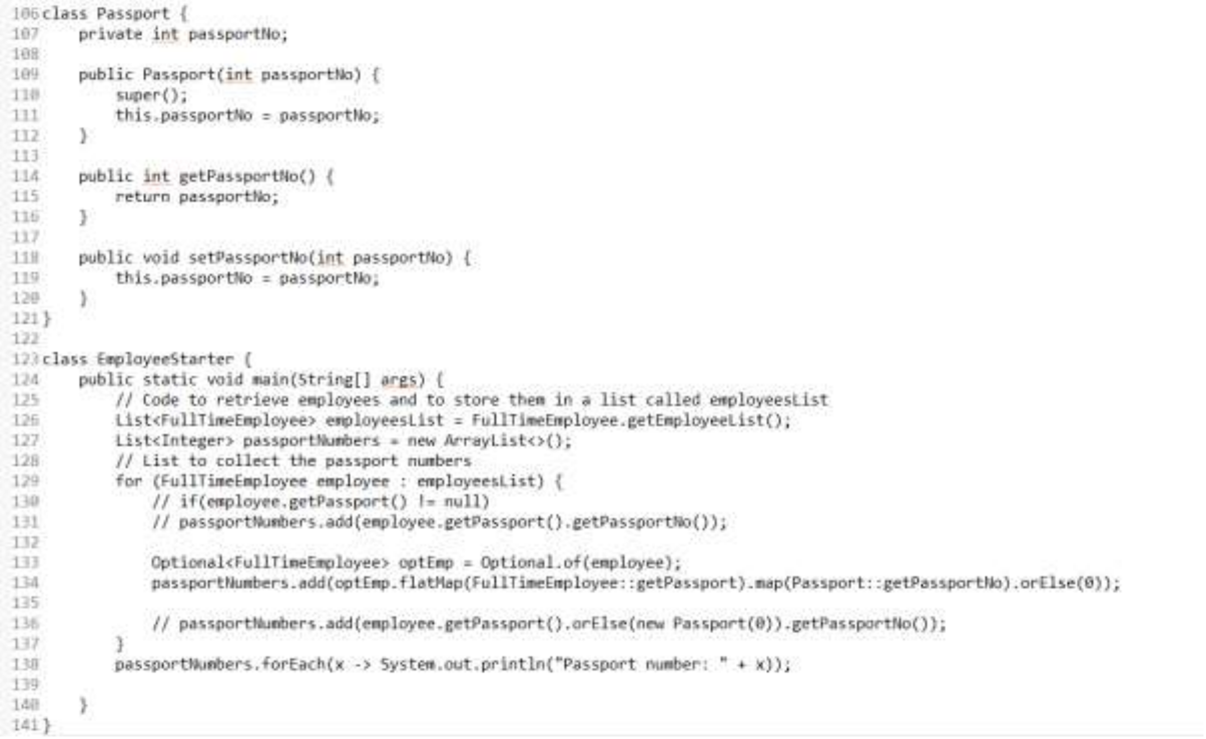
```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

class Employee {

    private String name;
    private Integer id;
    private Double sal;

    public Employee() {
        // Default constructor
    }

    public Employee(String name, Integer id, Double sal) {
        super();
        this.name = name;
        this.id = id;
        this.sal = sal;

    }

    public String getName() {
        return name;
    }

    public Double getSal() {
        return sal;
    }

    public void setSal(Double sal) {
        this.sal = sal;
    }

    public void setName(String name) {
        this.name = name;
```



```
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }


}

class FullTimeEmployee extends Employee {
    // Employee associated with passport
    private Passport passport;

    public FullTimeEmployee() {

    }

    public FullTimeEmployee(String name, Integer id, Double sal, Passport passport) {
        super(name, id, sal);
        this.passport = passport;
    }

    public Optional<Passport> getPassport() {
        // returns passport reference
        return Optional.ofNullable(passport);
        // return Optional.empty();
        // return Optional.of(passport);
    }

    public void setPassport(Passport passportt) {
        // sets passport reference
        this.passport = passportt;
```



```
    }

    public static List<FullTimeEmployee> getEmployeeList() {
        List<FullTimeEmployee> employeeList = new ArrayList<>();
        // Creating employee objects with passport

        Passport p1 = new Passport(10121);
        FullTimeEmployee e1 = new FullTimeEmployee("Robert", 102, 10000D, p1);

        Passport p2 = new Passport(10122);
        FullTimeEmployee e2 = new FullTimeEmployee("James", 105, 15000D, p2);

        // Creating employee objects without passport
        FullTimeEmployee e3 = new FullTimeEmployee();
        e3.setId(107);
        e3.setName("Tyrion");
        e3.setSal(21000D);

        FullTimeEmployee e4 = new FullTimeEmployee();
        e4.setId(110);
        e4.setName("Bruce");
        e4.setSal(8000D);

        // adding the objects into a list
        employeeList.add(e1);
        employeeList.add(e2);
        employeeList.add(e3);
        employeeList.add(e4);

        return employeeList;
    }
}
```



```java
class Passport {
    private int passportNo;

    public Passport(int passportNo) {
        super();
        this.passportNo = passportNo;
    }

    public int getPassportNo() {
        return passportNo;
    }

    public void setPassportNo(int passportNo) {
        this.passportNo = passportNo;
    }
}

class EmployeeStarter {
    public static void main(String[] args) {
        // Code to retrieve employees and to store them in a list called employeesList
        List<FullTimeEmployee> employeesList = FullTimeEmployee.getEmployeeList();
        List<Integer> passportNumbers = new ArrayList<>();
        // List to collect the passport numbers
        for (FullTimeEmployee employee : employeesList) {
            // if(employee.getPassport() != null)
            // passportNumbers.add(employee.getPassport().getPassportNo());

            Optional<FullTimeEmployee> optEmp = Optional.of(employee);
            passportNumbers.add(optEmp.flatMap(FullTimeEmployee::getPassport).map(Passport::getPassportNo).orElse(0));

            // passportNumbers.add(employee.getPassport().orElse(new Passport(0)).getPassportNo());
        }
        passportNumbers.forEach(x -> System.out.println("Passport number: " + x));

    }
}
```